2013-01-08

Improving regulator verification and compact representations in real quadratic fields

Silvester, Alan

UNIVERSITY OF CALGARY

Improving regulator verification and compact representations in real quadratic fields

by

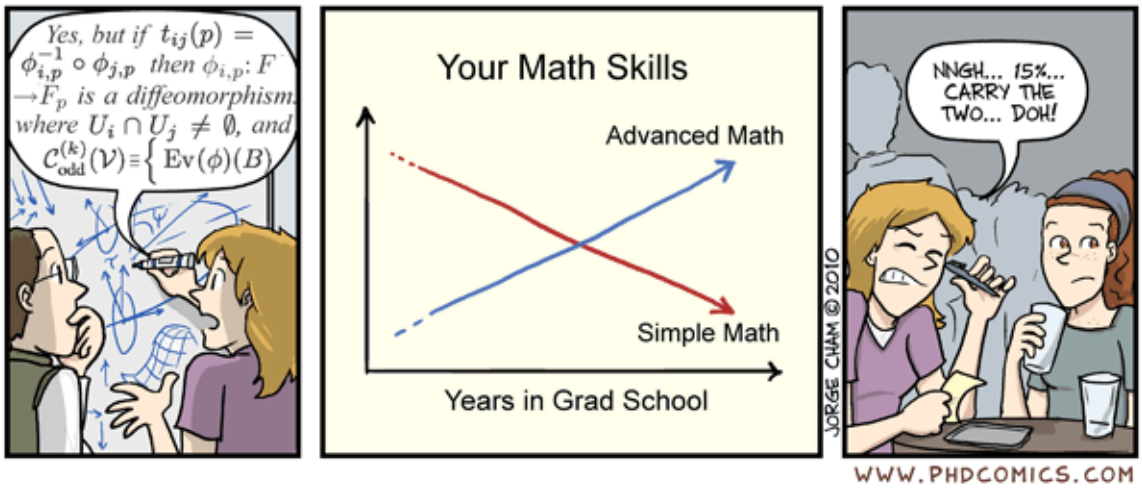
Alan Silvester

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF MATHEMATICS AND STATISTICS

CALGARY, ALBERTA
December, 2012

PREFACE



"Piled Higher and Deeper" by Jorge Cham.
www.phdcomics.com
Reprinted with permission.

Acknowledgements

I would like to express my sincere thanks to my supervisor Dr. Hugh Williams for his guidance throughout my graduate program, as well as for the generous funding support and research and travel opportunities I have enjoyed over the years. I would also like to thank Dr. Michael Jacobson, Jr., for agreeing to be my co-supervisor during the final years of my program and for all the advice, comments, corrections, and suggestions he has given me to enhance the quality of this thesis.

I would like to thank the Province of Alberta for the funding support I received through the Queen Elizabeth II Graduate Scholarship Program.

I would like to thank Compute Canada and WestGrid, the Western Canada Research Grid, for allowing me access to some of their high-performance computing resources. While I only started using these resources towards the tail-end of my program, they will be instrumental to my continuing work on the material presented in this thesis.

Thank you to Dan Bidulock, Patrick Boyle, Anne-Marie Bruzga, James Lange, Floribert Kamabu, Valerie McGillivray, Lindsay Penner, and Angela Vinturache of the Graduate Students' Association (GSA) and GSA Representative Council (GRC) for making my time serving on the GRC and the GSA's Objects and Bylaws Review Committee that much more enjoyable.

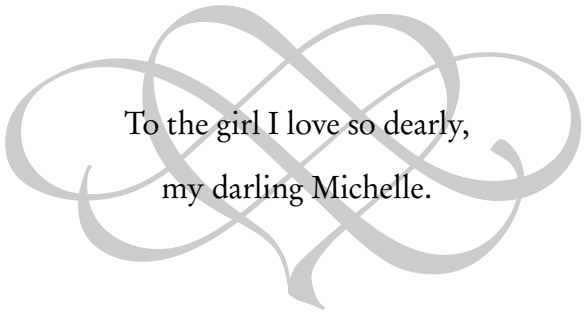
Finally, thank you to:

- The Penguins: Chris, Dave, and Tremayne, for their friendship and camaraderie over the years.

- Andrea, for always making us feel at home.
- Matt, for the prime rib requests.
- Tina, for taking care of the final paperwork for my degree.
- Big Rock Brewery: *Alberta's* other *natural resource.*
- The Yardhouse: Real food, real beer.
- The Last Defence Lounge.
- Digitally Imported, Inc.
- ISO 3103:1980, 6668:2008, 9899:1999/Cor 3:2007 and 32000-1:2008.
- Saccharomyces eubayanus.

—Alan Silvester

Dedication



To the girl I love so dearly,
my darling Michelle.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Epigraph

"Sir, our math shows that the bird is equal to or greater than the word."

"Check it again!"

"This is what happens, Larry! This is what happens when you find a stranger in the Alps!"

—Walter Sobchak

— Chapter 1 —
Introduction

A Diophantine equation is an indeterminate polynomial equation whose unknowns are restricted to integral values only. The study of these equations, named after the Hellenistic mathematician Diophantus of Alexandria, has a long and intriguing history that, unfortunately, we will not delve into. For the purposes of this thesis, we are most interested in a particular Diophantine equation, namely the *Pell equation*

$$T^2 - DU^2 = 1\,. \tag{1.1}$$

Due to an incorrect attribution by Euler, these equations were named after John Pell and though "it is both historically wrong and unjust to those early individuals who did make important contributions to its study" [48, p. 5], the name has stuck. It is clear that (1.1) has trivial solutions, that is $(T,U) = (\pm 1, 0)$, but what is not immediately evident is that it always has at least one non-trivial solution [48, Thm. 1.3, p. 6]. The *fundamental solution* of (1.1) is the solution (t,u) with $t,u > 0$ and $t + u\sqrt{D}$ minimal. Using this fundamental solution, we can characterize any other solutions: if (T,U) is a solution of (1.1), then for some $n \in \mathbb{Z}^+$ and choice of sign, $T + U\sqrt{D} = \pm(t + u\sqrt{D})^n$.

At this point, we also highlight a particular instance of the Pell equation: $T^2 - 410286423278424U^2 = 1$. This Pell equation represents the translation into a modern mathematical setting of Archimedes' Cattle Problem. In a letter addressed to Eratosthenes of Cyrene, Archimedes laid out a computational challenge in the form of a "lightly

satirical" [48, p. 23] epigram. The challenge to the reader is to find a solution to the number of cattle of the Sun, divided into four differently coloured herds (say W, X, Y, Z) of bulls and cows (upper- and lower-case, respectively) whose ratios are given by

$$W = \left(\frac{1}{2}+\frac{1}{3}\right)X + Z\,, \qquad X = \left(\frac{1}{4}+\frac{1}{5}\right)Y + Z\,, \qquad Y = \left(\frac{1}{6}+\frac{1}{7}\right)W + Z\,,$$
$$w = \left(\frac{1}{3}+\frac{1}{4}\right)(X+x)\,, \qquad x = \left(\frac{1}{4}+\frac{1}{5}\right)(Y+y)\,, \qquad y = \left(\frac{1}{5}+\frac{1}{6}\right)(Z+z)\,,$$
$$z = \left(\frac{1}{6}+\frac{1}{7}\right)(W+w)\,,$$

$W + X$ is a perfect square and $Y + Z$ is a *triangular number*; that is, an integer of the form $n(n+1)/2$ for some $n \geq 1$.

With the appropriate sequence of algebraic steps, we can combine and reduce these equations to the Pell equation we stated previously. The solutions related to this equation will serve as nice examples for a number of the calculations and algorithms that appear in the chapters to follow. They are large enough to illustrate the details of several techniques, but still easily manageable with only minor assistance from a computer. We end this section with the comment that the Cattle Problem has been used as an illustrative example previously in the literature. See, for example, [57] and of course [48].

Diophantine equations arise in several areas of mathematics, two applications of which we discuss here. The general binary quadratic Diophantine equation has the form

$$ax^2 + bxy + cy^2 + dx + ey + f = 0\,. \tag{1.2}$$

It was first solved by Joseph Lagrange over 200 years ago [54] and, as the authors of [73]

point out, the problem of solving (1.2) can be reduced to determining whether or not an ideal in a quadratic order is principal. This in turn requires us to determine, in essence, solutions to a Pell equation. If this ideal is in fact principal, solutions to (1.2) can be determined by exhibiting its generator.

Another interesting application is that of finding integer points on elliptic curves, also by way of principal ideal testing. For several parametric families of elliptic curves, it has been shown that the integral points on the curves are given by *Diophantine m-tuples*; for example, [28], [33], [47], [68] and [77]. These are a set of m positive integers such that the product of any two elements increased by 1 is a perfect square [28]. To prove the results cited above, the authors show that a given system of equations has no solutions. If we have such a solution, then we can construct a principal ideal of specific norm in a certain quadratic field. Working in the opposite direction, we instead find all the ideals with the given norm and test them for principality. If none are principal, then the system of equations is insoluble.

1.1. Unconditionally correct solutions of the Pell equation

Solutions to the Pell equation can be computed by looking at the solutions of a similar Diophantine equation, specifically

$$X^2 - DY^2 = 4\sigma \,, \tag{1.3}$$

where $\sigma \in \{\pm 1\}$.

In the case of (1.3), the fundamental solution is the solution (x, y) for which $x + y\sqrt{D} > 2$ and is least. Let $\eta_0 = (x + y\sqrt{D})/2$, a quantity called the *fundamental unit*. It can be shown [48, Cor. 1.10, p. 11, Tbl. 1.1, p. 13] that if (t, u) is the fundamental solu-

tion to (1.1) and (x, y) is the fundamental solution of (1.3), then for some $k \in \{1, 2, 3, 6\}$,

$$t + u\sqrt{D} = \eta_0^k \, .$$

However, as we will see in later chapters, just because a non-trivial solution to these equations exists, this does not mean it is easy to compute. Far from it, in fact. For computational reasons we will not go into at this point, we usually prefer to determine a quantity known as the regulator and use that to calculate η_0, if it is needed. It is known that computing the regulator, particularly when $D > 10^{25}$, is very difficult. As we will see from the results presented in Chapter 2, we expect for a large proportion of real quadratic fields that the regulator is much greater than $D^{1/2-\epsilon}$ [48, (9.26), p. 230]. In recent years, several authors have devised cryptographic applications whose security is based on the difficulty of these and other related computations. We refer the interested reader to such works as [8], [11], [38], [45], and [46], though this list is far from definitive.

Computing the regulator unconditionally is not important for these cryptographic applications, though it is important when trying to find solutions to Diophantine equations. As an example of this, we briefly discuss a problem considered by Jacobson and Williams in [47]. They wished to show that the equation

$$d_1 {x_3}^2 - d_3 {x_2}^2 = \frac{d_3 j_1 - d_1 j_2}{j_2} = c \, , \tag{1.4}$$

where d_1, d_3, j_1, j_2, and c are given integer constants, has no integer solutions. To do this, it is sufficient to show that the ideals of norm cd_1 in a certain quadratic order were not principal. Using the index-calculus algorithm, they determined an approximation to the regulator and found a total of 8 candidate ideals, none of which were principal. However,

the correctness of this approximation depends on a certain unproven Riemann hypothesis. The best they could say was that (1.4) had no integer solutions assuming the truth of this hypothesis. By using the unconditional regulator verification algorithm outlined later in this section—and in much more detail in Section 3.6—and an unconditional principal ideal testing algorithm [73, 77], they were able to show unconditionally that (1.4) has no integer solutions [73, p. 54–5].

In the remainder of this section, we will present an overview of the important advances made in regulator computations, both conditional and unconditional. Let $\mathbb{K} = \mathbb{Q}(\sqrt{D})$ be a real quadratic number field. By applying the continued fraction algorithm to $\sqrt{D}$, one can compute the regulator in time $O(D^{1/2+\epsilon})$. The development of this technique traces back through some well-known mathematicians, such as Euler and Lagrange, as well as many ancient Indian and Greek mathematicians. For those interested in this rich history, we refer to [48, Ch. 2 and §§3.1–3.3, pp. 19–62] and the citations found there. Using the infrastructure and the idea of baby-steps and giant-steps, Daniel Shanks [75] was able to produce an algorithm for computing the regulator unconditionally in time $O(D^{1/4+\epsilon})$. Using the infrastructure and an estimate of the character sum $L(1,(\frac{\Delta}{n}))$, where $(\frac{\Delta}{n})$ is the Kronecker symbol, Hendrik Lenstra in 1982 [56] was able to produce a regulator algorithm which runs in time $O(D^{1/5+\epsilon})$. The original ideas for the index-calculus algorithm, usually presented in terms of binary quadratic forms, were presented by Arjen and Hendrik Lenstra in 1987 [55], with more details filled in by James Hafner and Kevin McCurley in 1989 [35]. Johannes Buchmann further expanded the real quadratic field case [10]. The running time of this algorithm is $L_\Delta[1/2, \sqrt{2}+o(1)]$ where $L_\Delta[u,v] = \exp(v(\log|\Delta|)^u(\log\log|\Delta|)^{1-u})$. This increase in speed comes at a price, however: the output is conditional on the truth of the generalized Riemann hypothesis.

In 2003, Michael Jacobson, Jr., Ákos Pintér and Gary Walsh [44] presented an algorithm which unconditionally verifies the value of the regulator R' computed by the index-calculus algorithm. This was adapted by Robbert de Haan, Michael Jacobson, Jr., and Hugh Williams [21, 22] to work with (f, p) representations, and included a refined optimization of the algorithm's parameters. This algorithm, with expected running time in $O(\Delta^{1/6+\epsilon})$, is the one we are most interested in for this thesis.

The main ideas behind the algorithms in [21] and [22] are as follows. Unconditionally, we can say the approximation R' is a multiple of the actual regulator R. The first step is to verify that R' is less than an explicit upper bound and then determine a lower bound on the value of R by way a of a baby-step / giant-step algorithm. We also verify that R' is close enough to an integer multiple of R—specifically $|R' - cR| < 1$ for some $c \in \mathbb{Z}^+$—and, if not, determine a new value for R' that does satisfy this condition. Once we have that R' is close to an exact multiple of R and have bounds for R, we need to determine the value of c. In essence, this is done by trial dividing R' by a series of potential prime divisors. By keeping track of which numbers have been successfully tested, we compute the prime power factorization of c.

We obviously glossed over many details in the preceding description, in particular those surrounding the baby-step / giant-step algorithm used to find a lower bound on R. One particular optimization concerns storing the baby-step list. We compute and store a list of ideals $\mathcal{L} = \{\mathfrak{a}_1, \mathfrak{a}_2, \ldots, \mathfrak{a}_t, \mathfrak{a}_{t+1}, \mathfrak{a}_{t+2}\}$. When actually implementing this step, we have only a limited amount of memory available. If D becomes large enough, we will run out of room to store $\mathcal{L}$. To overcome this, de Haan noted that gaps could be introduced in $\mathcal{L}$ without adding much algorithm overhead. We instead store every l^{th} ideal in the list $\mathcal{L}' = \{\mathfrak{a}_1, \mathfrak{a}_l, \mathfrak{a}_{2l}, \ldots, \mathfrak{a}_{Nl}\} \cup \{\mathfrak{a}_t, \mathfrak{a}_{t+1}, \mathfrak{a}_{t+2}\}$, where N is such that $Nl \leq t <$

$(N+1)l$. After computing the giant steps, if we have an ideal $\mathfrak{a}_j \in \mathcal{L}'$—or $\overline{\mathfrak{a}}_j \in \mathcal{L}'$, but for simplicity's sake we will ignore this for now—then we must have at least one ideal in the set $\mathcal{N} = \{\mathfrak{a}_j, \mathfrak{a}_{j+1}, \ldots, \mathfrak{a}_{j+l-1}\}$ in $\mathcal{L}'$. Thus, we replace a search for $\mathfrak{a}_j$ in $\mathcal{L}$ by an iteration attempting to match an ideal in $\mathcal{N}$ with an ideal in $\mathcal{L}'$.

Using the $O(D^{1/6+\epsilon})$ algorithm and the preceding improvements, de Haan was able to unconditionally verify the regulator for a real quadratic field with a 60-decimal digit discriminant in 6 days and 23.5 hours. These results were expanded upon in [22], where the authors were able to unconditionally verify the regulator for a 65-decimal digit discriminant in 102 days and 7 hours.

Our contributions. The key idea behind our proposed change to this setup is as follows. Let $H(\mathfrak{a}_i)$ denote the hash of $\mathfrak{a}_i$ and set

$$\mathcal{L}'' = \{H(\mathfrak{a}_1), H(\mathfrak{a}_l), H(\mathfrak{a}_{2l}), \ldots, H(\mathfrak{a}_{Nl})\} \cup \left\{H(\mathfrak{a}_t), H(\mathfrak{a}_{t+1}), H(\mathfrak{a}_{t+2})\right\} ,$$

where again N is such that $Nl \leq t < (N+1)l$. When we are checking if the giant step $\mathfrak{b}_j$ is in $\mathcal{L}'$, we do not expect to find a match. Thus, most of the time, we can get away with only checking if $H(\mathfrak{b}_j) \in \mathcal{L}''$. If $H(\mathfrak{b}_j) \notin \mathcal{L}''$, then clearly we cannot have $\mathfrak{b}_j \in \mathcal{L}'$. If, however, we do find $H(\mathfrak{b}_j) \in \mathcal{L}''$, it does not immediately follow that $\mathfrak{b}_j \in \mathcal{L}'$. We may just be unlucky and have found a random hash collision. To determine which case we are faced with, a random collision or having found $\mathfrak{b}_j \in \mathcal{L}'$, we proceed through a resolution process. This process will terminate quite rapidly in the case of a random collision due to a probabilistic argument. If the process runs through to the end, however, an explicit ideal comparison determines which case we are in unconditionally. With an appropriate choice of hash function and length of hash stored in $\mathcal{L}''$, we will achieve some significant

memory savings and, hopefully, a noticeable decrease in run-time.

For the implementation of this hashed baby-step list, we take advantage of the refinements to the supporting algorithms which have been made since in [21] and [22]. In addition to using these improvements, our reimplementation is written in C using GMP for its multi-precision arithmetic. The former was written in C++ using NTL and it is well-known that the GMP arithmetic routines are faster than their NTL counterparts. In order to maximize the efficiency of the $O(\Delta^{1/6+\epsilon})$ algorithm, we must also ensure a computational balance between the various parts is maintained. To do this, we must optimize several algorithmic parameters. We have performed a more detailed analysis of these parameters and have enhanced the accuracy of their computation, resulting in some significant computational savings. For moderate-size discriminants, those in the 30–40 decimal digit range, our hashed baby-step list algorithm results in roughly a 10% run-time savings. For larger discriminants, due to two confounding factors, we see these savings disappear. Preliminary numerical tests in a shared high-performance computing environment we accessed through WestGrid seem to show these improvements can scale to larger discriminants. Unfortunately, due to some practical roadblocks discussed in later chapters, this work is still in its early stages and the results are limited, though promising.

1.2. Compact representations of certain quadratic integers

Once we have a correct approximation for the regulator of a real quadratic field, it can be useful to determine the associated fundamental unit η_0. We saw this was the case for determining solutions to the Pell equation. However, we can expect that

$$\eta_0 = \frac{x + y\sqrt{\Delta}}{2} \gg \exp(\Delta^{1/2+\epsilon}) \,.$$

Unfortunately, this means that when Δ is large, x and y can be so enormous that it is impossible to write them down in reasonable time. As two examples of this, we consider two specific values of Δ. Consider the Pell equation

$$T^2 - 410286423278424 \cdot U^2 = 1 \,,$$

the equation derived from Archimedes' Cattle Problem. If we solve this equation, the values computed for T and U have approximately 103,200 digits each. The resulting solution to the Cattle Problem contains approximately 206,500 digits and was first explicitly stated in 1965 by Hugh Williams, Gus German and Robert Zarnke [86]. Harry Nelson [69] repeated this calculation and published the complete solution in 1981 on twelve pages of fine print. To give the reader an idea of just how large this number is, Figure 1.1 shows a full-size excerpt of roughly the first 8,600 digits of the Cattle Problem solution. Nelson's original paper contains 12 pages of these fine-print pages; there were 47 of these subpages, scaled to one-third of their original size to fit four per page.

In [48, p. 62], the authors mention that for the 30-digit discriminant

$$\Delta = 990676090995853870156271607886 \,,$$

the values of x and y in η_0 are greater than $10^{2 \cdot 10^{15}}$. Attempting to print out these numbers, using the same font size and page layout as Nelson, would require approximately 6 billion pages.[1]

[1] Most of Nelson's pages are 64 × 70-digit blocks giving a total of 35,840 digits per double-sided page.

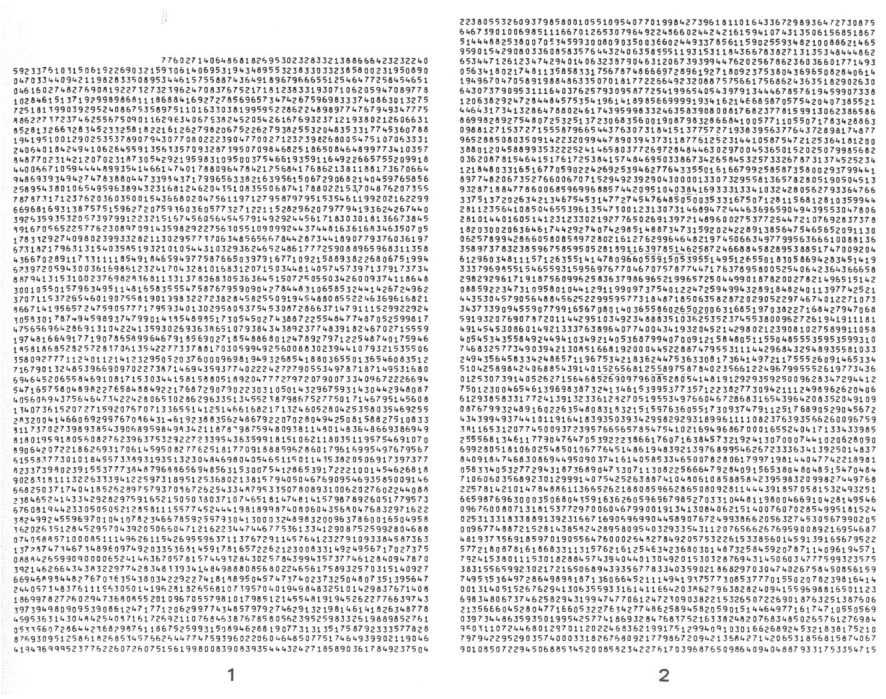


Figure 1.1: Excerpt of the explicit solution to Archimedes' Cattle Problem from [69].

The idea for a compact representation, originally presented by Johannes Buchmann, Christoph Thiel, and Hugh Williams in 1995 [12], is to represent an algebraic number in terms of a power product which satisfies a number of conditions. By doing so, they achieved a vast reduction in the number of digits needed to write down the number. Moreover, they showed how arithmetic operations could be performed on such representations, leading to more efficient calculations than those using the original numbers. We focus specifically on the compact representation of an algebraic integer θ in a real quadratic field.

From the literature, we know that an algorithm such as AX [48, Alg. 11.6, p. 279] will compute an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a} = (\theta)$ where $\log_2 \theta$ is approximately x using $O(\log x \log \Delta)$ elementary operations. AX works by determining the binary expansion of x and then

appropriately multiplying a series of ideals by way of a square-and-multiply routine. By storing the resulting relative generators λ_i and a sequence of ideal norms L_i, an explicit power product representation of θ can be computed:

$$\theta = \prod_{i=0}^{l} \left(\frac{\lambda_i}{L_i^2} \right)^{2^{l-i}} . \tag{1.5}$$

This is the essence of the algorithm CRAX. A representation such as (1.5) requires only $O((\log\log\theta)\log\Delta)$ bits to store, compared to the sizeable $O(\log\theta)$ bits needed to express θ as $(x + y\sqrt{\Delta})/2$.

An interesting application of these compact representations is as certificates for the decision problem known as the *principal ideal problem*: given an ideal $\mathfrak{a}$ of some order $\mathcal{O}$ of a real quadratic field, determine if there exists $\theta \in \mathcal{O}$ such that $\mathfrak{a} = (\theta)$ or not. We could, for example, use a principal ideal test such as the one described in [73, §7] to determine if the ideal $\mathfrak{a}$ was principal or not. If the test is successful—that is, $\mathfrak{a}$ is principal—the output of this algorithm gives an approximation of $\log\theta$. This value can then be passed as input to CRAX to produce a compact representation of θ. Since these representations can be produced in polynomial time, they serve to show that the principal ideal problem is an NP problem [16, 62]. For the interested reader, we point out that, in fact, this decision problem is in NP and co-NP [17, Thm. 1.1].

Our contributions. We present two modifications to the process of computing a compact representation which lead to significant reductions in their storage requirements. The first change is based on the observation on the size of the individual relative generator terms μ_i computed by EADDXY during the squaring portion of the square-and-

multiply routine of AX mentioned previously. From [48, pp. 445–6], we know

$$O(\Delta^{1/4}) < \mu_i < O(\Delta^{3/4})\,.$$

Thus, while the generators of the representation are bounded above and so cannot become too large, they are also bounded below and so cannot become too small. This situation comes as a result of the distance inaccuracies inherent to the giant-step computation at the heart of EADDXY. By altering the distances used as input to EADDXY, we can compensate for this error. Doing so reduces the lower bound on the relative generator produced, and so reduces the overall size of the compact representation.

The second modification comes about from the simple observation that if the giant steps computed in CRAX could traverse a greater distance, we would need to store fewer relative generators. In other words, we would reduce the value of l in (1.5) above. Rather than computing the square of an ideal as our giant-step, and hence doubling its distance, we can compute the cube or fourth power of the ideal. In this way, we can triple or quadruple the distance of the ideal. The end result is a significant reduction in the total number of relative generators needed to be stored in the compact representation.

There is a downside to this second modification, unfortunately. By travelling a greater distance during a giant step, we incur a greater distance inaccuracy and hence a larger individual relative generator than before. We can, of course, combine our two modifications to somewhat counteract this issue. This leads to a trade-off between smaller individual relative generator terms and a smaller number of overall terms which must be carefully balanced to ensure an optimal result.

We have developed a number of algorithms to allow the computation of these new

compact representations. Some are extensions of existing algorithms to allow them to work with (f, p) representations, while others are derived from the existing square-and-multiply routines to allow for cubing and quadrupling. Theoretically, we have determined that using a cube-and-multiply or a quadruple-and-multiply routine will give the most memory-efficient compact representation. Numerical testing indicates that, in fact, the quadruple-and-multiply routine is more efficient—on average, that is—and we have observed roughly a 37% reduction in memory storage requirement as compared to the standard compact representation.

We also mention in passing that these concepts can be used in other number field settings. In particular, they can be extended to complex cubic and totally complex quartic fields, both of which have unit rank one. However, in this thesis we focus exclusively on the case of real quadratic fields and will not delve further into these higher-degree fields.

1.3. Outline of thesis

In Chapter 2, we give a brief overview of the concepts and results needed to understand and develop both those algorithms presented previously in the literature which we draw on and our modifications and improvements to them. We touch on topics from algebraic number theory, particularly focusing on real quadratic fields and ideals of the maximal order of such fields, the theory of continued fractions, and some key analytic results needed to prove correctness and statements concerning expected run-times.

Chapter 3 is devoted to outlining the major advances in the process of computing the regulator of a real quadratic field, starting with the connection between computing the regulator and determining the continued fraction expansion of $\sqrt{D}$, the development of infrastructure techniques, analytic results concerning approximations of the value of

$L(1, \chi)$, the application of index calculus techniques, and ending with the regulator verification algorithm.

With this verification algorithm fresh in mind, we present our modifications to it in Chapter 4. We look at the selection of a suitable hash function, how our changes effect the process of optimum parameter selection, and give a practical assessment of the issues faced when actually implementing these changes. Detailed timing comparisons are presented as well. We compare our implementation to the results previously given in the literature. Further comparisons are made between the efficiency of various hash function choices, as well as how the length of hash stored effects the run-time.

The refinements to compact representations of certain quadratic integers is the topic of Chapter 5. We show the details of how the inputs to EADDXY are modified, how larger-distance giant steps can be effectively computed, and how we balance the trade-off between smaller relative generators and fewer relative generators. We also introduce a series of algorithms, extensions of some of the more well-known algorithms for computing with (f, p) representations of $\mathcal{O}_{\mathbb{K}}$-ideals, which allow us to compute cubes, fourth-, and higher powers of $\mathcal{O}_{\mathbb{K}}$-ideals. Proofs of correctness and an analysis of their run-times are, of course, provided. We end Chapter 5 with an analysis of the theoretical improvements achieved and supporting evidence in the form of results from numerical testing.

Chapter 6 presents a summary of the results we have achieved in this thesis, along with a discussion of several directions for future research on these topics. Finally in the appendices, we explain in much more detail our implementation and the various options that can be configured, discuss how the correctness of our code was tested, and present some additional graphs and figures that were left out of the main material.

— Chapter 2 —
Background & Notation

2.1. Introduction

For this chapter, we will assume that the reader has at least a basic knowledge of number fields and ideals. A number of the results given here are presented without proof as they are standard results which may be found in a number of texts. The specific notation we will adopt for the remainder of this thesis is that of [48]. In particular, symbols set in roman type $(a, b, c, \ldots)$ will generally represent integers; symbols set in Greek letters $(\alpha, \beta, \gamma, \ldots)$ will represent algebraic numbers; symbols set in fraktur type $(\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \ldots)$ will represent ideals; and symbols set in blackboard bold type $(\mathbb{N}, \mathbb{Z}, \ldots)$ or script type $(\mathcal{L}, \mathcal{O}, \ldots)$ will represent sets, rings, fields or other mathematical structures. The majority of the uncited definitions and results presented in this chapter come either from standard algebraic number theory works—[18] and [29], for example—or from [48] itself.

2.2. Quadratic number fields and their orders

Let $D \in \mathbb{Z}$ be an integer, not a perfect square, and greater than 1. The algebraic extension field $\mathbb{K} = \mathbb{Q}(\sqrt{D})$ is a *real quadratic number field*. The elements $\alpha \in \mathbb{K}$ are *quadratic numbers*, which have the form $\alpha = (a + b\sqrt{D})/c$ for integers a, b, and c. The *conjugate* $\overline{\alpha}$ of a $\alpha \in \mathbb{K}$ is given by $\overline{\alpha} = (a - b\sqrt{D})/c$ and the *norm* $N(\alpha)$ of α is $N(\alpha) = \alpha\overline{\alpha} = (a^2 - b^2 D)/c^2$. In particular, we are most interested in a subset of quadratic numbers: the quadratic integers. *Quadratic integers* are the elements of $\mathbb{K}$ that are *integral* over $\mathbb{Z}$, that

is, the elements which are zeroes of monic polynomials irreducible over $\mathbb{Z}$. If we let

$$r = \begin{cases} 1 & \text{if } D \not\equiv 1 \pmod 4, \\ 2 & \text{otherwise} \end{cases} \qquad \text{and} \quad \omega = \frac{r - 1 + \sqrt{D}}{r}, \tag{2.1}$$

then the quadratic integers of $\mathbb{K}$ take the form $\alpha = a + b\omega$.

The set of all quadratic integers in $\mathbb{K}$ forms a structure known as a module. If $\mathcal{A}$ is an additive abelian group, then M is a *$\mathbb{Z}$-module* of $\mathcal{A}$ if it is an additive abelian subgroup of $\mathcal{A}$. If $\zeta_1, \zeta_2, \ldots, \zeta_n \in \mathbb{K}$ and

$$\mathcal{M} = \left\{ \sum_{i=1}^{n} x_i \zeta_i \;\middle|\; x_i \in \mathbb{Z} \right\},$$

then we say the module $\mathcal{M}$ is *generated by* $\{\zeta_1, \zeta_2, \ldots, \zeta_n\}$, which is denoted as $\mathcal{M} = [\zeta_1, \zeta_2, \ldots, \zeta_n]$. We are particularly interested in modules generated by two elements. A *quadratic order* $\mathcal{O}$ of $\mathbb{K}$ is a module $\mathcal{M} = [\zeta_1, \zeta_2]$ which is a subring of $\mathbb{K}$ containing 1 and where $\zeta_i = a_i + b_i\sqrt{D}$ for $a_i, b_i \in \mathbb{Q}$ $(i = 1, 2)$ and $a_1 b_2 - a_2 b_1 \neq 0$. Using this concept of modules and orders, we can express the set of quadratic integers $\mathcal{O}_\mathbb{K}$ as the *maximal order* $[1, \omega]$ of $\mathbb{K}$, which is also called the *ring of integers*.

Finally, we also need to introduce an important invariant that will be quite useful in the following material. The *discriminant* $\Delta_\mathcal{O}$ of an order $\mathcal{O} = [\zeta_1, \zeta_2]$ is $\Delta_\mathcal{O} = (\zeta_1\overline{\zeta}_2 - \overline{\zeta}_1\zeta_2)^2$. An important fact to note is that the value of the discriminant is independent of the choice of basis elements ζ_i. In the case of the maximal order $\mathcal{O}_\mathbb{K}$, the *field discriminant* $\Delta_{\mathcal{O}_\mathbb{K}}$ can be explicitly determined as $\Delta_{\mathcal{O}_\mathbb{K}} = 4D/r^2$ and hence, $\mathbb{K} = \mathbb{Q}(\sqrt{\Delta_{\mathcal{O}_\mathbb{K}}})$. Another fact to keep in mind is that for every non-square Δ, there is only one order of $\mathbb{Q}(\sqrt{\Delta})$

with discriminant Δ, and that is

$$\mathcal{O} = \left[1, \frac{\Delta + \sqrt{\Delta}}{2}\right]. \tag{2.2}$$

2.3. Units and the fundamental unit

If $\alpha \neq 0$ and β are two elements of $\mathcal{O}_{\mathbb{K}}$, we say α *divides* β, written $\alpha \mid \beta$, if there exists a third element $\gamma \in \mathcal{O}_{\mathbb{K}}$ such that $\beta = \alpha\gamma$. The *units* of $\mathcal{O}_{\mathbb{K}}$ are those elements which divide 1; in other words, the invertible quadratic integers. The set of units forms a multiplicative group of $\mathcal{O}_{\mathbb{K}}$ called the *unit group* and denoted $\mathcal{O}_{\mathbb{K}}^*$. Units can be completely classified as η is a unit if and only if $|N(\eta)| = 1$.

There is one particular unit in which we are most interested in this thesis: the smallest unit of $\mathcal{O}_{\mathbb{K}}$ greater than 1. This unit is called the *fundamental unit* and is denoted η_Δ. The reason η_Δ is called *fundamental* is because we can write any other unit in terms of it. More precisely, if $\eta \in \mathcal{O}_{\mathbb{K}}^*$ is a unit then $\eta = \pm\eta_\Delta^n$ for some integer n. Thus, $\mathcal{O}_{\mathbb{K}}^* = \langle -1, \eta_\Delta \rangle$. As the size of η_Δ grows exponentially as Δ increases, we turn to a more manageable quantity called the *regulator*, denoted $R = \log \eta_\Delta$. The end goal in this thesis is to improve the efficiency of a computer implementation, and as such we prefer to use the base-2 regulator in our algorithms, denoted $\mathcal{R} = \log_2 \eta_\Delta$, and emphasize this by explicitly writing "$\log_2$." As we will see in later chapters, a great deal of effort has gone into the problems of computing and expressing these two quantities.

2.4. Ideals

Since $\mathcal{O}_{\mathbb{K}}$ is an integral domain, we can discuss the ideals of $\mathcal{O}_{\mathbb{K}}$. These are non-empty subrings of $\mathcal{O}_{\mathbb{K}}$ that are closed under external multiplication: $\mathfrak{i}$ is an $\mathcal{O}_{\mathbb{K}}$-ideal if $\mathfrak{i}$ is a subring of $\mathcal{O}_{\mathbb{K}}$ and for any $\alpha \in \mathcal{O}_{\mathbb{K}}$, $\alpha\mathfrak{i} \subseteq \mathfrak{i}$. It can be shown that if $\mathfrak{a}$ is an $\mathcal{O}_{\mathbb{K}}$-ideal,

then we can write it as generated as an $\mathcal{O}_{\mathbb{K}}$ module by at most two elements of $\mathcal{O}_{\mathbb{K}}$: $\mathfrak{a} = (\theta_1, \theta_2)$ for $\theta_1, \theta_2 \in \mathcal{O}_{\mathbb{K}}$. In fact, if $\mathfrak{a}$ is a non-zero $\mathcal{O}_{\mathbb{K}}$-ideal, then we can represent it as the $\mathbb{Z}$-module $[a, b + c\omega]$, where $a, b, c \in \mathbb{Z}$; $a, c > 0$; $0 \leq b < a$; $c \mid a, b$; and $ac \mid N(b + c\omega)$ [48, Thms. 4.21, 4.22, and 4.24]. The *conjugate ideal* $\overline{\mathfrak{a}}$ of an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ is $\overline{\mathfrak{a}} = [a, \overline{\beta}] = [a, b + c\overline{\omega}]$. By setting

$$S = c\,, \quad Q = \frac{ra}{c}\,, \quad \text{and} \quad P = \frac{r\,b}{c} + (r-1)\,,$$

an $\mathcal{O}_{\mathbb{K}}$-ideal can be represented as

$$\mathfrak{a} = S\left[\frac{Q}{r}, \frac{P + \sqrt{D}}{r}\right]\,, \tag{2.3}$$

where $S, Q, P \in \mathbb{Z}$, $r \in \{1, 2\}$, $r \mid Q$, and $rQ \mid D - P^2$. For the remainder of this thesis, we will commonly refer to an ideal as "$S[Q, P]$" where it is understood that S, Q, and P satisfy the conditions listed here.

Addition and product operations can be defined on $\mathcal{O}_{\mathbb{K}}$-ideals, though we will focus exclusively on the multiplication of ideals. If $\mathfrak{a}' = (\theta_1, \theta_2)$ and $\mathfrak{a}'' = (\psi_1, \psi_2)$ are $\mathcal{O}_{\mathbb{K}}$-ideals, then the *product* ideal $\mathfrak{a}'\mathfrak{a}''$ is defined as $(\theta_1\psi_1, \theta_2\psi_1, \theta_1\psi_2, \theta_2, \psi_2)$. An obvious question to ask at this point is, given the product ideal $\mathfrak{a}'\mathfrak{a}''$, how do we determine values for S, Q, and P such that $\mathfrak{a}'\mathfrak{a}'' = S[Q, P]$? We will defer answering this question until later in Section 2.10 and merely state here that it is relatively easy to do. We also remark that if $\mathfrak{a} = [\theta_1, \theta_2]$, then [48, p. 88]

$$(\alpha)\mathfrak{a} = \alpha(\theta_1, \theta_2) = (\alpha\theta_1, \alpha\theta_2) = [\alpha\theta_1, \alpha\theta_2]\,.$$

The concept of ideals can be further extended to what are called *fractional* ideals: a non-empty subring $\mathfrak{i}$ of $\mathbb{K}$—rather than $\mathcal{O}_{\mathbb{K}}$—that is closed under external multiplication from $\mathbb{K}$ and such that there exists a non-zero $\gamma \in \mathcal{O}_{\mathbb{K}}$ with $\gamma\mathfrak{i} \subseteq \mathcal{O}_{\mathbb{K}}$. In essence, this last condition says the elements of $\mathfrak{i}$ have γ as a "common denominator."

Theorem 2.1. *The set of all fractional $\mathcal{O}_{\mathbb{K}}$-ideals forms a multiplicative abelian group, denoted $\mathcal{F}(\mathbb{K})$.*

If $\mathfrak{a}$ and $\mathfrak{b}$ are two $\mathcal{O}_{\mathbb{K}}$-ideals, we say that $\mathfrak{a}$ *divides* $\mathfrak{b}$, denoted $\mathfrak{a} \mid \mathfrak{b}$, if there exists a non-zero $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{c}$ such that $\mathfrak{b} = \mathfrak{a}\mathfrak{c}$. This statement of ideal division can translated into a statement about sets.

Lemma 2.2 ("To divide is to contain")**.** *If $\mathfrak{a}$, $\mathfrak{b}$ are two $\mathcal{O}_{\mathbb{K}}$-ideals, then $\mathfrak{a} \mid \mathfrak{b}$ if and only if $\mathfrak{b} \subseteq \mathfrak{a}$.*

The concept of divisibility leads us to the idea of prime ideals. An ideal $\mathfrak{p}$ ($\neq \mathcal{O}_{\mathbb{K}}$) is *prime* if whenever $\mathfrak{p} \mid \mathfrak{a}\mathfrak{b}$ then $\mathfrak{p} \mid \mathfrak{a}$ or $\mathfrak{p} \mid \mathfrak{b}$.

In the ring of rational integers, we have unique factorization as guaranteed by the Fundamental Theorem of Arithmetic. However, this may not be the case in an arbitrary ring of integers. The reason unique factorization of elements can fail is because irreducible elements need not be prime in these extension fields. Although we lose unique factorization at an element level, it can be restored at the ideal level using, in particular, the prime ideals of $\mathcal{O}_{\mathbb{K}}$. $\mathcal{O}_{\mathbb{K}}$ is in fact a *Dedekind domain*, that is an integral domain in which every non-zero ideal can be written as a power product of finitely many distinct prime ideals (unique up to order). Moreover, one can show that prime ideals are *irreducible* ideals—ideals whose only divisors are $\mathcal{O}_{\mathbb{K}}$ and themselves—and vice versa, giving us unique factorization of ideals.

Two ideals are said to be *equivalent* if there exist non-zero $\alpha, \beta \in \mathcal{O}_{\mathbb{K}}$ such that $(\alpha)\mathfrak{a} =$

$(\beta)\mathfrak{b}$ and denote this by $\mathfrak{a} \sim \mathfrak{b}$. We remark that we will frequently abuse this notation by writing $\mathfrak{a} = (\gamma)\mathfrak{b}$, where it is understood that $(\gamma) = (\beta/\alpha)$ is a fractional $\mathcal{O}_{\mathbb{K}}$-ideal.

A *principal* ideal $\mathfrak{a}$ is an $\mathcal{O}_{\mathbb{K}}$-ideal which can be written as $\mathfrak{a} = (\theta)$ for some $\theta \in \mathcal{O}_{\mathbb{K}}$, in other words it has only a single generator.

Theorem 2.3. *The set of all principal fractional ideals, denoted $\mathcal{P}(\mathbb{K})$, forms a subgroup of $\mathcal{F}(\mathbb{K})$.*

It will be desirable to have nice representative ideals for the computations we do in later chapters. An $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ is *primitive* if it cannot be written as an integer multiple of another ideal $\mathfrak{b}$. More exactly, $\mathfrak{a}$ is primitive if $\mathfrak{a} \neq (m)\mathfrak{b}$ for any $m \in \mathbb{Z}$, where $|m| > 1$. Using the notation of (2.3), we say an ideal $\mathfrak{a}$ is primitive if $S = 1$, denoted as $\mathfrak{a} = [Q, P]$. Although restricting ourselves to primitive ideals is a start, this is not good enough as we will see shortly. Before continuing, we must first discuss norms. The *norm* $N(\mathfrak{a})$ of an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ is the index $|\mathcal{O}_{\mathbb{K}}/\mathfrak{a}|$ and when the ideal $\mathfrak{a}$ is written in the form of (2.3), we have

$$N(\mathfrak{a}) = S^2 Q/r \; . \tag{2.4}$$

Working in the rational integers, it is often useful to simplify calculations modulo some integer. In this way, intermediate results are reduced to numbers of a manageable size and will not become unwieldy. Returning to $\mathcal{O}_{\mathbb{K}}$-ideals, the norm can be used to measure the size of the generators of an ideal and the idea of "modulo" can be replaced with reduction. Thus, we will refer to an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ as *reduced* if it is a primitive ideal and there does not exist $\alpha \in \mathfrak{a}$, $\alpha \neq 0$, such that both $|\alpha| < N(\mathfrak{a})$ and $|\overline{\alpha}| < N(\mathfrak{a})$. A useful property of reduced $\mathcal{O}_{\mathbb{K}}$-ideals, when written in the form of (2.3), is that $0 < P < \sqrt{D}$ and $0 < Q < 2\sqrt{D}$ [48, Cor. 5.8.1, p. 101].

We are left with the question of how to compute a reduced ideal from a given prim-

itive ideal. To answer it, we must delve into the theory of continued fractions. As such, we will defer this discussion until after Section 2.8.

2.5. The ideal class group

Returning to the notion of ideal equivalence, we see it is actually an equivalence relation on the set of $\mathcal{O}_{\mathbb{K}}$-ideals. Let $[\mathfrak{i}]$ denote the *equivalence class of an ideal* $\mathfrak{i}$, or more simply the *ideal class* of $\mathfrak{i}$. A key result needed for the remainder of this chapter is the following.

Theorem 2.4. *Every ideal class contains a reduced ideal.*

For $\Delta < 0$, there will be at most two reduced ideals in each class. In the case of $\Delta > 0$, however, there can be many reduced ideals; in fact there tend to be roughly $O(R)$ of them. The following result can be used to show that there are finitely many reduced $\mathcal{O}_{\mathbb{K}}$-ideals.

Theorem 2.5 ([87, Thm. 3.5, p. 411])**.** $\mathfrak{i}$ *is a reduced* $\mathcal{O}_{\mathbb{K}}$*-ideal if and only if there exists some* $\beta \in \mathfrak{i}$ *such that* $\mathfrak{i} = [N(\mathfrak{i}), \beta]$ *where* $-N(\mathfrak{i}) < \overline{\beta} < 0 < \beta < N(\mathfrak{i})$.

As a consequence of Theorem 2.5, we see that if $\mathfrak{i}$ is a reduced $\mathcal{O}_{\mathbb{K}}$-ideal, we must have $N(\mathfrak{i}) < \beta - \overline{\beta} = \omega - \overline{\omega} = \sqrt{\Delta}$. This leads us to our desired result, the proof of which, presented below, is based on the results from [87].

Theorem 2.6. *The number of reduced* $\mathcal{O}_{\mathbb{K}}$*-ideals is finite.*

Proof. Let $\mathfrak{i}$ be a reduced $\mathcal{O}_{\mathbb{K}}$-ideal. We know the norm of $\mathfrak{i}$, $N(\mathfrak{i})$, is a positive integer which, by Theorem 2.5, is bounded above by $\sqrt{\Delta}$. Hence, there are only a finite number of values $N(\mathfrak{i})$ could assume. Let m be one of these values. By Lemma 2.2, $m \in \mathfrak{i}$ if and only if $\mathfrak{i} \mid (m)$, but since $\mathcal{O}_{\mathbb{K}}$ is a Dedekind domain, we know (m) has only finitely many divisors. This implies that (m) can belong to only a finite number of $\mathcal{O}_{\mathbb{K}}$-ideals. Now, if $N(\mathfrak{i}) = m$, then clearly $m \in \mathfrak{i}$, so only finitely many $\mathcal{O}_{\mathbb{K}}$-ideals can have a given norm. Hence, there can only be a a finite number of reduced $\mathcal{O}_{\mathbb{K}}$-ideals. □

Previously, we mentioned that the set of non-zero fractional $\mathcal{O}_{\mathbb{K}}$-ideals $\mathcal{F}(\mathbb{K})$ forms a group under ideal multiplication and that the principal $\mathcal{O}_{\mathbb{K}}$-ideals $\mathcal{P}(\mathcal{O}_{\mathbb{K}})$ form a subgroup. In fact, since $\mathcal{F}(\mathbb{K})$ is an abelian group, $\mathcal{P}(\mathcal{O}_{\mathbb{K}})$ is a normal subgroup and hence the factor group $\mathcal{F}(\mathbb{K})/\mathcal{P}(\mathbb{K})$ is well-defined and abelian. This quotient group $\mathcal{F}(\mathbb{K})/\mathcal{P}(\mathbb{K})$ is called the *ideal class group*, denoted by $\mathcal{C}l_{\mathbb{K}}$. Recalling Theorems 2.4 and 2.6, we see that $\mathcal{C}l_{\mathbb{K}}$ is a finite group. The order of $\mathcal{C}l_{\mathbb{K}}$ is commonly called the *class number* and denoted h. Using ideal multiplication, a closed, associative, commutative multiplication for $\mathcal{C}l_{\mathbb{K}}$ can be defined by $[\mathfrak{a}][\mathfrak{b}] = [\mathfrak{a}\mathfrak{b}]$. The identity element of $\mathcal{C}l_{\mathbb{K}}$ is the class $[\mathcal{O}_{\mathbb{K}}]$ of principal ideals and for every class $[\mathfrak{a}]$, there exists an *inverse* class $[\mathfrak{b}]$ such that $[\mathfrak{a}][\mathfrak{b}] = [\mathcal{O}_{\mathbb{K}}]$. In fact, this inverse class is given by $[\overline{\mathfrak{a}}]$.

2.6. Analytic results

Let G be a finite abelian group. A function $\chi : G \to \mathbb{C}$ is a *character* if $\chi(ab) = \chi(a)\chi(b)$ for $a, b \in G$, $\chi(c) \neq 0$ for some $c \in G$, and $|\chi(d)| = 1$ for all $d \in G$ for which $\chi(d) \neq 0$. The *principal character* of G is a character χ defined on G such that $\chi(a) = 1$ for all $a \in G$. One important property of a character is that if 1_G is the identity element of G, then $\chi(1_G) = 1$. The notion of characters leads us to L-functions. We specialize G to $G = (\mathbb{Z}/m\mathbb{Z})^*$ and take χ to be any non-principal character of G. If $\chi(n) = 0$ whenever $(m, n) > 1$, then χ is called a *Dirichlet character*. The *Dirichlet L-function* is

$$L(s, \chi) = \sum_{n=1}^{\infty} \frac{\chi(n)}{n^s}, \tag{2.5}$$

where χ is a Dirichlet character and $s = \sigma + it \in \mathbb{C}$.

Approximating the value of this L-function, particularly the value of $L(1, \chi)$, will be of great importance in the chapters that follow. To this end, we point out two key results.

The first is that $L(s,\chi)$ converges absolutely for all $\sigma > 1$. The second allows us to rewrite (2.5) in terms of an *Euler product* when $\sigma > 1$:

$$L(s,\chi) = \prod_p \left(1 - \frac{\chi(p)}{p^s}\right)^{-1} . \tag{2.6}$$

At this point, we will further specialize the Dirichlet character we are considering by taking $\chi = \chi_\Delta = (\frac{\Delta}{n})$, the *Kronecker symbol*. The Kronecker symbol is a generalization of the Jacobi symbol to all integers, which itself is a generalization of the Legendre symbol. The main result we have been building up to in this section is the *analytic class number formula*, which relates several of the important quantities we have discussed.

Theorem 2.7 (Analytic class number formula)**.**

$$\frac{2hR}{\sqrt{\Delta}} = L(1,\chi_\Delta) .$$

Proof. See [48, Thm. 8.35 and Cor. 8.35.1, pp. 204–5]. □

2.7. Consequences of the analytic class number formula and some Riemann hypotheses

The Riemann Hypothesis and its extended version are vital both in terms of guaranteeing correctness and run-time bounds for several algorithms in Chapter 3. This now well-known hypothesis was formulated by Georg Friedrich Bernhard Riemann in 1859, in connection with his work on the zeta function. It has remained as one of the most famous unsolved problems in mathematics. In this section, we introduce these hypotheses and briefly discuss some of their consequences, particularly in connection with the analytic class number formula.

Let $s = \sigma + it$ be a complex number. The *Riemann zeta function* is

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}\,.$$

It is convergent for $\sigma > 1$ and can be analytically continued to the entire complex plane, save for a pole of order 1 at $s = 1$. Considering the zeroes of $\zeta(s)$, there is a sequence of *trivial zeroes* at the negative even integers. The remaining non-trivial zeroes are known to lie in the *critical strip* $0 < \sigma < 1$. The *Riemann hypothesis* is the conjecture that if $\zeta(s) = 0$ and $\sigma > 0$, then $\sigma = 1/2$.

If the Riemann zeta function is generalized from its setting in the rational numbers $\mathbb{Q}$ to a quadratic number field, we arrive at the Dedekind zeta function. Let $\mathbb{K}$ be a quadratic number field. The *Dedekind zeta function* is defined by the series

$$\zeta_{\mathbb{K}}(s) = \sum_{\mathfrak{a}} \frac{1}{|N(\mathfrak{a})^s|}\,,$$

where $\mathfrak{a}$ varies over the non-zero integral ideals of $\mathbb{K}$. A well-known result, stated by Erich Hecke, is that $\zeta_{\mathbb{K}}(s) = \zeta(s)L\big(s, \left(\frac{\Delta}{n}\right)\big)$. As with the Riemann zeta function, $\zeta_{\mathbb{K}}(s)$ can be analytically continued to the entire complex plane except for a simple pole at $s = 1$. The residue of this pole is given by the analytic class number formula stated in Theorem 2.7.

Recalling the L-function $L(s, \chi)$ from (2.5), page 22, if χ is the principal character, then

$$L(s, \chi) = \sum_{n=1}^{\infty} \frac{\chi(n)}{n^s} = \sum_{n=1}^{\infty} \frac{1}{n^s} = \zeta(s)\,.$$

Thus, the L-function is a generalization of the Riemann zeta function. By replacing $\zeta(s)$

by $L(s,\chi)$ in the Riemann hypothesis, the *extended Riemann hypothesis* (ERH) can be derived: if $L(s,\chi)=0$ and $\sigma>0$, then $\sigma=1/2$.

Although we will not discuss it in any detail, we remark that this hypothesis can be generalized further. The aptly named *generalized Riemann hypothesis* (GRH) deals with any L-function, not just $L(s,\chi)$: no zeta function of a proper Hecke character has a zero with real part larger than $1/2$.

Assuming the ERH, John Littlewood [59] was able to derive a very tight bound on the value of $L(1,\chi)$. Specifically,

$$(1+o(1))(c_1 \log\log|\Delta|)^{-1} < L(1,\chi) < (1+o(1))c_2 \log\log|\Delta|\,,$$

where $c_1 = 4(r+1)e^{\gamma}/\pi^2$, $c_2 = re^{\gamma}$, and γ is the Euler-Mascheroni constant $0.577215\ldots$. If we desire an unconditional bound on $L(1,\chi)$, the best known bound comes from the work of Stéphane Louboutin [60]. He showed

$$|L(1,\chi)| \le \frac{\log\Delta+\kappa_0}{2}\,,$$

where $\kappa_0 \approx 0.046$.

Using the analytic class number formula and both Littlewood's and Louboutin's bounds, bounds on the regulator R can be derived. As presented previously, the fundamental unit can be expressed as

$$\eta_\Delta = \frac{x+y\sqrt{\Delta}}{2}\,.$$

Since $N(\eta_\Delta) = \pm 1$, we have $x = \sqrt{y^2\Delta \pm 4}$, and as $y \geq 1$—because $\eta_\Delta > \overline{\eta}_\Delta$—, we find

$$\eta_\Delta = \frac{\sqrt{y^2\Delta \pm 4} + y\sqrt{\Delta}}{2} \geq \frac{\sqrt{\Delta - 4} + \sqrt{\Delta}}{2},$$

and hence

$$R \geq \log \frac{\sqrt{\Delta - 4} + \sqrt{\Delta}}{2}.$$

Equality is achieved when $\Delta = x^2 + 4$ and $x^2 + 4$ is squarefree infinitely often [67]. This gives us a sharp lower bound on the size of R. To determine an upper bound on R, we must turn to the analytic class number formula. Looking at Theorem 2.7, R is maximal when $h = 1$, giving $R \leq \sqrt{\Delta} L(1, \chi)/2$. If we are willing to accept a conditional upper-bound, we can use Littlewood's bounds. For an unconditional bound, we turn to Louboutin's bounds, which give

$$R \leq \frac{\sqrt{\Delta}}{4}\left(\log \sqrt{\Delta} + \kappa_0\right) \approx \frac{\sqrt{\Delta}\log\Delta}{4}.$$

Combining the upper and lower bounds on R, we find $R = O(\Delta^{1/2+\epsilon})$.

A second consequence of the analytic class number formula is the Brauer-Siegel Theorem, which shows how the size of R grows with respect to Δ. The result was originally proved by Carl Siegel [76] for quadratic number fields, though Siegel conjectured it would hold for higher-degree fields; Richard Brauer provided a proof of this generalization [9].

Theorem 2.8 (Brauer-Siegel Theorem)**.**

$$\lim_{\Delta \to \infty} \frac{\log hR}{\log \sqrt{|\Delta|}} = 1.$$

2.8. CONTINUED FRACTIONS

At this point, we take a break from the analytic material and side-step into the study of continued fractions. Continued fractions play a large role in many aspects of number theory, in particular the study of Diophantine equations. Let $\phi = \phi_0$ a real number. By recursively defining $\phi_{j+1} = (\phi_j - q_j)^{-1}$ for $j = 0, 1, \ldots, i$ and $q_j \in \mathbb{Z}$, ϕ_0 can be expressed as the *continued fraction*[1]

$$\phi_0 = q_0 + \cfrac{1}{q_1 + \cfrac{1}{\ddots \; + \cfrac{1}{q_{i-1} + \cfrac{1}{q_i + \cfrac{1}{\phi_{i+1}}}}}} \, .$$

The q_i values are known as *partial quotients* and the number ϕ_{i+1} is a *complete quotient*. If $q_1, q_2, \ldots, q_i \geq 1$ and $\phi_{i+1} > 1$, then the continued fraction is *simple*. Since it is cumbersome to fully express continued fractions, we will adopt the short-hand notation $\langle q_0, q_1, \ldots, q_i, \phi_{i+1} \rangle$ for a continued fraction and $[q_0, q_1, \ldots, q_i, \phi_{i+1}]$ for a simple continued fraction. We will be focusing on simple continued fractions for most of the remainder of this section.

The finite continued fractions $[q_0, q_1, \ldots, q_k]$ $(k = 0, 1, 2, \ldots)$ are called the k^{th} *convergents* of the continued fraction. These convergents can also be recursively computed. If we set $A_{-2} = 0$, $A_{-1} = 1$, $B_{-2} = 1$, $B_{-1} = 0$, and recursively define (for $i \geq 0$)

$$A_i = q_i A_{i-1} + A_{i-2} \quad \text{and} \quad B_i = q_i B_{i-1} + B_{i-2} \, ,$$

[1] I much prefer Edsger Dijkstra's version: *The proper definition of a continued fraction is of course "a fraction whose numerator is an integer and whose denominator is an integer plus a continued fraction"* [23].

then A_i/B_i is the i^{th} convergent for $[q_0, q_1, \ldots, q_i, \phi_{i+1}]$.

If a continued fraction can be expressed as $[q_1, q_2, \ldots, q_n]$, then it is called a *finite continued fraction*. If not, then it is an *infinite continued fraction*. Infinite continued fractions are said to be *periodic* if we can find integers N and l, l minimal, such that $q_{N+l+i} = q_{N+i}$ $(i = 0, 1, \ldots)$. The integer l is the *length of the period* of the continued fraction and the sequence $q_1, q_2, \ldots, q_{N-1}$ is the *preperiod*. If there is no preperiod, in other words $N = 0$, then the infinite continued fraction is *purely periodic*.

So what kinds of numbers can be represented by continued fractions? The finite case is easy. It can be shown that every finite simple continued fraction represents a rational number and, conversely, that every rational number can be expressed by a finite simple continued fraction. The infinite case is a little bit more tricky, since it is not immediately clear that it converges to some value. However, it can be shown that infinite continued fractions represent irrational numbers and, conversely, that given an irrational number α, we can find a unique infinite simple continued fraction whose value is α.

The only remaining case is that of periodic infinite continued fractions. Joseph Lagrange categorized these continued fractions by showing that an infinite simple continued fraction is periodic if and only if the number it represents is a *quadratic irrational*: a real number α of the form $(a + b\sqrt{D})/c$, where a, b, and c are integers and $\sqrt{D} \notin \mathbb{Q}$. It can be shown that α is a quadratic irrational if and only if it can be expressed as

$$\alpha = \frac{P + \sqrt{D}}{Q},$$

where P, Q $(\neq 0)$, and D (> 0) are integers, $\sqrt{D} \notin \mathbb{Q}$, and $Q \mid (D - P^2)$.

At this point we hope the connection between periodic simple continued fractions

and the ideals of the maximal order $\mathcal{O}_{\mathbb{K}}$ is starting to come into focus. If $\alpha > 1$ is a quadratic irrational and $-1 < \overline{\alpha} < 0$, then α is known as a *reduced* quadratic irrational. Using this notion of a reduced quadratic irrational, Lagrange's result can be refined to the following statement: the infinite simple continued fraction of a quadratic irrational is purely periodic if and only if α is reduced.

Now that we know a quadratic irrational can be expressed by periodic simple continued fractions, an obvious question to ask is: how do we compute it? Suppose we take $\phi_0 = \sqrt{D}$ and form the sequence $\phi_1, \phi_2, \ldots$ where

$$\phi_{i+1} = \frac{1}{\phi_i - q_i} \quad \text{and} \quad q_i = \lfloor \phi_i \rfloor$$

for $i = 0, 1, 2, \ldots$. Clearly we can write

$$\phi_i = \frac{P_i + \sqrt{D}}{Q_i} > 1$$

where P_i and Q_i are integers, $P_0 = 0$, $Q_0 = 1$, and $Q_i \mid D - P_i^2$. As the ϕ_i are quadratic irrationals, the sequence $\{q_i\}_{i \geq 0}$ forms the periodic simple continued fraction $[q_0, q_1, \ldots, q_i, \phi_{i+1}]$. Using this fact, and some algebraic manipulations, the following recursive formulas for q_i, P_i, and Q_i can be derived:

$$q_i = \left\lfloor \frac{P_i + \sqrt{D}}{Q_i} \right\rfloor , \tag{2.7}$$

$$P_{i+1} = q_i Q_i - P_i , \tag{2.8}$$

$$Q_{i+1} = \frac{D - P_{i+1}^2}{Q_i} . \tag{2.9}$$

These formulas are at the heart of some of key algorithms we discuss in following chapters, and so we pause to consider their computational cost. Our first comment is that since $\sqrt{D}$ is an irrational number, computing q_i via (2.7), as stated, is inefficient. Instead, it is replaced by

$$\left\lfloor \frac{P_i + \sqrt{D}}{Q_i} \right\rfloor = \left\lfloor \frac{P_i + \lfloor \sqrt{D} \rfloor}{Q_i} \right\rfloor .$$

We are able to do this since we have $Q_i \geq 1$; see [77, Lem. 2.66, p. 22]. The second point we make is that division is an expensive operation. Considering the formula for Q_{i+1}, (2.9), we remark that this formula can be improved by making use of Tenner's algorithm [48, §3.4, p. 63]. Let q_i and R_i respectively be the quotient and remainder upon dividing $P_i + \lfloor \sqrt{D} \rfloor$ by Q_i. In other words, $P_i + \lfloor \sqrt{D} \rfloor = q_i Q_i + R_i$. Clearly we can rewrite (2.8) as

$$P_{i+1} = \lfloor \sqrt{D} \rfloor - R_i \tag{2.10}$$

and by substituting (2.8) into (2.9), we find

$$Q_{i+1} = Q_{i-1} - q_i(P_{i+1} - P_i) . \tag{2.11}$$

Since most computers provide both the quotient and remainder on division, this change saves us the cost of a division and a squaring operation at the expense of some extra storage.

2.9. Continued fractions and the ideals of $\mathcal{O}_{\mathbb{K}}$

The theory of continued fractions can now be united with that of the orders of a real quadratic field. At the end of Section 2.4, we were left with the question of how to compute a reduced $\mathcal{O}_{\mathbb{K}}$-ideal from a given ideal. We will answer that question now. Let

$\phi_0 = (P_0 + \sqrt{D})/Q_0$ and $\mathfrak{a} = \mathfrak{a}_1 = [Q_0, P_0]$ be an $\mathcal{O}_{\mathbb{K}}$-ideal. If we compute the continued fraction expansion of ϕ_0, it is clear from the material in the previous section that $\mathfrak{a}_{i+1} = [Q_i, P_i]$ will also be an $\mathcal{O}_{\mathbb{K}}$-ideal. We define

$$\psi_i = \frac{1}{-\overline{\phi}_i} = \frac{Q_i}{\sqrt{D} - P_i} = \frac{P_i + \sqrt{D}}{Q_{i-1}}$$

and point out that computing another step in the continued fraction expansion of ϕ_i is equivalent to multiplying $\mathfrak{a}_i$ by the fractional ideal (ψ_i) [48, §5.1, pp. 102–3]. Thus, we have the relation

$$\mathfrak{a}_{i+1} = (\psi_i)\mathfrak{a}_i \,. \tag{2.12}$$

If we further define

$$\theta_0 = -\overline{\phi}_0, \quad \theta_1 = 1, \quad \text{and} \quad \theta_{i+1} = q_{i-1}\theta_i + \theta_{i-1}\,,$$

then we can show $\theta_{i+1} = \psi_i \theta_i$ and hence [48, (3.17), §3.1, p. 46]

$$\theta_{i+1} = \prod_{k=1}^{i} \psi_k = \frac{Q_i}{Q_0} \prod_{k=1}^{i} \phi_k$$

for $i \geq 1$. Thus, (2.12) becomes

$$\mathfrak{a}_{i+1} = (\theta_i)\mathfrak{a}_1 \,.$$

From this, it is clear that $\mathfrak{a}_i \sim \mathfrak{a}_1$ for all $i \geq 1$. In other words, by applying the continued fraction algorithm to ϕ_0, we can produce a sequence of primitive ideals that are all equivalent to $\mathfrak{a}$. Of particular interest is that if we begin with a primitive, but not necessarily reduced, ideal $\mathfrak{a}_1$, we will eventually produce a reduced ideal $\mathfrak{a}_i = (\theta_i)\mathfrak{a}_1$ [48,

Prop. 3.4, p. 53; Thm. 3.5, p. 54; and Thm. 5.8, p. 103]. Furthermore, every subsequent ideal $\mathfrak{a}_{i+1}, \mathfrak{a}_{i+2}, \ldots$ will also be reduced.

What happens if we start with $\mathfrak{a}$ already a reduced ideal, say $\mathfrak{a} = \mathfrak{a}_1 = \mathcal{O}_{\mathbb{K}}$? As above, we can compute a sequence of ideals $\mathfrak{a}_i \sim \mathfrak{a}_1$. Since $\mathfrak{a}_1$ is a principal ideal and $\mathfrak{a}_{i+1} = (\psi_i)\mathfrak{a}_i$, we see that all of the ideals in this sequence are principal. By Theorem 2.6, page 21, since there are only finitely many reduced $\mathcal{O}_{\mathbb{K}}$-ideals, this sequence must also be periodic. Hence, we can find some minimal $p > 0$ such that $\mathfrak{a}_{p+1} = \mathfrak{a}_1$. By definition, we know

$$\psi_i = \frac{P_i + \sqrt{D}}{Q_i} > 1$$

and, since $\mathfrak{a}_1$ is reduced, $0 < P_i < \sqrt{D}$ and $0 < Q_i < 2\sqrt{D}$. So $1 < \psi_1 < \sqrt{D}$. In addition, we have [48, Prop. 3.16, p. 68]

$$\theta_{i+m} > F_{m+1}\theta_i \tag{2.13}$$

where F_j is the j^{th} Fibonacci number: $F_0 = 0, F_1 = 1$, and $F_{j+1} = F_j + F_{j-1}$. Thus,

$$1 = \theta_1 < \theta_2 < \theta_3 < \cdots < \theta_p < \cdots .$$

Using these inequalities, these ideals can be arranged into a cycle $\mathcal{C} = \{\mathfrak{a}_1, \mathfrak{a}_2, \ldots, \mathfrak{a}_p\}$ called the *cycle of reduced principal ideals*. A well-known fact—derived, for instance, from [48, (5.33), p. 113]—is that if the length of the principal ideal cycle is p, then the fundamental unit $\eta_\Delta = \theta_{p+1}$. Since $\psi_i > 1$, $\psi_i \psi_{i+1} > 2$—set $m = 2$ in (2.13)—and $\theta_{p+1} = \prod_{i=1}^{p} \psi_p$, we must have

$$\eta_\Delta > 2^{\lfloor p/2 \rfloor} \quad \text{or equivalently} \quad \mathcal{R} = \log_2 \eta_\Delta > \left\lfloor \frac{p}{2} \right\rfloor .$$

Hence, $p \approx \mathrm{O}(\mathcal{R}) = \mathrm{O}(\Delta^{1/2+\epsilon})$.

2.10. IDEAL PRODUCTS AND THE INFRASTRUCTURE

One of the key arithmetic operations we perform in later chapters is to compute a representation of the product of two ideals. Given two ideals $\mathfrak{a}'$ and $\mathfrak{a}''$, we know from the previous material that the product ideal $\mathfrak{a}'\mathfrak{a}''$ is generated by at most two generators. But how can we quickly determine them? Let

$$\mathfrak{a}' = \left[\frac{Q'}{r}, \frac{P' + \sqrt{D}}{r}\right] \quad \text{and} \quad \mathfrak{a}'' = \left[\frac{Q''}{r}, \frac{P'' + \sqrt{D}}{r}\right]$$

be two primitive $\mathcal{O}_{\mathbb{K}}$-ideals. The product of two primitive ideals need not be primitive, so we write it as $\mathfrak{a}'\mathfrak{a}'' = S[Q, P]$. By multiplying out the generators of $\mathfrak{a}'$ and $\mathfrak{a}''$ and applying some algebraic manipulations [48, §5.4, pp. 117–118], the following formulas for calculating S, Q, and P can be produced. We first solve

$$S = V\left(\frac{Q'}{r}\right) + W\left(\frac{Q''}{r}\right) + Y\left(\frac{P' + P''}{r}\right),$$

using the extended Euclidean algorithm for integers V, W, and Y. Setting $S = \gcd(Q', Q'', P' + P'')$, we see this equation has a solution. Next, we set

$$Q = \frac{Q'Q''}{S^2 r} \quad \text{and} \quad P \equiv P'' + \frac{UQ''}{rS} \pmod{Q},$$

where $U \equiv W(P' - P'') + YR'' \pmod{Q'/S}$ and $R'' = (D - P''^2)/Q''$.

If $\mathfrak{a}'$ and $\mathfrak{a}''$ are reduced ideals, most likely the product ideal will not be reduced. However, using the material in Section 2.9 we can easily compute a reduced ideal $\mathfrak{a}$ equivalent to $\mathfrak{a}'\mathfrak{a}''$. The main issue with using these *composition* formulas is that the intermediate values can be of size $O(\Delta)$.

Let $\mathfrak{a}_1 = [1, \omega]$ be the first ideal in $\mathcal{C}$. The *distance* of $\mathfrak{a}_i = (\theta_i)$ is defined as $\delta_i = \delta(\mathfrak{a}_i) = \log_2 \theta_i$. This concept of distance is quite natural, as we can see from a result independently discovered by Aleksandr Khintchine [51] and Paul Lévy [58]. They showed—specialized to our situation—that we can probabilistically expect $\lim_{i\to\infty} \sqrt[i]{\theta_i} = e^{\tau}$, where $\tau = \pi^2/(12\log 2) \approx 1.186569$. Taking base-2 logarithms of this expression gives $\delta_i = \log_2 \theta_i \approx i\tau \log_2 e \approx 1.712i$, showing that there is an approximately linear relationship between the index of an ideal in $\mathcal{C}$ and its distance. We also point out that since $\mathcal{C}$ is a cycle, we can always reduce a given distance δ_i modulo the regulator.

Let $\mathfrak{a}_i$ and $\mathfrak{a}_j$ be two reduced principal ideals in $\mathcal{C}$. Since they are principal, their product $\mathfrak{a} = \mathfrak{a}_i\mathfrak{a}_j = (\theta_i\theta_j)$ will also be principal. As we stated above, $\mathfrak{a}$ may no longer be reduced. However, we can compute $\mathfrak{a}_i\mathfrak{a}_j = (m)\mathfrak{b}_1$ for some $m \in \mathbb{Z}$ and primitive ideal $\mathfrak{b}_1$. We can then produce a sequence of ideals $\mathfrak{b}_1, \mathfrak{b}_2, \ldots$ by applying the continued fraction algorithm and, as we showed previously, we will eventually find a reduced ideal $\mathfrak{b}_k$. At this point we have

$$\mathfrak{b}_k = (\theta'_k)\mathfrak{b}_1 = \left(\frac{\theta'_k\theta_i\theta_j}{m}\right)$$

and as $\mathfrak{b}_k$ is reduced, we must have $\mathfrak{b}_k = \mathfrak{a}_l \in \mathcal{C}$ for some l. Considering the distance of $\mathfrak{a}_l$, we find

$$\delta_l = \delta(\mathfrak{a}_l) = \log_2\left(\frac{\theta'_k\theta_i\theta_j}{m}\right) \equiv \delta_i + \delta_j + \log_2\frac{\theta'_k}{m} \pmod{\mathcal{R}}. \tag{2.14}$$

The key observation to make is that we are tantalizingly close to forming a group out of $\mathcal{C}$ with the operation of a giant step—that is, ideal multiplication followed by reduction. Unfortunately, this operation does not satisfy one key property: it is not an associative operation. Instead of having $\delta_l = \delta_i + \delta_j$ in (2.14), we are stuck with the additional error

term $\log_2(\theta'_k/m)$ and so δ_l is only close to $\delta_i + \delta_j$. However, this error term can be bounded in absolute value, say by μ. This bound depends on the particular reduction algorithm selected, but for the method described above, it can be shown that $\mu < \mathrm{O}(\log \Delta)$ [48, p. 175], which is quite small compared to $\delta_i, \delta_j \approx \mathrm{O}(\mathcal{R})$.

This group-like structure in the cycle of reduced principal ideals, called the *infrastructure*, was discovered by Daniel Shanks [75]. He was also able to derive a more efficient method for computing the reduced product ideal $\mathfrak{a}'\mathfrak{a}''$ of two reduced principal ideals. Rather than multiplying then reducing, Shanks' method, called NUCOMP, combines these two steps and reduces the intermediate operands before the final product is computed. This keeps the operands of size roughly $\mathrm{O}(\sqrt{\Delta})$. Shanks' method was originally designed for ideal composition in imaginary quadratic fields, but it was later shown by Alf van der Poorten [82] to be effective in the real case, so long as the operands are large enough to compensate for the additional overhead. The particular version of NUCOMP we use is that described by Hugh Williams and Michael Jacobson, Jr., in [48, Alg. 5.1, pp. 122, 441–3].

In the remainder of this thesis, we will refer to a single application of the continued fraction algorithm to the ideal $\mathfrak{a}_i$ as a *(forward) baby step*, denoted $\mathfrak{a}_{i+1} = \rho(\mathfrak{a}_i)$. Although we will not derive the formulas here, given a reduced principal ideal $\mathfrak{a}_i \in \mathcal{C}$, we can also compute the *backward baby step* $\mathfrak{a}_{i-1} = \rho^{-1}(\mathfrak{a}_i)$ [48, §3.4, p. 64]. By using the infrastructure and NUCOMP, we can skip a number of applications of the continued fraction algorithm to move more quickly through the cycle of reduced principal ideals. We denote by $\mathfrak{a}' \star \mathfrak{a}''$ the computation of the reduced ideal equivalent to the product ideal $\mathfrak{a}'\mathfrak{a}''$ via an application of NUCOMP and refer to this process as a *giant step*.

2.11. (f, p) REPRESENTATIONS

While performing computations in the infrastructure, we need to keep track of the generators of the ideals we are working with or, equivalently, their distances. Both options have disadvantages. The size of the coefficients of the generators grow exponentially and so are infeasible to store for large Δ. As distances are logarithms of quadratic irrationals, and hence transcendental numbers, maintaining accurate distance approximations in the face of round-off and truncation errors is difficult. Michael Jacobson, Jr., Renate Scheidler, and Hugh Williams introduced the concept of an (f, p) representation [45] to work around these difficulties and provide provable bounds on the round-off and truncation errors accumulated during computations. This idea was later refined by the same authors [46] and we will use the procedures as described in [48, Ch. 11, p. 265].

Let $p \in \mathbb{N}$ and $f \in \mathbb{R}$ be such that $1 \leq f < 2^p$. If $\mathfrak{a}$ is a primitive $\mathcal{O}_{\mathbb{K}}$-ideal, then an (f, p) *representation* of $\mathfrak{a}$ is a triple $(\mathfrak{b}, d, k)$ where $\mathfrak{b} \sim \mathfrak{a}$, $d \in \mathbb{N}$ such that $2^p < d \leq 2^{p+1}$, and $k \in \mathbb{Z}$. In addition, there exists $\theta \in \mathbb{K}$ such that $\mathfrak{b} = (\theta)\mathfrak{a}$ with

$$\left| \frac{\theta}{2^{k-p} d} - 1 \right| < \frac{f}{2^p} .$$

In essence, an (f, p) representation stores both an approximation to the *relative generator* θ and an approximation of its distance, both with precision p. The parameter f is a measure of the approximation error, though it is rarely if ever explicitly computed. $k \approx \log_2 \theta$ and so measures the rough relative distance of $\mathfrak{b}$ with respect to $\mathfrak{a}$, and d holds the $p + 1$ most significant bits of an approximation to θ. The value of k can then be used to adjust d to give an explicit approximation of the relative generator θ as $\theta \approx 2^{k-p} d$. If $\mathfrak{b}$ is a reduced $\mathcal{O}_{\mathbb{K}}$-ideal, then $(\mathfrak{b}, d, k)$ is a *reduced* (f, p) *representation*.

There are a number of advantages to using (f,p) representations [48, p. 265]. The three most important ones, from our perspective, are that it is relatively easy to analyse the accuracy of operations on (f,p) representations, the precision needed for a given accuracy level tends to be lower than other methods, and all operations of (f,p) representations involve only integer arithmetic, avoiding explicit floating-point calculations involving logarithms. Because of this last point, implementations of (f,p) representations are very fast compared to methods based on floating-point arithmetic.

In the definition of an (f,p) representation presented above, there were no requirements on θ other than that $\mathfrak{b} = (\theta)\mathfrak{a}$. When performing operations on (f,p) representations of reduced ideals, it is advantageous to use representations with small relative generators. A *w-near* (f,p) representation is a a reduced (f,p) representation $(\mathfrak{b},d,k)$ of an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ with the following two conditions:

1. $k < w$ for some $w \in \mathbb{Z}^+$ and
2. if $\rho(\mathfrak{b}) = (\psi)\mathfrak{b}$ then there exist integers d' and k' such that $k' \geq w$, $2^p < d' \leq 2^{p+1}$ and
$$\left| \frac{\psi\theta}{2^{k'-p}d'} - 1 \right| < \frac{f}{2^p}\,.$$

Such representations have the useful property that $\theta \approx 2^w$ and $k \approx w$. Since this property will be used repeatedly in later material, particularly with respect to compact representations, we will state it more formally.

Lemma 2.9 ([48, Lem. 11.3, p. 270])**.** *Let $(\mathfrak{b},d,k)$ be a w-near (f,p) representation of some $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ with $p > 4$ and $f < 2^{p-4}$. If θ and ψ are defined as above, then*

$$\frac{15N(\mathfrak{b})}{16\sqrt{\Delta}} < \frac{15}{16\psi} < \frac{\theta}{2^w} < \frac{17}{16} \quad \textit{and} \quad -\log_2 \frac{34\psi}{15} < k - w < 0\,.$$

The second benefit to w-near representations is that they can be combined with NUCOMP to partially compensate for the $\log_2(\theta'_k/m)$ error term in (2.14) when performing a series of giant step computations. Determining a w-near representation in between each application of NUCOMP will limit the propagation of this error.

It can be shown [48, §11.3, p. 279] that if $\mathfrak{a}_i$ and $\mathfrak{a}_j$ are distinct ideals such that $(\mathfrak{a}_i, d_i, k_i)$ and $(\mathfrak{a}_j, d_j, k_j)$ are both w-near (f, p) representations of the same ideal, then $|i - j| \leq 2$. In other words, two w-near (f, p) representations of the same ideal are very close to each other in the infrastructure. As a notational convenience, we will use $\mathfrak{a}[w]$ to denote any of these ideals.

2.12. ALGORITHMS

In this final section, we will discuss a number of basic algorithms that are needed in the following chapters for performing various computations with (f, p) representations. The majority of these algorithms will not be explicitly presented, rather references to the appropriate sections of [21] or [48] will be given. Because of this, we present a flowchart in Figure 2.1 to help the reader better understand the dependencies between the algorithms. We point out that not all the algorithms included in this flowchart will be discussed in this thesis. Several algorithms are included strictly for those readers attempting to familiarize themselves with our implementation.

To start, an initial (f, p) representation for a given $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$ needs to be derived. There are two trivial (f, p) representations that can be immediately determined [48, §11.1, p. 267]. Taking $\mathfrak{b} = \mathfrak{a}$, we have $\mathfrak{b} \sim \mathfrak{a}$ with relative generator $\theta = 1$. Substituting this into the definition of an (f, p) representation gives two sets of values for d and k. The first representation, used most frequently in the algorithms that follow, is $(\mathfrak{a}, 2^p + 1, 0)$; alternatively, we can take $(\mathfrak{a}, 2^{p+1}, -1)$. Both are valid (f, p) representations

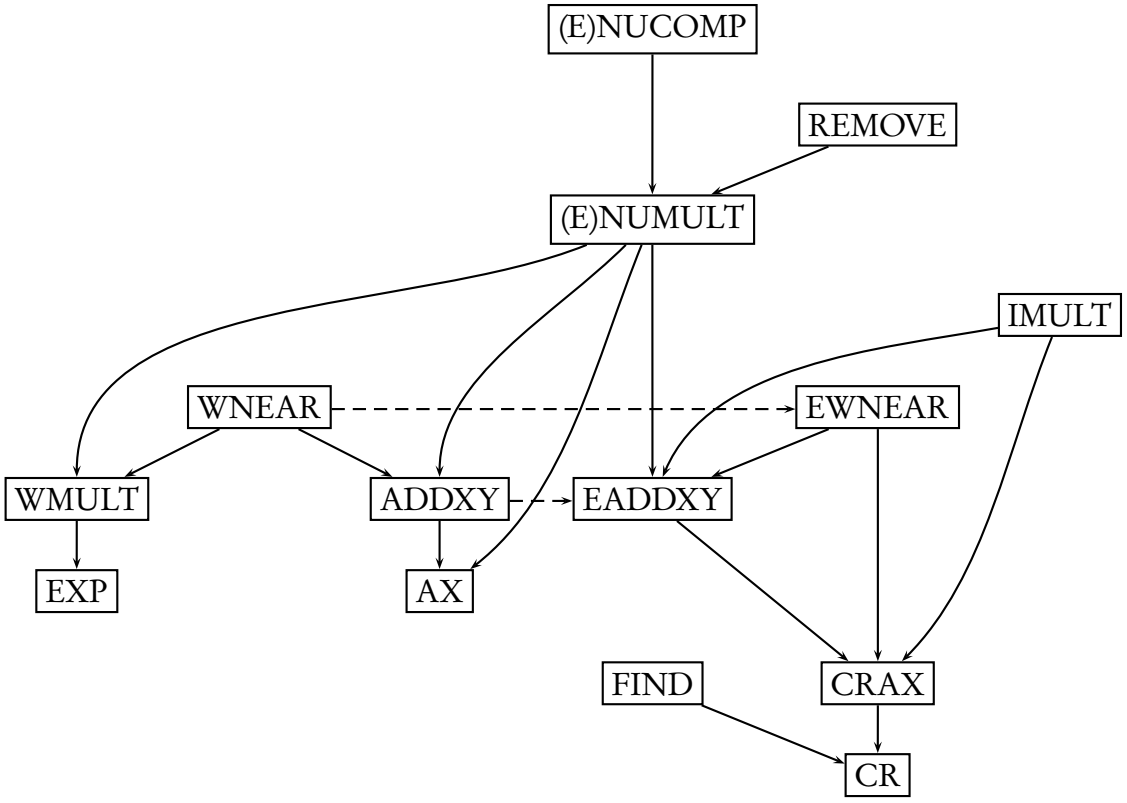


Figure 2.1: Algorithm dependencies.

for any value of f.

Given an (f,p) representation $(\mathfrak{b},d,k)$ of $\mathfrak{a}$, we can determine a w-near representation $(\mathfrak{c},g,h)$ of $\mathfrak{a}$ via the algorithm WNEAR [48, Alg. 11.2, pp. 275, 454–456]. This algorithm will produce a w-near $(f+9/8,p)$ representation of $\mathfrak{a}$. With a bit of work, WNEAR can be extended to return a relative generator $\kappa = (a+b\sqrt{D})/Q$ such that $\mathfrak{c} = (\kappa)\mathfrak{b}$ where $\mathfrak{b} = [Q,P]$. This enhanced algorithm is called EWNEAR [48, Alg. 12.1, pp. 286 and 457], though we remark that the version presented in [48] is restricted to the case when $k < w$. As the algorithmic modifications we propose in Chapter 5 also require EWNEAR to produce a relative generator in the $k > w$ case, we will present this further extension in Section 5, page 145.

We can also compute (f,p) representations of the forward and backward baby steps $\rho(\mathfrak{a})$ and $\rho^{-1}(\mathfrak{a})$ via FORWARD BABY STEP [21, Alg. 3.5, p. 28] and BACKWARD BABY STEP [21, Alg. 3.6, p. 30]. Given a reduced (f,p) representation $(\mathfrak{a}_i,d_i,k_i)$ of $\mathfrak{a}$, these two algorithms will produce an $(f+2,p)$ representation $(\mathfrak{a}_{i+1},d_{i+1},k_{i+1})$ or $(\mathfrak{a}_{i-1},d_{i-1},k_{i-1})$, respectively, of $\mathfrak{a}$. If $\mathfrak{a}_i=[Q_{i-1},P_{i-1}]$, these algorithms can be easily modified to return a relative generator $\psi=(P_i+\sqrt{D})/Q_{i-1}$ such that $\rho(\mathfrak{a}_i)=(\psi)\mathfrak{a}_i$ or $\psi=(-P_{i-1}+\sqrt{D})/Q_{i-1}$ such that $\rho^{-1}(\mathfrak{a}_i)=(\psi)\mathfrak{a}_i$.

We now turn to the issue of computing ideal products. Let $(\mathfrak{b}',d',k')$ and $(\mathfrak{b}'',d'',k'')$ be, respectively, reduced (f',p) and (f'',p) representations of $\mathfrak{a}'$ and $\mathfrak{a}''$. We can use NUMULT [48, Alg. 11.1, pp. 269, 448–9] to determine a reduced (f,p) representation $(\mathfrak{b},d,k)$ of $\mathfrak{a}'\mathfrak{a}''$ where $f=17/8+f'+f''+f'f''/2^p$. Optionally, NUMULT can return $\nu\in\mathcal{O}_{\mathbb{K}}$ where

$$\mathfrak{b}=\frac{(\nu)\mathfrak{b}'\mathfrak{b}''}{N(\mathfrak{b}')N(\mathfrak{b}'')}\,;$$

we refer to this enhanced algorithm as ENUMULT. As mentioned previously, we prefer to work with w-near representations. If $(\mathfrak{a}[x],d_x,k_x)$ and $(\mathfrak{a}[y],d_y,k_y)$ are respectively, x- and y-near (f',p) and (f'',p) representations of $\mathfrak{a}=(1)$, we can employ ADDXY [48, Alg. 11.5, p. 279] to produce an $x+y$-near (f,p) representation $(\mathfrak{a}[x+y],d,k)$ of $\mathfrak{a}$ where $f=13/4+f'+f''+f'f''/2^p$. EADDXY [48, Alg. 12.3, p. 286] is the enhanced version of this algorithm which returns $\lambda\in\mathcal{O}$ such that

$$\mathfrak{a}[x+y]=\left(\frac{\lambda}{N(\mathfrak{a}[x])N(\mathfrak{a}[y])}\right)\mathfrak{a}[x]\mathfrak{a}[y]\,.$$

Finally, if we are merely given some $x\in\mathbb{Z}^+$, we can use AX [48, Alg. 11.6, pp. 279–80] to compute an x-near (f,p) representation $(\mathfrak{a}[x],d,k)$ of $\mathfrak{a}=(1)$ for some $f\in[1,2^p)$.

2.13. Compact representations

Unlike the previous algorithms we have mentioned, we cannot extend AX to produce a simple relative generator θ such that $\mathfrak{a}[x] = (\theta)\mathfrak{a} = (\theta)$. In essence, the coefficients of such a generator would just be too large to express. In this section, we discuss why this is the case and how compact representations can be used to work around this problem. The idea for a compact representation was originally presented by Johannes Buchmann, Christoph Thiel, and Hugh Williams [12], though we will use the notation and algorithms of [48] in the description that follows. Also, although we will focus on the problem of representing the generator θ of an ideal $\mathfrak{a}[x] = (\theta)$ for $x > 0$, we remark that Michael Jacobson, Jr., and Hugh Williams discuss the compact representation of quadratic integers in a more general setting, as well as some arithmetic operations which can be applied to them [48, §§12.2–3, pp. 290–304].

Recall that the algorithm AX allows us to compute a reduced principal ideal $\mathfrak{a}$ at distance approximately x from $\mathfrak{a}_1 = (1)$. At the heart of AX is a square-and-multiply routine that uses the binary expansion of x to make a series of giant steps in the infrastructure. For each bit in the binary expansion, we compute the giant step $\mathfrak{a}_j \star \mathfrak{a}_j$—the squaring step—which results in an ideal with roughly double the distance from where we started. If the current bit is a 1, then we also adjust the resulting ideal via ρ to correct the distance we are at—the multiplying step.

How does this help us in reducing the number of bits we need to write down θ? At each stage of AX, suppose we were to keep track of the relative generator that appears. For the giant steps we would have μ_j such that $\mathfrak{a}'_{j+1} = (\mu_j / N(\mathfrak{a}_j)^2)\mathfrak{a}_j^2$ from EADDXY, and for the adjustment steps we would have ν_j such that $\mathfrak{a}_{j+1} = (\nu_j)\mathfrak{a}'_{j+1}$ from EWNEAR. We draw the reader's attention to the fact that the norm of the ideal $\mathfrak{a}_j$ is not a part of

the relative generator μ_j computed by EADDXY. As such, we must account for this and store them ourselves as $L_{j+1} = N(\mathfrak{a}_j)$.

At the end of AX, we will not only have an ideal $\mathfrak{a}_n = \mathfrak{a}[x] = (\theta)$ at distance approximately x, but also two sets of quadratic integers $\{\mu_1, \mu_2, \ldots, \mu_n\}$ and $\{\nu_1, \nu_2, \ldots, \nu_n\}$, and a set of ideal norms $\{L_1, L_2, \ldots, L_n\}$. Now, since we only perform an adjustment step for non-zero bits in the binary expansion of x, we would expect that roughly half of the ν_j values will be 1. Moreover, since the other ν_j values will be relatively small, at least compared to the μ_j values, we can quite easily combine μ_j and ν_j into a single relative generator λ_j, where $\mathfrak{a}_{j+1} = (\lambda_j / L_{j+1}^2)\mathfrak{a}_j^2$. In fact, if

$$\mu_j = \frac{a_1 + b_1\sqrt{\Delta}}{r} \quad \text{and} \quad \nu_j = \frac{a_2 + b_2\sqrt{\Delta}}{N(\mathfrak{a}'_{j+1})},$$

then

$$\lambda_j = \frac{\left(\dfrac{a_1 a_2 + \Delta b_1 b_2}{r N(\mathfrak{a}'_{j+1})}\right) + \left(\dfrac{a_1 b_2 + a_2 b_1}{r N(\mathfrak{a}'_{j+1})}\right)\sqrt{\Delta}}{r}.$$

This process is handled by the algorithm IMULT [48, Alg. 12.2, p. 286].

At this point it should be clear that if we combine the relative generators λ_j and ideal norms L_j by an appropriate combination of multiplications, divisions, and exponentiations, we will get the generator θ. In essence, what we have done so far is simply to find an explicit power product representation of θ:

$$\theta = \prod_{i=0}^{l} \left(\frac{\lambda_i}{L_i^2}\right)^{2^{l-i}}. \tag{2.15}$$

We know that for large values of Δ, it is infeasible to expand this power product. But,

what if we just left these terms in a formal power product? How much space do we need to represent these individual terms? As we will now show, these two sets of relative generators and ideal norms are exactly what we are looking to write down as a compact representation.

Let the binary representation of x be given as $x = \sum_{i=0}^{l} 2^{l-i} b_i$, where $b_0 = 1$ and $b_i \in \{0, 1\}$, and let L_{i+1}, λ_i be as above. Considering (2.15), as $l = \lceil \log_2 x \rceil$ and $x \approx \log_2 \theta$, we see that this product has $O(\log_2 \log_2 \theta)$ terms, and as the ideals we compute while executing AX are reduced, we know that $0 < L_{i+1} < \sqrt{\Delta}$. In order to say something about the size of λ_i, we must now introduce the concept of height.

Definition 2.10. *Let $\alpha \in \mathcal{O}_{\mathbb{K}}$. The* height *of α is $H(\alpha) = \max\{|\alpha|, |\overline{\alpha}|\}$.*

Recalling that $|N(\alpha)| = |\alpha\overline{\alpha}| \geq 1$, we see that $H(\alpha) \geq 1$ and so an element's height cannot be arbitrarily small. An upper bound for $H(\alpha)$ can also be found in our case, where α is the relative generator resulting from an application of ENUCOMP.

Let $\lambda = (m + n\sqrt{\Delta})/r$ where $m, n \in \mathbb{Z}$ and $r = 1, 2$ depending on $\Delta \pmod 4$. For each of the squaring steps, we compute $\mathfrak{a}[s_{i+1}] = \mathfrak{a}[2s_i]$ and if we let $\mathfrak{a}[s_i] = (\pi_i)$, then in light of (2.15) we find

$$\pi_{i+1} = \left(\frac{\lambda_{i+1}}{L_{i+1}^2} \right) \pi_i^2 \,. \tag{2.16}$$

Now, since the ideals $\mathfrak{a}[s_i]$ are reduced principal ideals, we must also have some θ_j in the simple continued fraction expansion of $\sqrt{\Delta}$ such that $\mathfrak{a}[s_i] = (\theta_j)$. By Lemma 2.9, page 37, we know

$$\frac{15N\left(\mathfrak{a}[s_i]\right)}{16\sqrt{\Delta}} 2^{s_i} < \pi_i < \frac{17}{16} 2^{s_i}$$

and, with a change of subscripts and slight rearrangement of terms,

$$\frac{1}{\pi_{i-1}} < \frac{16\sqrt{\Delta}}{15N(\mathfrak{a}[s_{i-1}])} 2^{-s_{i-1}} .$$

Combining these inequalities with (2.16), we see

$$\lambda_i = \frac{L_i^2 \pi_i}{\pi_{i-1}^2} < \left(\frac{16\sqrt{\Delta}}{15L_i} 2^{-s_{i-1}} \right)^2 \left(\frac{17L_i^2}{16} 2^{s_i} \right) = \frac{16 \cdot 17}{15^2} 2^{s_i - 2s_{i-1}} \Delta . \tag{2.17}$$

Since $s_i - 2s_{i-1} \in \{0, 1\}$, we have

$$\lambda_i < \frac{5}{2}\Delta .$$

Using a similar argument we can show

$$|\overline{\lambda}_i| \leq \frac{17^2}{15 \cdot 16}\sqrt{\Delta} < \frac{7}{5}\sqrt{\Delta} ,$$

and so

$$H(\lambda_i) < \frac{5}{2}\Delta . \tag{2.18}$$

Combining all of the previous observations gives us the definition of a compact representation that we are seeking.

Definition 2.11. *For any θ such that $(\theta) = \mathfrak{a}[x] \in \mathcal{O}_{\mathbb{K}}$, a* compact representation *of θ is*

$$\theta = \prod_{i=0}^{l} \left(\frac{\lambda_i}{L_i^2} \right)^{2^{l-i}}$$

where the following properties are satisfied:

1. $l = \mathrm{O}(\log\log\theta)$ *for large θ.*
2. $\lambda_i \in \mathcal{O}_{\mathbb{K}}$ *and L_i is an integer ($0 \leq i \leq l$).*

3. $0 < L_i \leq \Delta^{1/2}$ *and* $H(\lambda_i) = \mathrm{O}(\Delta)$ *(*$0 \leq i \leq l$*).*

4. $\pi_j \in \mathcal{O}_{\mathbb{K}}$, $L_j = |N(\pi_j)|$,

$$\pi_j = \prod_{i=0}^{j} \left(\frac{\lambda_i}{L_i^2}\right)^{2^{j-i}},$$

π_j generates a reduced ideal $\mathfrak{b}_j$, where $\mathfrak{b}_0 = \mathfrak{a}[1]$, and

$$L_{i+1}^2 \mathfrak{b}_{i+1} = \lambda_{i+1} \mathfrak{b}_i^2 \quad (0 \leq i \leq l-1).$$

In the remainder of this thesis, we will specify compact representations using the set of triples $\{(m_i, n_i, L_i)\}$, where $\lambda_i = (m_i + n_i\sqrt{D})/r$.

We remark that this definition is slightly different from that given in [48]. In particular, if we expand the product in Definition 2.11, we have

$$\theta = \left(\frac{\lambda_l}{L_l^2}\right)\left(\frac{\lambda_{l-1}}{L_{l-1}^2}\right)^2\left(\frac{\lambda_{l-2}}{L_{l-2}^2}\right)^4 \cdots \left(\frac{\lambda_2}{L_2^2}\right)^{2^{l-2}}\left(\frac{\lambda_1}{L_1^2}\right)^{2^{l-1}}\left(\frac{\lambda_0}{L_0^2}\right)^{2^l}.$$

Notice that upon substituting $d_i := L_{i+1}$, $\lambda := d_l$, $L_0 = N((1)) = 1$ and shifting the denominators right one term, we get

$$\begin{aligned}
\theta &= \left(\frac{\lambda_l}{d_{l-1}^2}\right)\left(\frac{\lambda_{l-1}}{d_{l-2}^2}\right)^2\left(\frac{\lambda_{l-2}}{d_{l-3}^2}\right)^4 \cdots \left(\frac{\lambda_2}{d_1^2}\right)^{2^{l-2}}\left(\frac{\lambda_1}{d_0^2}\right)^{2^{l-1}}\left(\frac{\lambda_0}{1}\right)^{2^l} \\
&= \frac{d_l}{d_l}\left(\frac{\lambda_l}{d_{l-1}^2}\right)\left(\frac{\lambda_{l-1}}{d_{l-2}^2}\right)^2\left(\frac{\lambda_{l-2}}{d_{l-3}^2}\right)^4 \cdots \left(\frac{\lambda_2}{d_1^2}\right)^{2^{l-2}}\left(\frac{\lambda_1}{d_0^2}\right)^{2^{l-1}}\left(\frac{\lambda_0}{1}\right)^{2^l} \\
&= d_l\left(\frac{\lambda_l}{d_l}\right)\left(\frac{\lambda_{l-1}}{d_{l-1}}\right)^2\left(\frac{\lambda_{l-2}}{d_{l-2}}\right)^4 \cdots \left(\frac{\lambda_2}{d_2}\right)^{2^{l-2}}\left(\frac{\lambda_1}{d_1}\right)^{2^{l-1}}\left(\frac{\lambda_0}{d_0}\right)^{2^l} \\
&= \lambda \prod_{i=0}^{l} \left(\frac{\lambda_i}{d_i}\right)^{2^{l-i}},
\end{aligned}$$

which is the power product given in [48, (12.8), p. 290].

Returning to the Cattle Problem, the base-2 regulator of $\mathbb{Q}(\sqrt{410286423278424})$ is approximately $\mathcal{R} = 343065.070997\ldots$, which can be verified via a number of software packages. Using the ceiling of this value as input to CRAX [48, Alg. 12.4, p. 287], along with an appropriate precision p, we obtain the compact representation given in Table 2.1. For brevity's sake, we will not include any trivial components—those relative generators $\lambda_i = 1$—in the representations generated by CRAX or its variants to be presented later. Moreover, the bit-counts listed in the text have been reduced by 3 bits per trivial generator removed, one for each of $m_i = 1$, $n_i = 0$, and $L_i = 1$.

In order to write down θ using standard decimal representation, we require $O(\log_2 \theta)$ bits. However, using a compact representation, we require only $O((\log_2 \log_2 \theta) \log_2 \Delta)$ bits to express θ. In our example, writing out the coefficients of the fundamental unit $\eta_{410286423278424}$ would require 686,106 bits; the compact representation in Table 2.1 takes only 1,212 bits.

Table 2.1: Compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i	i	m_i	n_i	L_i
5	692151654643	34171	1	13	105401119805274	5203573	14226959
6	1485823139703	73354	789265	14	63441154824	3132	87180
7	13018318544112	642705	4183599	15	58095534745996	2868132	14030668
8	116251981891416	5739272	15943256	16	106572415688343	5261399	3597385
9	91042564137939	4494702	3694935	17	50385403194207	2487489	3422487
10	99043143493614	4889685	12774777	18	125764468113497	6208896	3702695
11	407320198377372	20109088	10909076	19	—	—	1
12	14984774932785	739788	6772647				

— Chapter 3 —
Past Work

3.1. Introduction

In this chapter we will present an overview of the previous advancements that have been made to the process of computing the regulator of a real quadratic number field. We present the main ideas behind each method, develop explicit pseudocode for the algorithm, and give proofs of correctness and statements about their run-time complexity. We also point out the major refinements that have been made to each algorithm since its original introduction. An important point to take away from this chapter is that while significant achievements have been made in this area of computational number theory, the techniques used are increasingly complex. Each step forward draws on more and more topics from algebraic and analytic number theory, touching on some of their deep and unsolved problems.

We begin this chapter by looking at the connection between the continued fraction expansion of $\sqrt{D}$ and the cycle of reduced principal $\mathcal{O}_{\mathbb{K}}$-ideals (Section 3.2). This is followed by applying infrastructure techniques to the problem which deliver a significant reduction in algorithmic complexity (Section 3.3), as well as highlighting some practical improvements that have been made. Analytic number theory results are then considered to produce further enhancements (Section 3.4), though additional improvements from the use of index-calculus techniques comes at the expense of unconditional correctness (Section 3.5). We end the chapter with a verification procedure which takes these conditionally correct results and attempts to verify them unconditionally as efficiently as

possible (Section 3.6).

As mentioned in Chapter 1, as part of this thesis we have produced an implementation of the majority of the algorithms that follow. We will only give abbreviated signatures for the functions described in this chapter. For the complete signatures, and those of the supporting algorithms, we refer the reader to the C header files for our library.

3.2. The Continued Fraction Algorithm

As we noted in Section 2.5 (Theorem 2.6, page 21), there are only a finite number of reduced principal ideals in a real quadratic number field and we can arrange them in a cycle based on the distance of their generators (page 32). Moreover, if the length of this cycle is p, then the generator of $\mathfrak{a}_{p+1}$ is the fundamental unit, η_Δ. This leads us to the first algorithm we can devise for computing the regulator of a real quadratic number field. By starting with the ideal $\mathfrak{a}_1 = (1)$, computing the continued fraction expansion of ω and traversing the cycle of reduced principal ideals one-by-one, we can determine the regulator.

Before presenting the algorithm, we remark that an improvement can be made by using *ambiguous ideals*, that is $\mathcal{O}$-ideals $\mathfrak{a}$ satisfying $\mathfrak{a} = \overline{\mathfrak{a}}$. While computing the continued fraction expansion, when we encounter either $Q_s = Q_{s+1}$ or $P_s = P_{s+1}$, then we can immediately calculate the regulator [48, Thm. 5.20]. If $s \geq 1$ is minimal with one of the above equalities, we know

$$\eta_\Delta = \begin{cases} \dfrac{Q_s}{Q_0} \displaystyle\prod_{i=1}^{s} \phi_i^{\,2} & \text{if } P_s = P_{s+1}\,, \\ \dfrac{P_{s+1} + \sqrt{D}}{Q_0} \displaystyle\prod_{i=1}^{s} \phi_i^{\,2} & \text{if } Q_s = Q_{s+1}\,, \end{cases}$$

and hence,

$$\mathcal{R} = \begin{cases} \log_2 \dfrac{Q_s}{Q_0} + 2\sum_{i=1}^{s} \log_2 \phi_i & \text{if } P_s = P_{s+1}\,, \\ \log_2 \dfrac{P_{s+1} + \sqrt{D}}{Q_0} + 2\sum_{i=1}^{s} \log_2 \phi_i & \text{if } Q_s = Q_{s+1}\,. \end{cases}$$

The corresponding algorithm is presented in Algorithm 3.1, which has been implemented in our library as `mpz_qf_compute_regulator(..., ALG_CFRAC)`.

Algorithm 3.1: Regulator of a real quadratic number field (continued fraction)

Input: $D > 0$.
Output: An approximation of the base-2 regulator $\mathcal{R}$.

1: Set $Q_0 = r$, $P_0 = r\lfloor(\lfloor\sqrt{D}\rfloor - r + 1)/r\rfloor + r - 1$, $\mathfrak{a}_1 = [Q_0, P_0]$, $\mathcal{R} = 0$, and $i = 1$.
2: **while** $\mathfrak{a}_i \neq \mathfrak{a}_1$ **do**
3: Set $\mathfrak{a}_{i+1} = [Q_{i+1}, P_{i+1}] = \rho(\mathfrak{a}_i)$ and $\mathcal{R} \leftarrow \mathcal{R} + \log_2((P_{i+1} + \sqrt{D})/Q_i)$.
4: **if** $P_{i+1} = P_i$ **then**
5: **return** $\mathcal{R} = \log_2(Q_i/Q_0) + 2\mathcal{R}$.
6: **else if** $Q_{i+1} = Q_i$ **then**
7: **return** $\mathcal{R} = \log_2((P_{i+1} + \sqrt{D})/Q_0) + 2\mathcal{R}$.
8: **end if**
9: Set $i \leftarrow i + 1$.
10: **end while**
11: **return** $\mathcal{R}$.

Theorem 3.2. *Algorithm 3.1 executes in* $O(\Delta^{1/2+\epsilon})$ *elementary operations.*

Proof. The correctness of this algorithm follows from the discussion in [48, §5.3, p. 113]. The run-time follows from the observation that the length p of the cycle of reduced principal ideals is $O(\sqrt{\Delta}\log\Delta)$ [48, §3.4, p. 68]. □

3.3. The Baby-Step / Giant-Step Algorithm

The next improvement to the process of computing the regulator came with the discovery of the infrastructure of a real quadratic field by Shanks [75] (described on page 35). The main idea of this algorithm is to use the continued fraction expansion of ω to compute a

list of "baby steps" $\mathfrak{a}_1, \mathfrak{a}_2, \ldots$ and then use NUCOMP to compute a series of "giant steps" $\mathfrak{b}_1, \mathfrak{b}_2, \ldots$ until we find one in the baby-step list. Once we have found, say, $\mathfrak{b}_j = \mathfrak{a}_i$, we can calculate the regulator as $\mathcal{R} = \delta(\mathfrak{b}_i) - \delta(\mathfrak{a}_j)$. The key is to construct a sufficiently long list of baby-steps so that we are guaranteed to find this match. We compute the list $\mathcal{L} = \{\mathfrak{a}_1, \mathfrak{a}_2, \ldots, \mathfrak{a}_t, \mathfrak{a}_{t+1}, \mathfrak{a}_{t+2}\}$, where t is chosen such that $\delta(\mathfrak{a}_{t-1}) < \sqrt[4]{\Delta} < \delta(\mathfrak{a}_t)$. For the series of giant steps, we take $\mathfrak{b}_1 = \mathfrak{a}_t$ and compute $\mathfrak{b}_2, \mathfrak{b}_3, \ldots$ using $\mathfrak{b}_{i+1} := \mathfrak{b}_i \star \mathfrak{b}_1$.

How do we know that we will eventually find an ideal $\mathfrak{b}_j \in \mathcal{L}$? Let $\mathfrak{a}_i = [Q_{i-1}, P_{i-1}]$ and $\mathfrak{a}_j = [Q_{j-1}, P_{j-1}]$ be two reduced principal ideals. From the material in Section 2.10, we know that if we compute the product ideal $\mathfrak{a}_i\mathfrak{a}_j$, we get $(m)\mathfrak{c} = \mathfrak{a}_i\mathfrak{a}_j$, where $m \in \mathbb{Z}$ and $\mathfrak{c}$ is a primitive, but not necessarily reduced, principal ideal. We can then compute a reduced ideal $\mathfrak{c}'$ equivalent to $\mathfrak{c}$ and we know that $\mathfrak{c}' = \mathfrak{a}_k$ for some k. Thus, we have

$$\mathfrak{a}_k = \mathfrak{a}_i\mathfrak{a}_j\left(\frac{\theta'_k}{m}\right) \quad \text{or} \quad \delta(\mathfrak{a}_k) \equiv \delta(\mathfrak{a}_i) + \delta(\mathfrak{a}_j) + \log_2\left|\frac{\theta'_k}{m}\right| \pmod{\mathcal{R}}. \tag{3.1}$$

Let $\kappa = \log_2|\theta'_k/m|$. Since we are using NUCOMP to compute reduced ideal products, we know that at the end of that algorithm we have a reduced ideal $\mathfrak{b}$ such that $(\mu)\mathfrak{b} = \mathfrak{a}_i\mathfrak{a}_j$. Combining this with (3.1), we find $|\theta'_k/m| = 1/\mu$. It can be shown [48, §A.1] that $(1/2)\Delta^{-3/4} < 1/\mu \leq 1$, and so we can bound κ by

$$\frac{-3}{4}\log_2\Delta - 1 < \kappa \leq 0. \tag{3.2}$$

Now consider two consecutive ideals in our series of giant steps, say $\mathfrak{b}_i$ and $\mathfrak{b}_{i+1}$, and (3.1). We see that $\delta(\mathfrak{b}_{i+1}) - \delta(\mathfrak{b}_i) = \delta(\mathfrak{b}_1) + \kappa \leq \delta(\mathfrak{b}_1) + 1 = \delta(\mathfrak{a}_t) + 1$. Applying (2.13), page 32, we find that $\theta_{t+2} > F_3\theta_t = 2\theta_t$, where F_3 is the third Fibonacci number. Thus, $\log_2\theta_{t+2} > \log_2\theta_t + 1$ and we have

$$\delta(\mathfrak{b}_{i+1}) - \delta(\mathfrak{b}_i) < \delta(\mathfrak{a}_{t+2})\,.$$

This means that the distance we travel using NUCOMP to compute a giant step of distance $\delta(\mathfrak{a}_t)$ never exceeds $\delta(\mathfrak{a}_{t+2})$. Since our baby-step list $\mathcal{L}$ contains all the ideals up to and including $\mathfrak{a}_{t+2}$, we are guaranteed that a giant step will eventually land in $\mathcal{L}$. Thus, we will have a $\mathfrak{b}_i \in \mathcal{L}$, in particular say $\mathfrak{b}_i = \mathfrak{a}_j$, and we can calculate the regulator as $\mathcal{R} = \delta(\mathfrak{b}_i) - \delta(\mathfrak{a}_j)$. The corresponding algorithm is presented in Algorithm 3.3.

Algorithm 3.3: Regulator of a real quadratic number field (baby-step / giant-step)

Input: $D > 0$.
Output: An approximation of the base-2 regulator $\mathcal{R}$.

1: Set $Q_0 = r$, $P_0 = r\lfloor(\lfloor\sqrt{D}\rfloor - r + 1)/r\rfloor + r - 1$, $\mathfrak{a}_1 = [Q_0, P_0]$, and $\mathcal{L} = \{\mathfrak{a}_1\}$.
/ Generate baby-step list */*
2: Compute t such that $\delta(\mathfrak{a}_{t-1}) < \sqrt[4]{D} < \delta(\mathfrak{a}_t)$.
3: **for** $i = 2, 3, \ldots, t+2$ **do**
4: Compute $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathfrak{a}_i\}$ where $\mathfrak{a}_i = \rho(\mathfrak{a}_{i-1})$.
5: **if** $\mathfrak{a}_i \in \mathcal{L}$ with $P_i = P_{i-1}$ **then**
6: **return** $\mathcal{R} = \log_2(Q_0/Q_i) + 2\delta(\mathfrak{a}_i)$.
7: **end if**
8: **if** $\mathfrak{a}_i \in \mathcal{L}$ with $Q_i = Q_{i-1}$ **then**
9: **return** $\mathcal{R} = \log_2((P_i + \sqrt{D})/Q_0) + 2\delta(\mathfrak{a}_i)$.
10: **end if**
11: **end for**
/ Compute giant steps */*
12: Set $\mathfrak{b}_1 = \mathfrak{a}_t$, $\mathfrak{b}_2 = \mathfrak{b}_1 \star \mathfrak{a}_t$ and $i = 2$.
13: **while** $\mathfrak{b}_i \notin \mathcal{L}$ **do**
14: Compute $\mathfrak{b}_{i+1} = \mathfrak{b}_i \star \mathfrak{a}_t$ and set $i \leftarrow i + 1$.
15: **end while**
16: Find $\mathfrak{a}_j \in \mathcal{L}$ such that $\mathfrak{b}_i = \mathfrak{a}_j$.
17: Return $\mathcal{R} = \delta(\mathfrak{b}_i) - \delta(\mathfrak{a}_j)$.

Theorem 3.4. *Algorithm 3.3 executes in* $O(\Delta^{1/4+\epsilon})$ *elementary operations.*

Proof. See [48, Thm. 7.16, p. 179]. □

Shanks' baby-step / giant-step algorithm gives us a large efficiency gain over the continued fraction based algorithm shown previously. However, further gains are possible. The algorithm's efficiency depends on the accuracy of the upper bound on $\mathcal{R}$ we choose, and $\sqrt{\Delta}$ is a rather loose bound. With a tighter bound on $\mathcal{R}$, we can achieve a greater balance between the number of baby-step operations and the number of giant steps, and thus have a more efficient algorithm. The modification we describe now was originally presented by Buchmann and Williams [15].

Rather than computing $\mathcal{L}$ as $\mathcal{L} = \{\mathfrak{a}_1, \mathfrak{a}_2, \ldots, \mathfrak{a}_t, \mathfrak{a}_{t+1}, \mathfrak{a}_{t+2}\}$ with t as described previously, we instead choose a small value v and check if $0 < \mathcal{R} < v^2$. We do this by computing a baby-step list $\mathcal{L}$ of ideals up to distance v—let $\mathfrak{a}_t$ satisfy $\delta(\mathfrak{a}_t) > v > \delta(\mathfrak{a}_{t-1})$—and by taking giant steps of distance v using $\mathfrak{b}_1 = \mathfrak{a}_t$. If $\mathcal{R}$ is not found within this range, we extend $\mathcal{L}$ by setting $\mathcal{L} = \mathcal{L} \cup \{\mathfrak{a}_{t+3}, \mathfrak{a}_{t+4}, \ldots, \mathfrak{a}_u, \mathfrak{a}_{u+1}, \mathfrak{a}_{u+2}\}$, where $\mathfrak{a}_u$ satisfies $\delta(\mathfrak{a}_u) > 2v > \delta(\mathfrak{a}_{u-1})$, and then take giant steps of distance $2v$ by using $\mathfrak{b}_1 = \mathfrak{a}_u$. This doubling process is continued until we find some range such that $(2^{k-1}v)^2 < \mathcal{R} < (2^k v)^2$. At worst, we will require the computation of $2^k v$ baby steps and giant steps.

The corresponding algorithm is presented in Algorithm 3.5, which has been implemented in our library as `mpz_qf_compute_regulator(..., ALG_BSGS)`.

Algorithm 3.5: Regulator of a real quadratic number field (BS/GS improved)

Input: $D > 0$, v.
Output: An approximation of the base-2 regulator $\mathcal{R}$.

1: Set $Q_0 = r$, $P_0 = r\lfloor(\lfloor\sqrt{D}\rfloor - r + 1)/r\rfloor + r - 1$, $\mathfrak{a}_1 = [Q_0, P_0]$, $\mathcal{L} = \{\mathfrak{a}_1\}$, and $s = 2$.
/ Generate (or expand) baby-step list */*
2: Compute t such that $\delta(\mathfrak{a}_{t-1}) < v < \delta(\mathfrak{a}_t)$.
3: **for** $i = s, s+1, \ldots, t+2$ **do**
  4: Compute $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathfrak{a}_i\}$ where $\mathfrak{a}_i = \rho(\mathfrak{a}_{i-1})$.
  5: **if** $\mathfrak{a}_i \in \mathcal{L}$ with $P_i = P_{i-1}$ **then**
    6: **return** $\mathcal{R} = \log_2(Q_0/Q_i) + 2\delta(\mathfrak{a}_i)$.
  7: **end if**

8: **if** $\mathfrak{a}_i \in \mathcal{L}$ with $Q_i = Q_{i-1}$ **then**
9: **return** $\mathcal{R} = \log_2((P_i + \sqrt{D})/Q_0) + 2\delta(\mathfrak{a}_i)$.
10: **end if**
11: **end for**
12: Set $s \leftarrow t + 3$.
/ Compute giant steps up to distance v^2 */*
13: Set $\mathfrak{b}_1 = \mathfrak{a}_t$, $\mathfrak{b}_2 = \mathfrak{b}_1 \star \mathfrak{a}_t$ and $i = 2$.
14: **while** $\delta(\mathfrak{b}_i) < v^2$ **do**
15: Compute $\mathfrak{b}_{i+1} = \mathfrak{b}_i \star \mathfrak{a}_t$ and set $i \leftarrow i + 1$.
16: **if** $\mathfrak{b}_i \in \mathcal{L}$ **then**
17: **goto** Step 22
18: **end if**
19: **end while**
/ Increase the regulator upper bound and recompute */*
20: Set $v \leftarrow 2v$.
21: **goto** Step 2
22: Find $\mathfrak{a}_j \in \mathcal{L}$ such that $\mathfrak{b}_i = \mathfrak{a}_j$.
23: Return $\mathcal{R} = \delta(\mathfrak{b}_i) - \delta(\mathfrak{a}_j)$.

Theorem 3.6. *Algorithm 3.5 executes in* $O(\mathcal{R}^{1/2}\Delta^{\epsilon})$ *elementary operations.*

Proof. See the discussion following the proof of Theorem 7.16 in [48, p. 179]. □

An improved version of this method has been presented by Buchmann and Vollmer [13, 14], based on the work of Terr [78] for computing the order of an element in a finite abelian group. Terr showed that for an element g in a finite abelian group G, there exist integers $e \geq 0$ and $0 \leq f < e$ satisfying $g^{e(e+1)/2} = g^f$. Moreover, if e is minimal, then $\mathrm{ord}(g) = e(e+1)/2 - f$ [14, Lem. 9.7.7, p. 205].

In our case, this result can be applied to the infrastructure of a real quadratic field by taking note of its similarity to the cyclic group $\langle g \rangle$ [14, Prop. 10.2.1, p. 223]. We begin by computing a series of baby steps $\mathfrak{a}_1 = (\alpha_1), \mathfrak{a}_2 = (\alpha_2), \ldots, \mathfrak{a}_i = (\alpha_i), \ldots$ until we find L such that $\delta(\mathfrak{a}_{2(L+1)}) \geq (1/2)\log_2 \Delta$. The giant steps $\mathfrak{b}_1, \mathfrak{b}_2, \ldots$ are computed by determining $\gamma_i \geq 0$ such that $\mathfrak{b}_i = (\gamma_i/\alpha_{2(L+i)})\mathfrak{b}_{i-1} = (\beta_i)\mathfrak{b}_{i-1}$. Once we find $\mathfrak{b}_e = \mathfrak{a}_f$,

where $f \in \{1, 2, \ldots, 2(e + L + 1)\}$, then we can compute

$$\mathcal{R} = \log_2 \left(\frac{\alpha_f}{\prod_{i=1}^{e} \beta_i} \right).$$

This algorithm runs in time $O((\log \Delta + \sqrt{\mathcal{R}})(\log \Delta)^3)$ [14, Prop. 10.2.7, p. 228]. Further details, including a pseudocode implementation, can be found in Section 10.2 of [14, pp. 222–230]. Due to time constraints, this algorithm was not implemented in our library.

3.4. An Analytic Algorithm

The next significant advance in real quadratic field regulator computations came about from the work of H. W. Lenstra, Jr. [55]. It improves upon Shanks' baby-step / giant-step algorithm by combining it with some key analytic number theory results. The main idea of this algorithm is to compute an approximation of the product $h\mathcal{R}$ using the analytic class number formula and a sufficiently accurate approximation of $L(1, \chi)$. By bounding the error term of this approximation, we can produce a range in which $h\mathcal{R}$ is known to be (§3.4.1). We then employ infrastructure techniques to compute an integer multiple $h^*\mathcal{R}$ of the regulator by searching this interval (§3.4.2). By applying these techniques a second time, we can determine the value of the multiplier h^* and thus calculate $\mathcal{R}$ (§3.4.3). It is important to note that although the complexity of this algorithm is contingent on the truth of the extended Riemann Hypothesis (ERH), the computed regulator is unconditionally correct. The ERH is only used to estimate the error in approximating $h\mathcal{R}$.

3.4.1. Determining a range for $h\mathcal{R}$. The first stage of the algorithm is to determine a range in which $h\mathcal{R}$ is known to be. Recall from our discussion of $L(1,\chi)$ that we can write it as an Euler product: $L(1,\chi)=\prod_p \left(1-\chi(p)/p\right)^{-1}$. One way we can approximate $L(1,\chi)$ is by considering the truncated version of this product. In other words, we let

$$B(x,\chi)=\prod_{p<x}\left(1-\frac{\chi(p)}{p}\right)^{-1} \tag{3.3}$$

and try to find some x such that $B(x,\chi)$ is as close to $L(1,\chi)$ as we desire. To do this, we will need to derive bounds on the error of this approximation which, quite clearly, come from the "tail" of the Euler product—that is, the terms left off the product. Let

$$\overline{B}(x,\chi)=\prod_{p\geq x}\left(1-\frac{\chi(p)}{p}\right)^{-1}$$

represent the product of these terms. As noted in [48], we can show that, under the ERH, the relative error in using the truncated Euler product as an approximation is

$$\left|\log\overline{B}(x,\chi)\right| = O\left(\frac{\log|\Delta|\log x}{\sqrt{x}}\right).$$

This method has been improved by Bach [4] by reducing the relative error to $O(\log\Delta/(\sqrt{x}\log x))$ as follows. Rather than working with a single value of x, we instead average (3.3) over several nearby values. We take the values $x+i$ $(i=0,1,\ldots,x-1)$ and determine the corresponding values of $B(x+i,\chi)$, weighting each by the coefficient

$$a_i=\frac{(x+i)\log(x+i)}{\sum_{j=0}^{x-1}(x+j)\log(x+j)}.$$

Bach showed [4, Thm. 6.3] that under the GRH, these weighted terms satisfy the inequality

$$\left| L(1,\chi) - \sum_{i=0}^{x-1} a_i \log B(x+i,\chi) \right| < \frac{A\log|\Delta| + B}{\sqrt{x}\log x}\,, \tag{3.4}$$

for constants A and B, and it is this approximation that is used in modern implementations of Lenstra's algorithm. For convenience, we define

$$S(x) = \sum_{i=0}^{x-1} a_i \log B(x+i,\chi) \quad \text{and} \quad A(x) = \frac{A\log|\Delta| + B}{\sqrt{x}\log x}\,.$$

Remember that we are trying to produce a range in which we know that $h\mathcal{R}$ lies. By appealing to the analytic class number formula (Theorem 2.7, page 23), we see that (3.4) implies that

$$h\mathcal{R} \approx \frac{\sqrt{\Delta}\exp(S(x))}{2\log 2} =: E\,.$$

More precisely, we can show that the interval we are looking for is

$$|h\mathcal{R} - E| \le \frac{E}{\log 2}|\exp(A(x)) - 1| =: L^2\,.$$

As is shown in [48, Thm. 10.1, p. 242], $x = \Delta^{1/5}$ is the optimal value for this approximation in conjunction with the rest of Lenstra's algorithm.

3.4.2. Computing a multiple of $\mathcal{R}$. Once we have an approximation of $h\mathcal{R}$ and a range in which the true value lies, we can proceed to determine an integer multiple of the regulator. We begin by computing a list $\mathcal{L}$ of baby steps consisting of all the reduced principal ideals of distance less than $L+1$. We next compute a reduced principal ideal $\mathfrak{a}_m$ such that $\delta(\mathfrak{a}_m) \approx E$. It is this ideal that we will use to compute an integer multiple $h^*\mathcal{R}$.

Let $\mathfrak{a}_t \in \mathcal{L}$ such that $\delta(\mathfrak{a}_t) < L < \delta(\mathfrak{a}_{t+1})$. We compute a series of giant steps by setting $\mathfrak{c}_1 = \mathfrak{a}_m$ and computing $\mathfrak{c}_2, \mathfrak{c}_3, \ldots$ using $\mathfrak{c}_{i+1} := \mathfrak{c}_i \star \mathfrak{a}_t$. This differs slightly from Shanks' $O(\Delta^{1/4+\epsilon})$ algorithm where we computed the series of giant steps using $\mathfrak{b}_1 = \mathfrak{a}_t$ and $\mathfrak{b}_{i+1} := \mathfrak{b}_i \star \mathfrak{a}_t$ (note the subscripts on the $\mathfrak{a}$ ideals). A second difference is that we check for both $\mathfrak{c}_i \in \mathcal{L}$ and $\overline{\mathfrak{c}}_i \in \mathcal{L}$. If $\mathfrak{c}_i = \mathfrak{a}_k \in \mathcal{L}$, then

$$h^*\mathcal{R} = \delta(\mathfrak{c}_j) - \delta(\mathfrak{a}_k)\,.$$

Similarly, if $\overline{\mathfrak{c}}_j = \mathfrak{a}_k \in \mathcal{L}$, then

$$h^*\mathcal{R} = \delta(\mathfrak{a}_m) - \Big(\delta(\mathfrak{c}_1, \mathfrak{c}_j) - \delta(\mathfrak{a}_k)\Big) - \log_2\left(\frac{Q_{m-1}}{r}\right)\,,$$

where by $\delta(\mathfrak{c}_1, \mathfrak{c}_j)$ we mean the distance from $\mathfrak{c}_1$ to $\mathfrak{c}_j$. In the pseudocode to follow, we will refer to this multiple $h^*\mathcal{R}$ as $\mathcal{R}'$.

3.4.3. Determining h^*. Once we have computed the integer multiple $h^*\mathcal{R}$ of the regulator, we can proceed with determining the value of h^* and hence $\mathcal{R}$. First, we determine whether or not $\mathcal{R} > E/\sqrt{L}$. The reason for this is to ensure the complexity results by optimizing the balance between a baby-step / giant-step search for the regulator and trial division of the product $h^*\mathcal{R}$ to calculate h^*. Moreover, verifying this bound allows us to prevent complications with the determination of h^* as we will see later.

As with the previous part of this algorithm, verifying this lower bound is done via a baby-step / giant-step routine: set $\mathfrak{c}_1 = \mathfrak{a}_t$ and compute $\mathfrak{c}_2, \mathfrak{c}_3, \ldots$ using $\mathfrak{c}_{i+1} := \mathfrak{c}_i \star \mathfrak{a}_t$. We continue computing this series of ideals until either we find $\mathfrak{c}_j$ or $\overline{\mathfrak{c}}_j \in \mathcal{L}$ or we find that $\delta(\mathfrak{c}_j) > E/\sqrt{L}$. In the first case, we can immediately determine $\mathcal{R}$ as

$$\mathcal{R} = \begin{cases} \delta(\mathfrak{c}_j) - \delta(\mathfrak{a}_k) & \text{if } \mathfrak{c}_j \in \mathcal{L}\,, \\ \delta(\mathfrak{c}_j) + \delta(\mathfrak{a}_k) - \log_2\left(\dfrac{Q_{j-1}}{r}\right) & \text{if } \overline{\mathfrak{c}}_j \in \mathcal{L} \end{cases}$$

and exit the algorithm. In the second case, we know that $\mathcal{R}$ must then exceed $E/\sqrt{L}$ and so we must proceed to calculate h^*.

The basic idea for this second stage is to determine the prime power factorization of h^* by taking the primes that could possibly divide it and trial dividing to see if they do. More specifically, for all primes q that could divide h^*, we compute the ideal $\mathfrak{a}$ at distance $h^*\mathcal{R}/q$ and check if $\mathfrak{a} = (1)$. If the ideal at distance $h^*\mathcal{R}/q$ is (1), then we know that $h^*\mathcal{R}/q$ is also an integer multiple of the regulator and hence $q \mid h^*$. We then determine the highest power of q that divides h^* by computing the series of ideals $\widetilde{\mathfrak{a}}_2, \widetilde{\mathfrak{a}}_3, \ldots$ at distances $h^*\mathcal{R}/q^2$, $h^*\mathcal{R}/q^3$, … (resp.) until we find an ideal $\widetilde{\mathfrak{a}}_k = (1)$, but $\widetilde{\mathfrak{a}}_{k+1} \neq (1)$. This tells us that $q^k \,||\, h^*$. On the other hand, if the ideal $\mathfrak{a}$ at distance $h^*\mathcal{R}/q$ is not equal to (1), then we know that $q \nmid h^*$. Once we have worked our way through the possible prime divisors of h^*, we know the complete factorization of h^* and can compute $\mathcal{R}$ easily. Recall that at this point of the algorithm we know that $\mathcal{R} > E/\sqrt{L}$. This means that we need only consider primes q such that $h^*\mathcal{R}/q > E/\sqrt{L}$, or equivalently $q < h^*\mathcal{R}\sqrt{L}/E$.

The corresponding algorithm is presented in Algorithm 3.7, which has been implemented in our library as `mpz_qf_compute_regulator(..., ALG_ANALYTIC)`.

Algorithm 3.7: Regulator of a real quadratic number field (analytic improvements)

Input: Discriminant $\Delta > 0$.
Output: An approximation of the base-2 regulator $\mathcal{R}$.

1: Set $Q = \Delta^{1/5}$.
/ Approximation of $h\mathcal{R}$ */*
2: Compute $E = \sqrt{\Delta}\exp(S(Q,\Delta))/(2\log 2)$ and $L = \sqrt{(E/\log 2)|\exp(A(Q,\Delta)) - 1|}$.
/ Integer multiple $h^*\mathcal{R}$ $(= \mathcal{R}')$ */*

3: Set $\mathfrak{a}_1 = (1)$, $\mathcal{L} = \{\mathfrak{a}_1\}$, and $i = 1$.
4: **while** $\delta(\mathfrak{a}_i) < L + 1$ **do**
5: Compute $\mathfrak{a}_{i+1} = \rho(\mathfrak{a}_i)$ and set $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathfrak{a}_{i+1}\}$.
6: **if** $\mathfrak{a}_{i+1} = [Q_i, P_i]$ with $P_i = P_{i-1}$ **then**
7: **return** $\mathcal{R} = 2\delta(\mathfrak{a}_i) + \log_2(Q_0/Q_{i-1})$.
8: **end if**
9: **if** $\mathfrak{a}_{i+1} = [Q_i, P_i]$ with $Q_i = Q_{i-1}$ **then**
10: **return** $\mathcal{R} = 2\delta(\mathfrak{a}_i) + \log_2(Q_0\psi_i/Q_{i-1})$.
11: **end if**
12: Set $i \leftarrow i + 1$.
13: **end while**
14: Compute $\mathfrak{a}_m$ such that $\delta(\mathfrak{a}_m) \approx E$ and set $\mathfrak{c}_1 = \mathfrak{a}_m$ and $i = 1$.
15: **repeat**
16: Compute $\mathfrak{c}_{i+1} = \mathfrak{c}_i \star \mathfrak{a}_t$.
17: **if** $\mathfrak{c}_i = \mathfrak{a}_k \in \mathcal{L}$ **then**
18: Set $\mathcal{R}' = \delta(\mathfrak{c}_i) - \delta(\mathfrak{a}_k)$
19: **goto** Step 27
20: **end if**
21: **if** $\overline{\mathfrak{c}}_i = \mathfrak{a}_k \in \mathcal{L}$ **then**
22: Set $\mathcal{R}' = \delta(\mathfrak{a}_m) - \big(\delta(\overline{\mathfrak{c}}_i), \delta(\overline{\mathfrak{c}}_1) - \delta(\mathfrak{a}_k)\big) - \log_2(Q_{m-1}/r)$
23: **goto** Step 27
24: **end if**
25: Set $i \leftarrow i + 1$.
26: **end repeat**
/ Determine if $\mathcal{R} > E/\sqrt{L}$ */*
27: Set $\mathfrak{c}_1 = \mathfrak{a}_t$, $\mathfrak{c}_2 = \mathfrak{c}_1 \star \mathfrak{a}_t$, and $i = 2$.
28: **do**
29: **if** $\mathfrak{c}_i = \mathfrak{a}_k \in \mathcal{L}$ **then**
30: **return** $\mathcal{R} = \delta(\mathfrak{c}_i) - \delta(\mathfrak{a}_k)$.
31: **else**
32: Set $\mathfrak{c}_{i+1} = \mathfrak{c}_i \star \mathfrak{a}_t$ and $i \leftarrow i + 1$.
33: **end if**
34: **while** $\delta(\mathfrak{c}_i) < E/\sqrt{L}$
/ Compute the factorization of h^* */*
35: Set $h^* = 1$ and $q = 2$.
36: **while** $p < \sqrt{L} + L^{5/2}/E$ **do**
37: Set $i \leftarrow 1$ and compute $\mathfrak{a} = \mathfrak{a}[\mathcal{R}/q]$.
38: **while** $\mathfrak{a} = (1)$ **do**
39: Set $i \leftarrow i + 1$ and compute $\mathfrak{a} = \mathfrak{a}[\mathcal{R}/q^i]$.
40: **end while**

41: Set $h^* \leftarrow h^* \cdot q^{(i-1)}$ and q to the next prime greater than q.
42: **end while**
43: **return** $\mathcal{R} = \mathcal{R}'/h^*$.

Theorem 3.8. *Algorithm 3.7 executes in* $\mathrm{O}(\Delta^{1/5+\epsilon})$ *elementary operations.*

Proof. See [48, Thm. 10.1, pp. 242–3]. □

In practice, however, there is a more efficient method for computing the sequence of $\mathfrak{a}[\mathcal{R}/q_i]$ ideals, where q_i denotes the i^{th} prime. The key observation to make is that we can reuse the work expended while computing $\mathfrak{a}[h^*\mathcal{R}/q_{i+1}]$ in order to compute $\mathfrak{a}[h^*\mathcal{R}/q_i]$. This method was originally described in [43, pp. 214–5], though our description below follows the notation used in [48, §10.2, p. 244].

For the prime q_i, we need to compute a reduced principal ideal $\mathfrak{a}_n$ which satisfies

$$\frac{h^*\mathcal{R}}{q_i} < \delta(\mathfrak{a}_n) < \frac{h^*\mathcal{R}}{q_i} + \delta(\mathfrak{a}_t)\,.$$

Having already computed $\mathfrak{a}_m = \mathfrak{a}[h^*\mathcal{R}/q_{i+1}]$ for the preceding prime, we have an ideal such that

$$\frac{h^*\mathcal{R}}{q_{i+1}} < \delta(\mathfrak{a}_m) < \frac{h^*\mathcal{R}}{q_{i+1}} + \delta(\mathfrak{a}_t)$$

and, if we were to compute an ideal $\mathfrak{a}_s$ at distance $\delta(\mathfrak{a}_s) \approx h^*\mathcal{R}/q_i - \delta(\mathfrak{a}_m)$, then by setting $\mathfrak{a}_n = \mathfrak{a}_s \star \mathfrak{a}_m$ we would find

$$\frac{h^*\mathcal{R}}{q_i} < \delta(\mathfrak{a}_s) + \delta(\mathfrak{a}_m) \approx \delta(\mathfrak{a}_n) \leq \frac{h^*\mathcal{R}}{q_i} + \delta(\mathfrak{a}_t)$$

as desired.

Now $\mathfrak{a}_s$ could be determined via AX, however it is more efficient to compute it from

a list of precomputed ideals. Set $\mathcal{I} = \{\mathfrak{a}_{t_0}, \mathfrak{a}_{t_1}, \mathfrak{a}_{t_2}, \ldots, \mathfrak{a}_{t_k}\}$ where $\delta(\mathfrak{a}_{t_i}) \approx 2^i \delta(\mathfrak{a}_t)$ and k is sufficiently large. If we also set $r\delta(\mathfrak{a}_t) = h^*\mathcal{R}/q_i + \delta(\mathfrak{a}_m)$ for some real number r, then

$$\delta(\mathfrak{a}_s) \approx ([r]+1)\delta(\mathfrak{a}_t) = q\delta(\mathfrak{a}_t) \tag{3.5}$$

where $q \in \mathbb{Z}$. To compute $\mathfrak{a}_s$ with distance approximately $q\delta(\mathfrak{a}_t)$ using $\mathcal{I}$, we determine a reduced ideal $\mathfrak{b}$ equivalent to

$$\prod_{j=0}^{k} \mathfrak{a}_{t_j}{}^{b_j}$$

where the b_j come from the binary expansion of $q = \sum_{i=0}^{k} 2^i b_i$. Notice that

$$\begin{aligned}
\delta(\mathfrak{b}) &\approx b_k \delta\left(\mathfrak{a}_{t_k}\right) + b_{k-1}\delta\left(\mathfrak{a}_{t_{k-1}}\right) + \cdots + b_1\delta\left(\mathfrak{a}_{t_1}\right) + b_0\delta\left(\mathfrak{a}_{t_0}\right) \\
&\approx 2^k b_k \delta(\mathfrak{a}_t) + 2^{k-1} b_{k-1}\delta(\mathfrak{a}_t) + \cdots + 2b_1\delta(\mathfrak{a}_t) + b_0\delta(\mathfrak{a}_t) \\
&= \left(\sum_{i=0}^{k} 2^i b_i\right)\delta(\mathfrak{a}_t) = q\delta(\mathfrak{a}_t) \approx \delta(\mathfrak{a}_s)\,.
\end{aligned}$$

Due to time constraints, this improvement was not included in our implementation of Algorithm 3.7.

3.5. The Index-Calculus Method

Up until this point, all the algorithms we have discussed have exponential expected runtimes in $\log\Delta$. The index-calculus algorithm we present next has a subexponential runtime [10].

All index-calculus algorithms can be broken down into two major components: generating random smooth objects and solving a linear algebra problem. In our case, real quadratic number fields, the random objects we generate are smooth principal ideals.

These are principal ideals that factor into a product of prime ideals of small norm. Before we can describe the idea behind the index-calculus algorithm in more detail, we need to introduce a few definitions.

Definition 3.9. *Let $p_1, p_2, \ldots, p_k \in \mathbb{Z}$ be the first k rational primes that are not inert in $\mathbb{Q}(\sqrt{\Delta})$ (i.e., the Kronecker symbol $(\Delta/p_i) \neq -1$). For each p_i, let $\mathfrak{p}_i$ be a prime $\mathcal{O}_{\mathbb{K}}$-ideal such that $\mathfrak{p}_i \mid (p_i)$. If p_i ramifies, that is $(\Delta/p_i) = 0$, we have a unique choice for $\mathfrak{p}_i$. If p_i splits, $(\Delta/p_i) = 1$, then there are two ideals to choose from: $\mathfrak{p}_i$ and $\overline{\mathfrak{p}}_i$. It does not matter which we select, so long as we only select one. We call the set* $\mathrm{FB} = \{\mathfrak{p}_1, \mathfrak{p}_2, \ldots, \mathfrak{p}_k\}$ *a* factor base.

Definition 3.10. *Let (α) be a principal ideal. If (α) factors completely over the factor base, then we can write $(\alpha) = \prod_i \mathfrak{p}_i^{v_i}$ where $v_i \in \mathbb{Z}$. For convenience, we will denote this product as $\mathrm{FB}^{\vec{v}}$ and call the vector of exponents $\vec{v} = [v_i]$ a* relation. *The set of all relations Λ forms a sublattice of $\mathbb{Z}^k$ which we call the* relation lattice.

In the material that follows, we will actually be working with *extended relations* $(\vec{v}, \log_2|\alpha|)$ where $\mathrm{FB}^{\vec{v}} = (\alpha)$ and the *extended relation lattice* Λ', a sublattice of $\mathbb{Z}^k \times \mathbb{R}$. Consider the homomorphism $\Omega' : \mathbb{Z}^k \times \mathbb{R} \to \mathcal{C}l_{\mathbb{K}}$ that maps $(\vec{v}, \log_2|\alpha|)$ to the ideal class $[\mathrm{FB}^{\vec{v}}]$. Buchmann [14, Prop. 11.5.2] noted that if the ideal classes of the prime ideals in the factor base generate the class group, then Ω' is surjective and $\ker(\Omega') = \Lambda'$, so $\mathcal{C}l_{\mathbb{K}} \cong \mathbb{Z}^k/\Lambda$ and $\det(\Lambda') = h\mathcal{R}$. This is an extension of a result due to Pohst and Zassenhaus [70], who used $\Omega : \mathbb{Z}^k \to \mathcal{C}l_{\mathbb{K}}$ to show that $\det(\Lambda) = h$.

Now, in order for the ideal classes of the prime ideals in the factor base to generate the class group, we need to take all prime ideals whose norm is less than a given bound. For unconditional results, we may use the Minkowski bound of $(2/\pi)\sqrt{\Delta}$, though we incur a steep computational cost as this bound grows exponentially. If we are willing to

assume the generalized Riemann Hypothesis (GRH), then we may use Bach's bound [3] $6\log^2\Delta$.

Once we know that FB generates $\mathcal{C}l_{\mathbb{K}}$, we need to compute a set of random relations

$$L' = \{(\vec{v}_1, \log_2|\gamma_1|), (\vec{v}_2, \log_2|\gamma_2|), (\vec{v}_3, \log_2|\gamma_3|), \ldots, (\vec{v}_n, \log_2|\gamma_n|)\}$$

such that L' generates Λ' and not just a sublattice of Λ'. The most obvious method to verify this is to check if $\det(L') = h\mathcal{R}$, however this is not a computationally efficient method. By the work of several authors [48, pp. 170–1], it is known that the problem of integer factorization can be reduced to that computing the class number h of a quadratic field; in terms of complexity theory, integer factorization is known to be an NP problem and strongly suspected to not be in P—that is, solutions to an instance of the problem can be verified in polynomial time, but there is no known polynomial time algorithm to determine solutions. As such, computing h appears to be at least as hard as factoring. Instead, as in Lenstra's algorithm from Section 3.4, we will compute an approximation $\overline{h}$ of $h\mathcal{R}$ via an approximation of $L(1,\chi)$ such that the only multiple of $h\mathcal{R}$ in the open interval $(\overline{h}, 2\overline{h})$ is $h\mathcal{R}$ itself. Thus, once $\overline{h} < \det(L') < 2\overline{h}$ is satisfied, we know $\det(L') = h\mathcal{R}$ and hence that L' generates Λ'.

Once we know that L' generates the extended relation lattice, we form the *relation matrix* $M_L = [\vec{v}_1^{\,\mathsf{T}}, \vec{v}_2^{\,\mathsf{T}}, \ldots, \vec{v}_n^{\,\mathsf{T}}]$ and compute its Hermite normal form $\mathrm{HNF}(M_L) = [0 \mid H]$. An invertible square matrix $M = [m_{ij}]$ is in *Hermite normal form* if M is upper triangular, its diagonal entries m_{ii} are positive, and in a given row i the entries to the right of the diagonal satisfy $0 \leq m_{ij} < m_{ii}$ $(j > i)$. Since $\mathrm{HNF}(M_L)$ is upper-triangular, we can easily compute $\det(L) = \det(H)$ to find h by simply taking the product of the

diagonal entries of H. At this point, we also need to compute $\det(L')$, for which we need to compute the multiple of $\mathcal{R}$ corresponding to the real part of L'. To do this, we compute a basis $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$ of the null space of M_L, then we have $\det(L') = \det(L)\mathcal{R}'$ where $\mathcal{R}' = \text{rgcd}(\vec{r}\cdot\vec{x}_1, \vec{r}\cdot\vec{x}_2, \ldots, \vec{r}\cdot\vec{x}_n) = m\mathcal{R}$ and $\vec{r} = (\log_2|\gamma_1|, \log_2|\gamma_2|, \ldots, \log_2|\gamma_n|)$. The function rgcd represents the *real gcd* of two integer multiples of the same real number; in other words, for $x, y \in \mathbb{Z}$ and $R \in \mathbb{R}$, $\text{rgcd}(xR, yR) = \gcd(x, y)R$. Maurer [61] has described one method for calculating rgcd which uses the continued fraction expansion of $xR/yR = x/y$. This results in integers a and b such that $a(xR)+b(yR) = (ax+by)R = \gcd(x, y)R$. The corresponding algorithm is presented in Algorithm 3.11.

Algorithm 3.11: Regulator of a real quadratic number field (index-calculus)

Input: Discriminant $\Delta > 0$.
Output: An approximation of the base-2 regulator $\mathcal{R}$.

1: Compute a factor base $\text{FB} = \{\mathfrak{p}_1, \mathfrak{p}_2, \ldots, \mathfrak{p}_k\}$ of non-inert prime ideals such that $N(\mathfrak{p}_i) < B$ for some bound B and such that each $\mathfrak{p}_i$ divides a unique rational prime.
2: Compute an approximation $\overline{h}$ of $h\mathcal{R}$ such that $\overline{h} < h\mathcal{R} < 2\overline{h}$ and set $n = 0$.
3: **do**
  4: Increase n and compute $n > k$ random extended relations $(\vec{v}_i, \log_2|\gamma_i|)$.
  5: Set $M_L = [\vec{v}_i^{\,\mathsf{T}}]$, compute $[0 \mid H] = \text{HNF}(M_L)$, and set $h' \leftarrow \det(H)$.
  6: Compute a basis $\{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_{n-k}\}$ of $\text{null}(M_L)$.
  7: Set $\vec{r} = (\log_2|\gamma_1|, \log_2|\gamma_2|, \ldots, \log_2|\gamma_n|)$ and compute

$$\mathcal{R}' = \text{rgcd}(\vec{r}\cdot\vec{x}_1,\ \vec{r}\cdot\vec{x}_2,\ \ldots,\ \vec{r}\cdot\vec{x}_{n-k})\,.$$

8: **while** $h' = 0$ or $h'\mathcal{R}' > 2\overline{h}$.
9: **return** $\mathcal{R} = \mathcal{R}'$ and $h = h'$.

The steps in the previous algorithm are deliberately vague. Although we will not discuss them here, there have been a number of papers—see [10, 35] for the original algorithms and, for instance, the works cited in [48, Endnote 9, p. 350] and the those in [83, 84]—dealing with the implementation complexities of this algorithm, in both the

real and imaginary cases. For instance, in Step 4 we need to generate random extended relations. However we choose to do this, we need to verify that we are randomly sampling from the entire lattice. Moreover, as any particular ideal equivalence class contains multiple reduced ideals, once we have a relation, we must ensure that we are then randomly sampling from the reduced ideals in the resulting equivalence class. Because of these and other complexities, we have elected not to implement this algorithm in our library, but instead chose to interface with other preexisting implementations. These other implementations are available through `mpz_qf_compute_regulator(..., ALG_INDEXCALC)`.[1]

In order to discuss the run-time of this index-calculus algorithm, we require the following definition.

Definition 3.12. *For $a, b \in \mathbb{R}$ and $0 \leq a \leq 1$, define*

$$L_\Delta[a,b] := \exp\left(b(\log|\Delta|)^a(\log\log|\Delta|)^{1-a}\right) .$$

This function $L_\Delta[a,b]$ allows us to more accurately describe the range between a polynomial algorithm and an exponential one. If we let $a = 0$, then $L_\Delta[0,b] = (\log|\Delta|)^b$ is polynomial in $\log|\Delta|$. On the other hand, if $a = 1$, we see that $L_\Delta[1,b]$ is exponential in $\log|\Delta|$: $L_\Delta[1,b] = |\Delta|^b$. For the other values of a ($0 < a < 1$), we find that $L_\Delta[a,b]$ is asymptotically between a polynomial function and an exponential function; we refer to this as a *subexponential* function. The index-calculus algorithm described above is a *subexponential algorithm*, an algorithm whose running time is asymptotic to a subexponential function.

Theorem 3.13. *Assuming the truth of the ERH and GRH, Algorithm 3.11 will execute in*

[1] Although we interface against several libraries implementing the index-calculus algorithm, there is currently no way of selecting a particular implementation at run-time; one must be chosen at compile-time.

expected time $L_\Delta[1/2, \sqrt{2} + o(1)]$ *for all* $\Delta > 41$.

Proof. The lower bound on Δ ensures that the factor base bound—that is $\lceil L_\Delta[1/2, \sqrt{2} + o(1)]\rceil$—is large enough so that the ideal classes of the factor base primes generate the class group. See [14, Thm. 11.5.30, p. 267] for the remainder of the proof. □

At this point, we also need to highlight that the correctness of the output of Algorithm 3.11 is dependent on the ERH and GRH. Without the assumption of these hypotheses, the best we can say is that the value of the regulator returned is close to an integer multiple of the true regulator. The value of the class number returned will be a divisor of the true class number or, if the interval $(\overline{h}, 2\overline{h})$ is determined incorrectly, a multiple of it.

3.6. A Verification Algorithm

The next development in the line of regulator-computing algorithms for real quadratic fields is a verification algorithm presented by Michael J. Jacobson, Jr., Ákos Pintér and Gary Walsh [44]. This algorithm was then adapted to work with (f, p) representations by Robbert de Haan, Michael Jacobson, Jr., and Hugh Williams [21, 22], including a more detailed optimization of the algorithm's parameters. As we mentioned in the previous section, if we drop the GRH assumption, then the most we can say about the outputs are that $\mathcal{R}$ is an integer multiple of the true regulator and h is a divisor of the true class number. As the name implies, the idea of the verification algorithm is to use the index-calculus algorithm to generate a multiple of the regulator, which is then verified unconditionally.

We begin by assuming that we are given a multiple of the regulator $\mathcal{R}'$ that is close to

an integer multiple of the actual regulator $\mathcal{R}$. More specifically, we assume

$$|\mathcal{R}' - c\mathcal{R}| < 1 \tag{3.6}$$

for some $c \in \mathbb{Z}^+$. Our goal is to confirm this assumption and then to show that $c = 1$.

To verify that $\mathcal{R}'$ is close to an integer multiple of the regulator, we use AX($\mathcal{R}'$) to compute an (f, p) representation $(\mathfrak{a}_i, d_i, k_i)$ of $\mathfrak{a}_1$ $(= (1))$ and check if

$$\mathfrak{a}_1 \in \{\mathfrak{a}_i, \mathfrak{a}_{i+1}, \ldots, \mathfrak{a}_{i+8}\}\,. \tag{3.7}$$

If $\mathcal{R}'$ satisfies (3.6), then by Theorem 11.12 of [48, p. 282] we see that (3.7) must hold, however the converse need not be true. To work around this, we note that we can refine $\mathcal{R}'$ so that (3.6) does hold. By applying FIND to $(\mathfrak{a}_i, d_i, k_i)$, we compute an $(f + 1/4, p)$ representation $(\mathfrak{a}_j, d_j, k_j)$ of $\mathfrak{a}_1$ and by replacing $\mathcal{R}'$ by $k_j - p + y/2$, where $2^y < d_j^2 \leq 2^{y+1}$, we now have a value for $\mathcal{R}'$ that satisfies (3.6) [48, §15.1, p. 388]. Assuming (3.7) holds, we let $j \in \{i, i+1, \ldots, i+8\}$ be such that $\mathfrak{a}_j = \mathfrak{a}_1$.

The final part of the algorithm is to verify that $c = 1$, which we split into two steps. This splitting is identical to that done in Lenstra's algorithm when we determined h^*; see Section 3.4.3, page 59. The first step is to verify a precomputed lower bound K for $\mathcal{R}$, selected to minimize the run-time of the overall algorithm, which we accomplish by way of a baby-step / giant-step method. The second is to show $c = 1$ by determining its prime power factorization by iterating over all of the possible prime divisors. The rationale for verifying a lower bound on $\mathcal{R}$ is the same as before: we need it to ensure an optimum balance between the baby-step / giant-step search and the following trial division phase. As also mentioned in the Section 3.4.3, it also prevents finding a prime q with $\mathfrak{a}[\mathcal{R}'/q]$

close to (1), even though $q \nmid c$.

Let $\lambda = \lceil (1/2)\log_2 \Delta \rceil + 1$. We begin by precomputing a lower bound K for the regulator, again selected to minimize the overall run-time, and selecting values s and Q such that $2Qs \geq K$, selected to balance the run-times for the baby-step and giant-step phases. Next, we take $\mathfrak{a}_1 = (1)$ and compute a set of baby-steps $\mathcal{L} = \{\mathfrak{a}_1, \mathfrak{a}_2, \mathfrak{a}_3, \ldots, \mathfrak{a}_t\}$ where $\mathfrak{a}_t = \mathrm{AX}(s + \lambda)$. We then proceed to compute a series of Q giant steps $\mathfrak{a}[2s]$, $\mathfrak{a}[4s]$, $\mathfrak{a}[6s]$, …, $\mathfrak{a}[2Qs]$, at each step checking if $\mathfrak{a}[2qs]$ or $\overline{\mathfrak{a}}[2qs] \in \mathcal{L}$. If so, then we can use the techniques covered previously in Sections 3.3 and 3.4 to quickly determine $\mathcal{R}$. If not, then we must have $\mathcal{R} > 2Qs \geq K$ for any $c \in \mathbb{Z}^+$ which gives us the lower bound on the regulator we require.

For the second step, we try to confirm that $c = 1$ by determining its prime power factorization. Let $\mathcal{S} = \{\overline{\mathfrak{a}}_4, \overline{\mathfrak{a}}_3, \overline{\mathfrak{a}}_2, \mathfrak{a}_1, \mathfrak{a}_2, \mathfrak{a}_3, \mathfrak{a}_4\}$. We can show that if $\mathfrak{a}[\mathcal{R}'/q] \notin \mathcal{S}$, then we must have $q \nmid c$. However, if $\mathfrak{a}[\mathcal{R}'/q] \in \mathcal{S}$, we cannot immediately say that $q \mid c$. As before, it may be the case that if q is sufficiently large—$q \approx \mathcal{R}$—, then $\mathcal{R}'/q$ may be small enough that $\mathfrak{a}[\mathcal{R}'/q] \in \mathcal{S}$ even though $q \nmid c$. To prevent this from happening, we can use Theorem 15.6 of [48] to show that we can verify $c = 1$ by verifying that $\mathfrak{a}[\mathcal{R}'/q] \notin \mathcal{S}$ for all primes $q < \mathcal{R}'/K + 1$, assuming $\mathcal{R}$ is sufficiently large (see [48, p. 393] for details). For a given prime q, once we find that $q \mid c$, we determine the greatest power of q dividing c by computing the series of ideals $\mathfrak{a}[\mathcal{R}'/q^i]$ until we find $\mathfrak{a}[\mathcal{R}'/q^{i+1}]$ is not equal to (1). At that point, we know that $q^i \,||\, c$. We can also make use of the improvement to computing the sequence of $\mathfrak{a}[\mathcal{R}'/q_i]$ described in the paragraphs following Algorithm 3.7 on page 62, though here we use (f, p) representations to maintain accuracy. The corresponding algorithm is presented in Algorithm 3.14.

Algorithm 3.14: Regulator of a real quadratic number field (verification algorithm)

Precomputation: Select $K \approx (R')^{2/3}$ such that the total run-time of the algorithm is minimized. Select s, Q such that $2Qs \geq K$ and the run-times of the baby-step and giant-step phases are balanced.

Input: Discriminant $\Delta > 0$, bound $K > 0$, and integers s, Q.

Output: An approximation of the base-2 regulator $\mathcal{R}$.

/ Compute a value for $\mathcal{R}'$ */*

Use the index-calculus algorithm to determine an approximation $\mathcal{R}'$ of the regulator.

/ Verify that $\mathcal{R}'$ is close to an integer multiple of $\mathcal{R}$ */*

Compute an (f, p) representation $(\mathfrak{a}_i, d_i, k_i)$ of $\mathfrak{a}_1$ with distance $\mathcal{R}'$ via the algorithm AX.

Set $j = i$.

while $\mathfrak{a}_j \neq \mathfrak{a}_1$ **do**

  Set $\mathfrak{a}_j \leftarrow \rho(\mathfrak{a}_j)$ and $j \leftarrow j + 1$.

  if $j = i + 9$ **then**

    return "$\mathcal{R}'$ not close enough to a multiple of the regulator."

  end if

end while

/ Refine the approximation $\mathcal{R}'$ */*

Apply the algorithm FIND to the (f, p) representation $(\mathfrak{a}_i, d_i, k_i)$ to compute an (f, p) representation $(\mathfrak{a}_j, d_j, k_j)$ of $\mathfrak{a}_1$.

Set $\mathcal{R}' \leftarrow k_j - p + y/2$ where y is such that $2^y < d_j^2 \leq 2^{y+1}$.

/ Verify that $\mathcal{R} > 2Qs$ */*

Set $\lambda = \lceil (1/2)\log_2 \Delta \rceil + 1$, $\mathfrak{a}_1 = (1)$, $\mathfrak{a}_i = \mathfrak{a}_1$, $\mathcal{L} = \{\mathfrak{a}_1\}$, $\mathfrak{a}_t = \mathrm{AX}[s + \lambda]$, and $i = 1$.

while $\mathfrak{a}_i \neq \mathfrak{a}_t$ **do**

  Compute $\mathfrak{a}_{i+1} = \rho(\mathfrak{a}_i)$, set $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathfrak{a}_{i+1}\}$, and set $i \leftarrow i + 1$.

end while

for $q = 1, 2, \ldots, Q$ **do**

  Compute the ideal $\mathfrak{a}[2qs]$ via the algorithm AX.

  if $\mathfrak{a}[2qs] \in \mathcal{L}$ **then**

    return $\mathcal{R} = \delta(\mathfrak{a}[2qs]) - \delta(\mathfrak{a}_k)$, where $\mathfrak{a}[2qs] = \mathfrak{a}_k \in \mathcal{L}$.

  end if

  if $\overline{\mathfrak{a}}[2qs] \in \mathcal{L}$ **then**

    return $\mathcal{R}' = \delta(\mathfrak{a}[2s]) - \big(\delta(\overline{\mathfrak{a}}[2qs]), \delta(\overline{\mathfrak{a}}[2s]) - \delta(\mathfrak{a}_k)\big) - \log_2(\widehat{Q}/r)$, where $\overline{\mathfrak{a}}[2qs] = \mathfrak{a}_k \in \mathcal{L}$ and $\mathfrak{a}[2s] = (\widehat{Q}, \widehat{P})$.

  end if

end for

/ Determine the factorization of c */*

25: Set $c = 1$, q to be the largest prime less than $\mathcal{R}'/(2Qs)+1$, $\widetilde{\mathfrak{a}} = (1)$, $\gamma = 0$, and compute $\mathcal{S} = \{\overline{\mathfrak{a}}_4, \overline{\mathfrak{a}}_3, \overline{\mathfrak{a}}_2, \mathfrak{a}_1, \mathfrak{a}_2, \mathfrak{a}_3, \mathfrak{a}_4\}$.
26: Precompute the ideals $\mathfrak{a}[\delta_{t_0}], \mathfrak{a}[\delta_{t_0}], \ldots, \mathfrak{a}[\delta_{t_m}]$, where $\delta_{t_i} = 2^i(s-1)$.
27: **while** $q \geq 2$ **do**
28: Set $\delta \leftarrow \lceil \mathcal{R}'/q \rceil - \gamma$, $\widehat{\mathfrak{a}} = (1)$, and compute the binary expansion of

$$\left\lfloor \frac{\delta}{s-1} \right\rfloor + 1 = \sum_{k=0}^{r} b_k 2^k \, .$$

29: **for each** $b_i = 1$ $(i = 0, 1, \ldots, r)$ **do**
30: Compute $\widehat{\mathfrak{a}} \leftarrow \widehat{\mathfrak{a}} \star \mathfrak{a}[\delta_{t_i}]$ via the algorithm ADDXY.
31: **end for each**
32: Compute $\mathfrak{a}[\mathcal{R}'/q] = \widetilde{\mathfrak{a}} \star \widehat{\mathfrak{a}}$ via the algorithms ADDXY and WNEAR.
33: Set $\widetilde{\mathfrak{a}} \leftarrow \mathfrak{a}[\mathcal{R}'/q]$ and $\gamma \leftarrow \delta(\widetilde{\mathfrak{a}})$.
34: **if** $\mathfrak{a}[\mathcal{R}'/q] \in \mathcal{S}$ **then**
35: Set $e = 1$.
36: **do**
37: Set $e \leftarrow e+1$ and compute $\mathfrak{a}[\mathcal{R}'/q^e]$.
38: **while** $\mathfrak{a}[\mathcal{R}'/q^e] \in \mathcal{L}$
39: Set $c \leftarrow cq^{e-1}$.
40: **end if**
41: Set q to be the largest prime less than q, or set $q \leftarrow 1$ if $q = 2$.
42: **end while**
43: **return** $\mathcal{R} = \mathcal{R}'/c$.

This algorithm is implemented in our library as `mpz_qf_compute_regulator(..., ALG_VERIFY)`.

In order to maximize the efficiency of this algorithm, we must ensure that the number of operations required for its two parts are balanced. To verify the approximation $\mathcal{R}'$, we compute $t+2$ baby steps (stored in $\mathcal{L}$) and since $\mathfrak{a}_t = \mathfrak{a}[s+\lambda]$, we have $t \approx (s+\lambda)/1.7$. In other words, computing $\mathcal{L}$ requires $O(s)$ elementary operations. In order to show that $\mathcal{R} > K$, we compute the series of giant steps $\mathfrak{a}[2qs]$ $(q = 1, 2, \ldots, Q)$. We require one ADDXY invocation for each q and, recalling that Q and s satisfy $2Qs \geq K$, this gives us a cost of $O(s) + O(Q \log \Delta)$ elementary operations. Combining these two observations

by noting that we need $O(s) = O(Q) = O(\sqrt{K})$ in order to achieve an optimal run-time complexity for the baby-step and giant-step phases, we arrive at a cost of

$$\mathrm{O}(K^{1/2} + K^{1/2+\epsilon}) = \mathrm{O}(K^{1/2+\epsilon}) .$$

On the other hand, to verify that $c = 1$ we must trial divide by a range of possible prime divisors. The prime number theorem states that if $\pi(x)$ is the prime-counting function, then asymptotically $\pi(x) \sim x/\log(x)$. Using this as a guide, we would expect to have approximately $(\mathcal{R}'/K+1)/\log(\mathcal{R}'/K+1)$ primes less than $\mathcal{R}'/K+1$. So, this verification requires roughly $(\mathcal{R}'/K+1)/\log(\mathcal{R}'/K+1)$ invocations of AX at a cost of $\mathrm{O}(\log\mathcal{R}'\log\Delta)$ elementary operations each. This gives a cost of

$$\mathrm{O}\left(\left(\frac{\mathcal{R}'/K+1}{\log(\mathcal{R}'/K+1)}\right)\log\mathcal{R}'\log\Delta\right) = \mathrm{O}((\mathcal{R}'/K)^{1+\epsilon}) .$$

elementary operations. Thus, to balance the two parts of the verification algorithm, we need $K^{1/2} \approx \mathcal{R}'/K$ or equivalently $K \approx (R')^{2/3}$. With this value of K, the overall run-time is $\mathrm{O}((R')^{1/3+\epsilon}) = \mathrm{O}(\Delta^{1/6+\epsilon})$, assuming that $\mathcal{R}$ is close to $\mathcal{R}'$. As is mentioned by Jacobson and Williams [48, p. 394], we have an explicit upper bound on $\mathcal{R}$ from the analytic class number formula and the upper bound on $L(1,\chi)$ of $(1/2)\log|\Delta|+1$, so we would not execute the algorithm if our approximation $\mathcal{R}'$ exceeds this regulator bound by more than 1. Moreover, $\mathcal{R}$ must be sufficiently large in order for the second stage of this algorithm to give a correct answer. We must guarantee that when we are testing c for divisibility by a prime q, $\mathcal{R}$ is large enough as that we actually have $\mathfrak{a}[R'/q] \in \mathcal{S}$ because $q \mid c$ and not

because $R'/q \approx 1$. R is sufficiently large when [48, p. 394]

$$R > M\left(2\log_2 \Delta + \log\frac{17}{4}\right), \tag{3.8}$$

which will happen if K satisfies this lower bound (see also [48, Thm. 15.6, p. 392]). We determined previously that we need $K \approx (R')^{2/3}$ to balance the algorithm, so we also have $M = (R')^{1/3} + 1$. (3.8) will be satisfied for $\mathcal{R}' > 216(\log\Delta)^3$ [48, p. 394] and if this is not the case, $\mathcal{R}$ is small enough to be quickly verified by the continued fraction algorithm given in Section 3.2.

Theorem 3.15. *Assuming the ERH and GRH, Algorithm 3.14 executes in expected time* $O(\Delta^{1/6+\epsilon})$. *The output of Algorithm 3.14 is unconditionally correct.*

Proof. See [48, Thm. 15.7, p. 394]. □

3.6.1. Implementation Concerns. There are a number of implementation concerns for this verification algorithm, most of which have been discussed in other works such as [48, §15.3, pp. 397–399] and [21, Ch. 6].

As mentioned in the previous material in this section, we need to store a list of baby-steps $\mathcal{L}$. In practice, this list is actually stored as a hash table to allow for fast lookups, so searching for ideals in $\mathcal{L}$ is not an expensive operation. However, we have only a limited amount of storage space—in the form of RAM—available for it. If the size of $\mathcal{L}$ exceeds this limit, we are forced to swap portions of $\mathcal{L}$ out to a hard-disk cache which severely impacts the implementation's performance. We could restrict ourselves to a shorter baby-step list guaranteed to fit in RAM, however this would lead to an imbalance in work loads between the baby-step and giant-step phases of the algorithm. de Haan mentioned

[21, 22] another work-around for this problem, namely that we can get away with only storing a partial baby-step list

$$\mathcal{L}' = \left\{ \mathfrak{a}_1, \mathfrak{a}_l, \mathfrak{a}_{2l}, \ldots, \mathfrak{a}_{ml} \right\} \cup \left\{ \mathfrak{a}_t, \mathfrak{a}_{t+1}, \mathfrak{a}_{t+2} \right\},$$

with m such that $ml \leq t < (m+1)l$, and rather than checking if the giant step $\mathfrak{a} \in \mathcal{L}$, we check if $\rho^n(\mathfrak{a}_j) \in \mathcal{L}'$ for some $n = 1, 2, \ldots, l$. On a practical implementation note, the baby-step list is stored in a hash table and the extra lookups are not computationally expensive. In essence, we are trading a slight increase in computational overhead—ρ is a relatively efficient computation—for lower storage requirements. This change is implemented in our library as `mpz_qf_compute_regulator(..., ALG_VERIFY_PARTIAL_L)`.

A second concern is the selection of the particular values for the s, Q, and K parameters. We will hold off describing how these values are computed for the time being, however we should point out that they are highly implementation and machine dependent. The multi-precision arithmetic library chosen, compiler optimization flags selected, any CPU architecture differences, for example, have a great impact on the values of these parameters. Our implementation differs from de Haan's implementation—detailed in Section 4.3.2 and Appendix A—and as such, we were not able to reuse the parameter values he specified in [21].

As the $O(\Delta^{1/6+\epsilon})$ algorithm makes heavy use of (f, p) representations, we must determine a suitable precision p to ensure that the intermediate computations and results are numerically correct. de Haan has presented a detailed analysis of the precisions required for each component of the verification algorithm. Rather than repeating that work here, we summarize it below and direct the interested reader to [21, §6.3, pp. 73–83] for de-

tails of the analysis. For the regulator approximation refinement stage—Steps 2–11 of Algorithm 3.14—we required p such that $2^p > 21\mathcal{R}'\log_2\mathcal{R}'$, assuming $\mathcal{R}' > 10^6$. The precision p is largely irrelevant for the baby-step computation phase, save for when we compute the end-of-list bound $\mathfrak{a}[s+\lambda]$ in Step 12. We are only storing the coefficients of the baby-step ideals and stopping when we find an ideal matching $\mathfrak{a}[s+\lambda]$. Neither of these require us to compute (f,p) representations of each baby-step ideal, so we can use the version of ρ which operates on $\mathcal{O}_{\mathbb{K}}$-ideals and not the (f,p) representation version. Moreover, as de Haan points out, a larger precision is required for computing the initial giant step $\mathfrak{a}[2s]$ than is required for $\mathfrak{a}[s+\lambda]$. As such, we can ignore this precision calculation. For the giant-step phase, we refer to the following result for determining the necessary precision.

Theorem 3.16 ([21, Thm. 6.4, p. 74])**.** *Assume that $s > 16$, $Q > \max\{16, \log_2 s\}$ and that $(\mathfrak{b}_0, d_0, k_0) = \mathfrak{a}[2s]$, which is an (f_0, p) representation of $\mathfrak{a}_1$ such that $f_0 < 49s\sqrt{e}/2$ where $e \approx 2.71828\ldots$ is Euler's constant. Then, if $2^p \geq 221Q^2s$, we have an (f_{Q-1}, p) representation $(\mathfrak{b}_{Q-1}, d_{Q-1}, k_{Q-1})$ of $\mathfrak{a}_1$ at the end of [Step 24 of Algorithm 3.14], where $\mathfrak{b}_{Q-1} = \mathfrak{a}[2sQ]$ and $f_{Q-1} < 2^{p-4}$.*

For the final multiplier-finding phase, we can use another result from de Haan's work.

Theorem 3.17 ([21, Thm. 6.8, p. 80])**.** *Assume that $\mathcal{R}' > 10^6$, $(\mathcal{R}')^{1/2} < K < (\mathcal{R}')^{5/6}$ and $\max\{16, K^{2/5}\} < s < K^{3/5}$. If $2^p > 19\mathcal{R}'\max\{93, \log_2\mathcal{R}'\}$ then all (f,p) representations that are computed during [Steps 25–42 of Algorithm 3.14] satisfy $f < 2^{p-4}$.*

3.6.2. Parallelization. As the computations involved in Algorithm 3.14 are very time consuming for larger discriminants, an obvious question to ask is: can all or part of the

algorithm be parallelized?

We adopt the following standard terminology for the remainder of this thesis. A *compute cluster* is a collection of processors and memory, connected by a combination of memory buses and high-speed network interconnects. The cluster is divided into a number of *compute nodes*, or simply *nodes*, each having an equal number of processors and amount of memory. The nodes are connected by the high-speed network. Within each node, all the processors have shared access to all of the node's memory via a memory bus. We will refer to a machine or node with one single-core processor as a *single processor* machine. Machines or nodes with multiple single-core processors, a multi-core processor, or multiple multi-core processors will be called *multi-processor* machines.

With this terminology in mind, we can discuss two techniques for parallel communication which can be used in conjunction. *Message passing* involves creating separate processes on each node which then communicate with each other by literally passing messages back-and-forth. Generally, one process will control the division of tasks and handle collecting the final results after computations are finished. *Threading* is used to make a process on a single multi-processor machine or node run in parallel. The process is divided into a number of light-weight processes called *threads* which are inexpensive to create and share all the memory allocated to the parent process. However, care must be taken to ensure that data produced by one thread is not accidentally overwritten by another thread and therefore corrupted.

As the authors of [22] point out, the multiplier-finding phase is quite straight-forward to parallelize: each processor in the cluster works on a different interval of primes. de Haan found [21, §6.4.2, p. 85] that the best result comes from a heuristic that splits the range of possible prime divisors into three parts in a 1:5:6 distribution. That is, the first

part contains $1/12^{\text{th}}$ of the range and is comprised of the smallest primes, and so on. One-third of the number of processors are assigned to each part, and then the parts are further subdivided using the prime number theorem.

The baby-step and giant-step phases, however, are more difficult to parallelize. When using message passing to parallelize Algorithm 3.14 on a cluster, we have to keep in mind that each node needs access to the entire baby-step list during the giant-step phase. This means that each node must either compute a complete copy of $\mathcal{L}$ for itself, or must share its portion of $\mathcal{L}$ with every other node. As the communication overhead in the latter case would be prohibitively expensive, the authors of [21, 22] chose to use the former method. Unfortunately, this limits the size of baby-step list we can store to the amount of memory available on an individual node, rather than that available to the whole cluster.

For the baby-step phase on a cluster with multi-processor nodes, we can use threading to divide up the interval of $s + \lambda$ baby steps into equally sized intervals and so parallelize the computation. Each processor will also have to compute two appropriately sized giant-steps to determine the starting and ending ideals for its interval. The giant-steps can be parallelized in a similar fashion. As each node has a complete copy of the baby-step list, each processor can be given a range of giant steps to compute. In this phase, however, we only need to compute one additional giant-step to determine the appropriate starting ideal.

Theorem 3.18. *Using the parallelized version of the* $O(\Delta^{1/6+\epsilon})$ *regulator verification algorithm outlined above on a cluster consisting of* n *nodes, each with* r *shared-memory processors, will result in an expected speed-up of a factor* $n^{2/3}r$.

Proof. See [21, §6.5.6. pp. 92–93]. □

— Chapter 4 —
New Developments: The $O(\Delta^{1/6+\epsilon})$ verification algorithm

4.1. Introduction

Saving given an outline in the previous chapter of the current state of regulator computations in real quadratic numbers fields, we can now move on to the real heart of this thesis: the presentation of our modifications to the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm and to compact representations of algebraic integers. In this chapter, we focus on the former topic and our improvements to the storage requirements and, to a lesser extent, the run-time of certain algorithms from Chapter 3. This chapter will deal exclusively with improvements to the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm. In Section 4.2 we propose and discuss a method for reducing the memory requirements of the $O(\Delta^{1/6+\epsilon})$ algorithm by storing a list of hashes of the baby-step ideals in $\mathcal{L}$, rather than the ideals themselves or a subset of them. Section 4.3 deals with practical implementation concerns, in particular how we select an appropriate hash function and how we optimize our choice of parameters for the algorithm. Finally, in Section 4.4, we present some numerical evidence showing the practical improvements these modifications have allowed us to make.

4.2. Reducing Memory Usage in the $O(\Delta^{1/6+\epsilon})$ Algorithm

As was discussed in Section 3.6.1, for the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm, we need to store a list of baby-steps $\mathcal{L}$ in a limited amount of space. The length of this list

increases as the discriminant increases, and so for a large enough discriminant, we will run out of memory. We saw one way to squeeze a larger baby-step list into this space was by storing only every l^{th} ideal in the partial baby-step list $\mathcal{L}'$. In this section, we will describe our further improvement to this idea: storing a hash of the coefficients of each ideal.

Definition 4.1. *A* hash function H *is a function that takes as input a* message m *of arbitrary length and outputs a* digest $d = H(m)$ *of fixed length.*

Whichever hash function we choose to apply to the ideals in $\mathcal{L}$ needs to have some common properties, such as being

- efficient: given a message m, the digest $H(m)$ can be computed very quickly;
- deterministic: given a message m, the function H always outputs the same digest $H(m)$; and
- uniformly distributed: considering the set of expected messages, each digest should be generated with the same probability.

Later, in Section 4.3, we will look in more depth at the particular choice of hash function our implementation uses.

Let $H(\mathfrak{a}_i)$ be the *hash* of the ideal $\mathfrak{a}_i = [Q_{i-1}, P_{i-1}]$, which we store in a *hashed baby-step list*

$$\mathcal{L}'' = \left\{H(\mathfrak{a}_1), H(\mathfrak{a}_l), H(\mathfrak{a}_{2l}), \ldots, H(\mathfrak{a}_{Nl})\right\} \cup \left\{H(\mathfrak{a}_t), H(\mathfrak{a}_{t+1}), H(\mathfrak{a}_{t+2})\right\},$$

with N such that $Nl \leq t < (N+1)l$. In addition to this list, we store the two ideals $\mathfrak{a}_2 = [Q_1, P_1]$ and $\mathfrak{a}_{(N+1)l} = [Q_{(N+1)l-1}, P_{(N+1)l-1}]$; the reason for this will be explained later. The key observation behind this idea is that in reality we do not expect to find

a giant step—or its conjugate for that matter, but we will ignore this ideal for the time being—in the baby-step list. This allows us to replace the check for $\rho^i(\mathfrak{a}_j) \in \mathcal{L}'$ for some $i = 0, 1, \ldots, l$ with a check for $H(\mathfrak{a}_{j-1+i}) \in \mathcal{L}''$ for $i = 0, 1, 2, \ldots, l$. If $H(\mathfrak{a}_{j-1+i}) \notin \mathcal{L}''$, then clearly we cannot have $\rho^i(\mathfrak{a}_j) \in \mathcal{L}'$. However, what if we do find $H(\mathfrak{a}_{j-1+i}) \in \mathcal{L}''$? We cannot immediately say that $\rho^i(\mathfrak{a}_j) \in \mathcal{L}'$ as we may have the unfortunate luck of finding a random hash collision in our hash function H. How we differentiate between these two is the topic of the remainder of this section.

4.2.1. Resolving Hash Collisions. Before being able to discuss how we distinguish between a random hash collision and having found an actual giant step in the baby-step list, we need to present a series of lemmas. The first lemma tells us how the coefficients of two equal $\mathcal{O}_{\mathbb{K}}$-ideals compare to each other. Lemma 4.2(a) deals with the case of two reduced $\mathcal{O}_{\mathbb{K}}$-ideals, whereas 4.2(b) deals with a reduced $\mathcal{O}_{\mathbb{K}}$-ideal and the conjugate of a reduced $\mathcal{O}_{\mathbb{K}}$-ideal. These comparisons are key when we are testing if the ideal which results from a giant step is in the list of baby steps.

Lemma 4.2. *Let* $\mathfrak{a}_{i+1} = [Q_i, P_i]$ *and* $\mathfrak{a}_{j+1} = [Q_j, P_j]$ *be two reduced* $\mathcal{O}_{\mathbb{K}}$*-ideals. If* $Q_i = Q_j$ *and*

a.) $P_i \equiv P_j \pmod{Q_i}$, *then* $P_i = P_j$.

b.) $P_i \equiv -P_j \pmod{Q_i}$, *then* $P_i = P_{j+1}$.

Proof. Part (a) of this lemma follows from the fact that $\mathfrak{a}_{i+1}$ and $\mathfrak{a}_{j+1}$ are reduced and have a unique reduced basis. Part (b) follows from (a) and the symmetry property of the cycle of reduced principal ideals. □

The next lemma illustrates the connection between the continued fraction expansion of ω and the coefficients of the ideals in cycle of reduced principal ideals. Although we

do not use this lemma directly in our collision resolution process, it is needed to prove Lemma 4.4.

Lemma 4.3 ([87, Lem. 6.1, p. 418])**.** *If, in the continued fraction expansion of $\phi_0 = \phi$, we have $-1 < \overline{\phi}_1 < 0$, then*

$$q_k = \left\lfloor \frac{P_{k+1} + \sqrt{D}}{Q_k} \right\rfloor$$

for all $k \geq 1$.

As we briefly mentioned near the start of this section, we must test if the ideal resulting from a giant step *or its conjugate* is in the baby-step list. From the material in the Chapter 2, we already know how ρ affects an ideal in the cycle of reduced principal ideals. We have yet to discuss how it affects the conjugate of such an ideal, which is the subject of the final lemma.

Lemma 4.4. *If $\mathfrak{a}_j$ and $\mathfrak{a}_k$ are two reduced $\mathcal{O}_{\mathbb{K}}$-ideals and $\overline{\mathfrak{a}}_j = \mathfrak{a}_k$, then $\overline{\mathfrak{a}}_{j+1} = \mathfrak{a}_{k-1}$.*

Proof. Suppose $\overline{\mathfrak{a}}_j = \mathfrak{a}_k$. Considering the norm of each ideal, we see that $N(\overline{\mathfrak{a}}_j) = N(\mathfrak{a}_k)$ and in light of (2.4) (page 20), we have $N(\overline{\mathfrak{a}}_j) = N(\mathfrak{a}_j) = N(\mathfrak{a}_k)$. Thus, $Q_{j-1} = Q_{k-1}$. Now, considering the two ideals as $\mathbb{Z}$-modules, we see

$$\left(\frac{Q_{j-1}}{r}\right)\mathbb{Z} + \left(\frac{P_{j-1} - \sqrt{D}}{r}\right)\mathbb{Z} = \left(\frac{Q_{j-1}}{r}\right)\mathbb{Z} + \left(\frac{P_{k-1} + \sqrt{D}}{r}\right)\mathbb{Z}$$

and so we must have $P_{k-1} \equiv -P_{j-1} \pmod{Q_{j-1}}$. By Lemma 4.2(b), this means $P_{k-1} = P_j$.

Recalling the formulas given for the operation ρ ((2.7)–(2.9), page 29), we see that $D - P_j^2 = Q_j Q_{j-1}$ and $D - P_{k-1}^2 = Q_{k-1}Q_{k-2}$. Since $P_j = P_{k-1}$ and $Q_{j-1} = Q_{k-1}$, we must have $Q_j = Q_{k-2}$. By Lemma 4.3, we get

$$q_{k-2} = \left\lfloor \frac{P_{k-1} + \sqrt{D}}{Q_{k-2}} \right\rfloor = \left\lfloor \frac{P_j + \sqrt{D}}{Q_j} \right\rfloor = q_j \,.$$

And so rearranging the equation $P_{k-1} = q_{k-2}Q_{k-2} - P_{k-2}$, and making the appropriate substitutions, gives $P_{k-2} = q_j Q_j - P_j = P_{j+1}$. Thus, $(Q_j, P_{j+1}) = (Q_{k-2}, P_{k-2})$, or equivalently $\overline{\mathfrak{a}}_{j+1} = \mathfrak{a}_{k-1}$. □

At this point, we can discuss in detail how we can distinguish between a random hash collision and having found a giant step in the baby-step list. Suppose that during the giant step calculations we find an ideal $\mathfrak{a}_j$ or $\overline{\mathfrak{a}}_j$ in the list $\mathcal{L}'$. If $\mathfrak{a}_j \in \mathcal{L}'$, then we have $\mathfrak{a}_j = \mathfrak{a}_{kl}$ for some k such that $1 \leq k \leq N$ and $H(\mathfrak{a}_{j-1}) = H(\mathfrak{a}_{kl-1}) \in \mathcal{L}''$. If $k < N$ and we repeatedly apply ρ^l to $\mathfrak{a}_j$, it is clear that we will continue to find matches in $\mathcal{L}'$. More specifically, we will find

$$\begin{aligned}
\mathfrak{a}_{j+l} = \mathfrak{a}_{kl+l} = \mathfrak{a}_{(k+1)l} \qquad & H(\mathfrak{a}_{(j-1)+l}) = H(\mathfrak{a}_{(k+1)l-1}) \,, \\
\mathfrak{a}_{j+2l} = \mathfrak{a}_{(k+2)l} \qquad & H(\mathfrak{a}_{(j-1)+2l}) = H(\mathfrak{a}_{(k+2)l-1}) \,, \\
\vdots \qquad & \vdots \\
\mathfrak{a}_{j+il} = \mathfrak{a}_{(k+i)l} \qquad & H(\mathfrak{a}_{(j-1)+il}) = H(\mathfrak{a}_{(k+i)l-1}) \,, \\
\vdots \qquad & \vdots
\end{aligned}$$

so long as $k + i \leq N$. However if $k = N$, or $k + i > N$, then since we have stored the ideal $\mathfrak{a}_{(N+1)l}$, we can apply ρ^l and directly verify that $Q_{(j-1)+l} = Q_{(N+1)l-1}$ and $P_{(j-1)+l} = P_{(N+1)l-1}$.

If $\overline{\mathfrak{a}}_j \in \mathcal{L}'$, then we have $\overline{\mathfrak{a}}_j = \mathfrak{a}_{kl}$ for some k such that $1 \leq k \leq N$ and $H(\mathfrak{a}_{j-1}) = H(\mathfrak{a}_{kl-1}) \in \mathcal{L}''$. If $k > 1$, we can repeatedly apply ρ^l to $\mathfrak{a}_j$, we find a series of matches as

before. This time, keeping in mind Lemma 4.4, we find

$$\begin{aligned}
\mathfrak{a}_{j+l} &= \overline{\mathfrak{a}}_{kl-l} = \overline{\mathfrak{a}}_{(k-1)l} & H(\mathfrak{a}_{(j-1)+l}) &= H(\mathfrak{a}_{(k-1)l-1}),\\
\mathfrak{a}_{j+2l} &= \overline{\mathfrak{a}}_{(k-2)l} & H(\mathfrak{a}_{(j-1)+2l}) &= H(\mathfrak{a}_{(k-2)l-1}),\\
&\vdots & &\vdots\\
\mathfrak{a}_{j+il} &= \overline{\mathfrak{a}}_{(k-i)l} & H(\mathfrak{a}_{(j-1)+il}) &= H(\mathfrak{a}_{(k-i)l-1}),\\
&\vdots & &\vdots
\end{aligned}$$

so long as $k-i \geq 1$. If $k=1$, or $k-i<1$, then we have $\overline{\mathfrak{a}}_j = \mathfrak{a}_l$. Let p be the length of the principal ideal cycle; so $\mathfrak{a}_{p+1} = \mathfrak{a}_1$. On applying ρ^l to $\overline{\mathfrak{a}}_j$, we find $\overline{\mathfrak{a}}_{j+l} = \mathfrak{a}_{l-l} = \mathfrak{a}_p$, which implies that $\overline{\mathfrak{a}}_{j+l-1} = \mathfrak{a}_{p+1} = \mathfrak{a}_1 = \overline{\mathfrak{a}}_1$. Thus, we have $\mathfrak{a}_{j+l-1} = \mathfrak{a}_1$, or equivalently $\mathfrak{a}_{j+l} = \mathfrak{a}_2$. Since we have stored $\mathfrak{a}_2$, we can directly verify that $Q_{(j-1)+l} = Q_1$ and $P_{(j-1)+l} = P_1$.

Now as we mentioned previously, we do not expect to find $\mathfrak{a}_j$ or $\overline{\mathfrak{a}}_j \in \mathcal{L}'$. So most of the time, we can get away with asking if $H(\mathfrak{a}_{j-1}) \in \mathcal{L}''$. If $H(\mathfrak{a}_{j-i}) \notin \mathcal{L}''$, then neither $\mathfrak{a}_j$ nor $\overline{\mathfrak{a}}_j$ could be in $\mathcal{L}'$. In the rare case where $H(\mathfrak{a}_{j-1}) \in \mathcal{L}''$, we know from the previous discussion that $H(\mathfrak{a}_{j-1}) = H(\mathfrak{a}_{kl-1})$ for some $1 \leq k \leq N$. We proceed by cases. If $1 < k < N$, and neither $H(\mathfrak{a}_{j+l-1}) = H(\mathfrak{a}_{(k+1)l-1})$ nor $H(\mathfrak{a}_{j+l-1}) = H(\mathfrak{a}_{(k-1)l-1})$, then we know $\mathfrak{a}_j, \overline{\mathfrak{a}}_j \notin \mathcal{L}'$. If $k=N$, and either $Q_{j+l-1} \neq Q_{(N+1)l-1}$ or $P_{j+l-1} \neq P_{(N+1)l-1}$, then we know $\mathfrak{a}_j, \overline{\mathfrak{a}}_j \notin \mathcal{L}'$. If $k=1$, if neither $Q_{j+l-1} = Q_1$ nor $P_{j+l-1} = P_1$, then we know $\mathfrak{a}_j, \overline{\mathfrak{a}}_j \notin \mathcal{L}'$.

However, if $H(\mathfrak{a}_{j-1}) \in \mathcal{L}''$ and we find $H(\mathfrak{a}_{j+l-1}) = H(\mathfrak{a}_{(k\pm1)l-1})$, then we must also check if $H(\mathfrak{a}_{j+2l-1}) = H(\mathfrak{a}_{(k\pm2)l-1})$. We continue this process as long as we keep finding $H(\mathfrak{a}_{j+il-1}) = H(\mathfrak{a}_{(k\pm i)l-1})$ or until we make an explicit check if $\mathfrak{a}_{j+il} = \mathfrak{a}_2$ or $\mathfrak{a}_{(N+1)l}$. As we remarked above, finding $H(\mathfrak{a}_{j-1}) \in \mathcal{L}''$ is a rarity, even more so finding a series of them. Assuming the hash function we select has a uniformly distributed output, we

would expect for an m-bit digest that a random hash collision occurs with probability $1/2^m$. Probabilistically then, we can expect this method will rapidly determine hash collisions caused by simple bad luck. For the incredibly unlikely case where this series of hash comparisons cannot resolve a hash collision—this would occur with a probability of $1/2^{mN/2}$ on average—, the explicit ideal comparison with $\mathfrak{a}_2$ and $\mathfrak{a}_{(N+1)l}$ will certify whether we truly found a match or not.

4.3. Implementation concerns

In this section we will discuss a number of concerns which need to be addressed when implementing the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm and our refinements to it. For the computations that follow, we use the same set of D values that are used in [21]; they are reproduced in Table 4.1, page 86, for easy reference.

4.3.1. Hash function selection. One of the key components of creating the hashed baby-step list $\mathcal{L}''$ is, of course, a hash function. We analysed a number of options for functions to use in our implementation of the hashed baby-step list. The end goal of this effort was to find a hash function that

- is fast,
- has a low collision rate,
- produces hash values that are reasonably uniformly distributed, and
- whose hash values require a minimal number of bits to store.

By collision rate, we mean the percentage of hashes of giant steps that match the hash values stored in $\mathcal{L}''$ purely by chance. Although this bound on this rate is arbitrary, we will assume a collision rate of at most 1% is acceptable. The uniform distribution requirement is important for ensuring that the placement of hashed ideals into the hash

Table 4.1: Discriminants used throughout various calculations in this chapter.

$l(\Delta)$	D
15	124190375333324
20	69632554314051593309
25	4406306728804374025823929
30	738873402423833494183027176953
35	311644929703875673460240189868437 56
40	346109889478629021502039611124664826 4709
45	604021226554971154994549211891921127777412533
50	600526648930464396289639620337617106277732294553 37
55	876937929490527726393225830223773973487581185422921 4677
60	730189467691077755331569166053721301652308776100865143502041
65	392863757345425947497580508351516550921188485308333987435615684 81

table is balanced. To calculate which hash table bucket an ideal hash $H(\mathfrak{a})$ should be stored in, we simply truncate $H(\mathfrak{a})$ to a certain number of bits—the size of our hash table is always a power of two—and use the resulting value as an index to a bucket. If the hash values are uniformly distributed, we should only have a few ideal hashes stored in each bucket. However, if the hash values are clustered around a subset of possible values, we incur a non-negligible lookup cost as we are then forced to sift through many potential matches on every table lookup.

Although there are numerous hash functions we could have implemented and tested, we feel that the functions below give a decent, albeit introductory, overview of how certain classes of hash functions perform in our application. We will look at a basic truncation of bits, a cryptographic hash, reduction modulo a prime, and a generic hash table lookup function. Towards the end of this section we will provide a graphical comparison of the hash functions.

Our initial idea for a “hash function" was to simply take a number of the low-order bits of the P coefficient of $\mathfrak{a} = [Q, P]$. The hash values produced by this method, however, are not uniformly distributed—a chi-squared goodness of fit test showed a $\ll 0.01\%$ probability of the observed results being sampled from a discrete uniform distribution—and so during the giant-step phase of Algorithm 3.14, we observe a number of random collisions, varying of course with the size of $\mathcal{L}''$. For instance, with a 35-decimal digit discriminant and taking the lowest 32 bits of P, we found approximately 425 collisions; a 0.2% collision rate. Of course, for a larger discriminant we need to take additional low-order bits as part of the hash in order to keep the collision rate at an acceptable level. With a 55-digit discriminant, we found roughly 7.6 million collisions (6.45% collision rate). But, in terms of speed, this is clearly the best method: the hash overhead is just a

machine-word sized store of a value already residing in the processor's cache.

The first true hash function we chose to investigate was MD5. MD5 is a cryptographic hash function developed by Ron Rivest [72] that produces a 128-bit (16-byte) hash value from an arbitrary length input. Our implementation of MD5 is actually a slightly modified version of `md5deep` [53]. To compute the hash of $\mathfrak{a} = [Q, P]$, we take the byte-arrays used to store its Q and P coefficients, concatenate them as $Q||P$, and compute MD5$(Q||P)$. We then truncate the resulting 128-bit hash to the desired length. The resulting hash values are quite uniformly distributed, though this is not surprising as MD5 was designed not to exhibit bias. The goodness-of-fit tests showed distributions with a 35%–85% chance of being sampled from a discrete uniform distribution. In addition, MD5 exhibits a nice *avalanche effect*: small changes in an input produce substantial changes in the output hash value. From a computational aspect, we were concerned that a cryptographic hash would be overkill for this application. The basic operations used in MD5 are simple bit-wise logic operations and bit-shifts, which individually are extremely fast. However, the sheer number of these operations which must be performed, especially when weighed against the previously mentioned least-significant-bits hash, could be a hindrance. Is the trade-off between a high operation count versus better collision resistance and avalanche effect properties worthwhile?

The next hashing idea we chose to investigate was a simple remainder calculation using an appropriately-sized prime modulus. In particular, we looked at the Mersenne primes $M_n = 2^n - 1$ because of a very useful reduction algorithm for numbers of this form. The following theorem gives a more general reduction strategy, but clearly setting $c = 1$ gives the desired result for Mersenne primes.

Theorem 4.5 ([20, Thm. 9.2.12, p. 455])**.** *For an integer $N = 2^q \pm c$, where $q, c \in \mathbb{Z}$ with*

$q > 0$, *and for any integer* x, *we have* $x \equiv (x \pmod{2^q}) \mp c\lfloor x/2^q \rfloor \pmod{N}$.

By restricting ourselves to Mersenne primes, we can replace the rather costly remainder operation by a series of very fast left-shift, logical-AND, and subtraction operations since

$$x \pmod{2^q} \mapsto \texttt{x \& } \underbrace{0\cdots0\,1\cdots1}_{q} \quad \text{and} \quad \left\lfloor \frac{x}{2^q} \right\rfloor \mapsto \texttt{x >> q}\,.$$

Of course, there are only at most a couple Mersenne primes in the bit-length range we are interested in—M_{31} and M_{61}—, so to expand our choices, we can turn to *pseudo-Mersenne* primes.

Definition 4.6. *A* pseudo-Mersenne prime *is a prime* p *of the form* $2^q - c$, *where* $\log_2 c \leq \lfloor q/2 \rfloor$.

If we further restrict c to values with small Hamming weight, say < 3, we can replace the multiplication in Theorem 4.5 by further left-shift and addition operations. When implementing this Mersenne reduction technique using GMP, note that if q is less than the word size of the target processor, the operations can be implemented by using a logical-AND bit-mask on the low-order `mp_limb_t` of y and by calling the GMP function `mpz_tdiv_q_2exp(...)`, respectively.

Looking at the case of the Mersenne prime M_{31}, we find two issues. The first is that we do not achieve the same level of the avalanche effect inherent to MD5 and other cryptographic hashes. For instance, with a 35-decimal digit discriminant we found no random hash collisions while using a 31-bit truncated MD5 hash, whereas we had approximately 910 random collisions reducing modulo the 31-bit Mersenne prime, about a 0.4% collision rate. Using the 32-bit pseudo-Mersenne prime $2^{32} - 5$, we found about 460 collisions.

The second issue is that for even moderately-sized discriminants—over 30 decimal digits—our hash input is at least a few machine-words in length. This forces us to use the GMP functions `mpz_tdiv_q_2exp(...)` and `mpz_sub_ui(...)` to implement this fast-modulo algorithm, and thus adding a non-negligible overhead. For the 64-bit implementation, we selected the pseudo-Mersenne prime $2^{64} - 257$.

We also implemented a version of Robert (Bob) Jenkin, Jr.'s One-at-a-time hash [49] and Lookup3 hash [50]. Like MD5, these hash functions produce a 32-bit hash from an arbitrary-length input using only single-precision additions, bitwise-XORs, and shifts. In addition, although they are not cryptographic hashes, they are designed to have good avalanche characteristics. In our testing, the output hashes were decently uniformly distributed. The worst goodness-of-fit test we observed showed distributions that had an approximately 12% chance of being sampled from a discrete uniform distribution. We also remark that the One-at-a-time hash is missing from the 64-bit comparisons that follow as it is strictly a 32-bit hash, unlike the Lookup3 hash which has a 64-bit variant.

For each of the above hash functions, both 32-bit and 64-bit versions, we computed and stored 100,000 baby-steps. This was followed by 100,000 appropriately sized giant steps, as dictated by the optimized $s + \lambda$ value for the $O(\Delta^{1/6+\epsilon})$ algorithm. A summary of the timing results for hash table insertions and searches are presented in Figures 4.1 and 4.2 for the 32-bit hashes and Figures 4.3 and 4.4 for the 64-bit hashes.

From these figures, we can see that our initial concerns about the overhead of MD5 and the GMP functions needed for the Mersenne and pseudo-Mersenne reduction algorithms are justified. Additionally, the Lookup3 hash seems to be outperforming the One-at-a-time hash, but this is not too surprising as the Lookup3 hash is a refined, though slightly more complex, version of the One-at-a-time hash. As such, we will focus exclu-

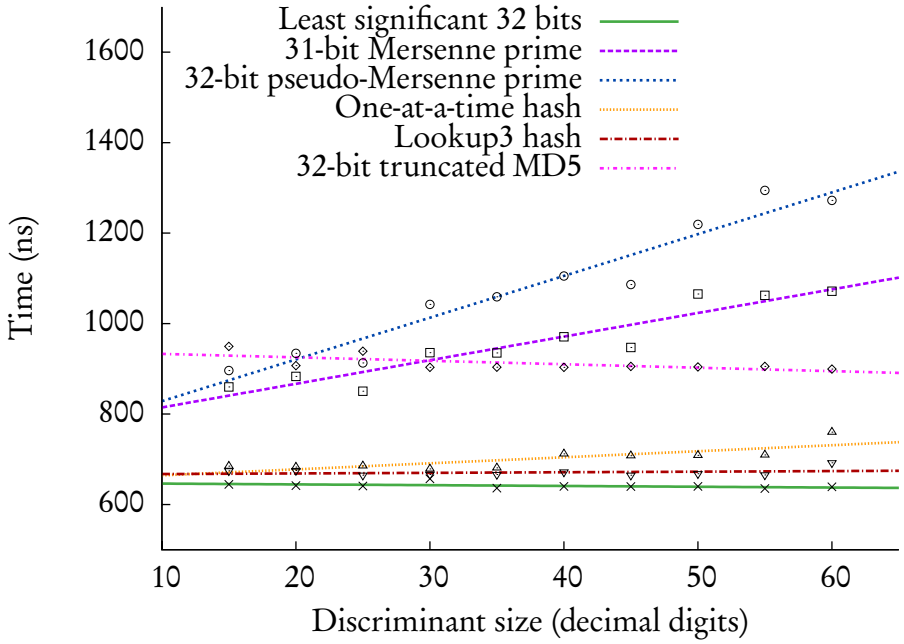


Figure 4.1: Hash table insertion times for various discriminant lengths (32-bit).

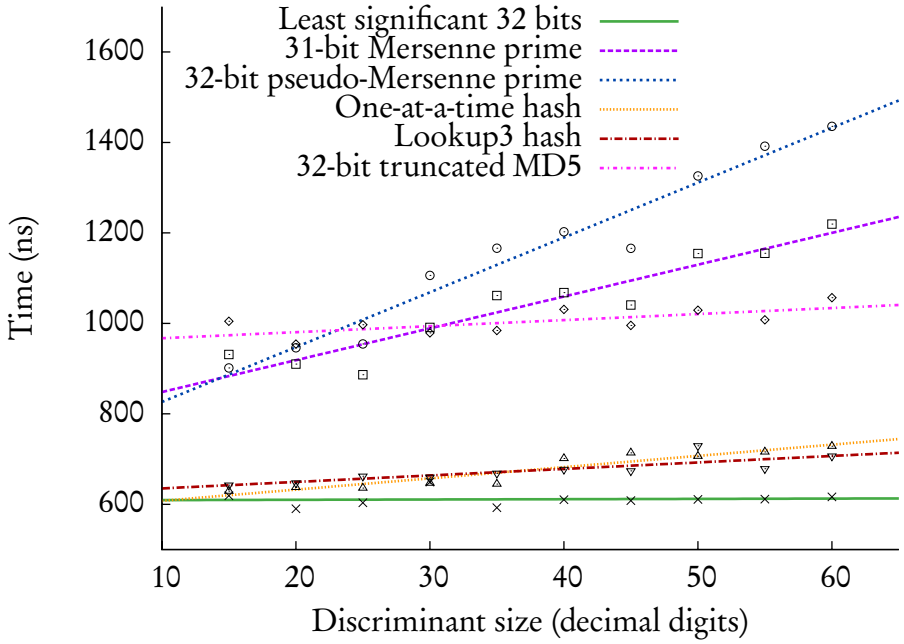


Figure 4.2: Hash table lookup times for various discriminant lengths (32-bit).

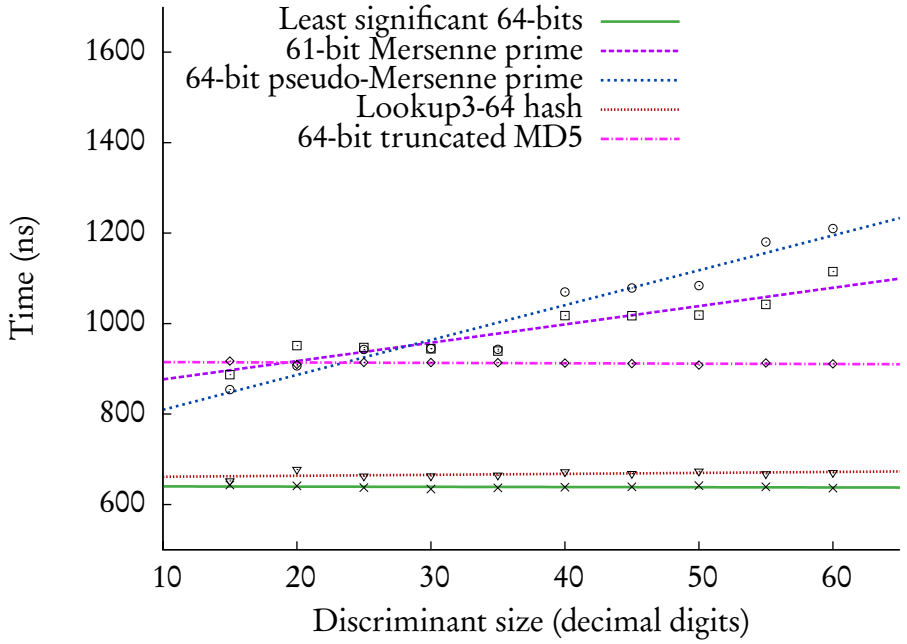


Figure 4.3: Hash table insertion times for various discriminant lengths (64-bit).

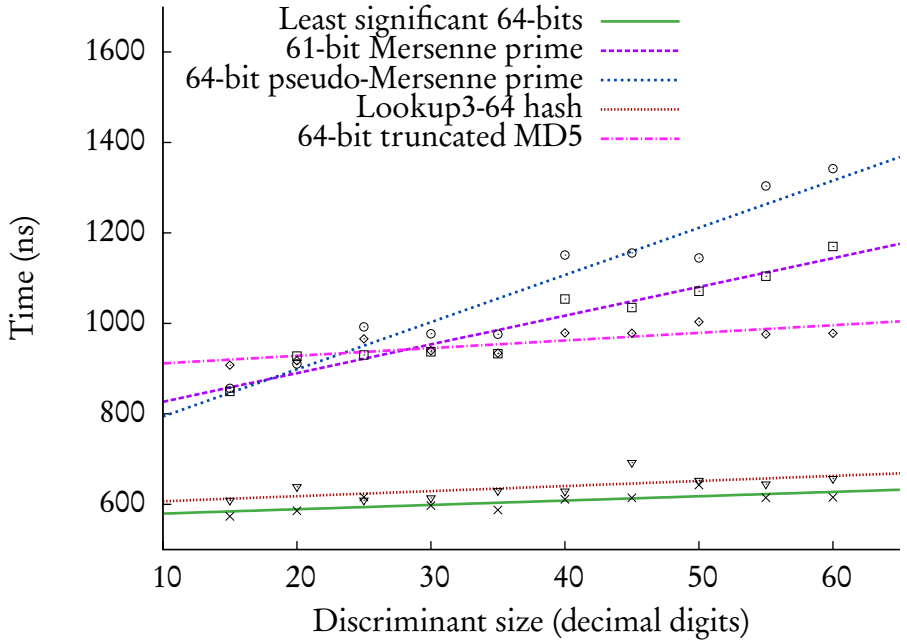


Figure 4.4: Hash table lookup times for various discriminant lengths (64-bit).

sively on the least significant 32 bits (LSB-32) hash and the Lookup3 hash in the remainder of this section.

4.3.2. Optimization parameters. There are a number of variables and run-time optimization parameters for the $O(\Delta^{1/6+\epsilon})$ algorithm. A few of these were introduced in Chapter 3—such as s, Q, and K—whereas others are important only when the algorithm is actually implemented, and so have been quietly glossed over until this point. As was done in [21], we normalize the running times in terms of the average time required to compute a baby-step in the analysis that follows. A detailed description of the variables and optimized parameters used in the following discussion is presented in Tables 4.2 and 4.3. Most of these variables and parameters were introduced in [21]. However, as we saw in the previous section, the amount of time needed to compute and store or lookup a hash varies based on the particular hash function selected. To account for this in our optimization formulas, we introduce the variables h and ν.

From the detailed analysis given in [21, §6.5, pp. 86–92], we know the overall run-time cost of the $O(\Delta^{1/6+\epsilon})$ algorithm can be expressed as the sum of the three quantities

$$Time_{BS} = \frac{s+\lambda}{1.7r}, \tag{4.1}$$

$$Time_{GS} = \frac{(Q-rn)(\mu+l-1)}{rn}, \text{ and} \tag{4.2}$$

$$Time_{FM} = \frac{\mathcal{R}'}{rnK\ln(2)}\left(\frac{\mu\log_2(\mathcal{R}'/s)+2(l-1)}{2\log_2(\mathcal{R}'/K)}\right). \tag{4.3}$$

We can easily determine l by computing

Table 4.2: Variables needed for optimizing the run-time of the $O(\Delta^{1/6})$ regulator verification algorithm.

Variable	Description
$\mathcal{R}'_2$	An approximation of the regulator.
λ	Additional distance needed to ensure a giant step or its conjugate will land in $\mathcal{L}$ ($\lambda = \lceil (1/2)\log_2 \Delta \rceil$).
μ	The time ratio between computing a baby step and computing a giant step.
h	The time ratio between computing the hash of an ideal and computing a baby step.
ν	The time ratio between searching for $H(\mathfrak{a})$ in a hash table and computing $H(\mathfrak{a})$.
N	Four (or six) less than the number of ideals we are able to store in memory. (We must store $\mathfrak{a}_1$, $\mathfrak{a}_t$, $\mathfrak{a}_{t+1}$, and $\mathfrak{a}_{t+2}$ in addition to every l^{th} ideal. To use the hashed baby-step list, we also need $\mathfrak{a}_2$ and $\mathfrak{a}_{(m+1)l}$.)
n	The number of compute nodes used during algorithm execution. For multi-processor or multi-core systems, we can count each processor or core as a separate machine if threading is disabled.
r	The number of processors and cores available on each compute node. If threading is disabled, we must set $r = 1$. (See n above.)

Table 4.3: Parameters determined by the optimization formulas for the $O(\Delta^{1/6})$ regulator verification algorithm.

Parameter	Description
l	The gap-size required to fit $\mathcal{L}'$ in the memory space available.
K	The regulator bound used to balance the algorithm work-load between the baby-step, giant-step, and multiplier-finding phases.
s	Used to determine the number of baby steps to store in $\mathcal{L}$.
Q	Used to determine the number of giant steps to compute.
M	Upper bound on the potential prime divisors of $\mathcal{R}'_2$ ($M = \mathcal{R}'_2/K$).

$$l = \frac{0.6(s+\lambda)}{N-(r-1)} \tag{4.4}$$

and, using the initial approximation

$$Q = \frac{K}{2s}\,, \tag{4.5}$$

we can substitute (4.4) and (4.5) into $Time_{BS} + Time_{GS}$, allowing us to solve for s in terms of K. Doing so gives

$$s = \sqrt{\frac{1.7\big((\mu-1)(N-(r-1))+0.6\lambda\big)}{2n(N-(r-1))}}\sqrt{K}\,. \tag{4.6}$$

Combining these equations back into (4.1)–(4.3) gives a function for the expected running time in terms of just the parameter K:

$$f(K) = Time_{BS} + Time_{GS} + Time_{FM}\,. \tag{4.7}$$

We do not present the fully-substituted formula here, both due to its sheer complexity and because it will not aid in understanding the optimization method. However, we do point out that a MAPLE script is presented in Section A.3.4, page 230, which will explicitly recreate this formula, if so desired. As an aid to the reader, however, we present a numerical example of $f(K)$ in Figure 4.5 for the discriminant $\Delta = 124190375333324$, using the parameters for a 100-node cluster of Intel Xeon E5530 processors.

Turning to our addition of ideal hashing to the $O(\Delta^{1/6+\epsilon})$ algorithm, we need to revise the preceding equations to account for the extra computational overhead. During the

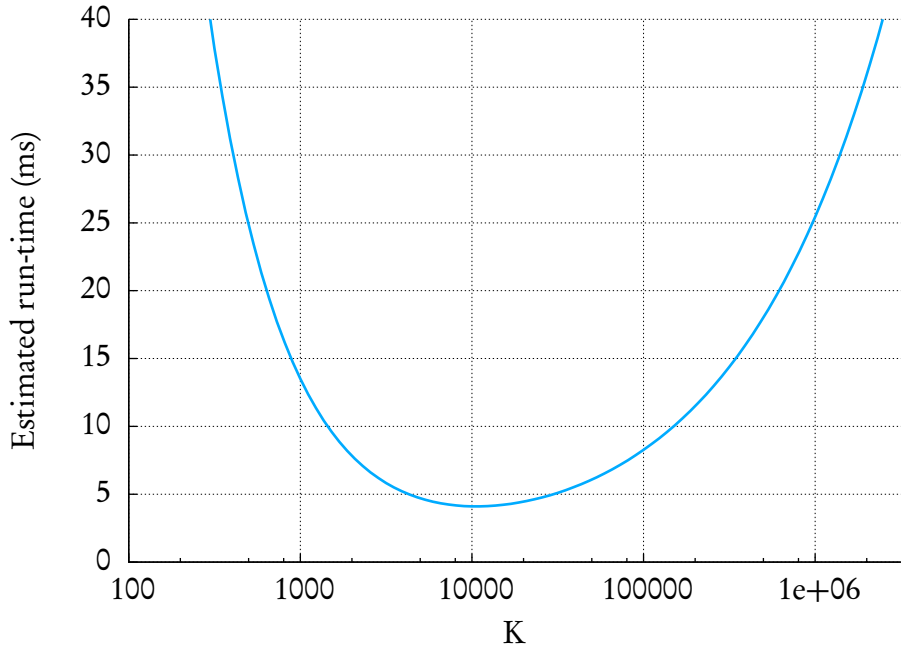


Figure 4.5: An instance of $f(K)$ for $\Delta = 124190375333324$.

baby-step phase, we need to compute an ideal hash for each ideal stored at a cost of h. Since we are only storing every l^{th} ideal, (4.1) becomes

$$Time_{BS} = \frac{s+\lambda}{1.7r}\left(1+\frac{h}{l}\right). \tag{4.8}$$

During the giant-step phase, we need to compute an ideal hash and perform a hash table lookup for each of the $l-1$ baby steps following every giant step produced. This means (4.2) becomes

$$Time_{GS} = \frac{(Q-rn)\big(\mu+(l-1)(1+h\nu)\big)}{rn}. \tag{4.9}$$

Similarly during the multiplier-finding phase, we perform $l-1$ baby steps following each $\mathfrak{a}[\mathcal{R}'/q_i]$ computation. As each of these baby steps now requires a hash computation and

hash table lookup, (4.3) becomes

$$Time_{FM} = \frac{\mathcal{R}'/K}{r\,n\ln(2)}\left(\frac{\mu\log_2(\mathcal{R}'/s)+2(l-1)(1+h\nu)}{2\log_2(\mathcal{R}'/K)}\right). \tag{4.10}$$

4.3.3. Optimization parameter selection. It is infeasible to symbolically find an expression for K which minimizes (4.7) in terms of the other variables and parameters involved. Instead, we minimize this equation by computing its derivative $f'(K)$ and numerically approximating $f'(K) = 0$. We then use the ceiling of this approximation as the optimum value for K, which in turn is used to approximate the optimum values of s, Q, and l.

In order to perform these calculations, however, we need to determine values for the μ, h, and ν constants specified in (4.1)–(4.3). This is done empirically and the results, of course, will vary from machine to machine. For μ, we take the average time needed to compute $\rho(\mathfrak{a}_i)$ divided by the average time needed to compute ADDXY$(\mathfrak{a}_j, \mathfrak{a}_t)$. h is determined by dividing the average time needed to compute and store $H(\mathfrak{a})$ by the average time needed to compute $\rho(\mathfrak{a})$. ν is determined by dividing the average time needed to call `hashtable_search(...)` by the average time needed to call `hashtable_insert(...)`.

For discriminants up to around 20 decimal digits, we were able to determine values for these constants by averaging the results from several thousand baby-step and giant-step computations over ten to twenty thousand random discriminants at each discriminant length. Beyond this range we are restricted to the discriminants from [21] and [22] as we were unable to produce further regulator approximations. The implemented index-calculus algorithms we had access to either could not scale beyond this range or were not functional.

We also need to determine an acceptable accuracy for the numerical approximation of the zero of $f'(K)$. Repeating the calculations for the parameters listed in Tables 7.2 and 7.5–7.7 of [21], we found that the zero of the derivative of (4.7) was calculated to an accuracy of roughly $|f'(K)| < 1 \times 10^{-6}$. By increasing the precision of MAPLE's floating point calculations to 25 decimal digits, we are able to recalculate these zeroes to an accuracy of about $|f'(K)| < 1 \times 10^{-19}$. This change alone gives us about a 20% reduction in the value of K, and hence a significant reduction in the number of baby and giant steps we need to compute. Using the MAPLE script mentioned in the preceding section and the empirical measurements mentioned above, we computed the optimal parameters for the single- and multi-processor implementations of the $O(\Delta^{1/6+\epsilon})$ algorithm using the LSB-32 and Lookup3 hashes. The values are listed in Tables 4.4–4.11.

Table 4.4: Optimal values for s (single-processor).

$l(\Delta)$	From [21]	Partial $\mathcal{L}$	LSB-32	Lookup3
15	2067	1680	1160	1143
20	28989	15841	11132	10925
25	500537	197669	139276	137000
30	2987168	1220382	861764	849835
35	17540584	9471963	6711231	6613907
40	92758446	40844765	30292922	29866303
45	742974984	294021503	233264657	231005360
50	5577266787	1992734173	1810501874	1786351716
55	—	9762388493	9376524778	9255211055

Table 4.5: Optimal values for Q (single-processor).

$l(\Delta)$	From [21]	Partial $\mathcal{L}$	LSB-32	Lookup3
15	82	57	70	70
20	472	492	598	604
25	3617	4906	5927	5990
30	19914	26404	31823	32077
35	113660	183850	222397	224105
40	569508	713199	857574	864079
45	3758569	4682471	6050500	6075211
50	24879661	29691656	43862754	43888758
55	—	136130482	212667923	213580199

Table 4.6: Optimal values for l (single-processor).

$l(\Delta)$	From [21]	Partial $\mathcal{L}$	LSB-32	Lookup3
15	1	1	1	1
20	1	1	1	1
25	1	1	1	1
30	1	1	1	1
35	1	1	1	1
40	5	3	1	1
45	34	19	4	4
50	251	125	27	27
55	—	655	140	140

Table 4.7: Optimal values for K (single-processor).

$l(\Delta)$	From [21]	Partial $\mathcal{L}$	LSB-32	Lookup3
15	335466	189110	160646	159688
20	27356281	15583757	13299272	13187010
25	3620796031	1939305208	1650710156	1641159405
30	118969311943	64444320739	54847020549	54519626467
35	3987312218758	3482837881003	2985113620557	2964416216543
40	105653501790311	58260863779734	51956806359377	51613683125421
45	5585044280443265	2753494225739124	2822735297272690	2806812565621660
50	277521007943539695	118335154832244015	158827194458235768	156801512960662300
55	—	2657917292592307000	3988172085477465173	3953459623944492093

Table 4.8: Optimal values for s (100 nodes, single processor).

$l(\Delta)$	Unthreaded [21]	Threaded [21]	Reimplemented	Lookup3
15	—	—	473	344
20	—	—	4689	3414
25	—	—	57180	41849
30	—	—	361335	264145
35	335671	410387	2791771	2045351
40	1842489	1968939	11705134	8818360
45	15536657	17168915	98049349	73086887
50	147933454	162324372	935434484	697295489
55	979952963	1056759343	6921903583	5166228351
60	—	5776794659	57234681978	42708057371
65	—	—	311979177889	249040653446

Table 4.9: Optimal values for Q (100 nodes, single processor).

$l(\Delta)$	Unthreaded [21]	Threaded [21]	Reimplemented	Lookup3
15	—	—	13	16
20	—	—	113	135
25	—	—	1167	1381
30	—	—	6291	7434
35	832829	756919	44778	52791
40	4154933	4052919	182364	211743
45	29526779	28273207	1319415	1539031
50	247800379	238411447	11709089	13651802
55	1468911639	1430111338	81491806	94885456
60	—	7757347522	618190516	718786976
65	—	—	3369680096	3147382632

Table 4.10: Optimal values for K (100 nodes, single processor).

$l(\Delta)$	Unthreaded [21]	Threaded [21]	Reimplemented	Lookup3
15	—	—	12184	10590
20	—	—	1057717	916149
25	—	—	133442699	115553714
30	—	—	4545798103	3926845192
35	559112756857	621259151670	250018355584	215951226594
40	15310835757647	15959900553825	4269174404922	3734437219338
45	917494866658340	970840569801550	258735401291404	224965887931234
50	73315931890405630	77399976766003286	21906170170976937	19038679021864581
55	2878928625823502223	3022567033942801739	1128156840544009744	980399864116469098
60	—	89625207458288884837	70763875069170128902	61395990810949500970
65	—	—	2102540051867409802425	156765245426341688 1427

Table 4.11: Optimal values for l (100 nodes, single processor).

$l(\Delta)$	Unthreaded [21]	Threaded [21]	Reimplemented	Lookup3
15	—	—	1	1
20	—	—	1	1
25	—	—	1	1
30	—	—	1	1
35	1	1	1	1
40	1	1	1	1
45	1	1	1	1
50	8	5	2	2
55	49	27	13	10
60	—	160	107	80
65	—	—	582	464

As de Haan noticed from his results, further run-time improvements can be realized with some empirical refinements to these computed values. For the single-processor implementation, we were able to improve the running time by about 5% on average. For the parallelized implementation, this improvement increased to about 13.9% on average for discriminants up to 35 decimal digits in length, but only to roughly 6.4% for 40-digit and larger discriminants. However, we cannot conclusively say whether this is a systemic property or merely a statistical anomaly due to the limited sample of discriminants we had to work with. These refined parameters are listed in Tables 4.12–4.13.

4.3.4. Implementation profiling. In this last subsection before we present our timing results, we will briefly discuss some of the practical optimizations made in our implemen-

Table 4.12: Empirically refined values for s (single-processor).

$l(\Delta)$	From [21]	Partial $\mathcal{L}$	LSB-32	Lookup3
15	2067	2016	1972	1944
20	28989	17426	18925	18573
25	500537	177903	236770	232900
30	2987168	1220382	1464999	1444720
35	17540584	10419160	11409093	11243642
40	92758446	57182671	51497968	50772716
45	742974984	382227954	396549917	392709112
50	5577266787		3077853186	3036797918

Table 4.13: Empirically refined values for Q (single-processor).

$l(\Delta)$	From [21]	Partial $\mathcal{L}$	LSB-32	Lookup3
15	82	48	42	42
20	472	448	352	356
25	3617	5452	3487	3524
30	19914	26404	18720	18869
35	113660	167137	130822	131827
40	569508	509428	504456	508282
45	3758569	3601901	3559118	3573654
50	24879661		25801620	25816917

tation of the algorithms in this thesis. For a more detailed discussion of these, we refer the reader to Appendix A.

The guiding principle behind how our implementation came together is based on a comment Donald Knuth made in his 1974 paper on structured programming with "go to" statements:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We* should *forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*
>
> *Yet we should not pass up our opportunities in that critical 3%. . . . A good programmer... will be wise to look carefully at the critical code; but only* after *that code has been identified. It is often a mistake to make a priori judgements about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.* [52, p. 268]

Whenever an algorithm was implemented from its pseudocode description, the structure and operations were preserved as much as possible. Of course, there were some minor pre-profiling optimizations made, but these were few and far between. The majority of the alterations and improvements were made only after extensive profiling, cache-simulation and analysis to determine bottle-necks in program execution.

One of the most significant pre-profiling changes was the decision to use a global pool of multi-precision integers. The repeated allocation, initialization and clearing of the multi-precision integers used as temporary variables in various algorithms is compu-

tationally expensive. In fact, as mentioned in the GMP documentation:

> *Initializing and Clearing*
>
> *Avoid excessive initializing and clearing of variables, since this can be quite time consuming, especially in comparison to otherwise fast operations like addition.*
>
> *A language interpreter might want to keep a free list or stack of initialized variables ready for use. It should be possible to integrate something like that with a garbage collector too.* [31, §3.11]

In some profiling runs, we saw the need for over 100,000 temporary variables being satisfied by maybe a few thousand pooled integers, leading to a substantial run-time savings. This idea was extended to a global pool of ideals for use in temporary calculations, but this is not as widely used in our library.

For the performance analysis of our programs, we used the VALGRIND tool suite [81], with visualization and data summaries produced by KCACHEGRIND [85]. Although the emulation VALGRIND performs can give an accurate picture of how a program executes, there are some caveats to be aware of. The first is a massive expansion in run-time. Because of the detail a profiler captures and, in the case of VALGRIND, the cache-simulation it performs, it is not uncommon to see a 100- to 150-fold increase in run-time. As an example, running our single-processor implementation of the $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing on the regulators from a 35-digit discriminant took roughly 20 seconds. When the program was placed under the profiler's control, it executed in roughly 42 minutes; a 127-fold increase.

The second point to keep in mind is that VALGRIND only tracks CPU instructions issued by the program it is profiling. It does not track the instruction counts for operating

system (kernel) functions—nor are they included in the cache simulation—and it does not track memory and disk I/O. For programs that read or write a lot of data to disk, this activity will not show up as a bottle-neck. In our case, our implementation has little operating system interaction and virtually no file I/O.

The third point is that while VALGRIND produces an instruction cost for each and every function call made by a program, it treats all instructions as having an equal execution time. This can be somewhat rectified by using the cache-simulation results and a measurement of a particular system's cache latencies to come up with an estimated instruction cost. Figure 4.6 shows the latency measurements from one compute node of the Institute for Security, Privacy and Information Assurance (ISPIA) cluster. From these measurements, we have produced a decently accurate model of the actual run-time using some of KCACHEGRIND's built-in tools. A more accurate analysis could be performed with a sampling profiler like OPROFILE, however due to time and system access constraints, this was not done.

One of our focal points for optimization was reducing CPU cache misses. As can be seen in Figure 4.6, if the data needed during the execution of our algorithms—temporary variables, various ideals, hash table entries, *etc.*—is not readily available in the L1 cache, we incur a considerable cost to copy it in from the lower L2 cache and an even greater cost if it must be brought in from RAM. In the particular case of an ISPIA cluster compute node, there is a 3 CPU clock-cycle wait for data residing in the L1 cache, a 15-cycle wait for L2 cache data, and roughly a 285-cycle wait for data in RAM. The delay for retrieving data from swap space is, at least, an order of magnitude greater than this. The sudden spikes in latency indicate where we have exceed the storage space in a particular cache: the Pentium 4 Xeon processors in our compute nodes have a 32KB L1 cache and 6MB L2

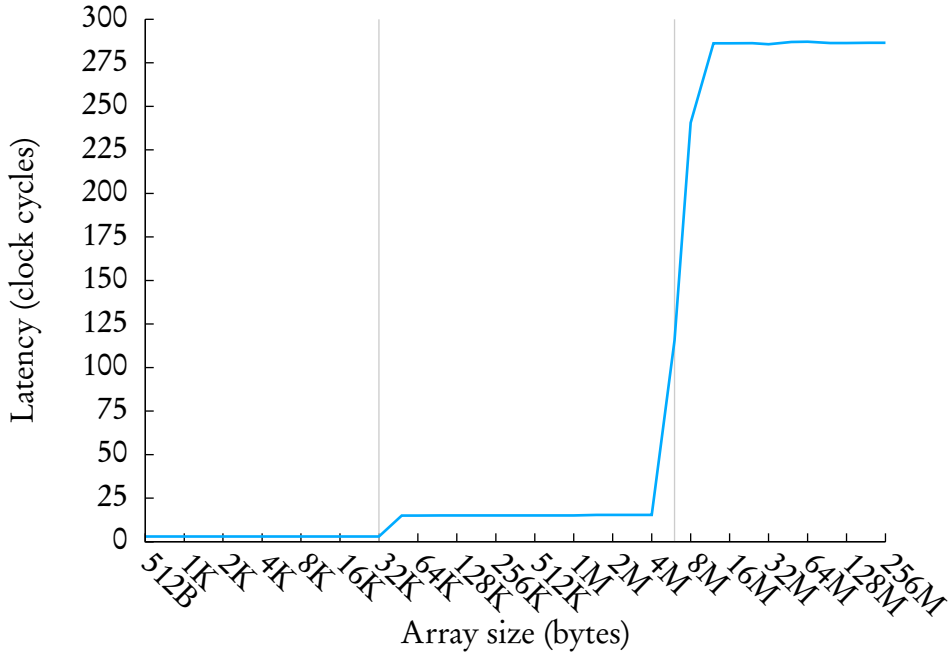


Figure 4.6: Cache latency measurements for an Intel Pentium 4 Xeon processor.

cache. The reason we see the first spike after 32KB, but the second one before the 6MB mark has to do with the particular cache structure on this processor. It has independent data and instruction caches at the L1 level, but a unified L2 cache. Thus, we can fill the L1 data cache without losing the measurement instructions, but when we approach the size of the L2 cache, we start swapping data and instructions back-and-forth from RAM.

In terms of absolute numbers of cache misses, by far the most expensive functions are those relating to initializing the hash table used for $\mathcal{L}''$ and then storing and retrieving ideal hashes from it. These account for roughly 98% to 99% of all cache read and write misses. This is not surprising as our implementation uses roughly 75% of the available system RAM for this hash table. When we initially create the hash table, we must zero out the block of memory allocated to store it. Though this ensures that the table indeed

starts empty, it is a very expensive process that is unfortunately only necessary to handle a rare situation. From a security standpoint, any modern operating system will not allocate memory pages to a process that it has not preemptively zeroed out. If the operating system did not do so, there is a very real risk of exposing privileged system information or another user's private data to a process which should not see this information. So the risk to our implementation comes not from the system, but actually from itself. If we ask for, use, and return memory before allocating the hash table, the operating system may allocate some of these now "dirty" memory pages to us rather than allocating a block of fresh pages. As our process was the last to write to them, there is no issue in returning them non-zeroed. Guaranteeing that this situation would not occur, though, is very difficult and so we are forced to explicitly zero out the hash table.

Turning to hash table storage and retrieval functions, because of the random accesses to such a large block of memory, the L1 and L2 caches are virtually useless in maintaining cache locality. On practically every store or retrieval we are forced to pull data in from RAM. Instead of trying to maintain locality, we have focused on preventing these, in a sense, one-time memory accesses from overwriting much more useful data that we wish to keep cached; temporary variables, some key ideals, and the instructions needed to execute the currently running function, for example. Protecting the cache in this way is done via so called non-temporal memory access functions which instruct the CPU to store or retrieve data from RAM, but not place a copy in its caches.

In the remainder of this section, we present a summary of the diagnostic and timing data from the cache-performance simulations. Because of the interactions and dependencies between the various algorithms used in the $O(\Delta^{1/6+\epsilon})$ algorithm—recall Figure 2.1 on page 39—we have chosen to illustrate this data in a *calling-context ring chart* [66].

This type of chart not only allows us to show the estimated operation cost of a function, but also how this cost relates to those of its parent and child functions in a condensed visual format; the greater an angle covered by a given wedge, the larger its relative cost. The calling-context ring chart for the single-processor implementation using LSB-32 ideal hashing for a 45-decimal digit discriminant is presented in Figure 4.7. Additional ring charts are presented in Appendix A for the 25-, 30-, 35-, and 40-decimal digit discriminants in Figures B.6–B.9, pages 242–245.

The innermost ring in the chart represents estimated costs of the immediate functions called by the $O(\Delta^{1/6+\epsilon})$ algorithm. The next larger ring presents the child functions called by those in the innermost ring—the "grandchild" functions of the $O(\Delta^{1/6+\epsilon})$ algorithm—and so on. The colours of each wedge have no meaning, save for the grey ones. These wedges represent an amalgamation of the functions that were too small to individually record; in this case, those functions whose estimated cost is less than around 1% of the overall run-time. To help simplify the labels, we have generally only labelled functions with an estimated cost of at least 4%.

From Figure 4.7 we obtain a fairly accurate picture of the operation costs of the various component functions used in the $O(\Delta^{1/6+\epsilon})$ algorithm. ADDXY occupies the majority of the run-time, with a mostly even split between its component functions NUMULT and EWNEAR. We note, however, that the estimated cost of WNEAR appears to be growing as the size of the discriminant is increased. The majority of NUMULT is spent executing ENUCOMP, which we would expect from glancing over its pseudocode description. While neither ENUCOMP nor EWNEAR has a significant overhead, the most expensive child functions are various GMP routines. A similar situation can be seen with the ρ operation. The end observation to make is that the most significant implemen-

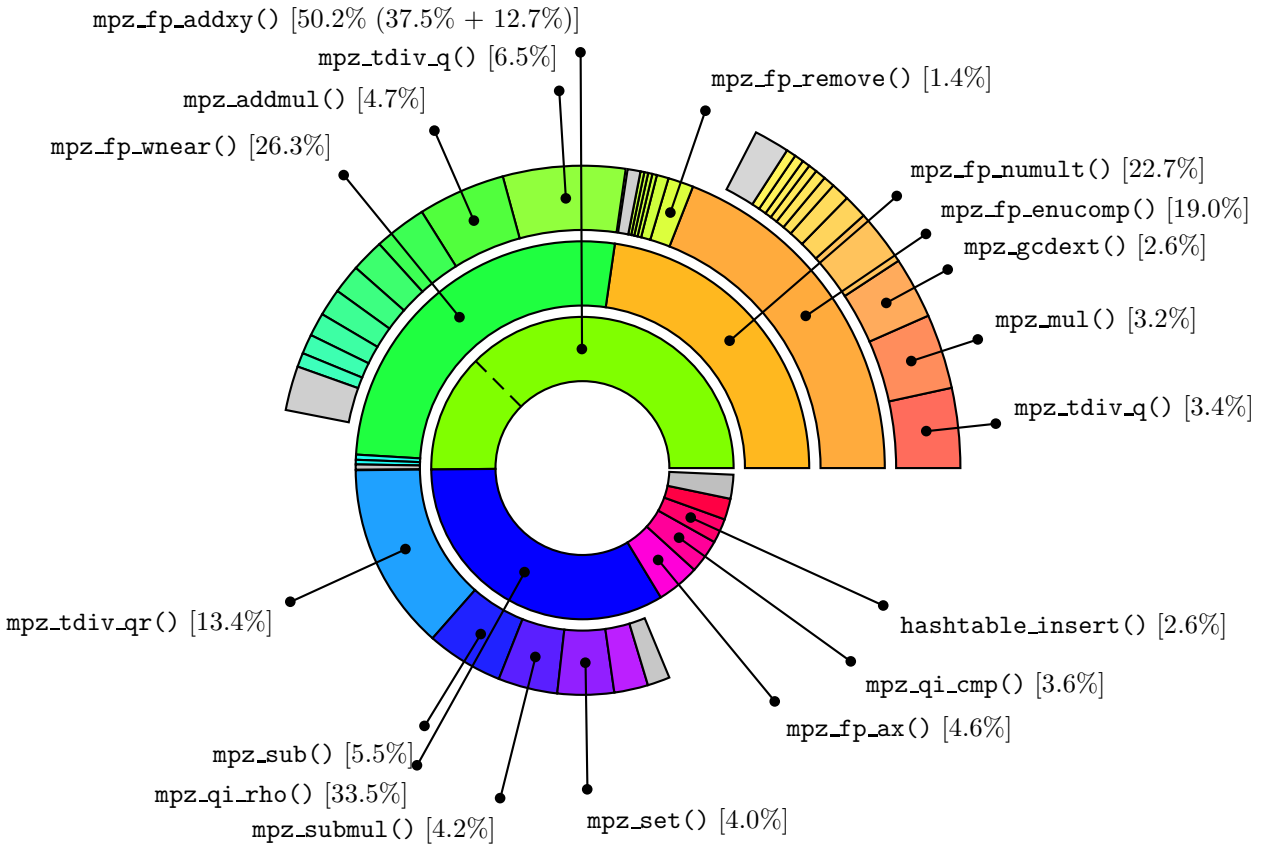


Figure 4.7: Estimated cycle costs with function calling contexts for the single-processor $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing (LSB-32, 45 decimal digit discriminant).

tation improvements will probably come from eliminating some arithmetic operations from ρ and from the inner-most while-loops in ENUCOMP and EWNEAR. However, this does not appear to be an easy task.

4.4. TIMING RESULTS

In this section we present the timing results for our numerical tests which were generated using the ISPIA Advanced Cryptography Lab cluster consisting of 152 IBM HS20 Blade nodes, each with dual 2.4GHz Intel Pentium 4 Xeon processors and 2 GB of RAM. The first table of timing results—Table 4.14, page 114—is meant to give a comparison between the implementation from [21] and our reimplementation of those algorithms. The key difference between the two is that the former was written in C++ using NTL for its multi-precision arithmetic. Our reimplementation is written in C using GMP. It is well-known that the GMP arithmetic routines are faster than their counterparts in NTL. The NTL routines are solely written in C++, with a strong focus on portability and a clean, consistent interface. On the other hand, the GMP routines tend to be written in hand-optimized assembly code. The NTL routines and data types also come with additional computational overhead from details like, for example, C++ class initialization and destruction. While NTL can be compiled to use GMP as its multi-precision integer package, these are generally wrapped with code to check function inputs before passing them off to various GMP subroutines. All of this combined makes the basic GMP routines, in particular, significantly faster than their NTL counterparts. As these are the routines we use most frequently in our implementation, we expect to see a marked improvement in run-time and want to properly attribute those gains coming from switching to GMP versus those from our modifications to the $O(\Delta^{1/6+\epsilon})$ algorithm. This comparison also takes into account the algorithmic refinements which were made to the base algorithms,

such as NUCOMP, and any compiler and optimization flag differences.

The second and third sets of timing results—Tables 4.15 and 4.16, pages 117 and 120, respectively—show the run-time improvements realized by our algorithmic modifications, using the LSB-32 and Lookup3 hashes. In the four figures that follow each of these three tables are graphical plots of the observed timings, the phase balance, an exponential regression (purple) with a 95% confidence interval (CI) (orange) for the mean and a 95% prediction interval (PI) (cyan) for future results, and extrapolated timings for larger discriminants (60–67 decimal digits). Table 4.17, page 123, gives a summary comparison of the three sets of results, which we will focus on now.

The vast majority of our implementation's speedup comes from the switch in the underlying multi-precision arithmetic library. For the single-processor implementation, the run-time savings due to switching from a partial baby-step list to a hashed list seem to top out around 10%. However, this is only for discriminants near the 30 to 35 decimal digit range. For larger discriminants, this savings rapidly disappears.

Starting at the 45-digit discriminant mark in the single-processor implementation, we exhaust the available memory storage and are forced to use a partial list as in [21, 22]. As the value of l increases, the balance between the baby-step, giant-step, and multiplier-finding phases changes drastically, as can be seen in Figures 4.9, 4.13, and 4.17 on pages 115, 118, and 121, respectively.

We can use the optimizing function from (4.7), page 95, to determine if this situation should be expected or not. Recall that, for a given value of K, $f(K)$ represents the overall cost of the $O(\Delta^{1/6+\epsilon})$ algorithm in terms of the number of baby-step operations. To calculate an estimated time per phase, we could multiply it by the actual average time to compute ρ at a given discriminant size. For our implementation, this constant varies

Table 4.14: Observed timings for single-processor implementation (partial baby-step list).

	Default parameters				Empirically-adjusted parameters				
$l(\Delta)$	BS Time	GS Time	FM Time	Total	BS Time	GS Time	FM Time	Total	Savings
15	0.00273s	0.00237s	0.00474s	0.00984s	0.00313s	0.00201s	0.00411s	0.00924s	6.10%
20	0.0213s	0.0217s	0.0158s	0.0588s	0.0230s	0.0195s	0.0154s	0.0579s	1.53%
25	0.282s	0.246s	0.114s	0.642s	—	—	—	—	0%
30	1.72s	1.45s	0.549s	3.72s	—	—	—	—	0%
35	14.5s	11.4s	3.58s	29.4s	14.7s	10.4s	3.46s	28.6s	2.72%
40	39.8s	54.2s	15.6s	1m 50s	54.7s	37.5	14.9s	1m 47s	2.73%
45	3m 49s	6m 42s	2m 23s	12m 53s	4m 26s	5m 1s	2m 15s	11m 41s	9.31%
50	21m 11s	45m 21s	42m 50s	1h 49m					

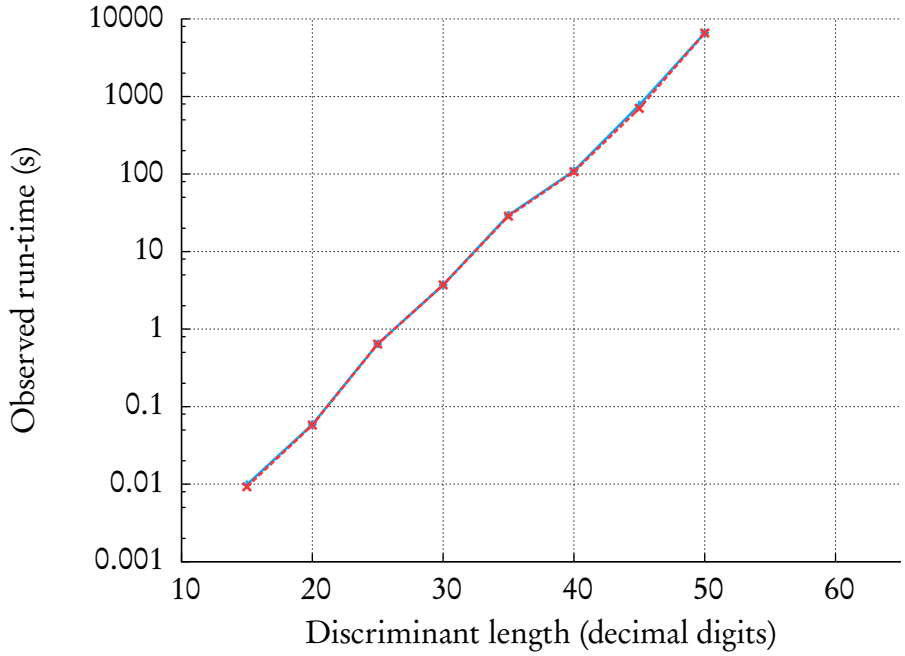


Figure 4.8: Observed timings for single-processor implementation (partial baby-step list).

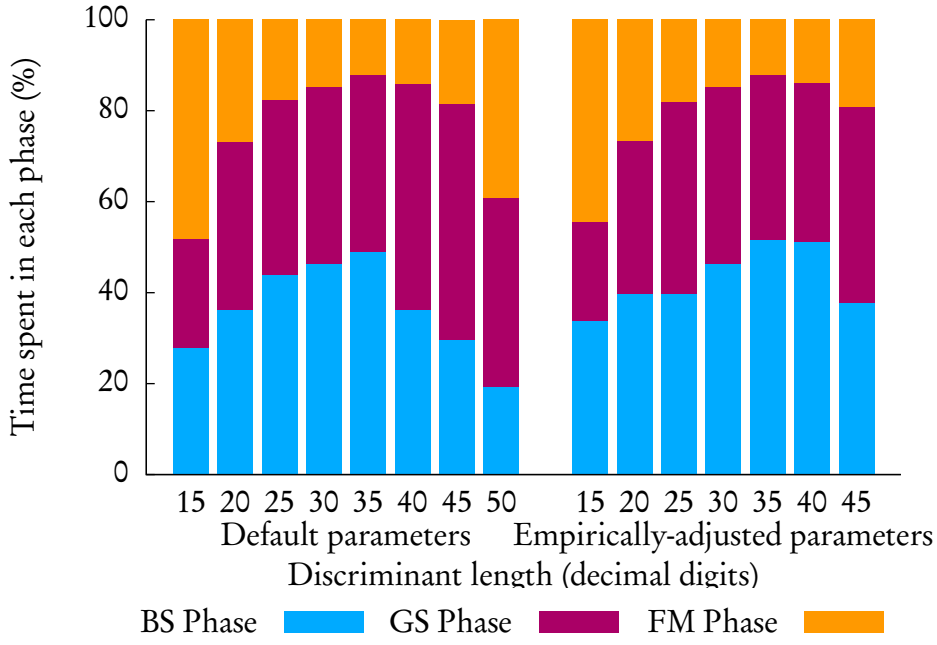


Figure 4.9: Phase balance for single-processor implementation (partial baby-step list).

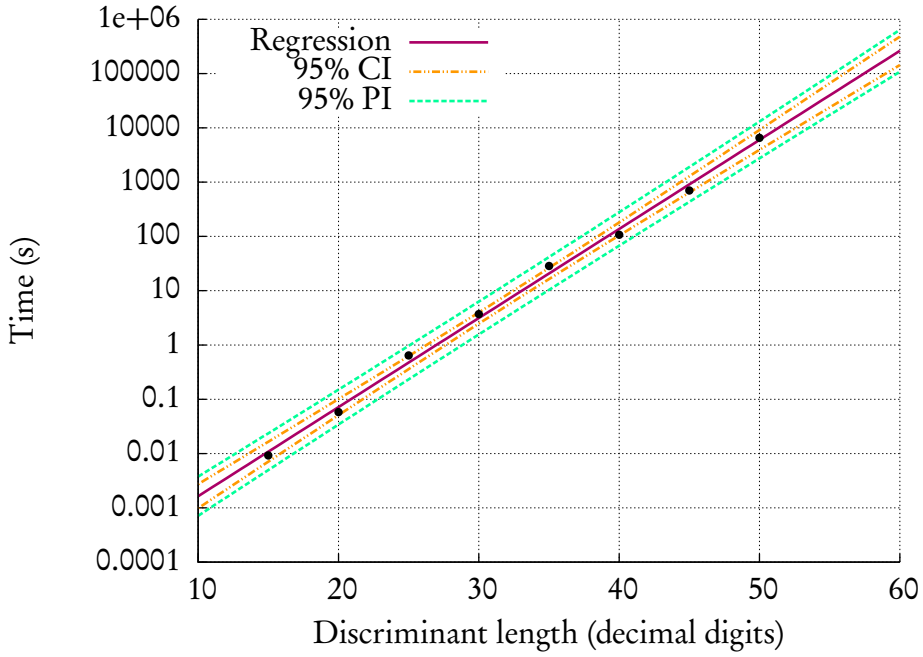


Figure 4.10: Exponential regression for single-processor implementation timings (partial baby-step list).

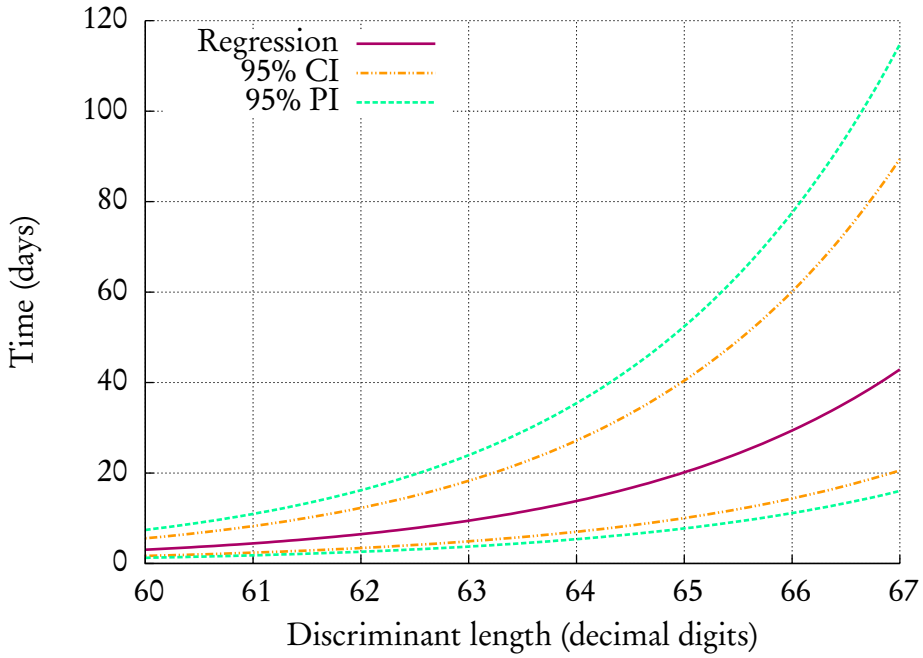


Figure 4.11: Estimated single-processor implementation run-time for larger discriminants (partial baby-step list).

Table 4.15: Observed timings for single-processor implementation (LSB-32 hashing).

	Default parameters				Empirically-adjusted parameters				
$l(\Delta)$	BS Time	GS Time	FM Time	Total	BS Time	GS Time	FM Time	Total	Savings
15	0.00105s	0.00256s	0.00430s	0.00791s	0.00149s	0.00188s	0.00404s	0.00741s	6.32%
20	0.00750s	0.0272s	0.0175s	0.0522s	0.0114s	0.0166s	0.0151s	0.0431s	17.4%
25	0.104s	0.301s	0.127s	0.532s	0.153s	0.180s	0.117s	0.450s	15.4%
30	0.622s	1.79s	0.611s	3.02s	0.967s	1.09s	0.563s	2.62s	13.2%
35	4.74s	14.7s	4.20s	23.6s	7.88s	8.03s	3.67s	19.6s	16.9%
40	25.7s	1m 5s	18.6s	1m 50s	39.0s	46.7s	19.3s	1m 45s	4.54%
45	2m 38s	8m 6s	2m 25s	13m 10s	4m 4s	5m 33s	2m 33s	12m 10s	7.59%
50	19m 20s	1h 6m	32m 24s	1h 57m	33m 31s	37m 42s	35m 26s	1h 47m	8.55%
55	1h 39m	5h 46m	7h 31m	14h 57m	2h 55m	3h 11m	8h 6m	14h 13m	4.91%

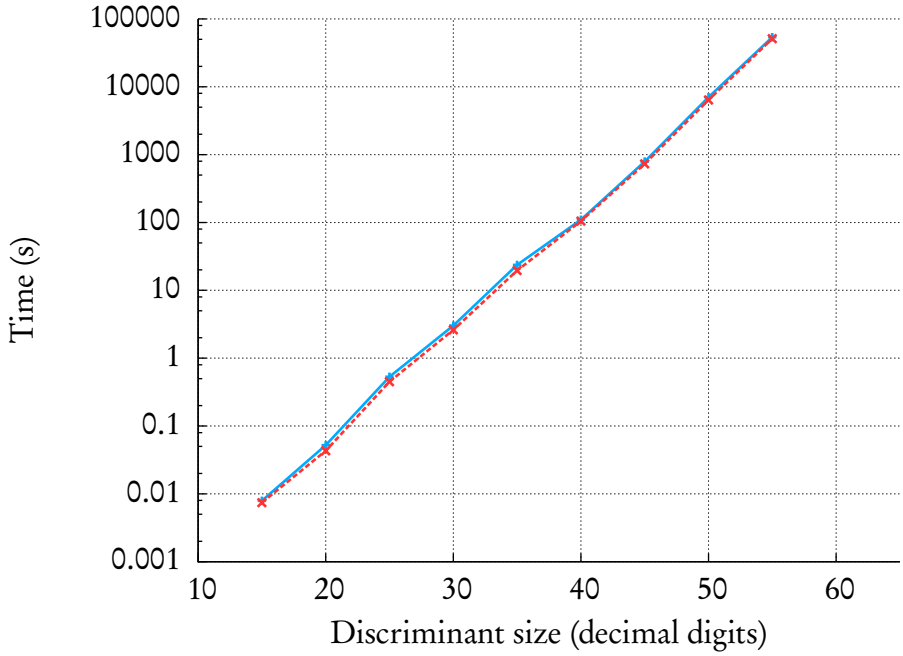


Figure 4.12: Observed timings for single-processor implementation (LSB-32 hashing).

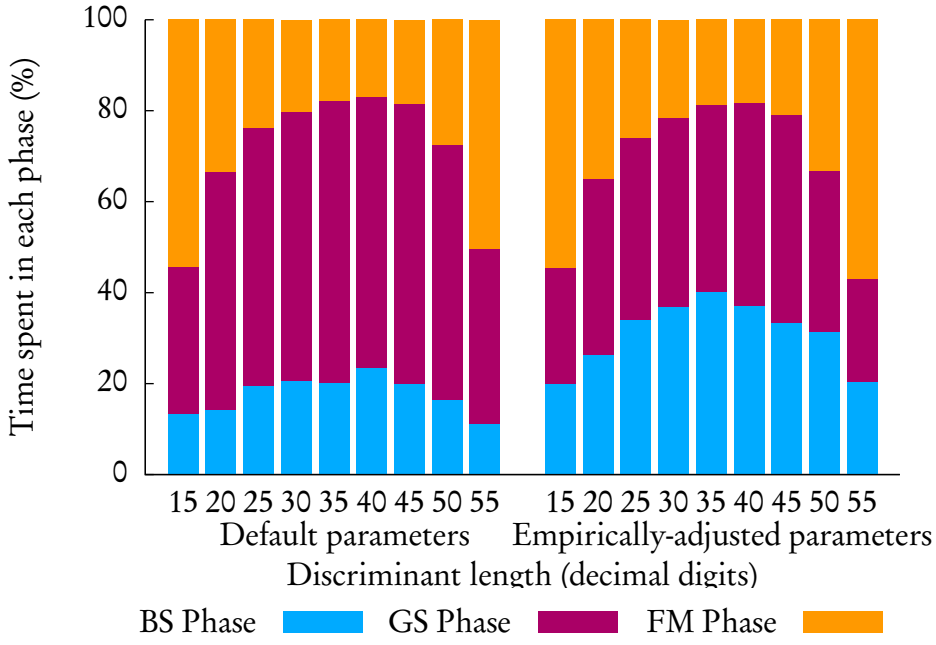


Figure 4.13: Phase balance for single-processor implementation (LSB-32 hashing).

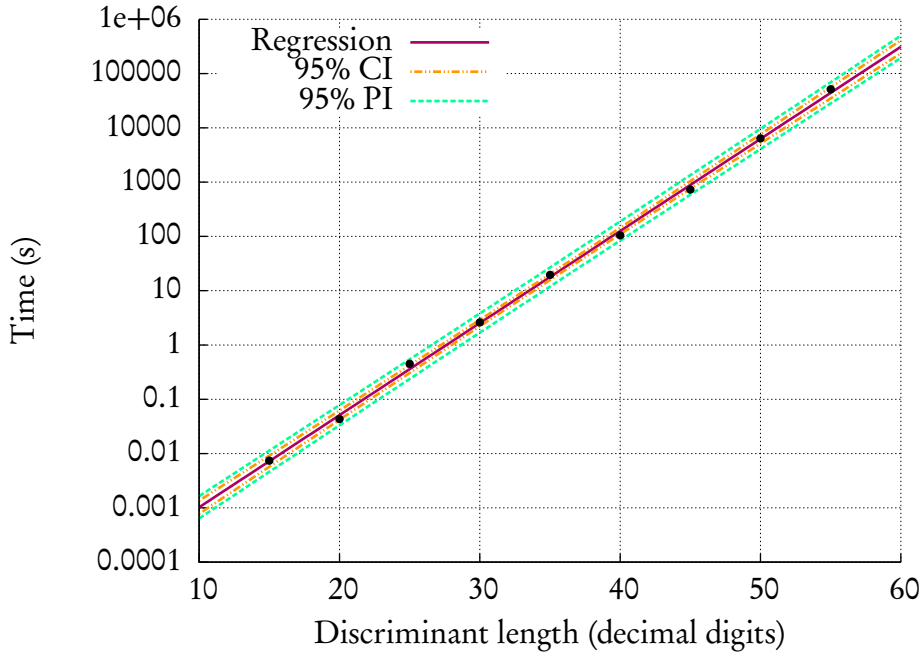


Figure 4.14: Exponential regression for single-processor implementation timings (LSB-32 hashing).

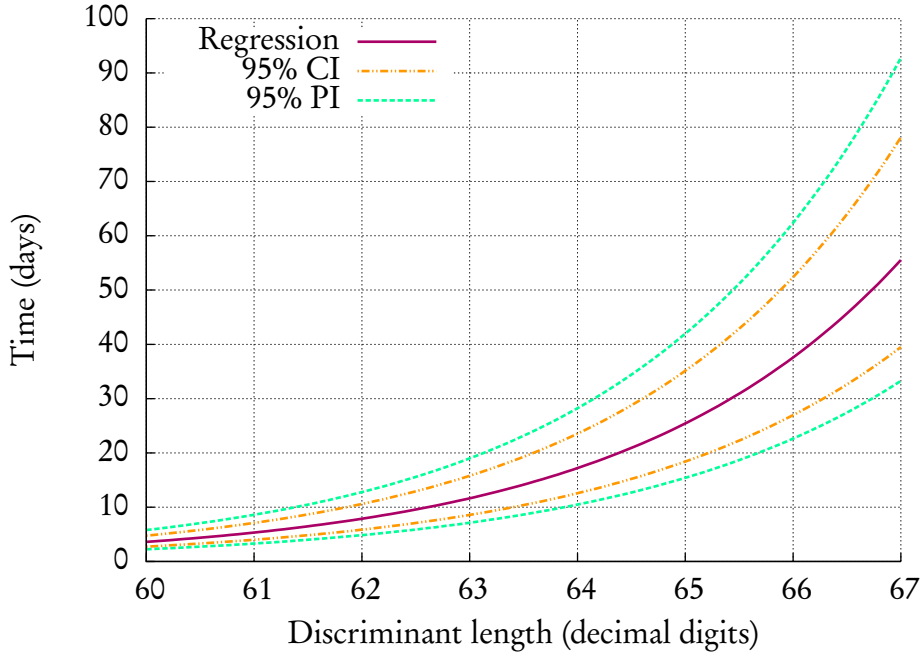


Figure 4.15: Estimated single-processor implementation run-time for larger discriminants (LSB-32 hashing).

Table 4.16: Observed timings for single-processor implementation (Lookup3 hashing).

	Default parameters				Empirically-adjusted parameters				
$l(\Delta)$	BS Time	GS Time	FM Time	Total	BS Time	GS Time	FM Time	Total	Savings
15	0.00106s	0.00274s	0.00441s	0.00820s	0.00152s	0.00185s	0.00404s	0.00740s	9.76%
20	0.00752s	0.0263s	0.0169s	0.0507s	0.0123s	0.0158s	0.0146s	0.0427s	15.8%
25	0.0992s	0.299s	0.127s	0.525s	0.152s	0.191s	0.116s	0.459s	12.6%
30	0.634s	1.72s	0.614s	2.97s	1.04s	1.04s	0.560s	2.64s	11.1%
35	4.82s	13.4s	4.00s	22.2s	7.74s	8.24s	3.70s	19.7s	11.3%
40	26.1s	1m 4s	17.4s	1m 48s	41.6s	41.2s	16.7s	1m 40s	7.41%
45	2m 41s	8m 0s	2m 17s	12m 58s	4m 31s	4m 58s	2m 16s	11m 46s	9.25%
50	19m 2s	1h 9m	29m 9s	1h 57m	32m 37s	37m 31s	28m 18s	1h 38m	16.2%
55	1h 35m	5h 42m	7h 0m	14h 17m	2h 45m	3h 21m	7h 6m	13h 12m	7.58%

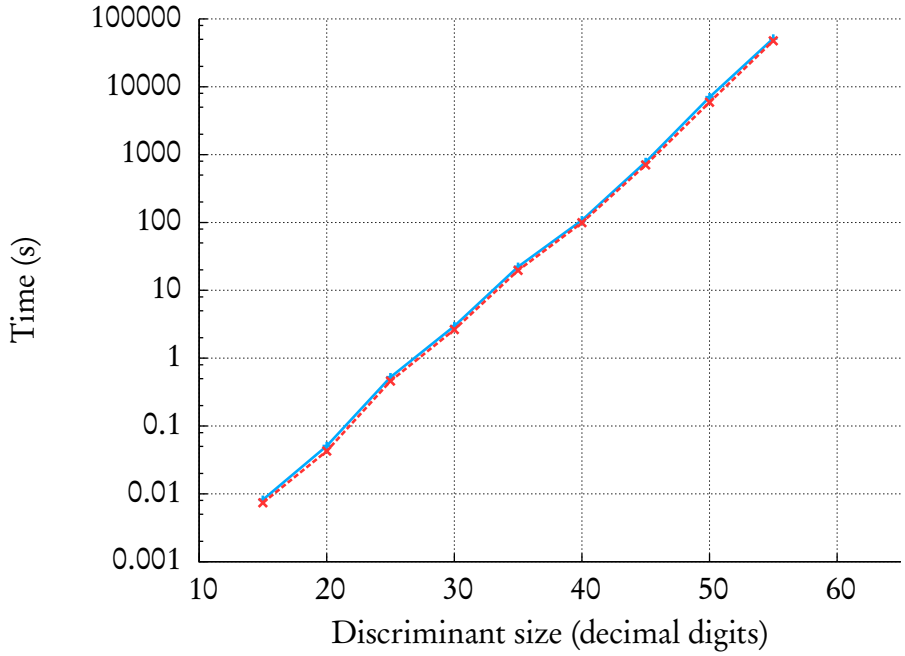


Figure 4.16: Observed timings for single-processor implementation (Lookup3 hashing).

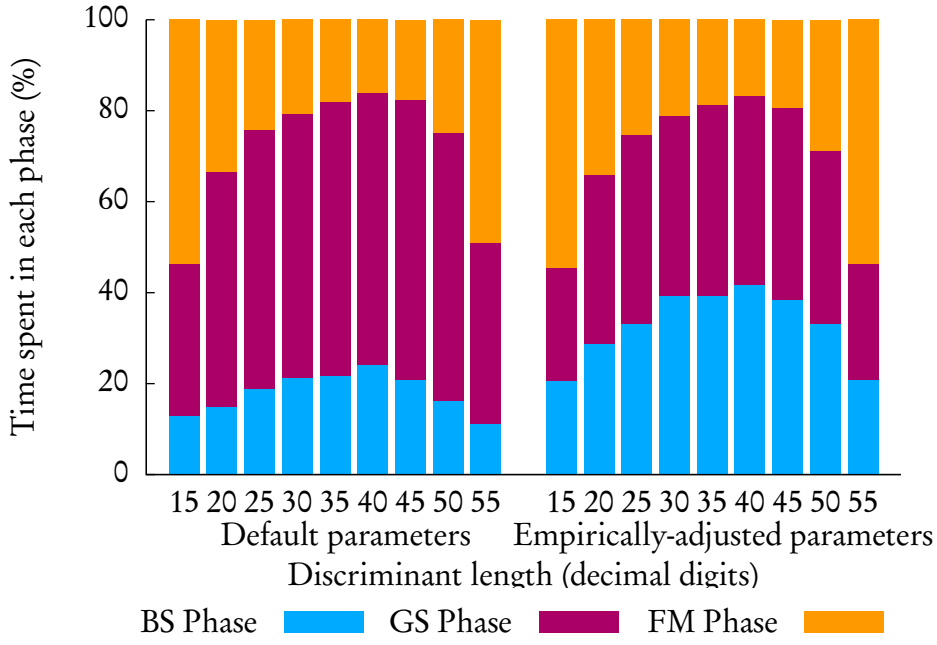


Figure 4.17: Phase balance for single-processor implementation (Lookup3 hashing).

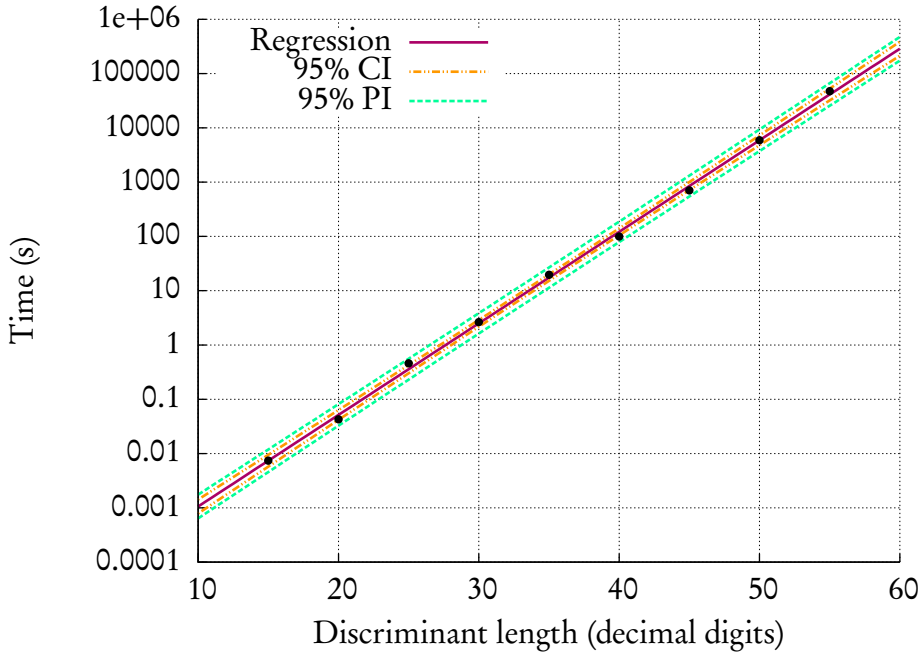


Figure 4.18: Exponential regression for single-processor implementation timings (Lookup3 hashing).

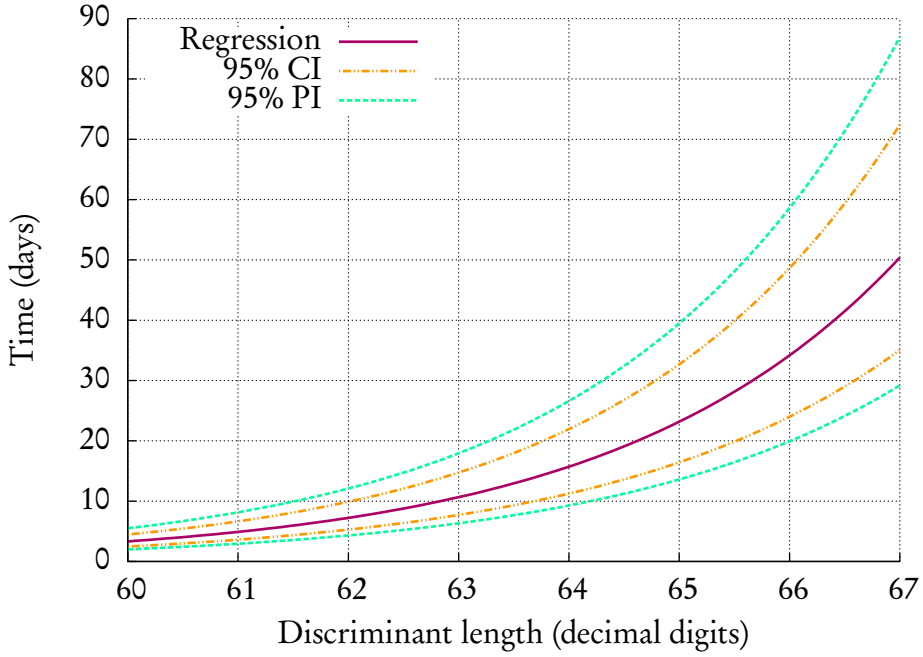


Figure 4.19: Estimated single-processor implementation run-time for larger discriminants (Lookup3 hashing).

Table 4.17: Summary of single-processor timing results (32-bit).

	Timings				Savings vs. [22]		
$l(\Delta)$	From [22]	Partial $\mathcal{L}$	LSB-32	Lookup3	Partial $\mathcal{L}$	LSB-32	Lookup3
15	0.42s	0.0092s	0.0074s	0.0074s	97.8%	98.2%	98.2%
20	0.93s	0.058s	0.043s	0.043s	93.8%	95.4%	95.4%
25	3.20s	0.64s	0.45s	0.46s	80.0%	85.9%	85.6%
30	14.60s	3.72s	2.62s	2.64s	74.5%	82.1%	81.9%
35	1m 27s	28.6s	19.6s	19.7s	67.1%	77.5%	77.4%
40	6m 12s	1m 47s	1m 45s	1m 40s	71.2%	71.8%	73.1%
45	1h 10m	11m 41s	12m 10s	11m 46s	83.3%	82.6%	83.2%
50	1d 9h	1h 49m[†]	1h 47m	1h 38m	94.5%[†]	94.6%	95.1%
55	—		14h 13m	13h 12m			

[†]: Due to suspected computer hardware issues, we were unable to determine this timing using empirically-adjusted parameters. In its place, we have included the timing determined using the default parameters.

between 0.8 μs and 1 μs. However, since we are more interested in a relative comparison between the three phases, this is not necessary at this point. Figure 4.20 shows the theoretical run-time percentages for each phase of the algorithm, assuming we use the single-processor implementation with LSB-32 hashing. We see that the optimum parameters computed should evenly balance the three phases of the algorithm, at least for smaller discriminants. However, once we are unable to store a complete baby-step list, a greater and greater percentage of the run-time is consumed by the multiplier-finding phase. Near the 70 decimal digit discriminant range, the relative time needed for the baby-step phase is negligible, whereas the giant-step and multiplier-finding phases are split roughly 1-to-2.

Our observed results largely match these predicted results. Rather than seeing an equal three-way split in phases for small discriminants, we have a roughly 40%–40%–

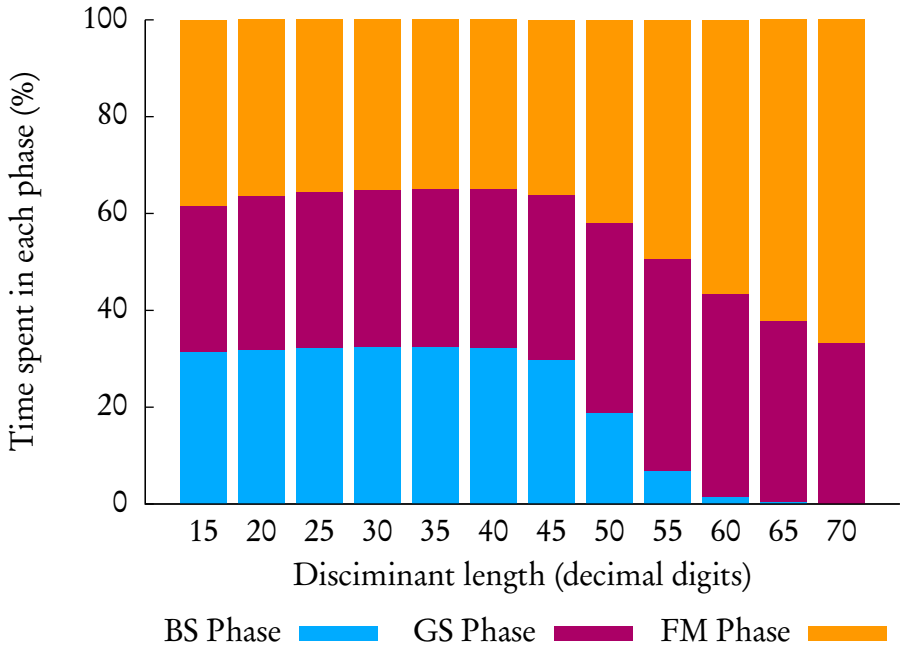


Figure 4.20: Theoretical phase balance for single-processor implementation (LSB-32 hashing).

20% split.[1] Once we exhaust our storage space, we see the predicted growth of the multiplier-finding phase. By the time we reach a 55-digit discriminant, the final phase of the $O(\Delta^{1/6+\epsilon})$ algorithm is consuming over half—roughly 55%—of the total run-time. Manual adjustment of the optimum parameters allows us to balance the baby-step and giant-step phases so that they consume roughly equal percentages of the run-time. But, this tweaking does not alter the percentage of run-time used in the multiplier-finding phase.

The second reason for the reduction in efficiency gains of the ideal hashing algorithm for larger discriminants has to do with the hash function itself, specifically the size of hash we are storing in $\mathcal{L}''$. In a nut shell, a 32-bit hash is insufficient for larger discriminants. Comparing the hash functions chosen for our implementation, we find using the LSB-32 hash gives slightly better performance than using the Lookup3 hash for discriminants less than 40 decimal digits, but the Lookup3 hash wins out for larger discriminants. For the smaller discriminants, the sheer speed of the LSB-32 hash is enough to overcome the small number of random hash collisions that need to be handled by the hash-collision resolution process. As the discriminant size increases, however, the hash collision rate increases significantly due to the non-uniform hash distribution of the LSB-32 hash. During our testing, we observed a $< 0.5\%$ collision rate for discriminants under 35 decimal digits, 2–4% for 35- to 50-digit discriminants, and 5–8% for 50- to 60-digit discriminants. The uniformly distributed output of the Lookup3 hash helps offset this cost and thus it becomes more efficient for larger discriminants, even given its slightly larger computational overhead.

Figures 4.21 and 4.22 show the estimated average run-time of the unmodified algo-

[1] We are ignoring the 15- and 20- digit results because it is hard to accurately measure the times-per-phase given that they are so fast.

rithm and the modified version using both the LSB-32 and Lookup3 hashes. As we expect from the numerical results summarized in Table 4.17, the best-fit curves in Figure 4.21 show that the $O(\Delta^{1/6+\epsilon})$ algorithm using the Lookup3 hash is the most efficient for discriminants in the 30–40 decimal digit range. Ultimately, however, a 32-bit hash is insufficient for the size of discriminants we are most interested in working with, those in the 60 to 70 decimal digit range. Looking at Figure 4.22, the unmodified $O(\Delta^{1/6+\epsilon})$ algorithm is more efficient when we are restricted to storing 32-bit hashes in our hashed baby-step list $\mathcal{L}''$. In fact, the run-time estimate curves for the unmodified algorithm and the Lookup3-based ideal hashing algorithm intersect for discriminants between 51 and 52 decimal digits.

The best efficiency gains for the $O(\Delta^{1/6+\epsilon})$ ideal hashing algorithm are made when we can keep the random hash collision rate below 0.5%. From Table 4.17, our best improvement in run-time comes at the 35 decimal-digit discriminant range: just over a 10% run-time savings. For the LSB-32 and Lookup3 hash functions, taking into account the optimum parameters computed for this discriminant size, the random collision rates are both 0.32%. Moving up to the 40 decimal-digit range gives a collision rate in the 2.75% to 3% range. The overhead in handling this increased number of random hash collisions eliminates our run-time savings. The only way to fix this situation is to increase the size of hash we store in $\mathcal{L}''$.

To better illustrate this point, we have taken a timing measurement for the $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing and artificially increased the random collision rate. We accomplish this by zeroing out a selected number of bits of the computed ideal hash. In essence, we artificially guarantee that those bits of the hashes will always collide with each other. Figure 4.23 shows how this change to the collision rate alters the observed

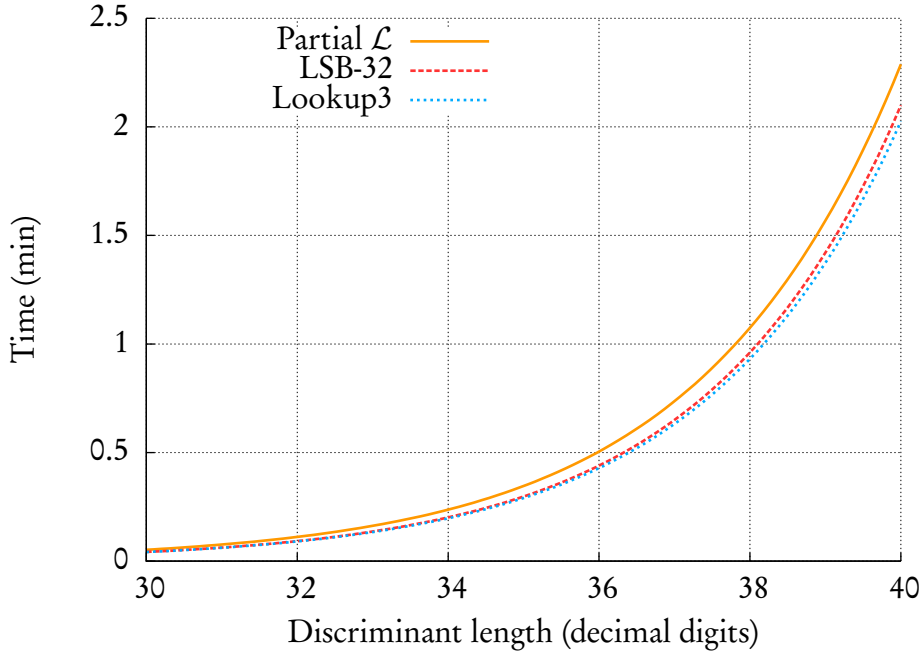


Figure 4.21: Comparison of single-processor implementation run-time estimates for medium-sized discriminants.

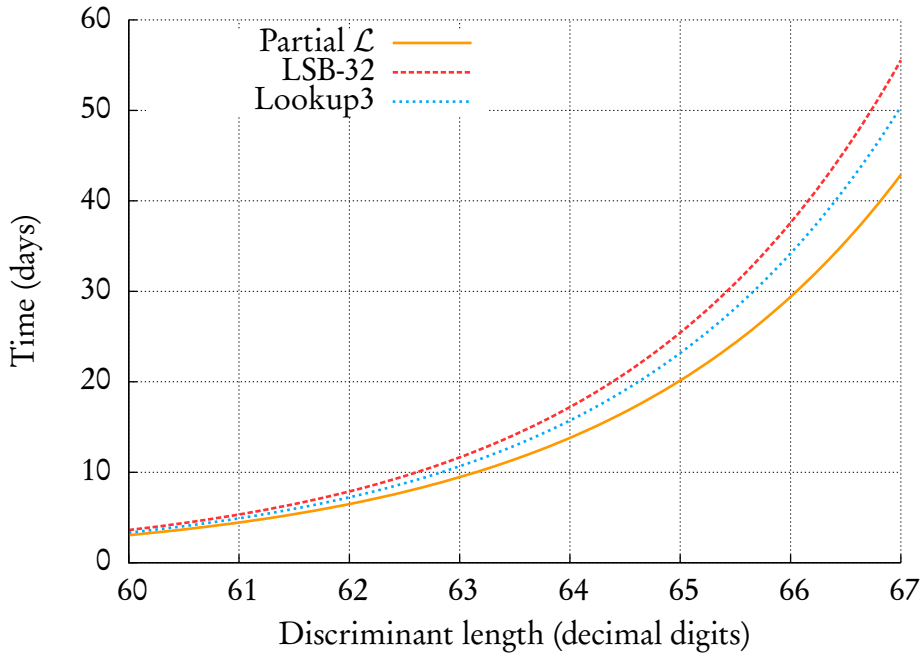


Figure 4.22: Comparison of single-processor implementation run-time estimates for larger discriminants.

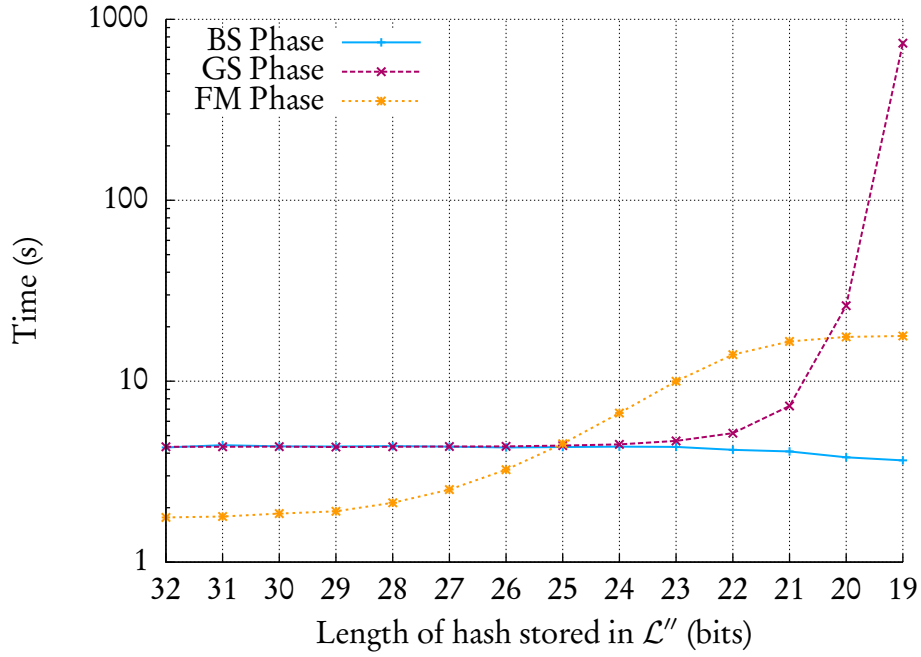


Figure 4.23: Changes in observed timings for an artificially inflated hash collision rate.

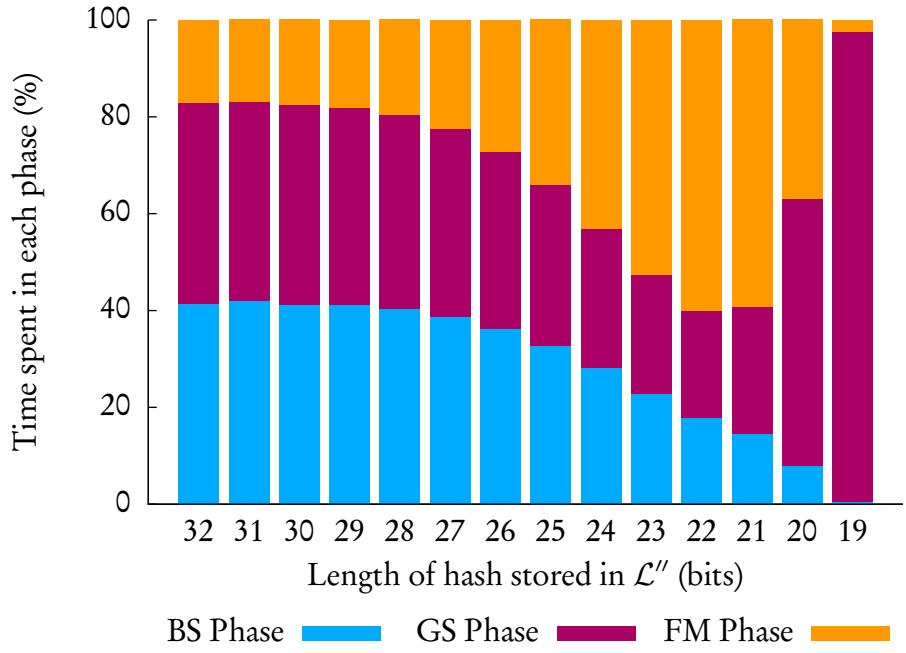


Figure 4.24: Phase balances for artificially inflated hash collision rate.

run-time. Initially we see a very gradual increase in the run-time of the multiplier-finding phase of the algorithm, while the baby-step and giant-step phases remain constant. Eventually, though, the increase in this phase levels off and the giant-step phase begins to dominate the computation. Figure 4.24 shows the same measurements in terms of the percentage of time spent in phase of the $O(\Delta^{1/6+\epsilon})$ algorithm. We point out the similarity between this graph—the middle portion of it, at least—and those in Figures 4.9, 4.13, and 4.17 on pages 115, 118, and 121, respectively. If we were to attempt to verify the regulator associated with a, say, 65-digit discriminant using our Lookup3-based $O(\Delta^{1/6+\epsilon})$ ideal hashing algorithm, we would expect a similar run-time distribution between the three phases as that seen on the right-hand side of Figure 4.24. That is, we would expect the baby-step and multiplier-finding phases to be completely dwarfed by the giant-step phase.

Turning to the parallel implementation, we chose to focus exclusively on the Lookup3 hash as it outperforms, in terms of the random collision rate, the LSB-32 hash for larger discriminant sizes. Due to the suspected hardware problems we encountered with the ISPIA cluster, the majority of our effort was focused on trying to optimize the larger 55- and 60-digit discriminant cases and get any results we could. The aftermath of this can be seen in the numerical timings summarized in Table 4.19 on page 133. While our implementation is significantly faster than the timings for the "regular" parameters given in [22], we are far from being competitive with the timings for "modified" parameters presented there, at least for the small to mid-sized discriminants.

However, we were able to produce a 60-digit discriminant run-time measurement for our $O(\Delta^{1/6+\epsilon})$ ideal hashing algorithm which is sizably smaller than the "modified" parameter timing presented in [22]. Moreover, this timing was recorded using the default

Table 4.18: Observed timings for parallel implementation (100 single-processor nodes, Lookup3 hashing).

	Default parameters				Empirically-adjusted parameters				
$l(\Delta)$	BS Time	GS Time	FM Time	Total	BS Time	GS Time	FM Time	Total	Savings
15	0.00263s	0.000867s	0.00584s	0.00934s	0.00268s	0.000879s	0.00575s	0.00930s	0.40%
20	0.00466s	0.00128s	0.0230s	0.0290s	0.00514s	0.00135s	0.0219s	0.0284s	2.00%
25	0.00308s	0.00255s	0.176s	0.209s	—	—	—	—	0%
30	0.188s	0.00657s	0.846s	1.04s	0.160s	0.00791s	0.871s	1.04s	0.24%
35	1.33s	0.0375s	5.48s	6.84s	0.735s	0.0703s	5.86s	6.67s	2.53%
40	7.06s	0.172s	23.9s	31.1s	3.18s	0.420s	26.3s	29.9s	3.82%
45	56.2s	1.63s	3m 11s	4m 9s	22.5s	3.3s	3m 9s	3m 34s	13.9%
50	6m 34s	15.8s	27m 55s	34m 44s	2m 42s	32.9s	27m 53s	31m 8s	10.4%
55	45m 42s	1m 58s	3h 26m	4h 14m	18m 20s	4m 4s	3h 22m	3h 44m	11.7%
60	6h 40m	13m 49s	1d 21.8h	2d 4.6h					

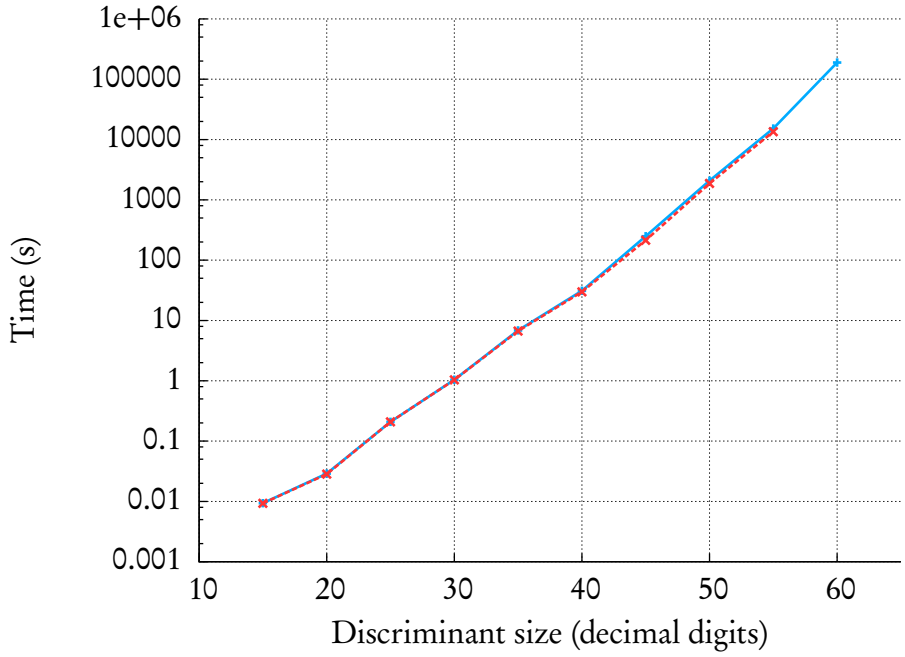


Figure 4.25: Observed timings for parallel implementation (Lookup3 hashing).

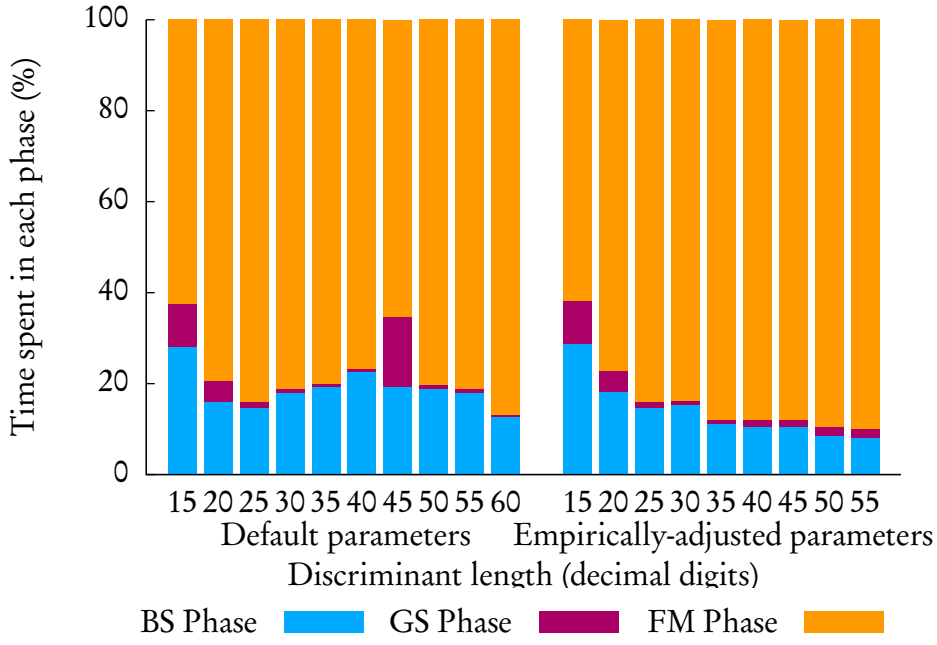


Figure 4.26: Phase balance for parallel implementation (Lookup3 hashing).

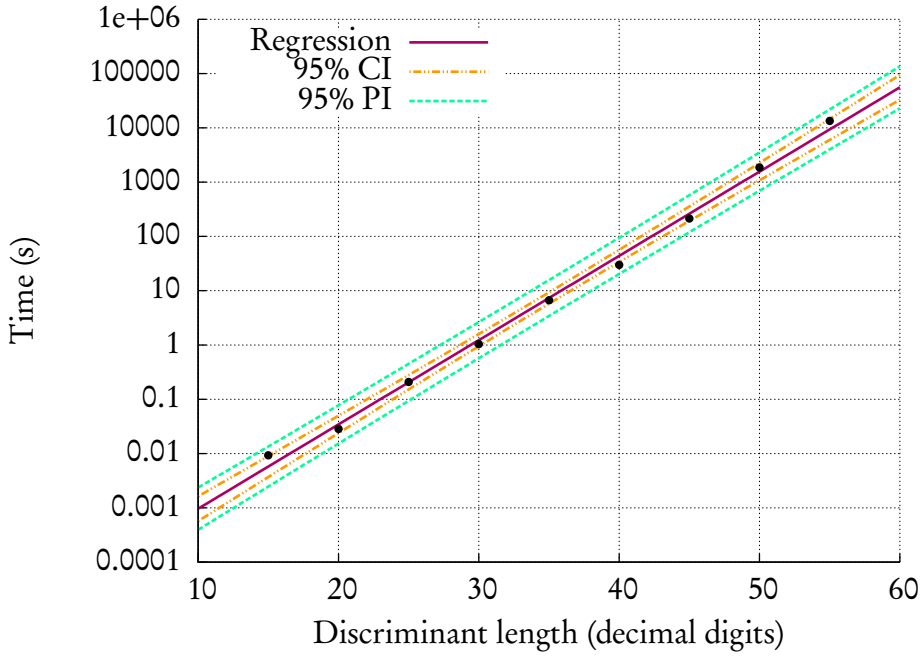


Figure 4.27: Exponential regression for parallel implementation timings (Lookup3 hashing).

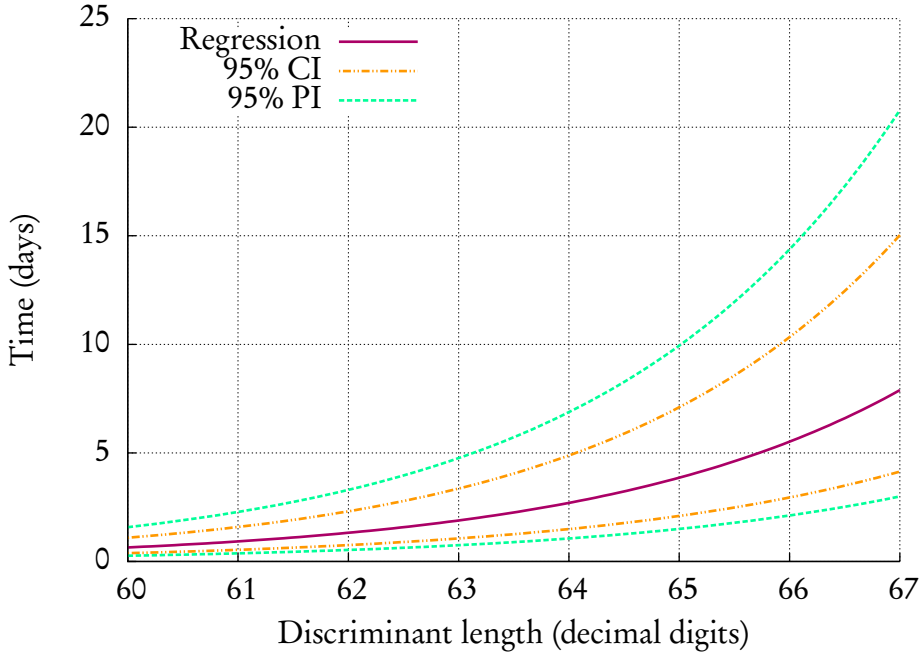


Figure 4.28: Estimated parallel implementation run-time for larger discriminants (Lookup3 hashing).

Table 4.19: Summary of parallel implementation timing results (32-bit).

$l(\Delta)$	[22] Regular	[22] Modified	Lookup3
15	—	—	0.00930s
20	—	—	0.0284s
25	—	—	0.209s
30	—	—	1.04s
35	1m 7s	2.35s	6.67s
40	1m 23s	11.02s	29.9s
45	3m 3s	1m 20s	3m 34s
50	25m 39s	12m 8s	31m 8s
55	5h 41m	2h 39m	3h 44m
60	7d 4h	4d 9h	2d 4.6h†

†: Due to suspected computer hardware issues, we were unable to determine this timing using empirically-adjusted parameters. In its place, we have included the timing determined using the default parameters.

values produced by the optimization formulas, not the empirically refined ones, and so we can expect to reduce this timing further still.

We also see a different situation with respect to the phase run-time balance. Since each node is required to compute its own copy of the baby-step list, we do not see a time reduction in this phase of the algorithm. With the giant steps distributed evenly across the available cluster nodes, we see a corresponding reduction in run-time. In fact, the giant-step phase consumes only a couple of percent of the overall algorithm run-time, at least for larger discriminants. By far, the most expensive phase of the parallelized algorithm is the multiplier-finding phase. For 35-digit and higher discriminants, roughly 90% of the algorithm run-time is spent in this phase.

However, this is at odds with the theoretical predictions. Figure 4.29 shows the theoretical run-time percentages for each phase of the algorithm, assuming we use 100 single-core processors with 32-bit Lookup3 hashing. We see that the optimum parameters computed should split the overall run-time roughly 2-to-1 between the baby-step and multiplier-finding phases of the algorithm. However, once we are unable to store a complete baby-step list, a greater and greater percentage of the run-time is consumed by the giant-step phase. Near the 70 decimal digit discriminant range, the relative time needed for the baby-step phase is negligible, whereas the giant-step and multiplier-finding phases are split roughly 1-to-2, just as in the single-processor case. Using our empirically-adjusted parameters, the relative percentage of the giant-step phase does seem to match the theoretical model. However, our only explanation for the observed 1-to-4 balance between

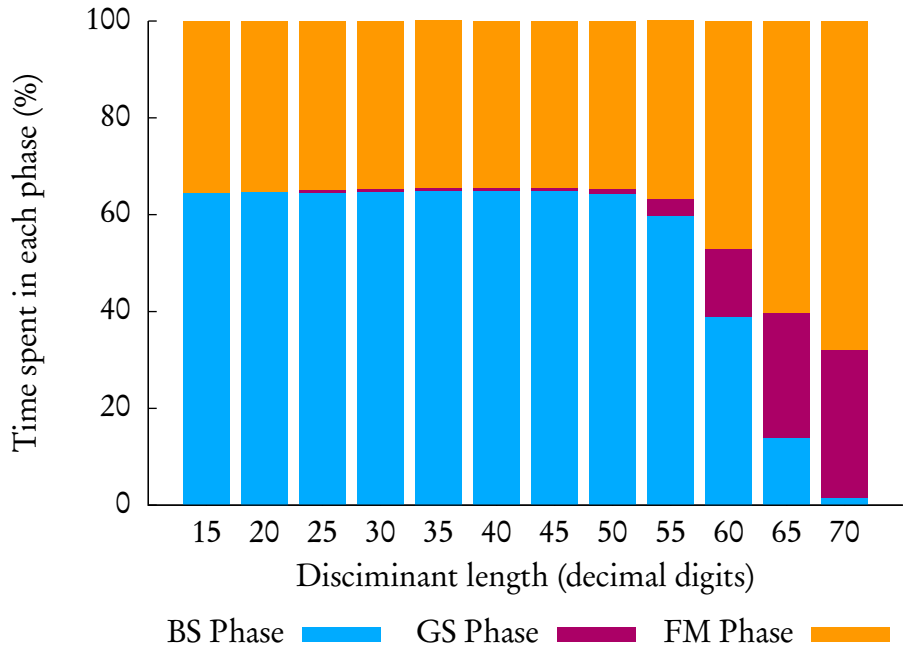


Figure 4.29: Theoretical phase balance for 100 single-core processor implementation (32-bit, Lookup3 hashing).

the baby-step and multiplier-finding phases is that we have a non-uniform distribution of work between the nodes during the later phase.

Because of the difficulties we were encountering with the ISPIA cluster, we decided to move our implementation to a more reliable system. Through Compute Canada, we were able to acquire an account with the Western Canada Research Grid (WestGrid) HPC consortium. This gave us access to a 288-node (2304-core) 64-bit cluster situated in the Research Computing Facility at the University of Victoria. Due to looming deadlines, we decided to focus our efforts exclusively on the 65-digit discriminant from [22]. For our reimplementation of the partial baby-step version of the $O(\Delta^{1/6+\epsilon})$ algorithm, we

Table 4.20: Optimal and empirically refined parameters for $l(\Delta) = 65$ (100 nodes, single processor, partial baby-step list).

Parameter	Theoretically optimal	Empirically refined
s	340193960077	219479974234
Q	2694953928	4177178588
K	1833614097520553081019	—
l	634	—

Table 4.21: Optimal and empirically refined parameters for $l(\Delta) = 65$ (100 nodes, single processor, Lookup3 hashing).

Parameter	Theoretically optimal	Empirically refined
s	249040653446	146494502027
Q	3147382632	5350550474
K	1567652454263416881427	—
l	464	—

Table 4.22: Observed phase balances for $l(\Delta) = 65$ (100 nodes, single processor).

	Run-time percentage	
Phase	Partial $\mathcal{L}$	Lookup3
BS Phase	8.31%	8.83%
GS Phase	0.61%	0.14%
FM Phase	93.24%	91.03%

achieved a run-time of 18 days and 11 hours. Using the hashed baby-step list version of the algorithm, we saw a roughly 4.61% improvement in the overall run-time: it ran in 17 days, 14.5 hours. Tables 4.20 and 4.21 lists the parameters used in these computations and the observed phase balances presented in Table 4.22 are similar to those observed on the ISPIA cluster—see Figure 4.26 on page 131—, which we mentioned above were at odds with the theoretical predictions.

4.5. Concluding Remarks

In this chapter we have presented our modification to the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm: a method for reducing the memory requirements of the algorithm by storing a list of hashes of the baby-step ideals in $\mathcal{L}$, rather than the ideals themselves or a subset of them. The key idea behind the change is that when we are attempting to find a given giant step in the baby-step list $\mathcal{L}'$, we do not actually expect to find a match. Because of this, it is usually sufficient to compare the hash of the giant-step ideal to a list of baby-step ideal hash values. From our single-processor tests, we saw that with an appropriate choice of parameters and hash function, we can achieve roughly a 10% improvement in run-time. Looking at the parallel timing results, it is difficult to say anything definitive about the

algorithm as we push into the 60 decimal-digit discriminant and larger range, at least for the time being. We observed a 4.6% improvement, but further parameter refinements and computations are certainly necessary.

When attempting to apply our modified algorithm to larger discriminants, one must keep two values in mind. The first is the value of the l parameter. If this grows too large, the balance between the baby-step, giant-step, and multiplier-finding phases of the algorithm becomes skewed. In our single-processor numerical tests, we saw the multiplier-finding phase consuming over half of the overall run-time. In the parallel-processor tests, we saw this approaching 90% of the overall run-time. The second value to watch is the rate of random hash collisions between the giant-step ideal hashes and the baby-step ideal hashes stored in $\mathcal{L}''$. Turning to our numerical results again, we find that the maximum collision rate we should allow is around 0.5%. As soon as we begin to exceed this limit, the efficiency gains made are rather quickly diminished by the additional overhead caused by the hash-collision resolution process. At a random collision rate of around 2–3%, we find that our modified algorithm is no faster than the unmodified algorithm.

— Chapter 5 —
New Developments: Compact representations

5.1. Introduction

Our next chapter presents several modifications to the idea of a compact representation introduced in Section 2.13. The first of these involves a change to CRAX which reduces the heights of the individual compact representation terms. This is discussed in Section 5.2. Section 5.3 outlines an adjustment to the definition of a compact representation which allows it to be computed in fewer iterations. As we will see in Section 5.4, a combination of these changes gives us a compact representation that is, quite likely, as small as we can expect to achieve with these methods. These analytical results are backed up by various numerical computations which we present in Section 5.5.

5.2. Reducing memory usage in compact representations

Recall from Section 2.13 that the compact representation of $\theta \in \mathcal{O}_{\mathbb{K}}$, where $(\theta) = \mathfrak{a}[x]$, is given by

$$\theta = \prod_{i=0}^{l} \left(\frac{\lambda_i}{L_i^2} \right)^{2^{l-i}}$$

where $\lambda_i = (m_i + n_i\sqrt{\Delta})/r$ $(m_i, n_i \in \mathbb{Z})$ and $L_i \in \mathbb{Z}$ such that $(L_i^2)\mathfrak{a}_{i+1} = (\lambda_i)\mathfrak{a}_i^2$. Also in that section, we determined a bound on the size of the λ_i, specifically,

$$1 \leq H(\lambda_i) \leq \frac{5}{2}\Delta\,.$$

We would like to reduce this upper bound and so reduce the amount of memory needed to store a compact representation of θ.

Consider for a moment the sequence of s_i values computed as AX executes. These are a sequence of values corresponding to the intermediate results produced by applying a square-and-multiply process according to the binary representation of x. Let $x = \sum_{i=0}^{l} 2^{l-i} b_i$ be such a representation and set $s_0 = b_0 \ (= 1)$. As we progress through AX computing giant steps, ideally we wish to compute

$$\mathfrak{a}[s_{i+1}]' = \mathfrak{a}[s_i]^2 .$$

However because of the way EADDXY[1] works—recall (3.1), page 52—, when we compute $\mathfrak{a}[s_i]^2$ we actually "fall short" of this ideal, computing instead

$$\mathfrak{a}[s_{i+1}]' = (\mu_i)\widetilde{\mathfrak{a}}[s_{i+1}] = (\mu_i)\mathfrak{a}[s_i]^2 .$$

See Figure 5.1 for an illustration of this process. Looking back at (2.14), page 34, it is the error term in computing a giant step which throws us off. The relative generator μ_i returned by EADDXY can thus be thought of as a correction factor. We then take the ideal $\mathfrak{a}[s_{i+1}]'$ and, depending on the value of b_i, either set $\mathfrak{a}[s_{i+1}] = \mathfrak{a}[s_{i+1}]'$ or compute a baby-step $\mathfrak{a}[s_{i+1}] = \rho(\mathfrak{a}[s_{i+1}]') = (\nu_i)\mathfrak{a}[s_{i+1}]'$. We also combine the relative generators μ_i and ν_i into a single generator λ_i so that

$$\mathfrak{a}[s_{i+1}] = (\mu_i \cdot \nu_i)\widetilde{\mathfrak{a}}[s_{i+1}] = (\lambda_i)\mathfrak{a}[s_i]^2 .$$

[1] Actually, it is (E)NUCOMP in particular that causes this.

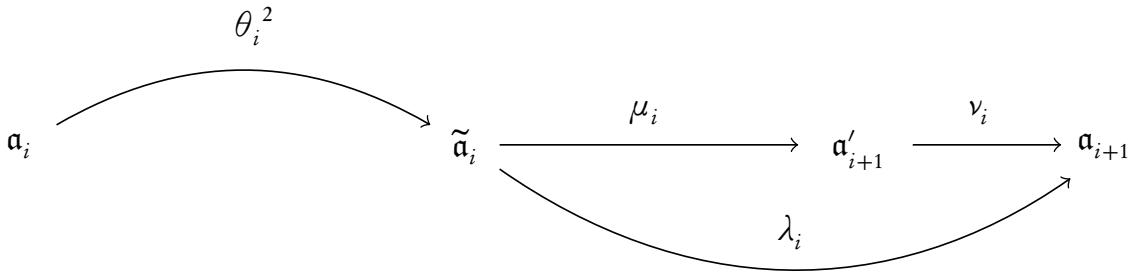


Figure 5.1: The main loop of CRAX, as presented in [48, Alg. 12.4, pp. 287–8].

As we also mentioned in Section 2.13, these μ_i values constitute the majority of the λ_i terms that we wish to store as a compact representation. So what if we could reduce the size of μ_i by some reasonably large amount? With some careful reasoning [48, pp. 445–6], one can show

$$\frac{\sqrt{2/r}\Delta^{1/4}}{q_i+3} < \mu_i < \frac{2\left(\sqrt{2/r}\Delta^{1/4}\right)^3}{q_i N(\mathfrak{a}[s_{i+1}])},$$

where $\mathfrak{a}[s_i] = [Q_i/r, (P_i+\sqrt{\Delta})/r]$ and $q_i = \lfloor (P_i+\sqrt{\Delta})/Q_i \rfloor$. The important point to take away from this inequality is that

$$O(\Delta^{1/4}) < \mu_i < O(\Delta^{3/4}). \tag{5.1}$$

In other words, while the relative generator μ_i is bounded and cannot become too large, it also cannot become very small.

In the following text, we will describe a method to adjust the s_i values—and hence the inputs to EADDXY—to exploit the short-fall we experience and so reduce the upper bound in (5.1). If we increase the s_i values at each step, we will compute ideals $\mathfrak{a}[s_{i+1}]'$ further along the infrastructure than we want. As before, we still experience a short-fall and end up computing $\widetilde{\mathfrak{a}}[s_{i+1}]$ instead. The key thing to note, however, is that the ideal

$\widetilde{\mathfrak{a}}[s_{i+1}]$ is now much closer to our goal of $\mathfrak{a}[s_{i+1}]$ than it was before. By using a larger and backwards EWNEAR step, we use the relative generator ν_i to cancel out, in a sense, a substantial portion of μ_i. See Figure 5.2 for an illustration of this idea.

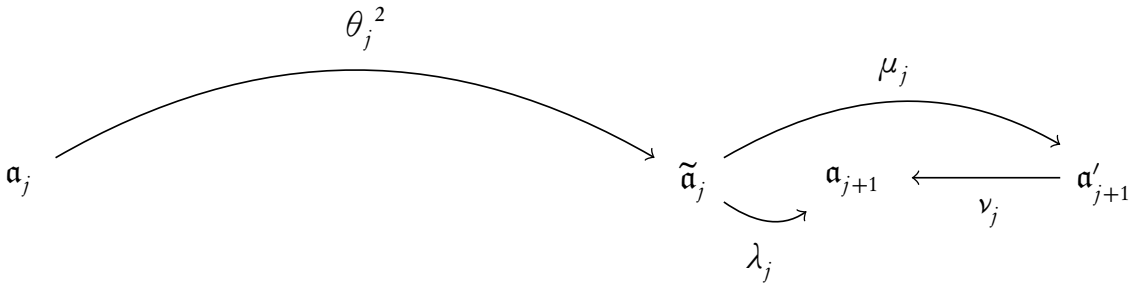


Figure 5.2: Proposed change to CRAX.

Let $h \in \mathbb{Z}^+$ and set $y = x + h$ assuming $x \geq 3h$. If we replace the binary representation of x in Step 1 of CRAX with that of y, we quite obviously end up computing a compact representation of $\mathfrak{a}[y]$. Now, regardless of the value of y we start with, at the end of CRAX we want to have computed a compact representation of $\mathfrak{a}[x]$. So we have to correct for the "$+h$" we have added by appropriately inserting a "$-h$." Looking at the last two iterations of the while-loop in Step 4 of CRAX, we compute

$$s_{l-1} = 2s_{l-2} + b_{l-1} = 2\Big(\sum_{i=0}^{l-2} 2^{l-2-i} b_i\Big) + b_{l-1} = \sum_{i=0}^{l-1} 2^{l-1-i} b_i \,,$$

and

$$s_l = 2s_{l-1} + b_l = \sum_{i=0}^{l} 2^{l-1} b_i = y = x + h \,;$$

clearly in this last computation is where we will need the "$-h$." We split the while-loop in two, iterating the first loop over $0 \leq i \leq l-2$ and the second over the singleton $i = l-1$.

Furthermore, for the second "loop," we modify Step 6 to set $s_{i+1} = 2s_i - h$. Thus when we compute s_l, we find $s_l = y - h = x + h - h = x$.

Consider if we were to add a "$-h$" to both the s_{l-1} and s_l terms and modified the while-loops accordingly. Then we find

$$s_{l-1} = 2s_{l-2} + b_{l-1} - h = 2\Big(\sum_{i=0}^{l-2} 2^{l-2-i} b_i\Big) + b_{l-1} - h = \sum_{i=0}^{l-1} 2^{l-1-i} b_i - (2^1 - 1)h\,,$$
$$s_l = 2s_{l-1} + b_l - h = \Big(\sum_{i=0}^{l} 2^{l-1} b_i - 2h\Big) - h = \sum_{i=0}^{l} 2^{l-1} b_i - (2^2 - 1)h\,,$$

which leads us to the generalized form of this modification. Let n be the largest integer such that $x \geq (2^n - 1)h$ and set $y = x + (2^n - 1)h$. We iterate the first while-loop over $0 \leq i < l - n$ and the second over $l - n \leq i < l$, with the modification to Step 6 as mentioned previously. Considering the s_i values computed, we have

$$\begin{aligned}
s_0 &= b_0\,,\\
s_1 &= 2s_0 + b_1 = 2b_0 + b_1\,,\\
&\vdots\\
s_{l-n-1} &= \sum_{i=0}^{l-n-1} 2^{l-n-1-i} b_i\,,\\
s_{l-n} &= 2s_{l-n-1} + b_{l-n} - h = \sum_{i=0}^{l-n} 2^{l-n-i} b_i - (2^1 - 1)h\,,\\
s_{l-n+1} &= 2s_{l-n} + b_{l-n+1} - h = 2\Big(\sum_{i=0}^{l-n} 2^{l-n-i} b_i - h\Big) + b_{l-n+1} - h\,,\\
&= \sum_{i=0}^{l-n+1} 2^{l-n+1-i} b_i - 2h - h = \sum_{i=0}^{l-n+1} 2^{l-n+1-i} b_i - (2^2 - 1)h\,,\\
&\vdots
\end{aligned}$$

$$s_{l-1} = \sum_{i=0}^{l} 2^{l-1-i} b_i - (2^{n-1}-1)h \,,$$
$$s_l = \sum_{i=0}^{l} 2^{l-i} b_i - (2^n-1)h = y - (2^n-1)h = x \,.$$

Thus at the end of CRAX, we will still compute $\mathfrak{a}[s_l] = \mathfrak{a}[x]$ as desired. All that remains is to determine an appropriate value for h and from that, determine how much the size of λ_i can be reduced.

Recalling (5.1), we can suppose that a reasonable maximal value for h would be something of size approximately $(1/4)\log_2 \Delta$. So, let $h = \lceil (1/4)\log_2 \Delta \rceil$. In order to determine how much λ_i is reduced, we must compute a revised bound for $H(\lambda_i)$. As CRAX executes, it finds a series of reduced principal $\mathcal{O}_{\mathbb{K}}$-ideals $\mathfrak{a}[s_i] = \mathfrak{a}_i = (\alpha_i)\mathfrak{a}_1$. For any fixed i $(1 \le i \le l)$, we must have some θ_j in the simple continued fraction expansion of $\sqrt{\Delta}$ such that $\alpha_i = \theta_j$. Recalling Lemma 2.9, we can conclude that for two consecutive ideals $\mathfrak{a}[s_{i-1}]$ and $\mathfrak{a}[s_i]$,

$$\frac{15N(\mathfrak{a}_i)}{16\sqrt{\Delta}} 2^{s_{i-1}} < \alpha_{i-1} < \frac{17}{16} 2^{s_{i-1}} \quad \text{and} \quad \frac{15L_{i+1}}{16\sqrt{\Delta}} 2^{s_i - h} < \alpha_i < \frac{17}{16} 2^{s_i - h} \,. \tag{5.2}$$

From the definition of a compact representation, we also know that

$$\alpha_{i+1} = \left(\frac{\lambda_{i+1}}{L_{i+1}^2}\right) \alpha_i^2 \quad \Longrightarrow \quad \lambda_i = \frac{L_i^2 \alpha_i}{\alpha_{i-1}^2}, \tag{5.3}$$

and so combining (5.2) and (5.3), we get

$$\lambda_i < L_i^2 \left(\frac{17}{16} 2^{s_i - h}\right) \left(\frac{16\sqrt{\Delta}}{15 L_i} 2^{-s_{i-1}}\right)^2 = \frac{16^2 \cdot 17}{15^2 \cdot 16} 2^{s_i - h - 2s_{i-1}} \Delta = \frac{16 \cdot 17}{15^2} 2^{s_i - 2s_{i-1} - h} \Delta, \tag{5.4}$$

and since $s_i - 2s_{i-1} \in \{0, 1\}$

$$\lambda_i < \frac{5}{2} 2^{-\lceil (1/4)\log_2 \Delta \rceil} \Delta \leq \frac{5}{2} \Delta^{-1/4} \Delta = \frac{5}{2} \Delta^{3/4}.$$

Hence, our modification to CRAX reduces the height of λ_i from $O(\Delta)$ to $O(\Delta^{3/4})$.

Before presenting our modified version of CRAX, we must deal with a technical issue. The algorithm EWNEAR as presented in [48, Alg. 12.1, pp. 286 and 457] is restricted to the case when $k < w$. With the changes to CRAX proposed above, we require EWNEAR to function in the $k > w$ case as well. Below we present our modifications of EWNEAR to allow this. As we are merely adding some key values which allow the determination of a relative generator, the proof of correctness of EWNEAR will remain unchanged.

Algorithm 5.1: EWNEAR

Input: $(\mathfrak{b}, d, k)$, w, p, where $(\mathfrak{b}, d, k)$ is a reduced (f, p) representation of some $\mathcal{O}$-ideal $\mathfrak{a}$. Here $\mathfrak{b}[Q/r, (P + \sqrt{D})/r]$, where $P + \lfloor \sqrt{D} \rfloor \geq Q$, $0 \leq \lfloor \sqrt{D} \rfloor - P \leq Q$.

Output: $(\mathfrak{c}, g, h)$ a w-near $(f + 9/8, p)$ representation of $\mathfrak{a}$ and a, b, where $\kappa = (a + b\sqrt{D})/Q$ and $\mathfrak{c} = \kappa\mathfrak{b}$.

1: **case 1:** $k < w$
2: Put $B_{-2} = 1, B_{-1} = 0$.
3: Find $s \in \mathbb{Z}^{\geq 0}$ such that $2^s Q \geq 2^{p+4}$. Put $Q_0 = Q$, $P_0 = P$, $M = \lceil 2^{p+s-k+w} Q_0/d \rceil$, $Q_{-1} = (D - P^2)/Q$, $T_{-2} = -2^s P_0 + \lfloor 2^s \sqrt{D} \rfloor$, $T_{-1} = 2^s Q_0$, $i = 1$.
4: **while** $T_{i-2} \leq M$ **do**
5: $q_{i-1} = \lfloor (P_{i-1} + \lfloor \sqrt{D} \rfloor)/Q_{i-1} \rfloor$
6: $P_i = q_{i-1} Q_{i-1} - P_{i-1}$
7: $Q_i = Q_{i-2} - q_{i-1}(P_i - P_{i-1})$
8: $T_{i-1} = q_{i-1} T_{i-2} + T_{i-3}$
9: $B_{i-1} = q_{i-1} B_{i-2} + B_{i-3}$
10: $i \leftarrow i + 1$
11: **end while**
12: Put $e_{i-1} = \lceil 2^{p-s+3} T_{i-3}/Q_0 \rceil$
13: **if** $d e_{i-1} \leq 2^{2p-k+w+3}$ **then**
14: Put $\mathfrak{c} = [Q_{i-2}/r, (P_{i-2} + \sqrt{D})/r]$, $e = e_{i-1}$,

$a = (T_{i-3} - \lfloor 2^s\sqrt{D}\rfloor)/2^s$, $b = B_{i-3}$.

else

Put $\mathfrak{c} = [Q_{i-3}/r, (P_{i-3} + \sqrt{D})/r]$, $e = \lceil 2^{p-s+3} T_{i-4}/Q_0 \rceil$, $a = (T_{i-4} - \lfloor 2^s\sqrt{D}\rfloor)/2^s$, $b = B_{i-4}$.

end if

Find t such that

$$2^t < \frac{ed}{2^{2p+3}} \le 2^{t+1} .$$

Put

$$g = \left\lceil \frac{ed}{2^{p+t+3}} \right\rceil , \quad h = k + t .$$

end case

case 2: $k > w$

Put $B^*_{-2} = 1$, $B^*_{-1} = 0$.

Put $s = p + 4$, $Q^*_0 = Q$, $P^*_0 = P$, $M^* = d2^{k-w+4}$, $Q^*_1 = (D - P^2)/Q$, $T^*_{-2} = 2^s Q^*_0$, $T^*_{-1} = 2^s P^*_0 + \lfloor 2^s\sqrt{D}\rfloor$, and $i = 1$.

while $T^*_{i-2} < Q^*_i M^*$ **do**

$q^*_i = \lfloor (P^*_{i-1} + \lfloor\sqrt{D}\rfloor)/Q^*_i \rfloor$

$P^*_i = q^*_i Q^*_i - P^*_{i-1}$

$Q^*_{i+1} = Q^*_{i-1} - q^*_i (P^*_i - P^*_{i-1})$

$T^*_{i-1} = q^*_i T^*_{i-2} + T^*_{i-3}$

$B^*_{i-1} = q^*_i B^*_{i-2} + B^*_{i-3}$

$i \leftarrow i + 1$

end while

Put $q^*_i = \lfloor (P^*_{i-1} + \lfloor\sqrt{D}\rfloor)/Q^*_i \rfloor$, $P^*_i = q^*_i Q^*_i - P^*_{i-1}$, $e = \lceil T^*_{i-2}/2Q^*_i \rceil$, $e' = \lceil T^*_{i-3}/2Q^*_{i-1} \rceil$, $j = 3$.

while $e' \ge d2^{k-w+3}$ **do**

$e \leftarrow e'$

$e' \leftarrow \lceil T_{i-2-j}/2Q^*_{i-j} \rceil$

$j \leftarrow j + 1$

end while

Find t (t') such that

$$2^{t-1} \le \frac{e}{8d} < 2^t . \quad \left(2^{t'-1} \le \frac{e'}{8d} < 2^{t'} . \right)$$

Put $\mathfrak{c} = [Q^*_{i-j+3}/r, (P^*_{i-j+3} + \sqrt{D})/r]$, $g = \lceil 2^{p+3+t} d/e \rceil$, $h = k - t$, $a = (T^*_{i-2} - B^*_{i-2}\lfloor 2^s\sqrt{\Delta}\rfloor)/2^s$, $b = B^*_{i-j+2}$.

40: **end case**

Our modified algorithm, HCRAX, is presented in Algorithm 5.2. Its correctness follows from the preceding discussion.

Algorithm 5.2: HCRAX

Input: x, p, where $x \in \mathbb{Z}^+$ and $2^p > 11.2x \max\{16 \log_2 x\}$.

Output: $(\mathfrak{a}[x], d, k)$, (m_i, n_i), and L_i, where $(\mathfrak{a}[x], d, k)$ is an x-near (f, p) representation of $\mathfrak{a} = (1)$ with $f < 2^{p-4}$, (m_i, n_i) are pairs of integers, and $L_i \in \mathbb{Z}^+$ for $i = 0, 1, \ldots, l$ where l is such that $x = \sum_{j=0}^{l} 2^{l-j} b_j$ and $b_0 = 1$, $b_j \in \{0, 1\}$.

1: Put $h = \lceil (1/4) \log_2 \Delta \rceil$, compute the maximal n such that $x/(2^n - 1) \geq h$, and put $y = x + (2^n - 1)h$.

2: Compute the binary representation of y with

$$y = \sum_{i=0}^{l} 2^{l-i} b_i \text{ and } b_0 = 1, b_i \in \{0, 1\} \quad (1 \leq i \leq l).$$

3: Put

$$Q = r, \qquad P = r \left\lfloor \frac{\lfloor \sqrt{\Delta} \rfloor - r + 1}{r} \right\rfloor + r - 1, \qquad (\mathfrak{b}, d, k) = ([Q, P], 2^p + 1, 0),$$

$s = b_0$, $L_0 = 1$, and $i = 0$.

4: Put $((\mathfrak{b}_0, d_0, k_0), m_0, n_0) = \text{EWNEAR}((\mathfrak{b}, d, k), s, p)$.

5: **while** $i < l - n$ **do**

6: Put $L_{i+1} = N(\mathfrak{b}_i)$ and

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m_{i+1}, n_{i+1}) = \text{EADDXY}((\mathfrak{b}_i, d_i, k_i), (\mathfrak{b}_i, d_i, k_i), s, s, p).$$

7: Set $s \leftarrow 2s + b_{i+1}$.

8: **if** $b_{i+1} \neq 0$ **then**

9: Put $N = N(\mathfrak{b}_{i+1})$ and set

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m'_{i+1}, n'_{i+1}) \leftarrow \text{EWNEAR}((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), s, p).$$

10: Set $(m_{i+1}, n_{i+1}) \leftarrow \text{IMULT}(m_{i+1}, n_{i+1}, m'_{i+1}, n'_{i+1}, N)$.

11: **end if**

12: Set $i \leftarrow i + 1$.

13: **end while**

14: **while** $i < l$ **do**
15: Put $L_{i+1} = N(\mathfrak{b}_i)$ and

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m_{i+1}, n_{i+1}) = \text{EADDXY}((\mathfrak{b}_i, d_i, k_i), (\mathfrak{b}_i, d_i, k_i), s, s, p).$$

16: Set $s \leftarrow 2s + b_{i+1} - h$.
17: Put $N = N(\mathfrak{b}_{i+1})$ and set

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m'_{i+1}, n'_{i+1}) \leftarrow \text{EWNEAR}((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), s, p).$$

18: Set $(m_{i+1}, n_{i+1}) \leftarrow \text{IMULT}(m_{i+1}, n_{i+1}, m'_{i+1}, n'_{i+1}, N)$.
19: Set $i \leftarrow i + 1$.
20: **end while**
21: Put $L_{l+1} = N(\mathfrak{b}_l)$ and $(\mathfrak{a}[x], d, k) = (\mathfrak{b}_l, d_l, k_l)$.

Theorem 5.3. *Let $\theta \in \mathcal{O}_{\mathbb{K}}$ such that $\mathfrak{a}[x] = (\theta)$ for some $x \in \mathbb{Z}^+$. The total number of bits required to express θ as an h-compact representation is* $\mathrm{O}((\log_2 \log_2 \theta) \log_2 \Delta^{3/4})$.

Proof. From the discussion preceding Algorithm 5.2, we know $H(\lambda_i) < (5/2)\Delta^{3/4}$. As $l = \lceil \log_2 x \rceil$ and $2^x < (16\sqrt{\Delta}/15)\theta$, we also have $l = \mathrm{O}(\log_2 \log_2 \theta)$. Thus, we require

$$\mathrm{O}(l \log_2 \Delta^{3/4}) = \mathrm{O}((\log_2 \log_2 \theta) \log_2 \Delta^{3/4})$$

bits to express θ as an h-compact representation. □

Returning once more to our running example, we can use HCRAX to produce the h-compact representation of the fundamental unit $\eta_{410286423278424}$ shown in Table 5.1. We invite the reader to compare it to the compact representation given by CRAX in Table 2.1, page 47. The representation in Table 5.1 uses only 974 bits, a substantial size reduction of 19.6% as compared to the standard compact representation.

Table 5.1: h-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i	i	m_i	n_i	L_i
5	1600186729	79	1	13	-11963646785	3648	20212201
6	126127662947	6188	12924743	14	-58391253311	2973	13014671
7	-6712468796	341	1183897	15	14890719660	755	1280375
8	-766498224	248	1891592	16	55817571288	-2016	7405320
9	16814222688	-684	6888180	17	-1791778223	4850	26406361
10	3566127224	-64	1912936	18	22988634517	-439	13835951
11	-2186792333	165	3016057	19	26413208897	1304	2347585
12	28814796035	1414	702239	20	—	—	1

5.3. Further reducing memory usage in compact representations

We saw in the last section a modification to the compact representation presented in [48, §11.1] which reduces the height of the individual terms to $O(\Delta^{3/4})$. An obvious question to ask is if we can do better than this? Trying to reduce the size of the individual terms further will be extremely hard, but what if we could reduce the total number of terms? Recalling CRAX again, for each step of the algorithm we compute an ideal ($\mathfrak{a}[s_{i+1}]$) at double the distance of the ideal we are currently at ($\mathfrak{a}[s_i]$), then store the relative generator between these two ideals (λ_i). In order to store fewer terms, we have to progress further from ideal to ideal. So what if we were to, say, compute an ideal at triple the distance we are at currently? In other words, instead of computing the binary expansion of x and applying a square-and-multiply routine, what if we computed a ternary expansion and used a cube-and-multiply routine?

The main hurdle to overcome is the cubing step. Thankfully Imbert, Jacobson, and

Schmidt have recently presented a dedicated ideal cubing algorithm: NUCUBE [41, Alg. 4]. We present this algorithm below and note that in addition to correcting a few minor typos, the pseudocode and formulas have been reformatted to match the style of presentation used in [48]. This was done to help the reader more clearly see how the formulas in NUCOMP were extended from computing an ideal product to computing an ideal cube. It is important to note, however, that these formulas are not the most efficient ones. Vanessa Dixon has presented a version of this algorithm [26] using the structure and formulas from [41, Alg. 4] which have been optimized to exploit the fact that $Q' = Q''$ and $P' = P''$. We encourage reimplementations to use Steps 1–22 from [26, Alg. 3.5, p. 45] instead of Steps 1–16 below. Finally, we also note that the algorithm has been extended to compute a relative generator as NUCOMP does.

Algorithm 5.4: NUCUBE

Input: $\mathfrak{a} = [Q'/r, (P' + \sqrt{D})/r]$, a reduced invertible $\mathcal{O}_{\mathbb{K}}$-ideal.

Output: A reduced $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{b} = [Q/r, (P + \sqrt{D})/r]$ such that $\mathfrak{b} \sim \mathfrak{a}^3$. (Optional output: A, B, C where $\mu = |(A + B\sqrt{D})/C|$ and $\mu\mathfrak{b} = \mathfrak{a}^3$.)

1: Compute $S = (Q'/r, 2P'/r)$ and solve $V(Q'/r) + Y(2P'/r) = S$ for $V, Y \in \mathbb{Z}$.

2: Put

$$R' = \frac{D - P'^2}{Q'}, \qquad Q'' = \frac{Q'^2}{rS^2}, \qquad P'' \equiv P' + \frac{YR'Q'}{rS} \pmod{Q''}.$$

3: Compute $S' = (SQ'/r, S(P' + P'')/r)$ and solve $K(SQ'/r) + L(S(P' + P'')/r) = S'$ for $K, L \in \mathbb{Z}$.

4: Put $U \equiv K(P'' - P') + LR' \pmod{Q''/S}$ where $0 \leq U < Q''/S$.

5: Put $R_{-2} = R_0 = K$, $R_{-1} = L$, $C_{-2} = C_0 = -1$, $C_{-1} = 0$, and $i = -1$.

6: **if** $R_{-1} < \lfloor \sqrt{Q'/r^2} D^{1/4} \rfloor$ **then**

7: Put

$$Q_{i+1} = Q'^3/rS^2, \qquad P_{i+1} = UQ'/rS + P' \pmod{Q_{i+1}}.$$

8: Go to Step 17.

9: **end if**

10: **while** $R_i > \lfloor \sqrt{Q'/r^2} D^{1/4} \rfloor$ **do**
11: $i \leftarrow i+1$.
12: $q_i = \lfloor R_{i-2}/R_{i-1} \rfloor$.
13: $R_i = R_{i-2} - q_i R_{i-1}$.
14: $C_i = C_{i-2} - q_i C_{i-1}$.
15: **end while**
16: Put $\widetilde{P} \equiv P' + UQ'/r \pmod{Q/S}$ and

$$M_1 = \frac{(Q'/rS)R_i + (\widetilde{P} - P')C_i}{Q/S}, \qquad M_2 = \frac{(P' + \widetilde{P})R_i + rSR'C_i}{Q/S},$$
$$Q_{i+1} = (-1)^{i-1}(R_i M_1 - C_i M_2), \qquad P_{i+1} = \frac{(Q'/rS)R_i + Q_{i+1}C_{i-1}}{C_i} - P'.$$

17: Put $j = 1$,

$$Q'_{i+1} = |Q_{i+1}|, \qquad k_{i+1} = \left\lfloor \frac{\lfloor \sqrt{D} \rfloor - P_{i+1}}{Q'_{i+1}} \right\rfloor, \qquad P'_{i+1} = k_{i+1}Q'_{i+1} + P_{i+1}.$$

($B_{i-1} = |C_{i-1}|, B_i = |C_i|$.)
18: **if** $P'_{i+1} + \lfloor \sqrt{D} \rfloor < Q'_{i+1}$ **then**
19: Set $j \leftarrow 2$ and put

$$q_{i+1} = \left\lfloor \frac{P_{i+1} + \lfloor \sqrt{D} \rfloor}{Q'_{i+1}} \right\rfloor, \qquad P_{i+2} = q_{i+1}Q'_{i+1} - P_{i+1},$$
$$Q_{i+2} = \frac{D - P_{i+2}^2}{Q'_{i+1}}, \qquad Q'_{i+2} = |Q_{i+2}|,$$
$$k_{i+2} = \left\lfloor \frac{\lfloor \sqrt{D} \rfloor - P_{i+2}}{Q'_{i+2}} \right\rfloor, \qquad P'_{i+2} = k_{i+2}Q'_{i+2} + P_{i+2}.$$

($B_i \leftarrow \operatorname{sign}(Q_{i+1})|C_i|, B_{i+1} = q_{i+1}B_i + B_{i-1}$.)
20: **if** $P'_{i+2} + \lfloor \sqrt{D} \rfloor < Q'_{i+2}$ **then**
21: Set $j \leftarrow 3$ and put

$$q_{i+2} = \left\lfloor \frac{P_{i+2} + \lfloor \sqrt{D} \rfloor}{Q'_{i+2}} \right\rfloor, \qquad P_{i+3} = q_{i+2}Q'_{i+2} - P_{i+2},$$

$$Q_{i+3} = \frac{D - P_{i+3}^2}{Q'_{i+2}}, \qquad Q'_{i+3} = |Q_{i+3}|,$$
$$k_{i+3} = \left\lfloor \frac{\lfloor \sqrt{D} \rfloor - P_{i+3}}{Q'_{i+3}} \right\rfloor, \qquad P'_{i+3} = k_{i+3} Q'_{i+3} + P_{i+3}.$$

$(B_{i+2} = q_{i+2} B_{i+1} + B_i.)$

22: **end if**

23: **end if**

24: Put $\mathfrak{b} = [Q'_{i+j}/r, (P'_{i+j} + \sqrt{D})/r]$. $(A = S(Q_{i+j} B_{i+j-2} + P_{i+j} B_{i+j-1}), B = -S B_{i+j-1}, C = Q_{i+j}.)$

It is relatively straight-forward to turn NUCUBE into an algorithm that can work with (f, p) representations. We begin with the following theorem for determining an (f, p) representation of the cube of an (f, p) representation.

Theorem 5.5 ([48, Thm. 11.2, p. 268])**.** *Let $(\mathfrak{b}, d', k')$ be an (f', p) representation of an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$. If $d'^3 \leq 2^{3p+1}$, put $d = \lceil d'^3/2^{2p} \rceil$ and $k = 3k'$. If $2^{3p+1} < d'^3 \leq 2^{3p+2}$, put $d = \lceil d'^3/2^{2p+1} \rceil$ and $k = 3k' + 1$. If $d'^3 > 2^{3p+2}$, put $d = \lceil d'^3/2^{2p+2} \rceil$ and $k = 3k' + 2$. Then $(\mathfrak{b}^3, d, k)$ is an (f, p) representation of the product ideal $\mathfrak{a}^3$, where $f = 1 + 3f' + 3f'^2/2^p + f'^3/2^{2p}$.*

Proof. Let $\mathfrak{b} = \theta\mathfrak{a}$ for $\theta \in \mathbb{K}$. By the definition of d in the theorem, it is easy to see that $2^p < d \leq 2^{p+1}$. From the definition of an (f, p) representation, we know

$$\left| \frac{2^{p-k'}\theta}{d'} - 1 \right| < \frac{f'}{2^p},$$

and rearranging this inequality gives

$$\frac{d'}{2^p}\left(1-\frac{f'}{2^p}\right) < \frac{\theta}{2^{k'}} < \frac{d'}{2^p}\left(1+\frac{f'}{2^p}\right).$$

As $2^p < d' \leq 2^{p+1}$ and $f'/2^p < 1/16$, we have

$$\frac{d'}{2^p}\left(1-\frac{f'}{2^p}\right) > 1\cdot\left(1-\frac{1}{16}\right) > 0 \quad \text{and} \quad \frac{d'}{2^p}\left(1+\frac{f'}{2^p}\right) < 2\cdot\left(1+\frac{1}{16}\right) < 4,$$

and thus

$$\left(1-\frac{f'}{2^p}\right)^3 < \frac{2^{3(p-k')}\theta^3}{d'^3} < \left(1+\frac{f'}{2^p}\right)^3.$$

If we set $f^* = 3f' + 3f'^2/2^p + f'^3/2^{2p}$ then

$$1-\frac{f^*}{2^p} = 1-\frac{3f'}{2^p}-\frac{3f'^2}{2^{2p}}-\frac{f'^3}{2^{3p}} < 1-\frac{3f'}{2^p}+\frac{3f'^2}{2^{2p}}-\frac{f'^3}{2^{3p}} = \left(1-\frac{f'}{2^p}\right)^3$$

and

$$\left(1+\frac{f'}{2^p}\right)^3 = 1+\frac{3f'}{2^p}+\frac{3f'^2}{2^{2p}}+\frac{f'^3}{2^{3p}} = 1+\frac{3f'+3f'^2/2^p+f'^3/2^{2p}}{2^p} = 1+\frac{f^*}{2^p}.$$

Hence

$$1-\frac{f^*}{2^p} < \frac{2^{3p-3k'}\theta^3}{d'^3} < 1+\frac{f^*}{2^p}. \tag{5.5}$$

Now suppose that $d'^3 \leq 2^{3p+1}$. Since $d = d'^3/2^{2p} + \epsilon$ for $0 \leq \epsilon < 1$, (5.5) becomes

$$1-\frac{f^*}{2^p} < \frac{2^{p-k}\theta^3}{d-\epsilon} < 1+\frac{f^*}{2^p}$$

and as $d-\epsilon = d(1-\epsilon/d)$,

$$\left(1-\frac{\epsilon}{d}\right)\left(1-\frac{f^*}{2^p}\right) < \frac{2^{p-k}\theta^3}{d} < \left(1-\frac{\epsilon}{d}\right)\left(1+\frac{f^*}{2^p}\right).$$

Looking at the right-hand side of this inequality, $(1-\epsilon/d) < 1$ so

$$\left(1-\frac{\epsilon}{d}\right)\left(1+\frac{f^*}{2^p}\right) < 1+\frac{f^*}{2^p} < 1+\frac{1}{2^p}+\frac{f^*}{2^p} = 1+\frac{f}{2^p};$$

considering the left-hand side, $\epsilon < 1$ and $2^p < d$ so $2^p\epsilon < d$. Rearranging this inequality gives $1-1/2^p < 1-\epsilon/d$ and thus

$$1-\frac{f}{2^p} = 1-\frac{1}{2^p}-\frac{f^*}{2^p} < 1-\frac{1}{2^p}-\frac{f^*}{2^p}+\frac{f^*}{2^{2p}} = \left(1-\frac{1}{2^p}\right)\left(1-\frac{f^*}{2^p}\right) < \left(1-\frac{\epsilon}{d}\right)\left(1-\frac{f^*}{2^p}\right).$$

It follows that

$$\left|\frac{2^{p-k}\theta^3}{d}-1\right| < \frac{f}{2^p}$$

and $(\mathfrak{b}^3, d, k)$ is an (f,p) representation of $\mathfrak{a}^3$, where $\mathfrak{b}^3 = \theta^3\mathfrak{a}^3$. The theorem follows by applying similar arguments when $2^{3p+1} < d'^3 \leq 2^{3p+2}$ and when $d'^3 > 2^{3p+2}$. □

We use the above result to produce the following algorithm which, given an (f,p) representation of an ideal $\mathfrak{a}$, computes an (f,p) representation of the cube $\mathfrak{a}^3$. Notice that it executes in $O(\log_2 \Delta)$ elementary operations.

Algorithm 5.6: FPCUBE

Input: $(\mathfrak{b}', d', k')$, p, where $(\mathfrak{b}', d', k')$ is a reduced (f', p) representation of an invertible $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}$. Here $\mathfrak{b}' = [Q'/r, (P'+\sqrt{D})/r]$.

Output: A reduced (f,p) representation $(\mathfrak{b}, d, k)$ of $\mathfrak{a}^3$, where $\mathfrak{b} = [Q/r, (P+\sqrt{D})/r]$, $(P+\sqrt{D})/Q > 1$, $-1 < (P-\sqrt{D})/Q < 0$, $k \leq 3k'+2$, and $f = f^* + 17/8$ with $f^* = 3f' + 3f'^2/2^p + f'^3/2^{2p}$. (Optional output: $a, b \in \mathbb{Z}$, where $\nu = (a+b\sqrt{D})/r \in \mathcal{O}_{\mathbb{K}}$

and $\mathfrak{b} = (\nu/N(\mathfrak{b}')^3)\mathfrak{b}'^3$.)
1: Put $(\mathfrak{b}, A, B, C) = \text{NUCUBE}(\mathfrak{b}')$ where $\mathfrak{b} = [Q/r, (P+\sqrt{D})/r]$.
2: **if** $A + B\sqrt{D} < 0$ **then**
3: Set $A \leftarrow -A$, $B \leftarrow -B$.
4: **end if**
5: Set $C \leftarrow |C|$
6: **if** $d'^3 \leq 2^{3p+1}$ **then**
7: Put $e = \lfloor d'^3/2^{2p} \rfloor$ and $h = 3k'$.
8: **else if** $2^{3p+1} < d'^3 \leq 2^{3p+2}$ **then**
9: Put $e = \lfloor d'^3/2^{2p+1} \rfloor$ and $h = 3k' + 1$.
10: **else**
11: Put $e = \lfloor d'^3/2^{2p+2} \rfloor$ and $h = 3k' + 2$.
12: **end if**
13: Find $s \geq 0$ such that $2^s Q > 2^{p+4} B$.
14: Put $T = 2^s A + B\lfloor 2^s \sqrt{D} \rfloor$.
15: Put $(\mathfrak{b}, d, k) = \text{REMOVE}((\mathfrak{b}, e, h), T, C, s, p)$. (If $A, -B < 0$, put $a = |A|$, $b = B$, otherwise put $a = A$, $b = -B$.)

Proof (of correctness of FPCUBE). By Theorem 5.5, $(\mathfrak{b}'^3, g, h)$ is an (f, p) representation of $\mathfrak{a}'^3$ with $f = 1 + f^*$. As the relative generator μ returned by NUCUBE is such that $|\mu| \geq 1$ and $|\mu|\mathfrak{b} = \mathfrak{b}'^3$, REMOVE will compute an (f, p) representation $(\mathfrak{b}, d, k)$ of $\mathfrak{a}'^3$ with $f = 1 + f^* + 9/8 = 17/8 + f^*$. Moreover, $|N(\mu)| = N(\mathfrak{b}')^3/N(\mathfrak{b})$ where $N(\mathfrak{b}) = |Q_{i+j}|/r$, so

$$\nu = \frac{N(\mathfrak{b}')^3}{\mu} = N(\mathfrak{b})|\overline{\mu}| = \left| \frac{a + b\sqrt{D}}{r} \right| \in \mathcal{O}.$$

□

The next step we need to make is to develop an algorithm similar to ADDXY; one which, given an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}[x]$, determines an $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a}[3x]$.

Algorithm 5.7: TRIPLEX

Input: $(\mathfrak{a}[x], d', k')$, x, p, where $(\mathfrak{a}[x], d', k')$ is an x-near (f, p) representation of the $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a} = (1)$.
Output: $(\mathfrak{a}[3x], d, k)$, a $3x$-near (f, p) representation of $\mathfrak{a}$ where $f = 13/4 + 3f' +$

$3f'^2/2^p + f'^3/2^{2p}$.

1: Put $(\mathfrak{c}, g, h) = \text{FPCUBE}((\mathfrak{a}[x], d', k'), p)$.
2: Put $(\mathfrak{c}', g', h') = \text{WNEAR}((\mathfrak{c}, g, h), 3x, p)$.
3: Put $\mathfrak{a}[3x] = \mathfrak{c}'$, $d = g'$, and $k = h'$.

Proof (of correctness of TRIPLEX). FPCUBE will return $(\mathfrak{c}, g, h)$, an (f, p) representation of $\mathfrak{a}[x]^3$ with $h \leq 3k' + 2 \leq 3x - 1$. Because of this, the first case of WNEAR will be called to produce a $3x$-near (f, p) representation of $\mathfrak{a}[x]^3$. □

We remark that by the same reasoning used to determine the computational complexity of ADDXY, TRIPLEX will execute in $O(\log_2 \Delta)$ elementary operations. As in the case of EADDXY, we can also make a slight modification to TRIPLEX to simultaneously compute the relative generator.

Algorithm 5.8: ETRIPLEX

Input: $(\mathfrak{a}[x], d', k')$, x, p, where $(\mathfrak{a}[x], d', k')$ is an x-near (f, p) representation of the $\mathcal{O}_{\mathbb{K}}$-ideal $\mathfrak{a} = (1)$.

Output: $(\mathfrak{a}[3x], d, k)$, a, b, where $(\mathfrak{a}[3x], d, k)$ is a $3x$-near (f, p) representation of $\mathfrak{a}$, $f = 13/4 + 3f' + 3f'^2/2^p + f'^3/2^{2p}$, and

$$\lambda = \frac{a + b\sqrt{D}}{r} \text{ such that } \mathfrak{a}[3x] = \left(\frac{\lambda\theta^3}{N(\mathfrak{a}[x])^3}\right)\mathfrak{a},$$

where $\mathfrak{a}[x] = (\theta)\mathfrak{a}$.

1: Put $((\mathfrak{c}, g, h), a', b') = \text{FPCUBE}((\mathfrak{a}[x], d', k'), p)$.
2: Put $((\mathfrak{c}', g', h'), a'', b'') = \text{EWNEAR}((\mathfrak{c}, g, h), 3x, p)$.
3: Put $\mathfrak{a}[3x] = \mathfrak{c}'$, $d = g'$, $k = h'$, and $(a, b) = \text{IMULT}(a', b', a'', b'', N(\mathfrak{c}))$.

Proof (of correctness of ETRIPLEX). In Step 1, we use the optional output of FPCUBE to find integers a', b' where

$$\mathfrak{c} = \left(\frac{\mu}{N(\mathfrak{a}[x])^3}\right)\mathfrak{a}[x]^3,$$

and $\mu = (a' + b'\sqrt{D})/r \in \mathcal{O}$. In the next step, we find $\mathfrak{c}' = \nu\mathfrak{c}$ and $\nu = (a'' + b''\sqrt{D})/r \in \mathcal{O}$.

Thus,

$$\lambda = \frac{\mu\nu}{N(\mathfrak{c})} = \frac{a + b\sqrt{D}}{r} \in \mathcal{O}$$

and

$$\mathfrak{a}[3x] = \mathfrak{c}' = \left(\frac{\lambda}{N(\mathfrak{a}[x])^3}\right)\mathfrak{a}[x]^3 = (\theta)\mathfrak{a}$$

where $\theta = \lambda\theta'^3/N(\mathfrak{a}[x])^3$ and satisfies

$$\left|\frac{2^p\theta}{2^k d} - 1\right| < \frac{f}{2^p}.$$

□

At this point we have all of the base algorithms needed to present our cube-and-multiply based routine for computing a compact representation. To help the reader better understand the dependencies between these new algorithms and the ones previously introduced in Chapter 2, we present an updated algorithm flowchart in Figure 5.3.

Algorithm 5.9: 3CRAX

Input: x, p, where $x \in \mathbb{Z}^+$ and $2^p > 11.2x\max\{16\log_2 x\}$.

Output: $(\mathfrak{a}[x], d, k)$, (m_i, n_i), and L_i, where $(\mathfrak{a}[x], d, k)$ is an x-near (f, p) representation of $\mathfrak{a} = (1)$ with $f < 2^{p-4}$, (m_i, n_i) are pairs of integers, and $L_i \in \mathbb{Z}^+$ for $i = 0, 1, \ldots, l$ where l is such that $x = \sum_{j=0}^{l} 3^{l-j} b_j$ and $b_0 \neq 0$, $b_j \in \{0, 1, 2\}$.

1: Compute the ternary representation of x with

$$x = \sum_{i=0}^{l} 3^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{0, 1, 2\} \quad (1 \leq i \leq l).$$

2: Put

$$Q = r, \qquad P = r\left\lfloor \frac{\lfloor\sqrt{\Delta}\rfloor - r + 1}{r} \right\rfloor + r - 1, \qquad (\mathfrak{b}, d, k) = ([Q, P], 2^p + 1, 0),$$

$s = b_0$, $L_0 = 1$, and $i = 0$.

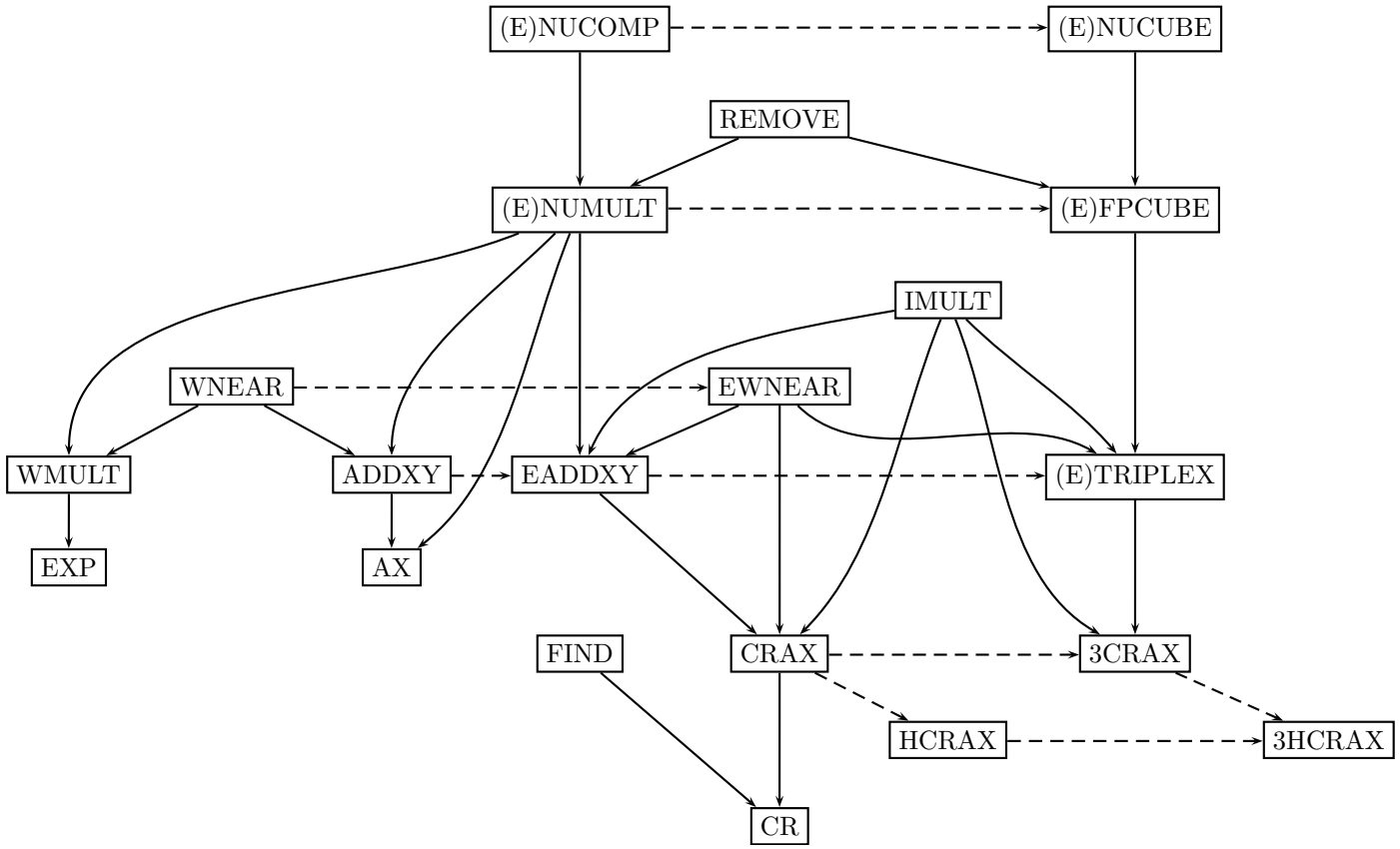


Figure 5.3: Updated algorithm dependencies

3: Put $((\mathfrak{b}_0, d_0, k_0), m_0, n_0) = \text{EWNEAR}((\mathfrak{b}, d, k), s, p)$.
4: **while** $i < l$ **do**
5: Put $L_{i+1} = N(\mathfrak{b}_i)$ and

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m_{i+1}, n_{i+1}) = \text{ETRIPLEX}((\mathfrak{b}_i, d_i, k_i), s, p).$$

6: Set $s \leftarrow 3s + b_{i+1}$.
7: **if** $b_{i+1} \neq 0$ **then**
8: Put $N = N(\mathfrak{b}_{i+1})$ and set

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m'_{i+1}, n'_{i+1}) \leftarrow \text{EWNEAR}((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), s, p).$$

9: Set $(m_{i+1}, n_{i+1}) \leftarrow \text{IMULT}(m_{i+1}, n_{i+1}, m'_{i+1}, n'_{i+1}, N)$.
10: **end if**
11: Set $i \leftarrow i + 1$.
12: **end while**
13: Put $L_{l+1} = N(\mathfrak{b}_l)$ and $(\mathfrak{a}[x], d, k) = (\mathfrak{b}_l, d_l, k_l)$.

Proof (of correctness of 3CRAX). As 3CRAX executes, it produces a series of reduced principal $\mathcal{O}_{\mathbb{K}}$-ideals $\mathfrak{a}[s_i] = \mathfrak{b}_i = (\pi_i)\mathfrak{a}_1$ ($\mathfrak{a}_1 = (1)$) where

$$\left| \frac{2^p \pi_i}{2^{k_i} d_i} - 1 \right| < \frac{f}{2^p}.$$

Moreover, $\pi_i \in \mathcal{O}_{\mathbb{K}}$, $|N(\pi_i)| = N(\mathfrak{a}[s_i]) = N(\mathfrak{b}_i) = L_{i+1}$ and since $2^p > 11.2x \max\{16, \log_2 x\}$, Theorem 11.9 of [48, p. 280] ensures that $f < 2^{p-4}$. If we set $\lambda_i = (m_i + n_i\sqrt{D})/r$, then

$$\pi_{i+1} = \left(\frac{\lambda_{i+1}}{L_{i+1}^3} \right) \pi_i^3 \tag{5.6}$$

where $\pi_0 = \lambda_0$. If we define $L_0 = 1$, then we get

$$\pi_j = \prod_{i=0}^{j} \left(\frac{\lambda_i}{L_i^3} \right)^{3^{j-i}}$$

for $j = 0, 1, \ldots, l$. When $j = l$, we have $s_l = x$, $\mathfrak{a}[x] = \mathfrak{b}_l = (\pi_l)$, and hence $\mathfrak{a}[x] = (\theta)$

where

$$\theta = \prod_{i=0}^{l} \left(\frac{\lambda_i}{L_i^3} \right)^{3^{j-i}} .$$

□

In order to formally state the definition of a 3-compact representation, we first need to derive bounds on the heights of the λ_i defined in the preceding proof. For any fixed i ($1 \leq i \leq l$), we must have some θ_j in the simple continued fraction expansion of ω such that $\pi_i = \theta_j$. By Lemma 2.9, we thus have

$$\frac{15N(\mathfrak{b}_{i+1})}{16\sqrt{\Delta}} 2^{s_i} < \theta_j < \frac{17}{16} 2^{s_i} \tag{5.7}$$

and hence,

$$\frac{15L_{i+1}}{16\sqrt{\Delta}} 2^{s_i} < \pi_i < \frac{17}{16} 2^{s_i} . \tag{5.8}$$

From (5.6) we see $\lambda_i = (L_i^3 \pi_i)/\pi_{i-1}^3$, which, when combined with (5.8), gives

$$\lambda_i < L_i^3 \left(\frac{17}{16} 2^{s_i} \right) \left(\frac{16\sqrt{\Delta}}{15L_i} 2^{-s_{i-1}} \right)^3 = \frac{16^3 \cdot 17}{15^3 \cdot 16} 2^{s_i - 3s_{i-1}} \Delta^{3/2} = \frac{16^2 \cdot 17}{15^3} 2^{s_i - 3s_{i-1}} \Delta^{3/2} \tag{5.9}$$

since $s_i - 3s_{i-1} \in \{0, 1, 2\}$. Hence,

$$\lambda_i < \frac{16^2 \cdot 17 \cdot 2^2}{15^3} \Delta^{3/2} < \frac{11}{2} \Delta^{3/2} .$$

Now, since $\overline{\lambda_i} = (L_i^3 \overline{\pi}_i)/\overline{\pi}_{i-1}^3$ and $|\pi_i \overline{\pi}_i| = L_{i+1}$, we find

$$|\overline{\lambda_i}| = \left| \frac{L_i^3 \overline{\pi}_i}{\overline{\pi}_{i-1}^3} \right| = \frac{L_i^3 (L_{i+1}/\pi_i)}{(L_i/\pi_{i-1})^3} = \frac{L_i^3 L_{i+1} \pi_{i-1}^3}{L_i^3 \pi_i} = \frac{L_{i+1} \pi_{i-1}^3}{\pi_i}$$

and so

$$|\overline{\lambda_i}| < L_{i+1}\left(\frac{17}{16}2^{s_{i-1}}\right)^3\left(\frac{16\sqrt{\Delta}}{15L_{i+1}}2^{-s_i}\right) = \frac{17^3}{16^2\cdot 15}2^{3s_{i-1}-s_i}\sqrt{\Delta} < \frac{3}{2}\sqrt{\Delta}\,. \tag{5.10}$$

since $3s_{i-1} - s_i \in \{0, -1, -2\}$. Thus,

$$H(\lambda_i) < \frac{11}{2}\Delta^{3/2}\,. \tag{5.11}$$

Considering λ_0, we see

$$\frac{15L_1}{16\sqrt{\Delta}} < \lambda_0 < \frac{17}{16}2^{s_0} = \frac{17}{8}\,,$$

as $s_0 = 1$, and so (5.11) holds for $i = 0$ as well. At this point, we can state the definition of a 3-compact representation.

Definition 5.10. *For any θ such that $(\theta) = \mathfrak{a}[x] \in \mathcal{O}$, a* 3-compact representation *of θ is*

$$\theta = \prod_{i=0}^{l}\left(\frac{\lambda_i}{L_i^3}\right)^{3^{l-i}}$$

where the following properties are satisfied:

1. $l = \lceil \log_3 \log_2 \theta \rceil$.
2. $\lambda_i \in \mathcal{O}_{\mathbb{K}}$ *and* L_i *is an integer* $(0 \le i \le l)$.
3. $0 < L_i \le \Delta^{1/2}$ *and* $H(\lambda_i) = \mathrm{O}(\Delta^{3/2})$ $(0 \le i \le l)$.
4. $\pi_j \in \mathcal{O}_{\mathbb{K}}$, $L_j = |N(\pi_j)|$,
$$\pi_j = \prod_{i=0}^{j}\left(\frac{\lambda_i}{L_i^3}\right)^{3^{j-i}}\,,$$

π_j generates a reduced ideal $\mathfrak{b}_j$, where $\mathfrak{b}_0 = \mathfrak{a}[1]$ and

$$L_{i+1}^3 \mathfrak{b}_{i+1} = \lambda_{i+1} \mathfrak{b}_i^3 \quad (0 \leq i \leq l-1)\,.$$

Theorem 5.11. *Let $\theta \in \mathcal{O}_{\mathbb{K}}$ such that $\mathfrak{a}[x] = (\theta)$ for some $x \in \mathbb{Z}^+$. The total number of bits required to express θ as a 3-compact representation is* $\mathrm{O}((\log_3 \log_2 \theta) \log_2 \Delta^{3/2})$.

Proof. From (5.11) we know $H(\lambda_i) < (11/2)\Delta^{3/2}$. If we set $s_i = x$, $\theta_j = \theta$ and $N(\mathfrak{b}_{i+1}) = 1$ in (5.7), we have $2^x < (16\sqrt{\Delta}/15)\theta$. Since $l = \lceil \log_3 x \rceil$, we require

$$\mathrm{O}(l \log_2 \Delta^{3/2}) = \mathrm{O}((\log_3 \log_2 \theta) \log_2 \Delta^{3/2})$$

bits to express θ as a 3-compact representation. □

Returning to our running example, we can employ 3CRAX to produce the 3-compact representation of the fundamental unit $\eta_{410286423278424}$ shown in Table 5.2. (The previous two compact representations are on pages 47 and 149.) As we can see, the number of terms has definitely been reduced, but the individual sizes have increased. However, this is to be expected: we are traversing a greater distance in the cycle of reduced principal ideals on each iteration of the main while-loop, but the short-fall we experience at the end of each giant-step has been magnified. Thus, the correction we need to make will be larger.

As we continue in this section, we will see just how far we can push this trade-off between increased term sizes and a decreased number of terms. Overall for our example, we require 1,186 bits, which is a 2.1% savings compared to the compact representation, but it is also 22.1% larger than the h-compact representation.

Table 5.2: 3-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i
3	461840850098620	22800731	1
4	393847945106440948064	19443973068552	2422664
5	1748981591548163056061	86345889031527	5221225
6	9417841462337825471	464951659728	1434745
7	61583040732846306699904	3040307815152853	37042535
8	9934001598858842146176	490434092525128	17298472
9	9857042566970449959	486634683738	898455
10	897726603790655651009	44320078659572	13953959
11	8561762325532279619533	422687684791656	3462265
12	—	—	1

Of course, we can also combine the ideas behind both the h-compact and 3-compact representations to reduce the number of bits needed to express θ even further. Working through the details of adding "$-h$" to the new Step 6 in 3CRAX, we find we must let n be the largest integer such that $x \geq ((3^n-1)/2)h$ and set $y = x+((3^n-1)/2)h$. Furthermore, in light of ETRIPLEX, we compute

$$\mathfrak{a}[s_{i+1}]' = (\mu_i)\mathfrak{a}[s_i]^3 = \left((\mu_i')\mathfrak{a}[s_i]\right)\left((\mu_i'')\mathfrak{a}[s_i]^2\right)$$

for each iteration of the main while-loop. Thus, as in the case of HCRAX, we have $\mathrm{O}(\Delta^{1/4}) < \mu_i', \mu_i'' < \mathrm{O}(\Delta^{3/4})$, and since $\mu_i = \mu_i'\mu_i''$, we see $\mathrm{O}(\Delta^{1/2}) < \mu_i < \mathrm{O}(\Delta^{3/2})$. So our choice of h needs to be increased to $h = \lceil (1/2)\log_2 \Delta \rceil$.

If we modify (5.9) and (5.10) as we modified (5.2), we find

$$\lambda_i < L_i^3 \left(\frac{17}{16}2^{s_i-h}\right)\left(\frac{16\sqrt{\Delta}}{15L_i}2^{-s_{i-1}}\right)^3 = \frac{16^3\cdot 17}{15^3\cdot 16}2^{s_i-h-3s_{i-1}}\Delta^{3/2} = \frac{16^2\cdot 17}{15^3}2^{s_i-3s_{i-1}-h}\Delta^{3/2},$$

since $s_i - 3s_{i-1} \in \{0,1,2\}$. So

$$\lambda_i < \frac{16^2\cdot 17\cdot 2^2}{15^3}\Delta^{3/2} < \frac{11}{2}2^{-\lceil (1/2)\log_2\Delta\rceil}\Delta^{3/2} = \frac{11}{2}\Delta\,. \tag{5.12}$$

Similarly, we have

$$\begin{aligned}|\overline{\lambda_i}| &< L_{i+1}\left(\frac{17}{16}2^{s_{i-1}}\right)^3\left(\frac{16\sqrt{\Delta}}{15L_{i+1}}2^{-s_i+h}\right) = \frac{16\cdot 17^3}{15\cdot 16^3}2^{3s_{i-1}-s_i+h}\Delta^{1/2}\\ &\le \frac{16\cdot 17^3}{15\cdot 16^3}\cdot 2^h\Delta^{1/2} < \frac{3}{2}2^{\lceil (1/2)\log_2\Delta\rceil}\Delta^{1/2} = \frac{3}{2}\Delta\,.\end{aligned} \tag{5.13}$$

Thus, we find that the heights of the λ_i produced by 3HCRAX are bounded by

$$H(\lambda_i) < \frac{11}{2}\Delta\,. \tag{5.14}$$

Algorithm 5.12: 3HCRAX

Input: x, p, where $x \in \mathbb{Z}^+$ and $2^p > 11.2x\max\{16\log_2 x\}$.

Output: $(\mathfrak{a}[x], d, k)$, (m_i, n_i), and L_i, where $(\mathfrak{a}[x], d, k)$ is an x-near (f, p) representation of $\mathfrak{a} = (1)$ with $f < 2^{p-4}$, (m_i, n_i) are pairs of integers, and $L_i \in \mathbb{Z}^+$ for $i = 0, 1, \ldots, l$ where l is such that $x = \sum_{j=0}^{l} 3^{l-j}b_j$ and $b_0 \neq 0$, $b_j \in \{0,1,2\}$.

1: Put $h = \lceil (1/2)\log_2\Delta\rceil$ and compute the maximal n such that

$$\frac{x}{(3^n-1)/2} \ge h,$$

and put $y = x + ((3^n-1)/2)h$.

2: Compute the ternary representation of y with

$$y = \sum_{i=0}^{l} 3^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{0,1,2\} \quad (1 \leq i \leq l).$$

3: Put

$$Q = r, \qquad P = r \left\lfloor \frac{\lfloor \sqrt{\Delta} \rfloor - r + 1}{r} \right\rfloor + r - 1, \qquad (\mathfrak{b}, d, k) = ([Q, P], 2^p + 1, 0),$$

$s = b_0$, $L_0 = 1$, and $i = 0$.

4: Put $((\mathfrak{b}_0, d_0, k_0), m_0, n_0) = \text{EWNEAR}((\mathfrak{b}, d, k), s, p)$.

5: **while** $i < l - n$ **do**

6: Put $L_{i+1} = N(\mathfrak{b}_i)$ and

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m_{i+1}, n_{i+1}) = \text{ETRIPLEX}((\mathfrak{b}_i, d_i, k_i), s, p).$$

7: Set $s \leftarrow 3s + b_{i+1}$.

8: **if** $b_{i+1} \neq 0$ **then**

9: Put $N = N(\mathfrak{b}_{i+1})$ and set

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m'_{i+1}, n'_{i+1}) \leftarrow \text{EWNEAR}((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), s, p).$$

10: Set $(m_{i+1}, n_{i+1}) \leftarrow \text{IMULT}(m_{i+1}, n_{i+1}, m'_{i+1}, n'_{i+1}, N)$.

11: **end if**

12: Set $i \leftarrow i + 1$.

13: **end while**

14: **while** $i < l$ **do**

15: Put $L_{i+1} = N(\mathfrak{b}_i)$ and

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m_{i+1}, n_{i+1}) = \text{ETRIPLEX}((\mathfrak{b}_i, d_i, k_i), s, p).$$

16: Set $s \leftarrow 3s + b_{i+1} - h$.

17: Put $N = N(\mathfrak{b}_{i+1})$ and set

$$((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), m'_{i+1}, n'_{i+1}) \leftarrow \text{EWNEAR}((\mathfrak{b}_{i+1}, d_{i+1}, k_{i+1}), s, p).$$

18: Set $(m_{i+1}, n_{i+1}) \leftarrow \text{IMULT}(m_{i+1}, n_{i+1}, m'_{i+1}, n'_{i+1}, N)$.

19: Set $i \leftarrow i + 1$.

20: **end while**

21: Put $L_{l+1} = N(\mathfrak{b}_l)$ and $(\mathfrak{a}[x], d, k) = (\mathfrak{b}_l, d_l, k_l)$.

Theorem 5.13. *Let $\theta \in \mathcal{O}_{\mathbb{K}}$ such that $\mathfrak{a}[x] = (\theta)$ for some $x \in \mathbb{Z}^+$. The total number of bits required to express θ as a $3h$-compact representation is* $O((\log_3 \log_2 \theta) \log_2 \Delta)$.

Proof. We know $H(\lambda_i) < (11/2)\Delta$ and, as stated in the proof of Theorem 5.11, we also have $l = \lceil \log_3 \log_2 \theta \rceil$. Thus we need

$$O(l \log_2 \Delta^{3/2}) = O((\log_3 \log_2 \theta) \log_2 \Delta)$$

bits to express θ as a $3h$-compact representation. □

Going back to our running examples one more time, we use 3HCRAX to produce the $3h$-compact representation of the fundamental unit $\eta_{410286423278424}$ shown in Table 5.3. (See pages 47, 149, and 163 for the other compact representations.) For this $3h$-compact representation, we need only 852 bits; a savings of 29.7% over the compact representation and of 12.5% over the h-compact representation. We would like to emphasize that the memory savings of using a $3h$-compact representation over a compact representation comes almost entirely from a reduced number of terms, and not a reduced term size. This is expected as both CRAX and 3HCRAX generate λ_i with $H(\lambda_i) = O(\Delta)$. On average, the compact representation in Table 2.1 uses 65.2 bits per λ_i, whereas the $3h$-compact representation in Table 5.3 uses 65.3 bits. On the other hand, the h-compact representation does derive its memory savings from a reduced term size; the representation in Table 5.1 uses 44.4 bits per λ_i on average.

In 3CRAX and 3HCRAX, as presented previously, we compute the ternary representation of x as

$$x = \sum_{i=0}^{l} 3^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{0, 1, 2\} \quad (1 \leq i \leq l).$$

Table 5.3: $3h$-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i
3	182299754	9	1
4	1222201773973992	62412364	22908172
5	128329945705305	4780740	8685255
6	331044493042892	15107681	10823735
7	163479422159904	-3917272	12575240
8	-136293163131663	7346283	10273359
9	189325393222681	9232568	3289273
10	99843396702949	4693418	24479305
11	915125032449	-45173	63457
12	688242823029	33978	874185
13	—	—	1

However, we can replace this with a signed ternary representation of x, where

$$x = \sum_{i=0}^{l} 3^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{0, \pm 1\} \quad (1 \leq i \leq l),$$

and with appropriate change to Step 1 of Algorithms 5.9 and 5.12, shave a few more bits off the compact representation. For a signed $3h$-compact representation, we have $s_i - 3s_{i-1} \in \{0, \pm 1\}$, and so (5.12) and (5.13) instead become

$$\lambda_i < \frac{16^2 \cdot 17}{15^3} 2^{s_i - 3s_{i-1} - h} \Delta^{3/2} \leq \frac{16^2 \cdot 17 \cdot 2}{15^3} \cdot 2^{-h} \Delta^{3/2} < \frac{11}{4} \Delta$$

and

$$|\overline{\lambda_i}| < \frac{17^3}{15 \cdot 16^2} 2^{3s_{i-1}-s_i+h} \Delta^{1/2} \leq \frac{17^3 \cdot 2}{15 \cdot 16^2} \cdot 2^h \Delta^{1/2} < \frac{11}{4} \Delta \,.$$

Thus, we find for a signed $3h$-compact representation that

$$H(\lambda_i) < \frac{11}{4} \Delta \,,$$

which is just slightly worse than the bound of $H(\lambda_i) < (5/2)\Delta$ for the regular compact representation.

Looking again at $\eta_{410286423278424}$, Tables 5.4 and 5.5 show the signed 3- and signed $3h$-compact representations of the fundamental unit which require 1,137 and 843 bits, respectively. Compared to the compact and h-compact representations respectively, the signed 3-compact representation gives us a savings of 6.2% and an increase of 16.7%; the signed $3h$-compact representation saves us 30.7% and 13.7%. As an aside, a summary of the results of this and the previous section will be given in Table 5.9 on page 179.

5.4. Extending compact representations to higher bases

At this point, we have said about all there is to say regarding 3-compact representations. However, a valid question to pose is: can we extend this idea further? What about a 4-compact, 5-compact, or higher representation? While we do not have dedicated algorithms like NUCUBE for computing higher powers, we can use some other algorithms at our disposal to model these. Computationally, of course, these are not as efficient as a dedicated algorithm, however since we are more interested in the resulting size of the compact representation, we will not concern ourselves with optimizing their run-time.

Furthermore, although we will not formally define a 4-compact, 5-compact, and

Table 5.4: Signed 3-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i
4	461840850098620	22800731	1
5	393847945106440948064	19443973068552	2422664
6	1748981591548163056061	86345889031527	5221225
7	10189739201431422114	503059663177	1434745
8	98000352201190176	4838202744	467916
9	9934001598858842146176	490434092525128	17298472
10	9857042566970449959	486634683738	898455
11	897726603790655651009	44320078659572	13953959
12	8561762325532279619533	422687684791656	3462265
13	—	—	1

Table 5.5: Signed 3h-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i
3	283577395	14	1
4	91022547145107	-3309911	7585079
5	128329945705305	4780740	8685255
6	331044493042892	15107681	10823735
7	163479422159904	-3917272	12575240
8	-136293163131663	7346283	10273359
9	189325393222681	9232568	3289273
10	99843396702949	4693418	24479305
11	915125032449	-45173	63457
12	688242823029	33978	874185
13	—	—	1

higher representations, they can be defined in a manner similar to the 3-compact representation (Definition 5.10 on page 161). We will also not provide pseudocode for the algorithms named below, though they are implemented in our library and their prototypes can be found in `fp-representation.h`.

For the time being, we will occupy ourselves with only the 4-compact and 5-compact representations. We can compute a quaternary representation of an integer x as

$$x = \sum_{i=0}^{l} 4^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{0,1,2,3\} \quad (1 \leq i \leq l),$$

and for the signed quaternary representation, we have a choice:

$$x = \sum_{i=0}^{l} 4^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{-1,0,1,2\} \quad (1 \leq i \leq l) \quad \text{or}$$
$$x = \sum_{i=0}^{l} 4^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{-2,-1,0,1\} \quad (1 \leq i \leq l).$$

For the quinary and signed quinary representation, of course, we have

$$x = \sum_{i=0}^{l} 5^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{0,1,\ldots,4\} \quad (1 \leq i \leq l) \quad \text{and}$$
$$x = \sum_{i=0}^{l} 5^{l-i} b_i \text{ and } b_0 \neq 0, b_i \in \{-2,-1,0,1,2\} \quad (1 \leq i \leq l).$$

For the 4CRAX algorithm to compute a 4-compact representation, we require

- an algorithm which computes a reduced $\mathcal{O}$-ideal $\mathfrak{b} \sim \mathfrak{a}^4$ and μ such that $(\mu)\mathfrak{b} = \mathfrak{a}^4$,
- a variant of this algorithm which works with (f, p) representations,
- and an algorithm which computes an ideal $\mathfrak{a}[4x]$ from an ideal $\mathfrak{a}[x]$.

For 5CRAX, we will need similar versions of these three algorithms. Once these algo-

rithms are in place, we can begin computing 4-compact and 5-compact representations.

Returning to the fundamental unit $\eta_{410286423278424}$, a 4-compact representation is given in Table 5.6, which requires 1,282 bits. Recall that we noticed an increase in the size of the individual representation terms when we moved from a compact representation to a 3-compact representation. As is expected, when moving to a 4-compact representation, we see this increase in term-size again. For a 4-compact representation, we can use the previously detailed formulas to show

$$H(\lambda_i) < \frac{45}{4}\Delta^2$$

and determine that for $\theta \in \mathcal{O}_{\mathbb{K}}$ such that $\mathfrak{a}[x] = (\theta)$ $(x \in \mathbb{Z}^+)$, the total number of bits required to express θ is $O((\log_4 \log_2 \theta)\log_2 \Delta^2)$.

Table 5.6: 4-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i
3	4570714250827732415837751	225652681196857720	1
4	3100391024098655612222148449	15306394339130632983	4183599
5	201217953830997167710786763 82	993397711937195910501	3694935
6	216803128672757353707614145296	10703405330581644340160	10909076
7	370923983038325258991916217558	18312234521696087195947	14226959
8	111177377246951965118728645488	5488742434438124188688	14030668
9	931518642043427967709423959 69	4598836584979894864110	3422487
10	—	—	1

Now, earlier we mentioned that we have a choice in implementing a signed 4-compact

representation, namely do we take $b_i \in \{-1,0,1,2\}$ or $b_i \in \{-2,-1,0,1\}$? Implementing both, we find that computing a signed 4-compact representation of $\eta_{410286423278424}$ using $b_i \in \{-1,0,1,2\}$ again requires 1,282 bits, whereas using $b_i \in \{-2,-1,0,1\}$ requires only 1,242 bits. In the general case, it seems hard to predict *a priori* which signed base will produce a shorter signed compact representation. Our initial guess was that the base-4 expansion with fewer non-zero terms—particularly twos—would lead to a smaller representation, since when $b_i = \pm 2, \pm 1$ we compute a slightly larger ν_i value than in the $b_i = 0$ case. However, the two base-4 expansions can result in slightly different sequences of $\mathfrak{a}[s_i]$ ideals being computed, which seems to have a larger effect on the overall representation size.

In fact, this occurs in our running example. During the signed 4CRAX computations, we find the signed base-4 expansions of $x = 343066$ as

$$\begin{aligned} 343066 &= \sum_{i=0}^{9} 4^{9-i} b_i \text{ where } \{b_0, b_1, \ldots, b_9\} = \{1,1,1,0,-1,0,0,1,2,2\} \\ &= \sum_{i=0}^{9} 4^{9-i} b_i \text{ where } \{b_0, b_1, \ldots, b_9\} = \{1,1,1,0,-1,0,1,-2,-1,-2\} . \end{aligned}$$

The second expansion has an extra non-zero term, but the overall compact representation is 40 bits shorter.

Looking at the heights of the individual λ_i generators in a signed 4-compact representation, we find

$$H(\lambda_i) < \frac{45}{8}\Delta^2 .$$

As with the 3-compact representation, we can extend the 4- and signed 4-compact representations to $4h$- and signed $4h$-compact representations as follows. As in the base-3

case, we need to increase h by a further factor of $(1/4)\log_2\Delta$ to $h = \lceil(3/4)\log_2\Delta\rceil$. We also must compute the maximal n such that

$$\frac{x}{(4^n-1)/3} \geq h,$$

and put $y = x + ((4^n-1)/3)h$. The remainder of 4HCRAX is structured similarly to 3HCRAX. For bounds on $H(\lambda_i)$, we find

$$H(\lambda_i) < \frac{45}{4}\Delta^{5/4} \quad \text{and} \quad H(\lambda_i) < \frac{45}{8}\Delta^{5/4}$$

for a $4h$- and signed $4h$-compact representation, respectively. Furthermore, for $\theta \in \mathcal{O}_{\mathbb{K}}$ such that $\mathfrak{a}[x] = (\theta)$ $(x \in \mathbb{Z}^+)$, the total number of bits required to express θ as a $4h$-compact representation is $O((\log_4\log_2\theta)\log_2\Delta^{5/4})$.

We can compute the $4h$-compact representation of $\eta_{410286423278424}$ given in Table 5.7 using 4HCRAX, which requires only 832 bits to store. Compared to the signed $3h$-compact representation, this represents an additional savings of 1.3%. As happened above, a signed $4h$-compact using $b_i \in \{-1, 0, 1, 2\}$ also requires only 832 bits. However, in the case of a signed $4h$-compact representation using $b_i \in \{-2, -1, 0, 1\}$ the resulting compact representation requires 843 bits.

Moving up to a 5-compact and signed 5-compact representation of $\eta_{410286423278424}$, we need 1,385 and 1,335 bits, respectively. As noted previously, this is caused by the increased size of the individual terms in the 5- and signed 5-compact representations:

$$H(\lambda_i) < 47\Delta^{5/2} \quad \text{and} \quad H(\lambda_i) < \frac{47}{4}\Delta^{5/2},$$

Table 5.7: $4h$- and signed $4h$-compact representation of $\eta_{410286423278424}$.

i	m_i	n_i	L_i
2	1600186729	79	1
3	4388379400321456612	216947663917	12924743
4	11399368843715584	-506977600	1891592
5	-6858207859883840	370889024	1912936
6	7270829486946172	359936327	702239
7	2998820192664699648	148049590976	1189832
8	-654539422395294144	33846945408	7405320
9	952354193589757649	47017000210	13835951
10	—	—	1

for the 5- and signed 5-compact representations, respectively. Overall, $O((\log_5 \log_2 \theta) \log_2 \Delta^{5/2})$ bits are needed to store the total representation. If we set $h = \lceil \log_2 \Delta \rceil$, compute the maximal n such that

$$\frac{x}{(5^n - 1)/4} \geq h,$$

and put $y = x + ((5^n - 1)/4)h$, we can compute a $5h$- and signed $5h$-compact representation. Looking one more time at the heights of the λ_i generators, we see that

$$H(\lambda_i) < 47\Delta^{3/2} \quad \text{and} \quad H(\lambda_i) < \frac{47}{4}\Delta^{3/2}$$

for these two representations, with $O((\log_5 \log_2 \theta) \log_2 \Delta^{3/2})$ bits needed to store the total representation. The $5h$- and signed $5h$-compact representations of $\eta_{410286423278424}$ require 877 and 875 bits, respectively, which is in fact larger than the storage needed for the $4h$-

and signed $4h$-compact representations.

At this point, we find the first indications that pursuing this idea to higher powers (i.e., 6-compact and higher representations) may not result in further memory savings. Unfortunately, the increase in size of the individual terms of the compact representations begins dominating over the savings from a decreased overall number of terms. In the remainder of this section, we look at an analytical argument to justify this claim. In the following section, we will present some calculations that confirm, numerically at least, that this analysis is valid.

Table 5.8 gives a summary of the bounds on $H(\lambda_i)$ for the various bases we have discussed previously. Since the signed h-compact representation is the most efficient for any given base, we will focus solely on these variants in our analysis.

Table 5.8: Summary of the $H(\lambda_i)$ bounds and number of terms for various compact representations.

Representation	$H(\lambda_i)$ bound	l
h-compact	$\frac{5}{2}\Delta^{3/4}$	$\log_2 \log_2 \theta$
Signed $3h$-compact	$\frac{11}{4}\Delta$	$\log_3 \log_2 \theta$
Signed $4h$-compact	$\frac{45}{8}\Delta^{5/4}$	$\log_4 \log_2 \theta$
Signed $5h$-compact	$\frac{47}{4}\Delta^{3/2}$	$\log_5 \log_2 \theta$
Signed $6h$-compact	$\frac{63}{5}\Delta^{7/4}$	$\log_6 \log_2 \theta$

To determine the overall expected size S_x of the signed base-x h-compact representation in bits, we multiply the base-2 logarithm of the $H(\lambda_i)$ bounds from Table 5.8 by the corresponding number of terms l. In general, these $H(\lambda_i)$ bounds are given by some constant B_x multiplied by $\Delta^{(x+1)/4}$. Expanding and standardizing the logarithm bases, we see that

$$\begin{aligned} S_x &= \log_2\left(B_x\Delta^{(x+1)/4}\right)\cdot\log_x\log_2\theta \\ &= \log_2 B_x\log_x\log_2\theta + \log_2\left(\Delta^{(x+1)/4}\right)\log_x\log_2\theta \\ &= \left(\frac{\log_2 B_x}{\log_2 x}\right)\log_2\log_2\theta + \left(\frac{x+1}{4\log_2 x}\right)\log_2\Delta\log_2\log_2\theta\,. \end{aligned} \tag{5.15}$$

The B_x values are given by

$$B_x = \max\left\{\frac{16^{x-1}\cdot 17}{15^x}, \frac{17^x}{15\cdot 16^{x-1}}\right\}2^{\lfloor x/2\rfloor} = \max\left\{\frac{17}{15}\frac{16^{x-1}}{15^{x-1}}, \frac{17}{15}\frac{17^{x-1}}{16^{x-1}}\right\}2^{\lfloor x/2\rfloor}$$

$$= \frac{17}{15}\max\left\{\left(\frac{16}{15}\right)^{x-1}, \left(\frac{17}{16}\right)^{x-1}\right\}2^{\lfloor x/2\rfloor} = \frac{17}{15}\max\left\{\frac{16}{15}, \frac{17}{16}\right\}^{x-1}2^{\lfloor x/2\rfloor}$$

as 17/15, 16/15, and 17/16 are all greater than 1. Thus for either signed or unsigned compact representations,

$$B_x < \frac{17}{15}\left(\frac{16}{15}\right)^{x-1}2^{\lfloor x/2\rfloor} < \frac{17}{15}\left(\frac{16}{15}\right)^{x-1}2^{x-1} = \frac{17}{15}\left(\frac{32}{15}\right)^{x-1}$$

and $\log_2 B_x$ is of size $O(x)$. Asymptotically then, the $(x+1)/(4\log_2 x)$ coefficient will dominate this expression as the discriminant increases. Looking at Figure 5.4, we see this coefficient has a minimum between $x=3$ and $x=4$.

In this thesis, we are most interested in computing compact representations where

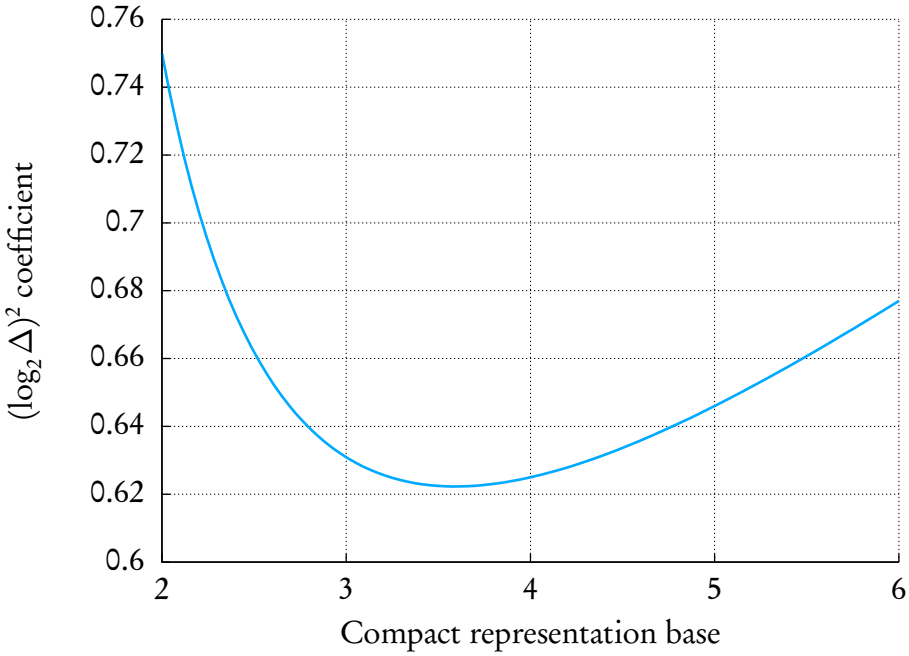


Figure 5.4: Plot of $(x+1)/(4\log_2 x)$.

$\theta = \eta_\Delta$, the fundamental unit. As such, the column of l values in Table 5.8 can be rewritten as

$$\log_x \log_2 \theta = \log_x \log_2 \eta_\Delta = \log_x \mathcal{R},$$

where $x \in \{2, 3, \ldots, 6\}$. Recall that we can loosely bound the regulator by $\sqrt{\Delta}$ and, after substitution, we are left with

$$l < \log_x \sqrt{\Delta} \tag{5.16}$$

as an upper bound on the number of terms in our various compact representations. Specializing (5.15) using (5.16), we find

$$S_x = \left(\frac{\log_2 B_x}{2\log_2 x}\right) \log_2 \Delta + \left(\frac{x+1}{8\log_2 x}\right) (\log_2 \Delta)^2 .$$

Again, asymptotically, the $(x+1)/(8\log_2 x)$ coefficient will dominate this expression as the discriminant increases and the minimum still occurs between $x = 3$ and $x = 4$. In fact, if we compare the two functions S_3 and S_4, we find that $S_4 < S_3$ for discriminants greater than $10^{16.5}$. In other words, for discriminants larger than about 16 decimal digits, the signed $4h$-compact representation is the most efficient one.

This conclusion supports our initial impression that base-5 and higher representations are not worth considering. From an analytic viewpoint, the trade-off between increasing the heights of the individual compact representation terms and gaining a representation with a fewer number of terms is no longer working in our favour. Because of this, we will not provide numerical results for the base-5 or higher compact representations in the next section.

5.5. Numerical testing

Since the preceding discussion only shows the savings in one particular case, we turn to some empirical results to further support our memory-saving claims. Figures B.1–B.5 (pages 237–241) show a statistical analysis of a random sampling of 28,000 discriminants evenly spread from length 5 through 18. We calculated an approximation of the associated regulator for each discriminant and used this to compute various compact representations of the fundamental unit. For each of the regular, h-, signed $3h$-, signed $4h$-, and signed $5h$-compact representations, we computed a best-fit regression line (red) for the data, as well as provided distribution box plots,[2] a 95% confidence interval (blue) for our regression line, and a 95% prediction interval (cyan) for further data points. Figure 5.5 shows a summary comparison of the average representation length for each discriminant length, along with the associated best-fit curves. Figure 5.6 shows the average relative savings of each

[2]For each box plot, potential outliers have been marked with a "×" symbol.

Table 5.9: Summary of sizes and relative memory savings for the various compact representations of $\eta_{410286423278424}$.

Representation	Bits	Relative savings	Representation	Bits	Relative savings
Standard	1,212	—	Signed 4 $\{-2,\ldots,1\}$	1,242	-2.5%
h	974	19.6%	$4h$	832	31.4%
3	1,186	2.1%	Signed $4h$ $\{-1,\ldots,2\}$	832	31.4%
Signed 3	1,141	5.9%	Signed $4h$ $\{-2,\ldots,1\}$	843	30.4%
$3h$	852	29.7%	5	1,385	-14.3%
Signed $3h$	843	30.4%	Signed 5	1,335	-10.1%
4	1,282	-5.8%	$5h$	877	27.6%
Signed 4 $\{-1,\ldots,2\}$	1,282	-5.8%	Signed $5h$	875	27.8%

new type of compact representation, as compared to the regular compact representation. In the previous section, our analytical discussion concluded that the signed $4h$-compact representation should be the most efficient for large enough discriminants. The numerical results that follow seems to show that the signed $4h$-compact representation is more efficient all the time. Why is this?

We initially speculated that this discrepancy is caused by our conservative bound on $H(\lambda_i)$. However, this only provides a piece of the answer. When we computed the bounds shown in Table 5.8 (on page 175), we followed the process as outlined by (5.2) (on page 144) and the equations following it. Specifically, for the signed base-3 case we have

$$\lambda_i < \frac{16^2 \cdot 17}{15^3} 2^{s_i - 3s_{i-1} - h} \Delta^{3/2} \quad \text{and} \quad |\overline{\lambda_i}| < \frac{17^3}{15 \cdot 16^2} 2^{3s_{i-1} - s_i + h} \Delta^{1/2}$$

and since $s_i - 3s_{i-1} \in \{0, \pm 1\}$,

$$H(\lambda_i) = \max\left\{ \frac{16^2 \cdot 17}{15^3} 2,\ \frac{17^3}{15 \cdot 16^2} 2 \right\} \Delta < \frac{11}{4} \Delta \,.$$

Similarly for the signed base-4 case, we have

$$\lambda_i < \frac{16^3 \cdot 17}{15^4} 2^{s_i - 4s_{i-1} - h} \Delta^2 \quad \text{and} \quad |\overline{\lambda_i}| < \frac{17^4}{15 \cdot 16^3} 2^{4s_{i-1} - s_i + h} \Delta^{1/2}$$

and since $s_i - 4s_{i-1} \in \{-1, 0, 1, 2\}$ (or equivalently $\{-2, -1, 0, 1\}$),

$$H(\lambda_i) = \max\left\{ \frac{16^3 \cdot 17}{15^4} 2^2,\ \frac{17^4}{15 \cdot 16^3} 2^1 \right\} \Delta^{5/4} < \frac{45}{8} \Delta^{5/4}$$
$$\left(H(\lambda_i) = \max\left\{ \frac{16^3 \cdot 17}{15^4} 2^1,\ \frac{17^4}{15 \cdot 16^3} 2^2 \right\} \Delta^{5/4} < \frac{45}{8} \Delta^{5/4} \right) .$$

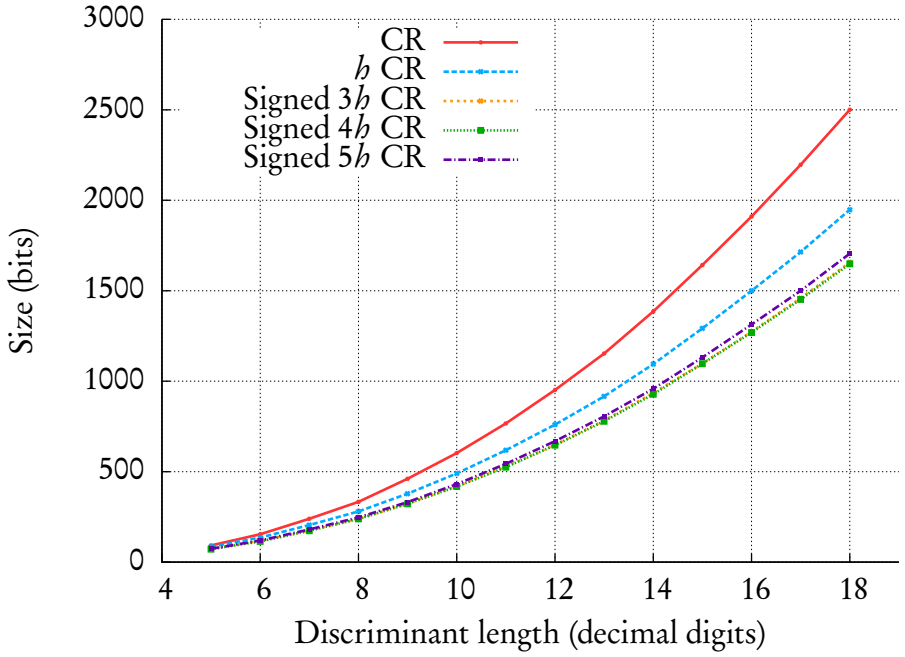


Figure 5.5: Comparison of the sizes of some various compact representations for random discriminants (1,000 discriminant values for each length from 5–18).

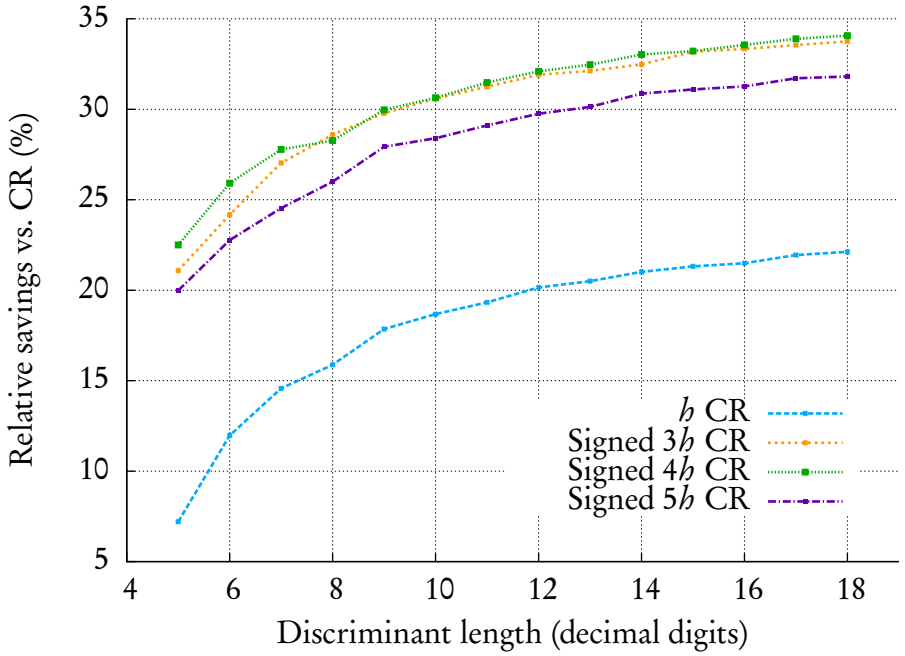


Figure 5.6: Relative savings of the various compact representations in Figure 5.5 as compared to the standard compact representation.

However, as we said, this is a conservative bound. In essence, we are assuming that each digit in the base-3 representation is a 1 and each digit in the base-4 representation is a 2 (or -2). Turning to a probabilistic argument, for a randomly selected number we would expect the proportions of the digits in its representation to approximately be equal. For example, for a signed base-3 representation, we would expect to see 0, 1, and -1 each roughly 33% of the time. If we take this into account, we derived the following bound on $H(\lambda_i)$ for the signed base-3 case:

$$H(\lambda_i) = \max\left\{\frac{16^2 \cdot 17}{15^3},\ \frac{17^3}{15 \cdot 16^2}\right\}\left(\frac{1}{3}2^{-1} + \frac{1}{3}2^0 + \frac{1}{3}2^1\right)\Delta < \frac{43}{20}\Delta\,.$$

Similarly for the signed base-4 case, we have

$$H(\lambda_i) = \max\left\{\frac{16^3 \cdot 17}{15^4},\ \frac{17^4}{15 \cdot 16^3}\right\}\left(\frac{1}{4}2^{-1} + \frac{1}{4}2^0 + \frac{1}{4}2^1 + \frac{1}{4}2^2\right)\Delta^{5/4} < \frac{13}{5}\Delta^{5/4}\,.$$

Using these probabilistic bounds in S_3 and S_4 in place of the bounds from Table 5.8, we find that S_4 is now strictly less than S_3 for discriminants larger than roughly 10^{14}.

We can extend these empirical results further by using the series of discriminants from [21, Tbl. 7.8, p. 101], as well as the regulator approximations given there, to produce the same variety of compact representations as above for the associated fundamental units. A comparison of the sizes of these representations is shown in Figure 5.7 with a comparison of the relative savings versus the standard compact representation shown in Figure 5.8.

We must be a little more cautious with these extended results, however. Because of the limited sampling, it is difficult to make a definitive claim on the relative efficiencies of the various h-compact representations at this point. For the majority of these larger discriminants, the signed $4h$-compact representation is the most efficient. However, for a

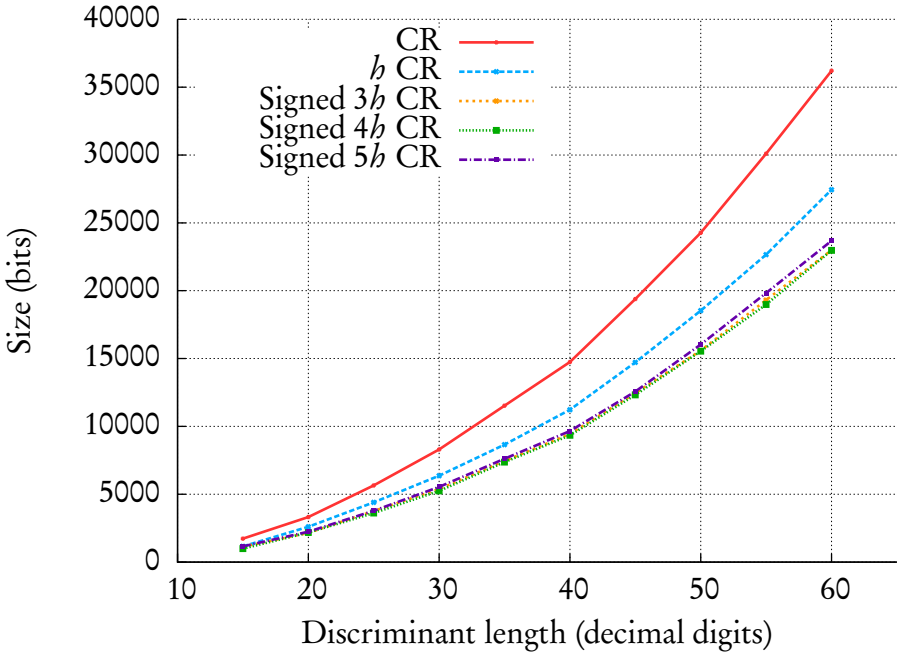


Figure 5.7: Comparison of the sizes of some various compact representations for the series of discriminants presented in [21, Tbl. 7.8, p. 101].

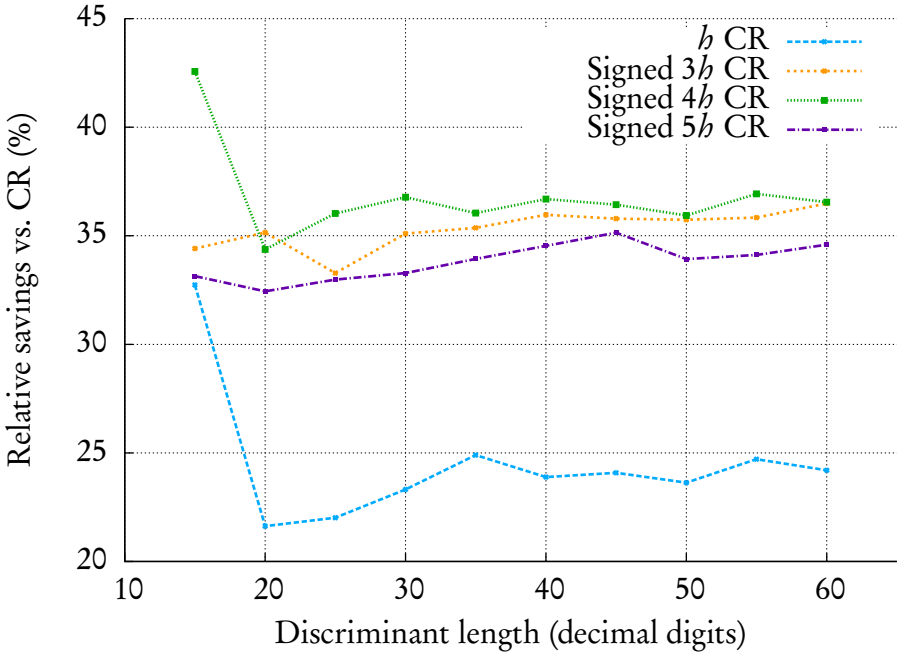


Figure 5.8: Relative savings of the various compact representations in Figure 5.7 as compared to the standard compact representation.

given discriminant, the signed $3h$-compact representation may be just as, or even slightly more, efficient. With further measurements, we expect to find that the signed $4h$-compact representation is most efficient on average.

As a final point of comparison, we turn to the regulator

$$\mathcal{R}' \approx 707950740910597226082932276551846667487998785334803 99.6730200233$$

for the quadratic field of 110-digit discriminant $\Delta = 4(10^{110} + 3)$ recently computed by Michael Jacobson, Jr., and Jean-François Biasse [6, p. 63]. After extensive testing, we determined that this regulator approximation is incorrect. However, at the time of writing, we have not been able to compute the correct value. We can still use this value to

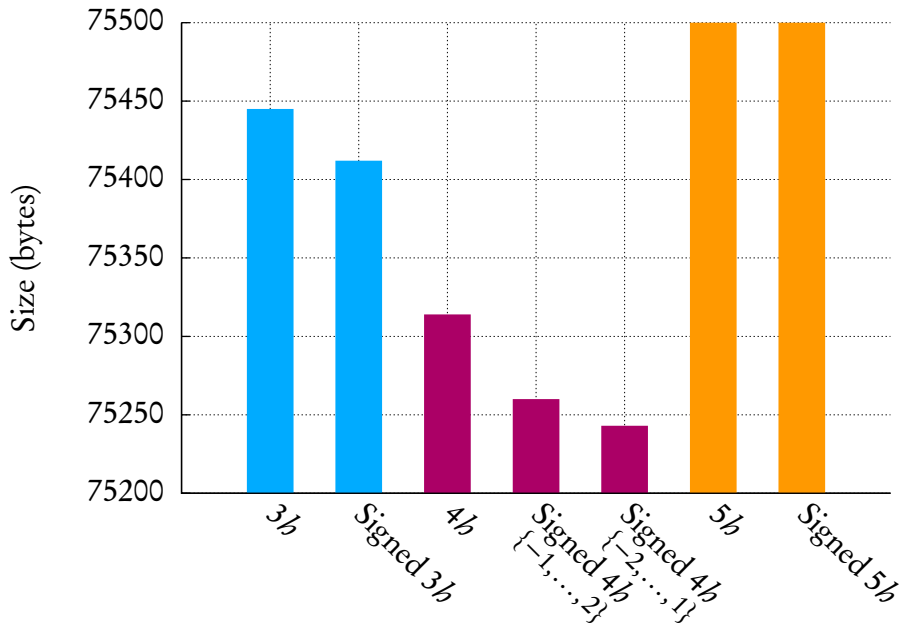


Figure 5.9: Size of various compact representations of an algebraic integer in the field with discriminant $\Delta = 4(10^{110} + 3)$.

generate compact representations, though they will be representations of some algebraic integer in the field, not of the fundamental unit. Figure 5.9 shows a comparison of the five compact representation varieties we have focused on. The standard compact representation requires 119,115 bytes to express this element, whereas the signed $4h$-compact representation, using the digits $\{-2,-1,0,1\}$, needs only 75,243 bytes, a 36.8% savings. This is in line with the relative savings suggested by Figure 5.8.

5.6. Concluding Remarks

In this chapter, we presented two substantial refinements to the idea of a compact representation for certain quadratic integers. The first was noticing that the size of the individual compact representation terms could be reduced by a substantial factor. This was done by modifying the inputs to EADDXY and exploiting a key behaviour of (E)NUCOMP. Specifically, by increasing the distances input to EADDXY, we force NUCOMP to take a slightly larger giant step than is needed. When this giant-step is adjusted by EWNEAR and the two resulting relative generators are combined, they partially cancel. Thus, each of the relative generators stored in the compact representation are reduced in size.

The second refinement was to notice that the overall number of terms could be reduced by computing larger giant steps on each iteration of CRAX. Rather than taking the binary expansion of input distance x and applying a square-and-multiply routine, we can consider the base-3, -4, or higher expansion of x and apply a cube-and-multiply, quadruple-and-multiply, *etc*. routine. By doing so, we progress through the infrastructure more quickly and so need to store fewer relative generators in the compact representation. There is a trade-off in doing this, however: the size of the individual relative generators increases as larger and larger giant steps are taken. For these higher-base compact representations, we are also able to slightly further reduce the size of the individual terms by

allowing for a signed representation of x.

Asymptotically, the signed $4h$-compact representation results in the most efficient balance between larger individual compact representation terms and a reduced overall number of terms. Numerical testing supports this conclusion. In the large-discriminant tests we performed, we found overall memory savings of around 37% as compared to the standard compact representation.

— Chapter 6 —
Conclusion & Future Directions

6.1. Summary

The first result we presented was our attempt at improving the run-time of the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm. The key to our modification is the observation that when we are checking for a match between a particular giant step in the list of baby-step ideals, we do not actually expect to succeed. Because of this, we can replace the explicit comparison of ideals with a "fuzzy comparison" of ideals by comparing the hashes of the two ideals. In the majority of cases, this is sufficient to determine that the ideals are different. The resulting savings in storage leads to some slight reductions in the computational time required by the algorithm.

There is an important caveat, however. If the rate of random hash collisions increases too much, particularly if the gap-length in a partial baby-step list is also large, the efficiency gains are quickly diminished.

Turning to the problem of representing the generators of reduced principal ideals, we made significant achievements. We crafted changes to CRAX, the algorithm for determining such compact representations, which are designed to minimize the distance short-fall we experience when computing giant steps with EADDXY and EWNEAR. By taking a giant step of slightly greater distance than was previously specified, we can use the relative generator from EWNEAR to reduce some of the size of the relative generator from EADDXY. This h-compact representation has a significantly reduced storage

requirement as compared to the regular compact representation.

In addition, we developed algorithms for computing (f, p) representations of the cube and higher powers of an (f, p) representation, though the explicit pseudocode for most of these was omitted due to their extreme similarity to the cubing algorithms. We used these algorithms to show how shifting from a square-and-multiply to a cube- or fourth-power-and-multiply routine reduces the overall number of terms as compared to the regular representation. Unfortunately, these shorter base-3 and base-4 compact representations come at the expense of larger individual terms. Combining h-compact representations with these base-3 and base-4 representations results in the best-case scenario. Theoretically, the signed $4h$-compact representation gives us the largest storage savings. Numerical testing, in the case of larger discriminants, supports this conclusion and we have observed a fairly consistent storage savings of around 37%.

As a concluding remark, we reiterate a statement made in the introduction to this work: these concepts can be used in other settings. They can be extended higher-degree number fields which have unit rank one, in particular complex cubic and totally complex quartic fields. Furthermore, we can translate these results to *algebraic function fields*; that is, finite algebraic extensions $\mathbb{K}$ of $\mathbb{F}_q(t)$, the quotient field of the polynomial ring $\mathbb{F}_q[t]$ where q is a prime power and $\mathbb{F}_q$ is a finite field. These fields share many similar properties and often problems posed in a number field setting have an analogous problem in a function field setting. It has been shown that compact representations exist for real quadratic congruence fields [74], work which has since been generalized to function fields of arbitrary degree [30].

6.2. Future directions: Improving our 64-bit implementation of the $O(\Delta^{1/6+\epsilon})$ algorithm and refining the optimal parameter formulas

The numerical testing results presented in Chapter 4 show that a 32-bit hash is probably insufficient for our modified $O(\Delta^{1/6+\epsilon})$ ideal hashing algorithm to achieve its maximum efficiency for larger discriminants. The excessive random hash collision rate in our single-processor tests, coupled with the large values determined for the parameter l, more than eliminate any run-time gains we achieved for smaller discriminants. In fact, in this situation, the unmodified $O(\Delta^{1/6+\epsilon})$ algorithm is superior.

As we mentioned in Chapter 4, we transferred our implementation from the ISPIA cluster at the University of Calgary to the Nestor cluster at the University of Victoria, made available to us via WestGrid. While our parallelized implementation now runs on this machine, there is a significant amount of further tuning and base-line calculations that need to be performed before we can show how far our modified algorithm can be pushed. We observed 4.6% improvement in overall run-time as compared to the previous $O(\Delta^{1/6+\epsilon})$ algorithm, and further investigation will show more clearly how the efficiency gains of our modified algorithm scale to higher discriminants.

One of the largest hurdles we faced in generating the results in Chapter 4—and the 64-bit results we were unable to generate—was determining optimal values for the parameters used by the $O(\Delta^{1/6+\epsilon})$ algorithm. As we have seen, a lot of work has been done in this area, but we would like to take this further. The optimization formulas given in Section 4.3.2 get us fairly close to the optimal values for s, Q, K and l, but manual refinements are unfortunately still needed. Ultimately, these refinements need to be incorporated back into the optimization formulas.

One other key area we would like to investigate is whether some of the assumptions

made when deriving the various optimizing formulas need to be changed. For instance, we currently assume that if there is enough memory, we should take $l = 1$ and store a complete copy of the baby-step list. However, some of our numerical tests show that, in fact, we can get a faster run-time by taking $l > 1$, even when we have the space to store $\mathcal{L}'$ or $\mathcal{L}''$ in its entirety. Figure 6.1 shows one such result for a 40-digit discriminant. In this case, we have the memory space for a complete list, but we can achieve an approximately 15% improvement in run-time by forcing $l = 10$.

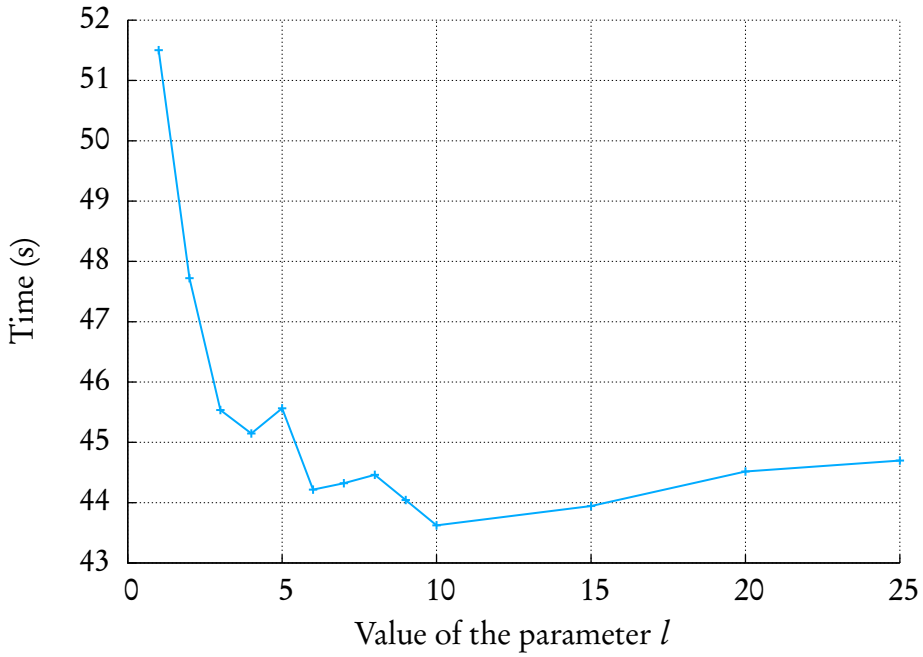


Figure 6.1: Observed run-time improvement when forcing the storage of a partial baby-step list.

6.3. Future directions: Parallelizing the storage of $\mathcal{L}$

In his thesis, de Haan remarked:

> *Because the baby steps need to be stored and then later on accessed during the*

> *giant steps this means that each process has to either compute its own baby step list or that the processes have to send the part of the baby step list they computed and to receive the rest. Communicating parts of the baby step list is expected to be slower than having the processes compute their own list, which is why we have decided to use the latter method.* [21, §6.4.1, pp. 84–5]

However, we feel that we have come up with two methods to distribute the baby-step list $\mathcal{L}$ that may have an acceptable level of internode communication overhead.

In the current parallelized algorithm, the giant-step phase is distributed by assigning each node a contiguous range of roughly $(Q-1)/n$ giant steps to compute. What if we tried to distribute $\mathcal{L}$ in a similar fashion? We could try assigning each node a range of $(s+b)/n$ baby steps, starting at the ideal $\mathfrak{a}_{start} = \mathrm{AX}[i(s+b)/n]$ and ending at $\mathfrak{a}_{end} = \mathrm{AX}[(i+1)(s+b)/n - 1]$. However, we need to keep in mind that in order to simultaneously distribute the giant-step phase, we must ensure that each giant step $\mathfrak{b}_j$ is compared against, in essence, the entire list $\mathcal{L}$. As stated above, each node would only compare its set of giant steps against its set of baby steps; there are "gaps" in the comparison. Instead, we need to assign the same baby-step and giant-step ranges to multiple nodes rather than assigning each unique ranges. If mn computational nodes are available to us, we can assign the first range of baby-steps to nodes $1, 2, \ldots, n$, the second to $n+1, n+2, \ldots, 2n$, and so on. Similarly, the first range of giant steps are assigned to nodes $1, n+1, \ldots, (mn-1)+1$, the second to $2, n+2, \ldots, (mn-1)+2$, and so on. See Figure 6.2 for an illustration of this assignment process. By careful selection of the start and end ideals, we can guarantee that each giant step will be compared against the entire baby-step list as required.

Unfortunately, there is a serious drawback to this process. Let node $bn+g$, where $1 \leq b \leq m$ and $1 \leq g \leq n$, be the node storing the b^{th} portion of $\mathcal{L}''$ and computing the

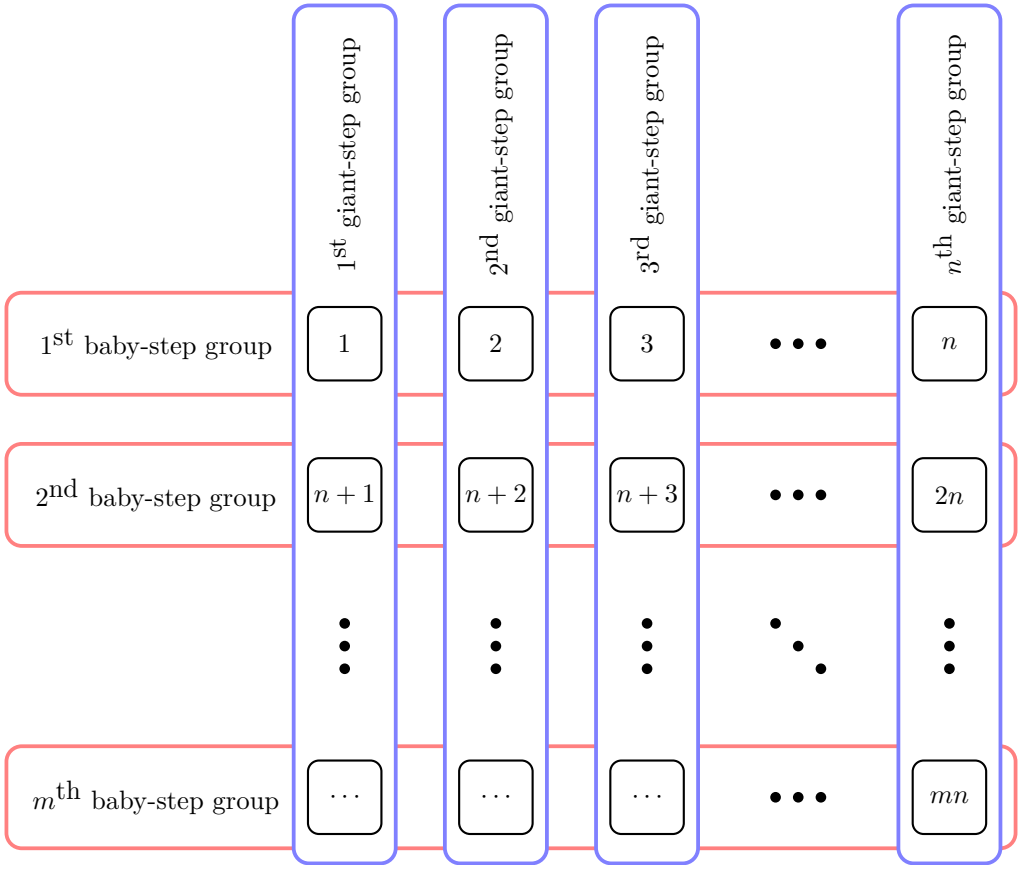


Figure 6.2: Parallelizing both the baby-step and giant-step phases. The horizontal (red) blocks represent nodes sharing a common portion of the (partial or hashed) baby-step list, while the vertical blocks (blue) represent nodes sharing a common portion of the giant-step list.

g^{th} set of giant steps. For a fixed value of b, the nodes $\{bn+1, bn+2, bn+3, \ldots\}$ comprise the b^{th} *baby-step group*; for a fixed value of g, the nodes $\{g, n+g, 2n+g, \ldots\}$ comprise the g^{th} *giant-step group*. If node $b'n+g'$ finds a giant step $\mathfrak{b}_j$ such that $H(\mathfrak{b}_j) \in \mathcal{L}''$, it is not immediately clear which group of nodes will be storing $H(\rho^l(\mathfrak{b}_j))$. This node would have to "ask"—using a pair of `MPI_Comm_bcast(...)` and `MPI_Comm_recv(...)` function calls—each node in the g'^{th} giant-step group whether or not they are storing the value $H(\rho^l(\mathfrak{b}_j))$. Given a low random collision rate, this may not be too expensive of a communication overhead. But, if the collision rate rises, it may quickly become prohibitively expensive.

For the second method of distributing the baby-step ideals across multiple nodes, we exploit the hashing modification introduced in Chapter 4. With negligible overhead, we can distribute Δ and a few key parameters—an approximation of the regulator, s, and Q—to each of the cluster nodes and have each begin generating $\mathcal{L}$. However, rather than store the entire list, we do the following. For each ideal $\mathfrak{a}_i \in \mathcal{L}$, the nodes compute $H(\mathfrak{a}_i)$ and rather than immediately storing this hash, they examine its low-order bits and compare them against their own MPI node id. By *node id*, we mean the integer returned by the `MPI_Comm_rank(...)` function call. If the bits match, then that particular node stores $H(\mathfrak{a}_i)$ in its baby-step hash list $\mathcal{L}''_{node\ id}$ while the rest discard it. If the values $H(\mathfrak{a}_i)$ are uniformly distributed, we should have a uniform distribution of $\mathcal{L}''$ across the cluster nodes. Moreover, since the MPI *node id* values are guaranteed to be unique, we have

$$\mathcal{L}'' = \mathcal{L}''_1 \cup \mathcal{L}''_2 \cup \cdots \cup \mathcal{L}''_{cluster\ size} \quad \text{where} \quad \mathcal{L}''_i \cap \mathcal{L}''_j = \varnothing \text{ if } i \neq j \ .$$

If we restrict the number of nodes n used by our implementation to a power of two, we can truncate $H(\mathfrak{a}_i)$ to an exact bit-length for comparison to the MPI *node id* using a fast

bitwise-AND operation. It is important to note that each node is still computing a full baby-step list, even though they only store $1/n^{\text{th}}$ of them. Because of this, it is not clear if this idea for generating and storing $\mathcal{L}''$ will lead to a computational improvement. It will certainly be slower than computing only $1/n^{\text{th}}$ of $\mathcal{L}$ as in the previous method, but faster than storing a complete list. As most of the computed $H(\mathfrak{a}_i)$ values are discarded, we will incur a much smaller memory-latency penalty in storing that $1/n^{\text{th}}$ portion in RAM.

Once the baby-step list is generated, the giant steps are assigned in contiguous ranges of $(Q-1)/n$ steps and computed as before. If node a finds a giant step $\mathfrak{b}_j$ such that $H(\mathfrak{b}_j) \in \mathcal{L}''_a$, we are in a better position than in the previous method. Upon computing $H(\rho^l(\mathfrak{b}_j))$, the node immediately knows on which node that hash value is stored, if it is stored at all. Node a can directly ask node, say, b if it is storing the value $H(\rho^l(\mathfrak{b}_j))$ or not via a pair of `MPI_Comm_send(...)` and `MPI_Comm_recv(...)` function calls. This will greatly reduce the chances of node intercommunication conflicts and may result in a tolerable communication overhead even in the face of a moderate random hash collision rate.

In a computing environment where each node has multiple processors or cores, we can improve this method slightly further. After each node has finished generating its portion of $\mathcal{L}''$, it can spawn two one-way communication threads: one for sending messages, the other for receiving. While the main process is computing giant steps on one processor or core, the sending and receiving threads will generally be idle. If $H(\mathfrak{b}_j) \in \mathcal{L}''_a$ is found on node a, the main process signals the sending thread to contact the node b which is potentially storing $H(\rho^l(\mathfrak{b}_j))$. The receiving thread on node b gets this query, performs the lookup, and signals its sending thread to reply "yes" or "no." The receiving thread on node a gets this response and passes it to the main process, which can then determine

how to proceed. The advantage to this setup is that the main process on node b will keep computing giant steps and not be interrupted by the communication and lookup activity generated by the request from node a. Furthermore, if the main process on node b is waiting for a reply to its own query, the receiving and sending threads can handle the query from node a without any additional delay.

We end this section by repeating an earlier comment. Although both of these methods will allow the baby steps and giant steps to be parallelized, it is hard to say if a computational improvement will be seen. It is not clear what the best distribution of nodes into baby-step and giant-step groups will be. Roughly an even number of each? Significantly more giant-step groups than baby-step groups, since the giant steps are more computationally intensive? It is also unclear to which extent the duplication of computations will slow down the algorithm. On the bright side, either method should more effectively maximize the memory storage available and, thus, help keep the optimizing parameter l smaller. In this way, we should be able to better maintain the balance between the three phases of the $O(\Delta^{1/6+\epsilon})$ algorithm and, hopefully, better minimize its run-time.

6.4. Future directions: Alternative types of compact representations

There are other types of number representations we did not investigate which may lead to further memory savings for compact representations. For example, we could consider using a *non-adjacent form* (*NAF*) representation. The NAF representation of the integer x, introduced by George Reitwiesner [71], is a signed base-2 representation such that for any two consecutive digits b_i and b_{i+1}, we have $b_i b_{i+1} = 0$ [36, p. 98]. That is, any two non-zero values in the representation are separated by a zero. These representations have some important properties: namely, they are unique, have minimal Hamming weight,

and the average density of non-zero digits among all NAFs of a given length is approximately 1/3 [36, Thm. 3.29]. As a comparison for this last point, we mention that the average density of non-zero digits in the standard binary representation is approximately 1/2. While this reduction in the non-zero digits of the representation would be beneficial to reducing the memory requirements of a compact representation, there is a more critical property to keep in mind. The length of a NAF representation of an integer is at most one more than the length of its binary representation [36, Thm. 3.29].

The idea of a NAF representation has been generalized to a *width-w NAF* (wNAF), by using the following two conditions:

1. there is at most one non-zero digit amongst any w consecutive digits in the representation and
2. each non-zero digit appearing in the representation is odd and less than 2^{w-1} in absolute value.

The NAF representation described in the previous paragraph is a wNAF representation with $w = 2$. The obvious benefit to this generalized representation is to further reduce the number of non-zero digits that appear: the average density of non-zero digits is $1/(w+1)$. In terms of an overall length, however, the wNAF representation of an integer is again at most one more than the length of its binary representation. Our intuition is that, on average, the extra storage needed for an additional μ_i term will outweigh any savings obtained by not storing even several of the ν_i terms, particularly when combined with the higher-base compact representations.

Another example is the *double-base* representation of the integer x, given by

$$x = \sum_{i,j} b_{i,j} 2^i 3^j ,$$

where $b_{i,j} \in \{0,1\}$. It is important to note that, for any given integer x, we can have multiple equivalent representations. Even restricting ourselves to *canonic double-base representations*, those using the minimum number of $2^i 3^j$ terms, we can be left with multiple choices. For instance, if $x = 127$ there are 783 representations and 3 canonic representations:

$$127 = 2^2 3^3 + 2^1 3^2 + 2^0 3^0 = 2^2 3^3 + 2^4 3^0 + 2^0 3^1 = 2^5 3^1 + 2^0 3^3 + 2^2 3^0 \quad [40] .$$

These representations only require $O(\log n / \log\log n)$ digits to store and near-canonic representations can be computed via a number of methods [2, 24, 25, 27]. The advantage to this representation over a standard binary or ternary representation is that we require fewer terms.

It could be quite beneficial if this numeric representation could be applied to create a double-base compact representation. First, by reducing the overall number of terms of the representation, we would reduce the overall storage requirements, just as we did in Chapter 5 by moving from a compact to a 3-compact to a 4-compact representation. Furthermore, by restricting ourselves to the bases 2 and 3, we have the potential of avoiding the per-term expansion we encountered due to the increasing bound on $H(\lambda_i)$.

In order to actually use the representation above to compute a compact representation, we require the concept of a chain. A *double-base chain* for an integer x is an expansion of the form

$$x = \sum_{i=1}^{m} b_i 2^{a_i} 3^{b_i} ,$$

where $b_i = \pm 1$ and such that $a_1 \geq a_2 \geq \cdots \geq a_m$ and $b_1 \geq b_2 \geq \cdots \geq b_m$. Vassil Dimitrov, Laurent Imbert, and Pradeep Mishra [24] presented an efficient elliptic curve point

multiplication algorithm using double-base chains, which could be adapted to compute a compact representation. The outline of such an algorithm would resemble that of Algorithm 6.1 below, which is based on CRAX for simplicity's sake. Of course, in addition to storing the λ_i and L_i values of the compact representation, we must also store one additional bit per term to specify whether it must be squared or cubed.

Algorithm 6.1: Double-base CRAX

Input: x, p, where $x \in \mathbb{Z}^+$.

Output: $(\mathfrak{a}[x], d, k)$, an x-near (f, p) representation of $\mathfrak{a} = (1)$ and a set of integer pairs (m_i, n_i) and integers $L_i \in \mathbb{Z}^+$, $i = 0, 1, 2, \ldots, l$, where $l = \lfloor \log_2 x \rfloor$.

1: Compute a double-base chain representation of x with

$$x = \sum_{i=0}^{l} c_i 2^{a_{l-i}} 3^{b_{l-i}},\ c_0 = 1,\ c_i = \pm 1 \quad (1 \le i \le l)\,.$$

2: Put

$$Q = r,\ P = r \left\lfloor \frac{\lfloor \sqrt{D} \rfloor - r + 1}{r} \right\rfloor + r - 1,\ \mathfrak{b} = \left[1, \frac{P + \sqrt{D}}{r} \right],$$

$d = 2^p + 1$, $k = 0$, $i = j = 0$, $s_0 = 1$, $L_0 = 1$, $a_{-1} = 0$, $b_{-1} = 0$.

3: Put $((\mathfrak{b}_0, d_0, k_0), m_0, n_0) = \text{EWNEAR}((\mathfrak{b}, d, k), 1, p)$.

4: **while** $i < l$ **do**

5: Set $a = a_{l-i} - a_{l-(i-1)}$, $b = b_{l-i} - b_{l-(i-1)}$.

6: **while** $a \ge 0$ **do**

7: Put $L_{j+1} = N(\mathfrak{b}_j)$ and

$$((\mathfrak{b}_{j+1}, d_{j+1}, k_{j+1}), m_{j+1}, n_{j+1}) = \text{EADDXY}((\mathfrak{b}_j, d_j, k_j), (\mathfrak{b}_j, d_j, k_j), s_j, s_j, p)\,.$$

8: Put $s_{j+1} = 2s_j$.

9: Set $j \leftarrow j + 1$, $a \leftarrow a - 1$.

10: **end while**

11: **while** $b \ge 0$ **do**

12: Put $L_{j+1} = N(\mathfrak{b}_j)$ and

$$((\mathfrak{b}_{j+1}, d_{j+1}, k_{j+1}), m_{j+1}, n_{j+1}) = \text{ETRIPLEX}((\mathfrak{b}_j, d_j, k_j), s_j, p)\,.$$

13: Put $s_{j+1} = 3s_j$.
14: Set $j \leftarrow j+1$, $b \leftarrow b-1$.
15: **end while**
16: Put $s_{j+1} \leftarrow s_{j+1} + c_{j+1}$, $N = N(\mathfrak{b}_{j+1})$, and

$$((\mathfrak{b}_{j+1}, d_{j+1}, k_{j+1}), m', n') \leftarrow \text{EWNEAR}((\mathfrak{b}_{j+1}, d_{j+1}, k_{j+1}), s_{j+1}, p)\,.$$

17: Put $(m_{j+1}, n_{j+1}) = \text{IMULT}(m_{j+1}, n_{j+1}, m', n', N)$.
18: $i \leftarrow i+1$, $j \leftarrow j+1$.
19: **end while**
20: Put $\mathfrak{a}[x] = \mathfrak{b}_j$, $d = d_j$, $k = k_j$.

As an example, we use Algorithm 6.1 to compute the compact representation of the generator of AX[1717]. We point out that this example is based on the elliptic curve point multiplication example given in [40, §II, pp. 1–2]. Following the s_j values in the algorithm above, we see it computes

$$\begin{array}{lll} s_0 \leftarrow 1 & s_4 = 2s_3 = 16 & s_7 = 2s_6 = 286 \\ s_1 = 2s_0 = 2 & s_5 = 3s_4 = 48 & s_8 = 2s_7 = 572 \\ s_2 = 2s_1 = 4 & s_6 = 3s_5 = 144 & s_9 = 3s_8 = 1716 \\ s_3 = 2s_2 = 8 & s_6 \leftarrow s_6 - 1 = 143 & s_9 \leftarrow s_9 + 1 = 1717 \end{array}$$

as desired. At the end of Algorithm 6.1, we have the tuples $\{\lambda_i, (m_i, n_i)\}$ for $i = 0, 1, \ldots, 9$. Expressing these as a product, we have γ equal to the product

$$\frac{\lambda_9}{L_9^2}\left(\frac{\lambda_8}{L_8^3}\right)^3\left(\frac{\lambda_7}{L_7^2}\right)^6\left(\frac{\lambda_6}{L_6^2}\right)^{12}\left(\frac{\lambda_5}{L_5^3}\right)^{36}\left(\frac{\lambda_4}{L_4^3}\right)^{108}\left(\frac{\lambda_3}{L_3^2}\right)^{216}\left(\frac{\lambda_2}{L_2^2}\right)^{432}\left(\frac{\lambda_1}{L_1^2}\right)^{864}\left(\frac{\lambda_0}{L_0^2}\right)^{1728},$$

where, of course, $\text{AX}[1717] = (\gamma)$. We can simplify this statement by observing that the exponents are actually the coefficients resulting from a staircase walk for the double-

base chain $1717 = 2^6 3^3 - 2^2 3^1 + 2^0 3^0$. If we visualize the double-base chain terms on a 2x2 grid, a *staircase walk* is simply the path traced out by moving from the bottom-right corner to the top-left [40]. This concept is illustrated in Figure 6.3. In general, using the coefficients from a staircase walk to simplify a product like the one given above results in the formal product

$$\gamma = \prod_{i=0}^{l} \left(\frac{\lambda_i}{L_i^{e_i}} \right)^{2^{f_i} 3^{g_i}}$$

where $e_l = 2$, $e_i = (2^{f_i} 3^{g_i})/(2^{f_{i+1}} 3^{g_{i+1}})$, and $2^{f_i} 3^{g_i}$ is the i^{th} staircase walk coefficient for the double-base chain representation of x ($i = 0, 1, \ldots, l-1$). It would be quite interesting to see what storage savings could be realized by such a double-base compact representation.

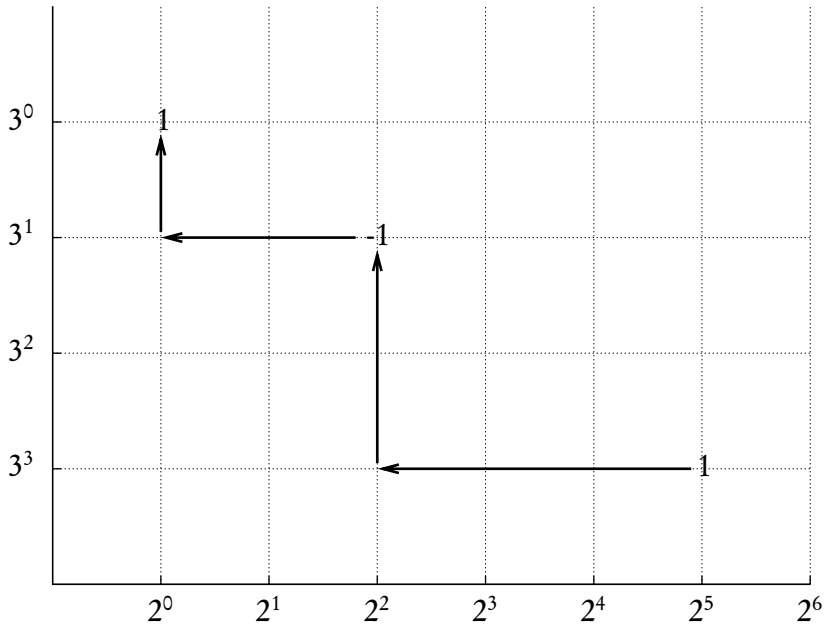


Figure 6.3: A staircase walk for the double-base chain $1717 = 2^6 3^3 - 2^2 3^1 + 2^0 3^0$.

BIBLIOGRAPHY

[1] Argonne National Laboratory, *MPICH2, v. 1.2.7*, 2002, `http://www.mcs.anl.gov/research/projects/mpi/mpich2`.

[2] R. Avanzi, V. Dimitrov, C. Doche, and F. Sica, *Extending scalar multiplication using double bases*, Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3–7, 2006, LNCS, vol. 4284, Springer, 2006, pp. 130–144.

[3] E. Bach, *Explicit bounds for primality testing and other problems*, Math. Comp. **55** (1990), 355–380.

[4] ———, *Improved approximations for Euler products*, Number Theory: Fourth Conference of the Canadian Number Theory Association (Dalhousie University, Halifax, Canada, July 2–8, 1994) (Karl Dilcher, ed.), CMS Conf. Proc., vol. 15, AMS, 1995, pp. 13–28.

[5] E. Bach and J. Shallit, *Algorithmic Number Theory Volume I: Efficient Algorithms*, Foundations of Computing, MIT Press, Cambridge MA, 1996.

[6] J.-F. Biasse and M. J. Jacobson, Jr., *Practical improvements to class group and regulator computation of real quadratic fields*, Algorithmic Number Theory: 9th International Symposium (ANTS-IX, Nancy, France, July 19–23, 2010) (G. Hanrot, F. Morain, and E. Thomé, eds.), LNCS, no. 6197, Springer, 2010, pp. 50–65.

[7] J.-F. Biasse, M. J. Jacobson, Jr., and A. K. Silvester, *Security estimates for quadratic*

field based cryptosystems, Information Security and Privacy: 15th Australasian Conference on Information Security and Privacy Proceedings (ACISP 2010, Sydney, Australia, July 5–7, 2010), LNCS, vol. 6168, Springer, 2010.

[8] I. Biehl, J. Buchmann, and C. Thiel, *Cryptographic protocols based on discrete logarithms in real-quadratic orders*, Advances in Cryptology—CRYPTO '94 (Y. G. Desmedt, ed.), LNCS, vol. 839, Springer-Verlag, 1994, pp. 56–60.

[9] R. D. Brauer, *On the zeta-functions of algebraic number fields*, Amer. J. Math. **69** (1947), no. 2, 243–250.

[10] J. Buchmann, *A subexponential algorithm for the determination of class groups and regulators of algebraic number fields*, Séminaire de Théorie des Nombres, 1989, pp. 27–41.

[11] J. Buchmann, M. Maurer, and B. Möller, *Cryptography based on number fields with large regulator*, vol. 12, 2000, pp. 293–307.

[12] J. Buchmann, C. Thiel, and H. C. Williams, *Short representation of quadratic integers*, Mathematics and its Applications, vol. 325, pp. 159–185, Kluwer Academic Publishers, Amsterdam, 1995.

[13] J. Buchmann and U. Vollmer, *A Terr algorithm for computations in the infrastructure of real-quadratic number fields*, Journal de Théorie des Nombres de Bordeaux **18** (2006), no. 3, 559–572.

[14] ———, *Binary Quadratic Forms*, Algorithms and Computation in Mathematics, vol. 20, Springer, 2007.

[15] J. Buchmann and H. C. Williams, *On the infrastructure of the principal ideal class of an algebraic number field of unit rank one*, Math. Comp. **50** (1988), no. 182, 569–579.

[16] J. Buchmann and H. C. Williams, *On the existence of a short proof for the class num-*

ber of a real quadratic number field, Number Theory and Applications, NATO Advanced Science Institutes Series C, vol. 265, Kluwer, Dordrecht, 1989, pp. 327–345.

[17] ———, *Some remarks concerning the complexity of computing class groups of quadratic fields*, Journal of Complexity **7** (1991), no. 3, 311–315.

[18] H. Cohen, *A Course in Computational Algebraic Number Theory*, 4 ed., Graduate Texts in Mathematics, vol. 138, Springer, New York, 2000.

[19] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual: Combined volumes 1, 2, and 3*, Tech. Report 325462-040US, 2011, `http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-manual-325462.pdf`.

[20] R. E. Crandall and C. Pomerance, *Prime Numbers: A Computational Perspective*, 2nd ed., Springer, 2005.

[21] R. de Haan, *A fast, rigourous technique for verifying the regulator of a real quadratic field*, Master's thesis, University of Amsterdam, 2004.

[22] R. de Haan, M. J. Jacobson, Jr., and H. C. Williams, *A fast, rigorous technique for computing the regulator of a real quadratic field*, Math. Comp. **76** (2007), no. 260, 2139–2160.

[23] E. W. Dijkstra, *A new science, from birth to maturity*, ETH Zürich: 20-year Anniversary of Institut für Informatik, ETH Zürich, Institut für Informatik, October 1988, available at `http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1024.PDF`.

[24] V. Dimitrov, L. Imbert, and P. K. Mishra, *Efficient and secure elliptic curve point multiplication using double-base chains*, Advances in Cryptology - ASIACRYPT 2005, Proceedings of the 11th International Conference on the Theory and Application of Cryptology and Information Security (Chennai, India, December 4–8, 2005)

(B. Roy, ed.), LNCS, vol. 3788, Springer, 2005, pp. 59–78.

[25] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, *An algorithm for modular exponentiation*, Information Processing Letters **66** (1998), 155–159.

[26] Vanessa Dixon, *Fast exponentiation in the infrastructure of a real quadratic field*, Master's thesis, University of Calgary, 2011.

[27] C. Doche and L. Imbert, *Extended double-base number system with applications to elliptic curve cryptography*, Progress in Cryptology - INDOCRYPT 2006 (R. Barua and T. Lange, eds.), LNCS, vol. 4329, Springer, 2006, pp. 335–348.

[28] A. Dujella and A. Pethő, *Integer points on a family of elliptic curves*, Publicationes Mathematicae, Institutum Mathematicum Universitatis Debreceniensis **56** (2000), 321–335.

[29] D. S. Dummit and R. M. Foote, *Abstract Algebra*, 2nd ed., Prentice-Hall, Inc., 1999.

[30] K. Eisenträger and S. Hallgren, *Computing the unit group, class group and compact representations in algebraic function fields*, Proceedings of the Tenth Algorithmic Number Theory Symposium (ANTS X), UC San Diego, July 9–13, 2012 (E. Howe and K. Kedlaya, eds.), OBS, vol. 1, Mathematical Sciences Publishers, 2012, to appear.

[31] Free Software Foundation, Inc., *GNU MP: The GNU Multiple Precision Arithmetic Library manual*, 2009, `http://gmplib.org/index.html\#DOC`.

[32] ———, *GNU MP: The GNU Multiple Precision Arithmetic Library, v. 4.2.2*, 2009, `http://gmplib.org/`.

[33] Y. Fujita, *The d(1)-extensions of d(−1)-triples* $\{1, 2, c\}$ *and integer points on the attached elliptic curves*, Acta Arithmetica **128** (2007), 349–375.

[34] Google, Inc., *Cityhash, v. 1.0.3*, 2011, `http://code.google.com/p/cityhash/`.

[35] J. L. Hafner and K. S. McCurley, *A rigorous subexponential algorithm for computation of class groups*, J. Amer. Math. Soc. **2** (1989), 839–850.

[36] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag, New York, 2004.

[37] Hipp, Wyrick & Company, Inc. (Hawci), *SQLite*, 2010, `http://www.sqlite.org/`.

[38] D. Hühnlein and S. Paulus, *On the implementation of cryptosystems based on real quadratic number fields (extended abstract)*, Proceedings of the 7th Annual Workshop on Selected Areas in Cryptography (SAC '00), Centre for Applied Cryptographic Research (CACR), University of Waterloo, Waterloo, ON, Canada, August 14–15, 2000 (D. R. Stinson and S. Tavares, eds.), LNCS, vol. 2012, Springer-Verlag, 2001, pp. 288–302.

[39] IEEE and The Open Group, *IEEE Std 1003.1-2008 / The Open Group technical standard base specifications, Issue 7*, 2008, `http://www.opengroup.org/onlinepubs/9699919799/mindex.html`.

[40] L. Imbert and C. Doche, *The double-base number system in elliptic curve cryptography*, Conference Records of the 42nd Asilomar Conference on Signals, Systems and Computers (Pacific Grove, CA, October 26–29, 2008) (M. B. Matthews, ed.), IEEE, 2008, pp. 777–780.

[41] L. Imbert, M. J. Jacobson Jr., and A. Schmidt, *Fast ideal cubing in imaginary quadratic number and function fields*, Advances in Mathematics of Communications **4** (2010), no. 2, 237–260, preprint available at `http://www.lirmm.fr/~imbert/pdfs/cubing_amc_2010.pdf`.

[42] M. J. Jacobson, Jr., *Computing discrete logarithms in quadratic orders*, Journal of Cryptology **13** (2000), 473–492.

[43] M. J. Jacobson, Jr., R. F. Lukes, and H. C. Williams, *An investigation of bounds for the regulator of quadratic fields*, Experimental Mathematics **4** (1995), no. 2, 211–225.

[44] M. J. Jacobson, Jr., Á. Pintér, and P. G. Walsh, *A computational approach for solving* $y^2 = 1^k + 2^k + \cdots + x^k$, Math. Comp. **72** (2003), no. 244, 2099–2110.

[45] M. J. Jacobson, Jr., R. Scheidler, and H. C. Williams, *The efficiency and security of a real quadratic field based key exchange protocol*, Public-Key Cryptography and Computational Number Theory: September 11–15, 2000, Warsaw, Poland (K. Alster, J. Urbanowicz, and H. C. Williams, eds.), Walter de Gruyter GmbH & Co., 2001.

[46] M. J. Jacobson, Jr., R. Scheidler, and H. C. Williams, *An improved real quadratic field based key exchange procedure*, J. Cryptology **19** (2006), 211–239.

[47] M. J. Jacobson, Jr. and H. C. Williams, *Modular arithmetic on elements of small norm in quadratic fields*, Designs, Codes and Cryptography **27** (2002), 93–110.

[48] ———, *Solving the Pell Equation*, CMS Books in Mathematics, Springer-Verlag, 2009.

[49] B. Jenkins, *Algorithm alley*, Dr. Dobb's Journal, 1997, `http://drdobbs.com/database/184410284`.

[50] ———, *Lookup3 hash*, 2006, `http://burtleburtle.net/bob/c/lookup3.c`.

[51] A. Y. Khintchine, *Zur metrischen Theorie der Diophantischen Approximationen*, Math. Z. **24** (1936), 706–714.

[52] D. Knuth, *Structured programming with go to statements*, Computing Surveys **6** (1974), no. 4, 261–301.

[53] J. Kornblum, *md5deep*, 2012, `http://md5deep.sf.net`.

[54] J. L. Lagrange, *Sur la solution des problèmes indéterminés du second degré*, Oeuvres, vol. II, Gauthier-Villars, Paris, 1868, pp. 377–535.

[55] H. W. Lenstra, Jr., *On the calculation of regulators and class numbers of quadratic fields*, London Math. Soc. Lecture Note Series **56** (1982), 123–150.

[56] H. W Lenstra, Jr., *Factoring integers with elliptic curves*, Annals of Math. (2) **126** (1987), 649–673.

[57] H. W. Lenstra, Jr., *Solving the Pell equation*, Notices of the AMS **49** (2002), no. 2, 182–192.

[58] P. Lévy, *Sur le développement en fraction continue d'un nombre choisi au hasard*, Compositio Math. **3** (1936), 286–303, reprinted in *Oeuvres de Paul Lévy*, Vol. 6, Gauthier-Villars, Paris, 1980, pp. 285–302.

[59] J. E. Littlewood, *On the class number of the corpus* $p(\sqrt{-k})$, Proc. Lon. Math. Soc. **27** (1928), 358–372.

[60] S. Louboutin, *Explicit upper bounds for* $|L(1,\chi)|$ *for primitive even Dirichlet characters*, Acta Arith. **101** (2002), 1–18.

[61] M. Maurer, *Regulator approximation and fundamental unit computation for real-quadratic orders*, Diplomarbeit, Technische Universität Darmstadt, Darmstadt, Germany, 2000.

[62] K. S. McCurley, *Cryptographic key distribution and computation in class groups*, Number Theory and Applications, NATO Advanced Science Institutes Series C, vol. 265, Kluwer, Dordrecht, 1989, pp. 459–479.

[63] L. McVoy and C. Staelin, *Lmbench: Tools for performance analysis, v. 3.0-a9*, 2011, `http://www.bitmover.com/lmbench/`.

[64] Message Passing Interface Forum, *MPI: A message-passing interface standard, Version 2.2*, 2009, `http://www.mpi-forum.org/`.

[65] J. Milan, *TIFA: Tools for Integer FActorization*, `http://www.lix.polytechnique.`

fr/Labo/Jerome.Milan/tifa/tifa.xhtml.

[66] P. Moret, W. Binder, A. Villazón, D. Ansaloni, and A. Heydarnoori, *Visualizing and exploring profiles with calling context ring charts*, Software: Practice and Experience **40** (2010), no. 9, 825–847.

[67] T. Nagell, *Zur Arithmetik der Polynome*, Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg **1** (1922), 179–194.

[68] F. Najman, *Compact representation of quadratic integers and integer points on some elliptic curves*, Rocky Mountain J. Math. **40** (2010), no. 6, 1979–2002.

[69] H. L. Nelson, *A solution to Archimedes' Cattle Problem*, J. of Recreational Math. **13** (1981), no. 3, 162–176.

[70] M. Pohst and H. Zassenhaus, *Über die Berechnung von Klassenzahlen und Klassengruppen algebraischer Zahlkörper*, J. Reine Angew. Math. **365** (1985), 50–72.

[71] G. W. Reitwiesner, *Binary arithmetic*, Advances in Computers **1** (1960), 231–308.

[72] R. Rivest, *The MD5 Message Digest Algorithm*, Tech. Report 1321, Internet Engineering Task Force, 1992, http://www.ietf.org/rfc/rfc1321.txt.

[73] R. E. Sawilla, A. K. Silvester, and H. C. Williams, *A new look at an old equation*, Algorithmic number theory, Proceedings of the 8th International Symposium (ANTS-VIII), Banff Centre, Banff AB, Canada, May 17–22, 2008, LNCS, vol. 5011, Springer, 2008, pp. 39–59.

[74] R. Scheidler, *Compact representation in real quadratic congruence fields*, Algorithmic number theory (Talence, 1996), LNCS, vol. 1122, Springer, Berlin, 1996, pp. 323–336.

[75] D. Shanks, *The infrastructure of a real quadratic field and its applications*, Proc. 1972 Number Theory Conference, University of Colorado, Boulder, 1972, pp. 217–224.

[76] C. L. Siegel, *Über die Klassenzahl quadratischer Zahlkörper*, Acta Arith. **1** (1935), 83–86.

[77] A. Silvester, *Fast and unconditional principal ideal testing*, Master's thesis, University of Calgary, 2006.

[78] D. C. Terr, *A modification of Shanks' baby-step giant-step algorithm*, Math. Comp. **69** (2000), no. 230, 767–773.

[79] The Eclipse Foundation, *Eclipse C/C++ Development Tooling (CDT) project, v. 3.1.2.*, 2009, `http://www.eclipse.org/cdt/`.

[80] ———, *Eclipse SDK, v. 3.2.2.*, 2009, `http://eclipse.org`.

[81] The Valgrind Developers, *Valgrind 3.3.1*, 2009, `http://valgrind.org/`.

[82] A. J. van der Poorten, *A note on NUCOMP*, Math. Comp. **72** (2003), no. 244, 1935–1946.

[83] U. Vollmer, *Asymptotically fast discrete logarithms in quadratic number fields*, Algorithmic Number Theory—ANTS-IV, LNCS, vol. 1838, Springer, Berlin, 2000, pp. 581–594.

[84] ———, *An accelerated Buchmann algorithm for regulator computation in real quadratic fields*, Algorithmic Number Theory—ANTS-V, LNCS, vol. 2369, Springer, Berlin, 2002, pp. 148–162.

[85] J. Weidendorfer, *Kcachegrind*, 2012, `http://kcachegrind.sf.net`.

[86] H. C. Williams, R. A. German, and C. R. Zarnke, *Solution of the Cattle Problem of Archimedes*, Math. Comp. **19** (1965), 671–674.

[87] H. C. Williams and M. C. Wunderlich, *On the parallel generation of the residues for the continued fraction factoring algorithm*, Math. Comp. **48** (1987), 405–423.

— Appendix A —
Implementation

There's nothing more permanent than a temporary fix.

A.1. Development

The various algorithms listed and cited in this thesis have been implemented in a two-part process. The algorithms were initially implemented in Maple and later in C using GMP, the GNU Multiple Precision Arithmetic Library [32], and MPI, the Message Passing Interface [64]. Maple was chosen for the initial prototype as in the past, I have found that it hides a number of the annoying, but necessary, complexities of programming like memory management, data structure implementation, and data conversion and other such routines. Moreover, since its syntax is based on Pascal, it is relatively straight-forward to translate into lower-level languages such as C and C++.

The C implementation was developed using the Eclipse software development kit [80] and the Eclipse CDT plug-in [79] in conjunction with Valgrind [81], a debugging and profiling tool suite that can not only automatically detect many memory management and threading bugs, but also perform very detailed profiling to help find execution bottlenecks.

The actual development machine varied over the course of this thesis, from a server with 2 dual-core AMD Opteron 2214 processors and 6 GB of RAM, to a desktop machine

with a single-core Intel Pentium 4 2.40GHz processor and 1 GB of RAM, and then to the Institute for Security, Privacy and Information Assurance's (ISPIA) cluster with 152 IBM HS20 Blades each configured with a dual-core Intel Pentium 4 Xeon processor and 2 GB of RAM. All of the timing results presented in this thesis were generated from programs run on the ISPIA cluster. As mentioned in Chapters 4 and 6, we have recently added WestGrid's NESTOR cluster to this list. This capability cluster, hosted by the Research Computing Facility of the University of Victoria, has 288 IBM iDataplex nodes, each configured with two Intel Xeon x5550 quad-core processors and 24 GB of RAM.

A.2. SOURCE CODE

A.2.1. Licensing. The source code files for the C implementation are available from the author via email: `aksilves@math.ucalgary.ca` or `aksilvester@gmail.com`. The library and testing suite is distributed under the following ISC-style license.

```
— Library license: ————————————————————————————————————————
Copyright (c) 2010, Alan Silvester <aksilvester@gmail.com>

Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted, provided that the above
copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES
WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR
ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF
OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
————————————————————————————————————————————————————————
```

The MD5 routines used in the library have been placed in the public domain:

```
— MD5 library license: ————————————————————————————————————
An implementation of the MD5 message-digest algorithm.

This file is placed in the public domain.

The code in this file is based on md5deep (http://md5deep.sf.net),
```

```
a cross-platform set of programs to compute various message digests.

md5deep was written by Special Agent Jesse Kornblum, United States Air
Force Office of Special Investigations.  As such, it is a work of the
US Government. In accordance with 17 USC 105, copyright protection is
not available for any work of the US Government.
```

The SQLite interface and code used in the library have also been placed in the public domain:

```
— SQLite license: ——————————————————————————————————————
All of the deliverable code in SQLite has been dedicated to the public domain
by the authors. All code authors, and representatives of the companies they
work for, have signed affidavits dedicating their contributions to the public
domain and originals of those signed affidavits are stored in a firesafe at
the main offices of Hwaci.  Anyone is free to copy, modify, publish, use,
compile, sell, or distribute the original SQLite code, either in source code
form or as a compiled binary, for any purpose, commercial or non-commercial,
and by any means.
```

The $O(\Delta^{1/6+\epsilon})$ algorithm requires a routine that will solve an instance of the infrastructure discrete logarithm problem. Our implementation uses a portion of the program written by Jean-François Biasse for [6, 7] that uses an (as of yet) unreleased integer factorization library by Jérôme Milan [65]. However, as both authors have requested that I not distribute the code, those routines are not available in our library and the relevant functions have been replaced with stubs. Anyone wishing to compile and use this library will have to either implement an infrastructure DLP solver by themselves or contact Biasse and Milan.

In the remainder of this section, rather than list the full source code, we will briefly discuss the structure of the library and some of the more interesting functions.

A.2.2. Pooling integers. Before using a GMP integer, an `mpz_t`, it must be initialized by passing it as a parameter to the function `mpz_init`. Once you are finished with it,

you should call `mpz_clear` to release any memory that may have been allocated while using the `mpz_t`. Unfortunately, repeatedly initializing and clearing GMP variables can be very computationally expensive. Many of the algorithms presented in this thesis, as well as those cited from other sources, use several temporary variables. To reduce the computational overhead associated with these, we have implemented an `mpz_t` pool data type, `mpz_pool_t`, and supporting functions. The library `mpz-pool.h` implements the pool using two singly-linked lists with linear searching. The linked lists are actually used as stacks, named `free` and `used`, and a linear search is only used to remove nodes from `used`. The other addition and removal operations are implemented as pops and pushes. By being careful in how temporary variables are acquired and released in each function, the linear search will degrade nicely into a pop operation with only one extra pointer comparison as overhead. Any number of pools may be defined and used in a program, but there is always a global pool, aptly named `pool`, available for use. I exclusively use this pool in my program's implementation.

Rudimentary statistics are maintained by each pool and can be printed out using the `mpz_pool_stats` function. Currently, four counters are used to track the number of integers in the pool, the number of new integers that have been created since the pool was initialized (using `mpz_pool_init`), the number of times an integer has been removed from the pool, and the number of times one has been returned. In order, these counters are `nodes`, `allocs`, `borrows`, and `returns`. Although it has not been done, it should be possible to implement a basic garbage collector using these statistics in addition to the `mpz_pool_clear_free` function, if the memory usage of this pool becomes an issue.

A.2.3. Debugging. As mentioned in the Development section at the start of this chapter, Valgrind was heavily used to hunt down memory management and threading bugs. To aid the end-user, should they not wish to use Valgrind, there is also a great deal of debugging code in the library. The various commands are (normally) defined as blank macros, so there is no overhead incurred to have these statements in the code; your C compiler will optimize them away.

If you would like to enable the debugging routines, simply remove the first number sign from the following line

```
DEBUG := #-DDEBUG  # Uncomment to enable debugging output
```

in the top-level Makefile and rebuild the library.

Before using any of the debugging functions, a call to `DEBUG_INIT` must be made. There are five levels of debugging output available: trace, info, warning, error, and fatal. By default, all five levels will send their output to `stderr`. You can redirect these by using the `DEBUG_SET_*` commands. Debugging output for the the trace, info, warning, and error levels can also be temporarily disabled with the `DEBUG_DISABLE_*` commands; they can be re-enabled with a call to the appropriate `DEBUG_ENABLE_*` command. To output a debugging message, simply add a call to the appropriate `DEBUG_*` command. You should be aware, though, the `DEBUG_FATAL` command will not return; after printing its message, it calls `exit(1)` and terminates the program. If you would like to return a different return code, you may use the `DEBUG_FATAL_RC` command.

`DEBUG_TRACE_IN` and `DEBUG_TRACE_OUT` are two specialized tracing commands. If these commands are called, they output a message similar to the `DEBUG_TRACE` command, however they then indent (or unindent) all following messages from any of the `DEBUG_*`

commands.

A.2.4. Checkpointing. A major concern with any high-availability computers is the unexpected loss of work. Looking at the timing data in [21, 22], some of the calculations can take days, if not weeks or months. How can we protect ourselves against accidental job cancellation, hardware failures, unexpected power outages, or even something as esoteric as stray cosmic rays flipping bits in RAM? The simple answer is to use process checkpointing to save the state of our program at a regular interval. If a problem occurs, we can roll-back to the most recent checkpoint and continue as though nothing happened. Instead of losing weeks of computations, perhaps we only loose an hour or so. However, it is one thing to say that we will save checkpoints. It is another matter entirely to actually come up with a method that works, does not add much overhead to our computations, and is quick at restoring our program's state after an interruption.

The simplest method of checkpointing a process is to halt it, take a complete snapshot of everything stored in memory, store the memory image to a file, and resume the process. For a one-time checkpoint, this method can work rather well. For repeated checkpoints, however, we run into issues. When working with a program processing large amounts of data, this method can use a lot of disk space. We could refine our checkpointing process to have a function that, when called, writes out certain pieces of memory to a file, but avoids saving temporary variables and data it does not need to keep. However, this means we must be very careful in how we write out the data, how we import it back in, and how we determine which variables need saving. Once we have this done, we still have to devise a method to resume our program at the point where the checkpoint left off. This can be quite a lot of work to figure out.

One of the more popular types of checkpointing programs are so called *transparent* checkpointing programs. Rather than coming up with routines like those described previously and deciding when and how to use them, you can link your program against a special library and have virtually everything handled automagically. Generally these libraries handle checkpointing more than just the memory a program is using. Most can save and restore file descriptors that your program may be reading from or writing to; network sockets, including any unprocessed data in buffers; shared libraries that your program has loaded while running; child processes, and all the memory, file descriptors, network sockets, and more that they are using. Moreover, usually you can specify what things not to save, like temporary variables. Some more modern checkpointing programs can be used without even altering the source code of your program. You can simply load your program with the checkpointing program and it runs as normal, but with regular automatic snapshots taken. These work well even with distributed programs using MPI-style libraries. The checkpointing program used for my implementation, once it was placed on the ISPIA cluster, was DISTRIBUTED MULTITHREADED CHECKPOINTING (DMTCP).

A.3. DEPENDENCIES

This library has some substantial dependencies which are discussed in this section.

The Debugger code is a standalone program that only depends on some standard C headers (`stdio`, `stdlib`, and `string`). There are only two portability issues that may arise:

1. A number of routines rely on the `__FUNCTION__` pre-processor macro and
2. A write-only filehandle is opened for `/dev/null` (`debugger.c`, lines 87 and 106).

The HashTable implementation depends on Debugger as well as some standard headers

(`stdlib`, `string`, `math`, and `inttypes`).

At a number of places in the library, we have used functions provided by the GNU C Library, GLIBC, that are GNU extensions to either the C90/C99 or POSIX standards. Accordingly, the library is set to compile with the flag `-D_GNU_SOURCE`. The GNU extensions used are described in Subsection A.3.2, as well as rough work-arounds if you wish to compile the library with a compiler other than GCC.

As mentioned at the start of this appendix, the library depends heavily on both GMP and MPI. Unfortunately, the most recent versions of these libraries (at the time of this writing) are missing a few key functions needed for the algorithms presented or referred to in this thesis. These are described in Subsection A.3.3.

In order to execute the $O(\Delta^{1/6+\epsilon})$ algorithm efficiently, we first need to calculate a number of parameters that allow us to find the optimal balance between the running times of the various pieces of the algorithm. However, as these parameters depend on the discriminant of the field we are working in, we must recalculate them when the discriminant changes. Subsection A.3.4 describes how we can automate this process.

A.3.1. LINUX timing extension to the POSIX standard. A formerly excellent high-resolution, low-latency method of getting CPU timing information was the Time Stamp Counter (TSC). Introduced with the INTEL Pentium processor, and present on all x86-compatible processors since then, the TSC is a 64-bit register that counts the number of processor ticks since it was reset. Unfortunately, with the advent of computers with multiple CPUs and multi-core / hyperthreaded CPUs, the TSC can no longer be relied on to provide an accurate timestamp. Since the introduction of the Pentium Pro, INTEL processors have supported out-of-order execution, where the instructions in a program

are not necessarily executed in the order they appear. This can cause a read of the TSC to be executed earlier or later than expected, producing a misleading result.

On a system with multiple CPUs or with a multi-core processor, a program may migrate from one CPU or core to another during its execution. However, the TSCs on each CPU or core are not necessarily synchronized which can lead to inaccurate timing. Even if a process is locked to a single CPU or core, power-saving measures by the operating system (such as CPU-frequency scaling) may alter the speed of the CPU or core, thereby making the TSC tick at a non-constant rate.

On newer INTEL CPUs, a constant-rate TSC has been included that runs at the maximum frequency of the CPU, regardless of its current frequency. On LINUX-based systems, this can be determined by the `constant_tsc` flag in `/proc/cpuinfo`. However, even on these systems, using the TSC for timing may skew the results as a computation may have some "spin-up" time where the OS keeps the CPU at a lower frequency before switching to a higher one. This has the effect of making computations seem like they require more processor cycles than they actually do.

Because of these issues, direct use of the TSC is discouraged. On LINUX-based systems, the operating system[1] can detect whether the TSC is reliable, detect alternatives like a high precision event timer (HPET) or advanced configuration and power interface (ACPI) timer, and automatically select the best one.

The POSIX-standard `gettimeofday` function is one common way of getting the current time from a LINUX operating system. This function returns the current time expressed as the number of seconds and microseconds elapsed since the UNIX Epoch.[2] But as of Issue 7 of the POSIX-standard [39], `gettimeofday` has been deprecated and applica-

[1] Since kernel version 2.6.18, that is.
[2] January 1, 1970, at 00:00:00 UTC.

tions are encouraged to use its replacement `clock_gettime`. According to the standard:

> *All implementations shall support a* clock_id *of CLOCK_REALTIME as defined in* <time.h>. *This clock represents the real-time clock for the system. For this clock, the values returned by* clock_gettime() *and specified by* clock_settime() *represent the amount of time (in seconds and nanoseconds) since the Epoch. An implementation may also support additional clocks. The interpretation of time values for these clocks is unspecified.* [39]

The `clock_gettime` function, like `gettimeofday`, returns the current time elapsed since the UNIX Epoch. However, the time returned by `clock_gettime` is measured in seconds and nanoseconds. Moreover, in a multiple CPU or multi-core system, the time returned by `clock_gettime(CLOCK_REALTIME, ...)` will be consistent across all processors and cores.

However, `gettimeofday` and `clock_gettime` have their own problems. Both functions are affected by any adjustments made to the system clock, such as those made by `settime`, `settimeofday`, `adjtimex`, `clock_settime(CLOCK_REALTIME, ...)`. This is an issue not because a user may change the system time during the execution of our program, but because a large portion of modern computers keep their system clocks synchronized to an external time source via mechanisms like the Network Time Protocol (NTP). On LINUX-based systems, the usual program doing this work is the Network Time Protocol Daemon `ntpd`. `ntpd` works by occasionally getting the current time from a remote, highly-accurate timeserver (usually attached to an atomic clock or a fixed-location GPS receiver) and calculating the difference between the local and remote current times, correcting for any latency involved in the communication. Once `ntpd` knows how far off the local clock is, it will either jump the clock forward or backwards suddenly, if the lo-

cal time is off by a large amount (> 0.5 seconds), or will begin slowly slewing the local clock, by altering its frequency to make it run slightly faster or slightly slower, until it matches up with the time from the remote timeserver. Obviously, this will effect any timing we attempt to do with `gettimeofday` or `clock_gettime` and unfortunately, both my development machine and the ISPIA cluster have `ntpd` running.

Thankfully, though, GCC supports an additional (and optional) clock that is guaranteed to not jump forwards or backwards: `CLOCK_MONOTONIC`. From the POSIX standard:

> *This clock represents the monotonic clock for the system. For this clock, the value returned by* clock_gettime() *represents the amount of time (in seconds and nanoseconds) since an unspecified point in the past (for example, system start-up time, or the Epoch). This point does not change after system start-up time. The value of the CLOCK_MONOTONIC clock cannot be set via* clock_settime(). [39]

However, although `CLOCK_MONOTONIC` is not effected by `ntpd`'s sudden time jumps, it is still susceptible to its clock-slewing frequency adjustments. Because of this, `LINUX`-based operating systems[3] provide an additional clock: `CLOCK_MONOTONIC_RAW`. This clock is guaranteed to have a constant frequency. If it is available, my library will attempt to use `CLOCK_MONOTONIC_RAW` and will fall-back to `CLOCK_MONOTONIC` and then, as a last-resort `CLOCK_REALTIME`.

A.3.2. GNU extensions to the C90, C99 and POSIX standards. Most of the GNU-extension functions used in the library are string-processing and timing functions. In `mpz_qf_regulator_dehaan`, when calling MAPLE via `execl` (see Subsection A.3.4 for why we do this), the arguments are specified as C-strings. We use the GNU extension

[3] Since kernel version 2.6.28.

asprintf to automatically allocate the storage needed for strings representing three of the arguments. This can be worked around by using the C99 function snprintf. The C90 function sprintf should not be used due to buffer-overflow concerns.

Library code:
```
if (UNLIKELY(asprintf(&strN, "%l", N)) < 0)
{
        DEBUG_ERROR("Unable to allocate memory calling to Maple script");
        return 0;
}
```

C99-compliant code:
```
int len = snprintf(strN, 0, "%ld", N);
strN = (char*) malloc((len+1)*sizeof(char));
if (UNLIKELY(strN == NULL))
{
        DEBUG_ERROR("Unable to allocate memory calling to Maple script");
        return 0;
}
snprintf(strN, len+1, "%ld", N);
```

Also in the same function, we use the GNU extension strndup to make a copy of a portion of a string. This can be worked around by again using the C99 function snprintf.

Library code:
```
lhs = strndup(str, exp);
if (UNLIKELY(lhs == NULL))
{
        DEBUG_ERROR("Unable to allocate memory calling to Maple script");
        return 0;
}
rhs = strndup(str+exp, strlen(str)-exp);
if (UNLIKELY(rhs == NULL))
{
        DEBUG_ERROR("Unable to allocate memory calling to Maple script");
        return 0;
}
```

C99-compliant code:
```
int len = snprintf(rhs, 0, "%s", str+exp);
rhs = (char*) malloc((strlen(str)-exp+1)*sizeof(char));
```

```
if (UNLIKELY(rhs == NULL))
{
        DEBUG_ERROR("Unable to allocate memory calling to Maple script");
        return 0;
}
snprintf(rhs, strlen(str)-exp+1, "%s", str+exp);
str[exp] = '\0';
len = snprintf(lhs, 0, "%s", str);
lhs = (char*) malloc((exp+1)*sizeof(char));
if (UNLIKELY(lhs == NULL))
{
        DEBUG_ERROR("Unable to allocate memory calling to Maple script");
        return 0;
}
snprintf(lhs, exp+1, "%s", str);
```

A.3.3. GMP: additional functions. In the second portion of the $O(\Delta^{1/6+\epsilon})$ regulator verification algorithm, we need to trial divide by a range of potential prime divisors in order to determine if $c = 1$. For efficiency reasons, we begin with the largest prime q below the given bound M and work backwards until we reach $q = 2$. GMP has a built-in function `mpz_nextprime` that returns the smallest prime[4] greater than the input integer. Unfortunately, GMP does not have a "previous prime" function; the following is a patch to rectify this situation. It is derived from GMP's `mpz_nextprime` function and, accordingly, is hereby licensed under the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Source code: libgmp3c2-mpz_prevprime.patch

```
1 --- gmp-4.3.1+dfsg/mpz/nextprime.c        2009-05-12 00:12:12.000000000 -0600
2 +++ gmp-4.3.1+dfsg/mpz/nextprime.c        2010-02-02 14:57:11.000000000 -0700
3 @@ -118,3 +118,95 @@
4   done:
5    TMP_SFREE;
```

[4]The function, as of version 4.3.1 of GMP, actually calculates the smallest odd integer greater than the input and sieves this value over the primes less than 1010. If the number passes this test, GMP proceeds to perform ten Miller-Rabin primality tests. If successful, the integer is returned.

```
 }
+
+void
+mpz_prevprime (mpz_ptr p, mpz_srcptr n)
+{
+  unsigned short *moduli;
+  unsigned long difference;
+  int i;
+  unsigned prime_limit;
+  unsigned long prime;
+  int cnt;
+  mp_size_t pn;
+  unsigned long nbits;
+  unsigned incr;
+  TMP_SDECL;
+
+  /* First handle tiny numbers */
+  if (mpz_cmp_ui (n, 5) < 0)
+    {
+      mpz_set_ui (p, 2);
+      return;
+    }
+  mpz_sub_ui (p, n, 2);
+  mpz_setbit (p, 0);
+
+  if (mpz_cmp_ui(p, 9) == 0)
+    {
+      mpz_set_ui(p, 7);
+      return;
+    }
+  if (mpz_cmp_ui (p, 11) <= 0)
+    return;
+
+  pn = SIZ(p);
+  count_leading_zeros (cnt, PTR(p)[pn - 1]);
+  nbits = pn * GMP_NUMB_BITS - (cnt - GMP_NAIL_BITS);
+  if (nbits / 2 >= NUMBER_OF_PRIMES)
+    prime_limit = NUMBER_OF_PRIMES - 1;
+  else
+    prime_limit = nbits / 2;
+
+  TMP_SMARK;
+
+  /* Compute residues modulo small odd primes */
+  moduli = TMP_SALLOC_TYPE (prime_limit * sizeof moduli[0], unsigned short);
+
+  for (;;)
+    {
```

```
+      /* FIXME: Compute lazily? */
+      prime = 3;
+      for (i = 0; i < prime_limit; i++)
+        {
+          moduli[i] = mpz_fdiv_ui (p, prime);
+          prime += primegap[i];
+        }
+
+#define INCR_LIMIT 0x10000        /* deep science */
+
+      for (difference = incr = 0; incr < INCR_LIMIT; difference += 2)
+        {
+          /* First check residues */
+          prime = 3;
+          for (i = 0; i < prime_limit; i++)
+            {
+              signed r;
+              /* FIXME: Reduce moduli + incr and store back, to allow for
+                 division-free reductions.  Alternatively, table primes[]'s
+                 inverses (mod 2^16).  */
+              r = (moduli[i] - incr);
+              while (r < 0)
+                 r += prime;
+              r %= prime;
+              prime += primegap[i];
+
+              if (r == 0)
+                goto next;
+            }
+
+          mpz_sub_ui (p, p, difference);
+          difference = 0;
+
+          /* Miller-Rabin test */
+          if (mpz_millerrabin (p, 10))
+            goto done;
+        next:;
+          incr += 2;
+        }
+      mpz_sub_ui (p, p, difference);
+      difference = 0;
+    }
+ done:
+  TMP_SFREE;
+}
--- gmp-4.3.1+dfsg/gmp-h.in        2009-05-12 00:12:12.000000000 -0600
+++ gmp-4.3.1+dfsg/gmp-h.in        2010-02-02 14:57:18.000000000 -0700
@@ -999,6 +999,9 @@
```

```
102  #define mpz_nextprime __gmpz_nextprime
103  __GMP_DECLSPEC void mpz_nextprime __GMP_PROTO ((mpz_ptr, mpz_srcptr));
104
105 +#define mpz_prevprime __gmpz_prevprime
106 +__GMP_DECLSPEC void mpz_prevprime __GMP_PROTO ((mpz_ptr, mpz_srcptr));
107 +
108  #define mpz_out_raw __gmpz_out_raw
109  #ifdef _GMP_H_HAVE_FILE
110  __GMP_DECLSPEC size_t mpz_out_raw __GMP_PROTO ((FILE *, mpz_srcptr));
```

Two common calculations we need to perform are determining $t \geq 0$ such that $2^t \leq x < 2^{t+1}$ for a given integer x and, similarly, determining $t \geq 0$ such that $2^t y \leq x < 2^{t+1} y$ for given integers x and y. To solve these inequalities, we have implemented two functions: `mpz_find_2exp_range` and `mpz_find_2exp_range_z`, respectively. Additionally, these functions take an input flag (the parameter "equality" in the following code) which will cause them to compute x such that $2^t < x \leq 2^{t+1}$ or $2^t y < x \leq 2^{t+1} y$. Although these functions are currently implemented in our library in `utilities.c`, they will eventually be converted into a patch against GMP.

Source code: libgmp3c2-mpz_find_2exp_range.patch

```
1  unsigned int
2  mpz_find_2exp_range(mpz_ptr x, int equality)
3  {
4      if (mpz_sgn(x) == 0)
5      {
6      return 0;
7      }
8
9    mp_limb_t msb_x, t;
10
11   // Pointer to the most significant limb of x
12   mp_limb_t *hi = x->_mp_d + (x->_mp_size - 1);
13
14   /*
15    * Get the most significant bit index in the most
16    * significant limb of x.
17    */
18   asm("bsrl %1, %0" : "=r" (msb_x) : "r" (*hi));
19
20   // Use this to determine the most significant bit index
```

```
21  t = msb_x + (x->_mp_size - 1) * sizeof(mp_limb_t) * 8;
22
23  /*
24   * Compute masks for most significant bit in this limb and
25   * see if any other bits are set.  If none are (ie, limb == 0)
26   * then we need to check the lower significance limbs
27   * to see if x is a power of two.
28   */
29  mp_limb_t upper, lower;
30  switch (msb_x)
31  {
32    case 0:
33      upper = GMP_NUMB_MAX << 1;
34      lower = 0;
35      break;
36    case GMP_LIMB_BITS-1:
37      upper = 0;
38      lower = GMP_NUMB_MAX >> 1;
39      break;
40    default:
41      upper = GMP_NUMB_MAX << (msb_x + 1);
42      lower = GMP_NUMB_MAX >> (GMP_LIMB_BITS - msb_x);
43      break;
44  };
45
46  if ((x->_mp_d[x->_mp_size - 1] & (upper | lower)) == 0)
47  {
48    // x might be a power of two
49
50    // Are there lower limbs that we need to check?
51    if (x->_mp_size > 1)
52    {
53      do
54      {
55        if (*hi-- != 0)
56        {
57          // Found a non-zero limb, x not a power of two.
58          goto DONE;
59        }
60      }
61      while (hi > x->_mp_d);
62    }
63
64    /*
65     * x is a power of 2.  Do we need 2^t < x or 2^t <= x?
66     */
67    if (equality > 0)
68    {
```

```
69        // 2^t < x.
70        t--;
71      }
72    }
73
74  DONE:
75    return t;
76  }
```

Source code: libgmp3c2-mpz_find_2exp_range_z.patch

```
1  int
2  mpz_find_2exp_range_z(mpz_ptr x, mpz_ptr y, int equality)
3  {
4    if (mpz_sgn(x) == 0)
5    {
6      return 0;
7    }
8
9    mp_limb_t msb_x, msb_y, t;
10
11   // Pointers to the most significant limb of x and y
12   mp_limb_t *hi_x = x->_mp_d + (x->_mp_size - 1);
13   mp_limb_t *hi_y = y->_mp_d + (y->_mp_size - 1);
14
15   /*
16    * Get the most significant bit index in the most
17    * significant limbs of x and y.
18    */
19   asm("bsrl %1, %0" : "=r" (msb_x) : "r" (*hi_x));
20   asm("bsrl %1, %0" : "=r" (msb_y) : "r" (*hi_y));
21
22   // Use this to determine the most significant bit index
23   msb_x += (x->_mp_size - 1) * sizeof(mp_limb_t) * 8;
24   msb_y += (y->_mp_size - 1) * sizeof(mp_limb_t) * 8;
25
26   /*
27    * Temporarily left-shift y or x, depending on t.
28    * We also need to re-grab a pointer to MSL(y) (or
29    * MSL(x)) as it will change if mpz_mul_2exp()
30    * calls gmp_realloc().
31    */
32   t = msb_x - msb_y;
33   if (t >= 0)
34   {
35     mpz_mul_2exp(y, y, t);
36     hi_y = y->_mp_d + (y->_mp_size - 1);
37   }
```

```
38  else
39  {
40    mpz_mul_2exp(x, x, t);
41    hi_x = x->_mp_d + (x->_mp_size - 1);
42  }
43
44  /*
45   * Compare the MSLs of x and y.  If MSL(x) < MSL(y), we
46   * have 2^t * y < x < 2^(t+1) * y and are done.  If MSL(x)
47   * > MSL(y), we need to set t <- t+1 to satisfy the above
48   * inequality.  If MSL(x) = MSL(y), then we need to check
49   * the lower significance limbs as well.
50   */
51  if (*hi_x < *hi_y)
52  {
53    t--;
54  }
55  else if (*hi_x == *hi_y)
56  {
57    // Are there lower limbs that we need to check?
58    if (x->_mp_size > 1)
59    {
60      // Need to check the lower limbs
61      do
62      {
63        hi_x--;
64        hi_y--;
65
66        if (*hi_x < *hi_y)
67        {
68          // LIMB(x) < LIMB(y)
69          t--;
70          goto DONE;
71        }
72        else if (*hi_x > *hi_y)
73        {
74          goto DONE;
75        }
76
77      }
78      while (hi_x > x->_mp_d);
79    }
80
81    // x and y are equal.  Check if this is allowed.
82    if (equality > 0)
83    {
84      // No, we want 2^t * y < x.
85      t--;
```

```
86      }
87    }
88
89  DONE:
90    /*
91     * Undo the temporary left-shift of y (or x).  Note, we cannot
92     * use t as its value may have changed.
93     */
94    if (msb_x - msb_y >= 0)
95    {
96      mpz_tdiv_q_2exp(y, y, msb_x - msb_y);
97    }
98    else
99    {
100     mpz_tdiv_q_2exp(x, x, msb_y - msb_x);
101   }
102
103   if (t < 0)
104     t = 0;
105
106   return t;
107 }
```

It may be possible to squeeze a bit more out of the `mpz_find_2exp_range_z` implementation by working limb-by-limb, making temporary copies as needed. We would then avoid the `mpz_mul_2exp` call, though it remains to be seen if this will lead to a significant improvement or not.

A.3.4. Optimizing the $O(\Delta^{1/6+\epsilon})$ parameters. How these global variables are set depends on a couple of compile-time flags. To use the automated method described below, the library must be compiled with the `-DUSE_MAPLE` flag. If it is unset, these parameters will not be computed by the `compute_delta16_parameters` function. All of the functions for the various $O(\Delta^{1/6+\epsilon})$ algorithms assume that these parameters have been set before being calling. Not doing so will, most likely, result in your program crashing.

If MPI has also been enabled, via the `-DUSE_MPI` flag, the `-DUSE_MAPLE` flag will split the global MPI group in two. The first group contains a number of dedicated

MAPLE computing nodes, the second contains the computing nodes that will actually work on the $O(\Delta^{1/6+\epsilon})$ algorithm. This is done to allow load-balancing of the available MAPLE computing nodes, as the number of worker nodes should far exceed the number of MAPLE computing nodes. Respectively the two MPI groups are called `MPI_worker_group` and `MPI_maple_group`. Two MPI intra-communicators are initialized, `MPI_COMM_MAPLE` and `MPI_COMM_WORKER`, though the former is rarely used. An MPI inter-communicator `MPI_INTERCOMM_WORKER_MAPLE` is also available and is used for worker nodes to request that optimum parameters be computed on their behalf. The details of which communicator to use given which compile-time flags have been abstracted behind a number of the functions available in the ISPIA-MPI library. Any extensions to the library should use the `cluster_*` functions presented in `ispia-mpi.h` in order to ensure compatibility with existing functions.

There are two parts to the automated process for calculating the optimum parameters for the $O(\Delta^{1/6+\epsilon})$ algorithms. The first is a generic shell script that takes a filename followed by any number of space-delineated strings as arguments. It then translates the space-delimited strings into MAPLE's array syntax, starts MAPLE, passes it the array, and instructs it to open the script file specified by the filename in the first argument.

Source code: runmaple.sh

```
1  #!/bin/bash
2
3  theargs="\"$0\"";
4
5  for var in "$@"
6  do
7      theargs="$theargs,\"$var\""
8  done
9
10 /usr/local/maple9/bin/maple -q -s <<__EOM__
11   NARGS:=$#-1:
12   ARGS:=[$theargs][3..-1]:
```

```
  read("$1");
__EOM__
```

The second part is the MAPLE script that was mentioned in the previous paragraph. This script turns off "pretty-printing" and uses the input variables in the ARGS array to calculate the optimal parameters via the formulas described in Section 4.3.2. The four integers s, Q, K, and l are then printed out, in that order.

Source code: optimize-d-16.maple

```
#!/usr/bin/env /usr/local/maple9/bin/maple

# Disable the pretty-printer
interface(prettyprint=0):

# Set up the optimization formulas
f:=BSTime + GSTime + FMTime;
BSTime:=((s + lambda)/(17/10) + 2)*(1 + h) / r;
GSTime:=(Q-r*n)*(mu + (l-1)*(1+h*nu))/(r*n);
FMTime:=1/(r*n*ln(2)) * (R/K) * ( (mu*log[2](R/s) + 2*(l-1)*(1+h*nu))
        / (2*log[2](R/K)) );

# Set up optimum parameter relations
s:=sqrt(34)/2 * 1/(sqrt( 51*r^2*n*(1+h*nu) - 50*(1+h)*(N-r+1) ))
   * sqrt(r*K*( 5*(N-r+1)*((1+h*nu)-mu) - 3*lambda*(1+h*nu) ));
Q:=K/(2*s);
l:=(6/10)*(s + lambda)/(N-(r-1));

# Set input variables from command-line arguments
# Note: R2 argument converted to base-2 as all index-calculus algorithms we have
#       access to produce base-e regulator approximations.  Change as needed.
Delta:=parse(ARGS[1]):
lambda:=ceil((1/2)*log[2](Delta)) + 1;
R2:=parse(ARGS[2])/ln(2):
mu:=parse(ARGS[3]):
h:=parse(ARGS[4]);
nu:=parse(ARGS[5]);
N:=parse(ARGS[6]):
n:=parse(ARGS[7]):
r:=parse(ARGS[8]):

# Explicit formula to optimize
#f;

# Convert formula to function.
```

```
g:=unapply(f, K):

# Compute derivative f' of formula to optimize.
gp:=unapply(diff(g(x),x),x):

# Increase precision of calculations.
Digits:=25;

# Numerically compute zero of derivative
initial_zero_approximation:=??;
fsolve(gp(10^x)=0, x=initial_zero_approximation);
newK:=10^(%);

# Print out the optimum parameters
ceil(subs(K=newK, s));
ceil(subs(K=newK, Q));
ceil(newK);
ceil(subs(K=newK, 1));
```

To use the automated process, we calculate or estimate the input variables—μ, h, ν, N, n, and r—, redirect the standard input and output streams to a temporary pipe, use a fork-and-exec block to run the `runmaple.sh` script, and print the input variables to the pipe. After MAPLE has finished its computations, the output is then read back from the pipe, parsed via some standard C string functions, and converted back into GMP integers. The explicit `execlp` function call to open a connection to MAPLE is:

```
— Running MAPLE from library: ——————————————————————
execlp("runmaple.sh", "runmaple.sh", "optimize-d-16.maple", strDelta, strReg,
       strmu, strh, strnu, strN, strn, strr, NULL);
```

Note, if MAPLE is not available on your particular computer or compute node, but is available on a remote machine, this technique will work over an SSH connection provided that password-less logins are used.

A.4. TESTING

To aid in testing this implementation, the MAPLE program GENERATEPIC was created. It explicitly computes the ideals in the principal ideal cycle, the relative generator θ_i

for each ideal (where $\mathfrak{a}_i = (\theta_i)\mathfrak{a}_1$), and an approximation of the distance of each ideal ($\log_2 \theta_i$). The program also, on input of a precision p, generates an (f, p) representation of each ideal by setting $k = \lfloor \log_2 \theta_i \rfloor$ and computing a range of acceptable d values. The program outputs the data as a table using LaTeX markup. To typeset the resulting file, the `geometry`, `xtab`, `amssymb` and `xcolor` packages are required. Figure A.1 shows an excerpt of the resulting document.

Principal ideal cycle for $\mathbb{Q}(\sqrt{10000003})$

i	$[Q, P]$	θ_i	θ_i approx.	$\log_2 \theta_i$	d range	k
1	$[1, 0]$	1	1	0	$[15421, 17476]$	0
2	$[1759, 3162]$	$3162 + \sqrt{10000003}$	6324.28	12.6267	$[23810, 26983]$	12
3	$[3142, 2115]$	$9487 + 3\sqrt{10000003}$	18973.8	14.2117	$[17858, 20238]$	14
4	$[2847, 1027]$	$12649 + 4\sqrt{10000003}$	25298.1	14.6267	$[23810, 26984]$	14
5	$[2349, 1820]$	$22136 + 7\sqrt{10000003}$	44271.9	15.4341	$[20834, 23611]$	15
6	$[731, 2878]$	$56921 + 18\sqrt{10000003}$	113842	16.7967	$[26787, 30357]$	16
7	$[1613, 2970]$	$477504 + 151\sqrt{10000003}$	955008	19.8652	$[28089, 31833]$	19
8	$[4034, 1869]$	$1489433 + 471\sqrt{10000003}$	2.97887e+06	21.5063	$[21904, 24823]$	21
9	$[1317, 2165]$	$1966937 + 622\sqrt{10000003}$	3.93387e+06	21.9075	$[28926, 32782]$	21
10	$[282, 3103]$	$9357181 + 2959\sqrt{10000003}$	1.87144e+07	24.1576	$[17201, 19494]$	24
11	$[1361, 3101]$	$207824919 + 65720\sqrt{10000003}$	4.15650e+08	28.6308	$[23877, 27060]$	28
12	$[3314, 2343]$	$840656857 + 265839\sqrt{10000003}$	1.68131e+09	30.6469	$[24146, 27365]$	30
13	$[2733, 971]$	$1048481776 + 331559\sqrt{10000003}$	2.09696e+09	30.9657	$[30115, 34130]$	30
14	$[2523, 1762]$	$1889138633 + 597398\sqrt{10000003}$	3.77828e+09	31.8151	$[27131, 30747]$	31
15	$[3734, 761]$	$2937620409 + 928957\sqrt{10000003}$	5.87524e+09	32.452	$[21094, 23906]$	32
16	$[311, 2973]$	$4826759042 + 1526355\sqrt{10000003}$	9.65352e+09	33.1684	$[17330, 19640]$	33

Figure A.1: Excerpt of output from GENERATEPIC(10000003,14).

Source code: GENERATEPIC

```
GeneratePIC:=proc(d :: posint, p :: posint)
   local r, Q, P, i, q, l, psi, theta, fd, k, thistheta;
   Q:=table();
   P:=table();
   r:=1: Q[0]:=1: P[0]:=0:
   if (d mod 4 = 1) then r:=2; Q[0]:=2; P[0]:=1; fi:
   #
   # Prevent this loop from running away if there are too many ideals
   #
   for i from 0 to 5000 do
      q[i]:=floor((P[i] + sqrt(d))/Q[i]);
      P[i+1]:=q[i]*Q[i] - P[i];
```

```
13       Q[i+1]:=(d - P[i+1]^2)/Q[i];
14       if (i <> 1) and (P[i] = P[1]) and (Q[i] = Q[1]) then break; fi;
15    od:
16    l:=i;
17    psi:=table():
18    for i from 1 to l do
19       psi[i]:=(P[i] + sqrt(d))/Q[i-1];
20    od:
21    theta[0]:=(sqrt(d)-P[0])/Q[0]: theta[1]:=1:
22    for i from 2 to l do
23       theta[i]:=expand(psi[i-1]*theta[i-1]);
24    od:
25    fd:=fopen(cat("D-", d, "-p-", p, ".tex"), WRITE);
26    fprintf(fd, "\\documentclass{article}\n"):
27    fprintf(fd, "\\usepackage[landscape,margin=1in]{geometry}\n"):
28    fprintf(fd, "\\usepackage[table]{xcolor}\n"):
29    fprintf(fd, "\\usepackage{xtab}\n"):
30    fprintf(fd, "\\usepackage{amssymb}\n");
31    fprintf(fd, "\\definecolor{lightlightgray}{rgb}{0.85,0.85,0.85}\n"):
32    fprintf(fd, "\\pagestyle{empty}\n\n"):
33    fprintf(fd, "Principal ideal cycle for $\mathbb{Q}(\sqrt{%d})$", d);
34    fprintf(fd, "\\bigskip\n");
35    fprintf(fd, "\n");
36    fprintf(fd, "\\begin{document}\n"):
37    fprintf(fd, "\\tablefirsthead{\\hline $i$ & $[Q,P]$ & $\\theta_i$ &
38                 $\\theta_i$ approx. & $\\log_2 \\theta_i$ & $d$ range &
39                 $k$ \\\\\\hline}\n"):
40    fprintf(fd, "\\tablehead{\\hline $i$ & $[Q,P]$ & $\\theta_i$ &
41                 $\\theta_i$ approx. & $\\log_2 \\theta_i$ & $d$ range &
42                 $k$ \\\\\\hline}\n"):
43    fprintf(fd, "\\rowcolors{0}{}{lightlightgray}\n"):
44    fprintf(fd, "\\begin{xtabular}{ccccccc}\n"):
45    for i from 0 to l-1 do
46       k:=floor(log[2](theta[i+1])):
47       thistheta:="";
48       if (nops(theta[i+1]) > 1) then
49          if type(op(2,theta[i+1]), `*`) then
50             if type(op(1, theta[i+1]), `fraction`) then
51                thistheta:=cat("\\frac{", op(1,op(1,theta[i+1])), "}{",
52                               op(2,op(1,theta[i+1])), "} + \\frac{",
53                               op(1,op(1,op(2,theta[i+1]))), "\\sqrt{", d,
54                               "}}{", op(2,op(1,op(2,theta[i+1]))), "}");
55             else
56                thistheta:=cat(op(1,theta[i+1]), " + ", op(1,op(2,theta[i+1])),
57                               "\\sqrt{", d, "}");
58             fi;
59          else
60             thistheta:=cat(op(1,theta[i+1]), "+\\sqrt{", d, "}");
```

```
61          fi;
62       else
63          thistheta:=convert(theta[i+1], string);
64       fi;
65       fprintf(fd, "%d & $[%d, %d]$ & $%s$ & %g & %g & $[%d, %d]$ &
66               %d \\\\\n", i+1, Q[i], P[i], thistheta, evalf(theta[i+1]),
67               evalf(log[2](theta[i+1])), ceil(2^(p+4-k)*theta[i+1]/17),
68               floor(2^(p+4-k)*theta[i+1]/15), k);
69    od:
70    fprintf(fd, "\\end{xtabular}\n"):
71    fprintf(fd, "\\end{document}\n"):
72    fclose(fd);
73 end proc:
```

— Appendix B —
Additional Tables and Figures

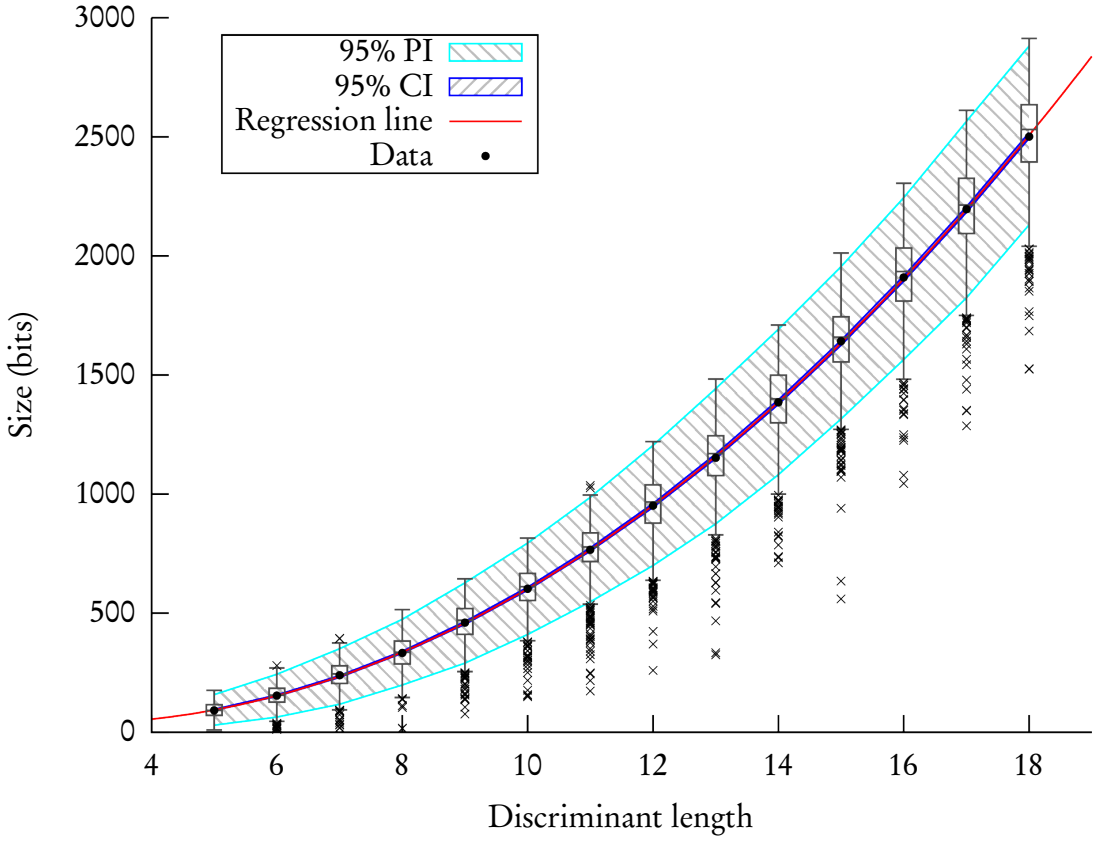


Figure B.1: Statistical analysis of random compact representations.

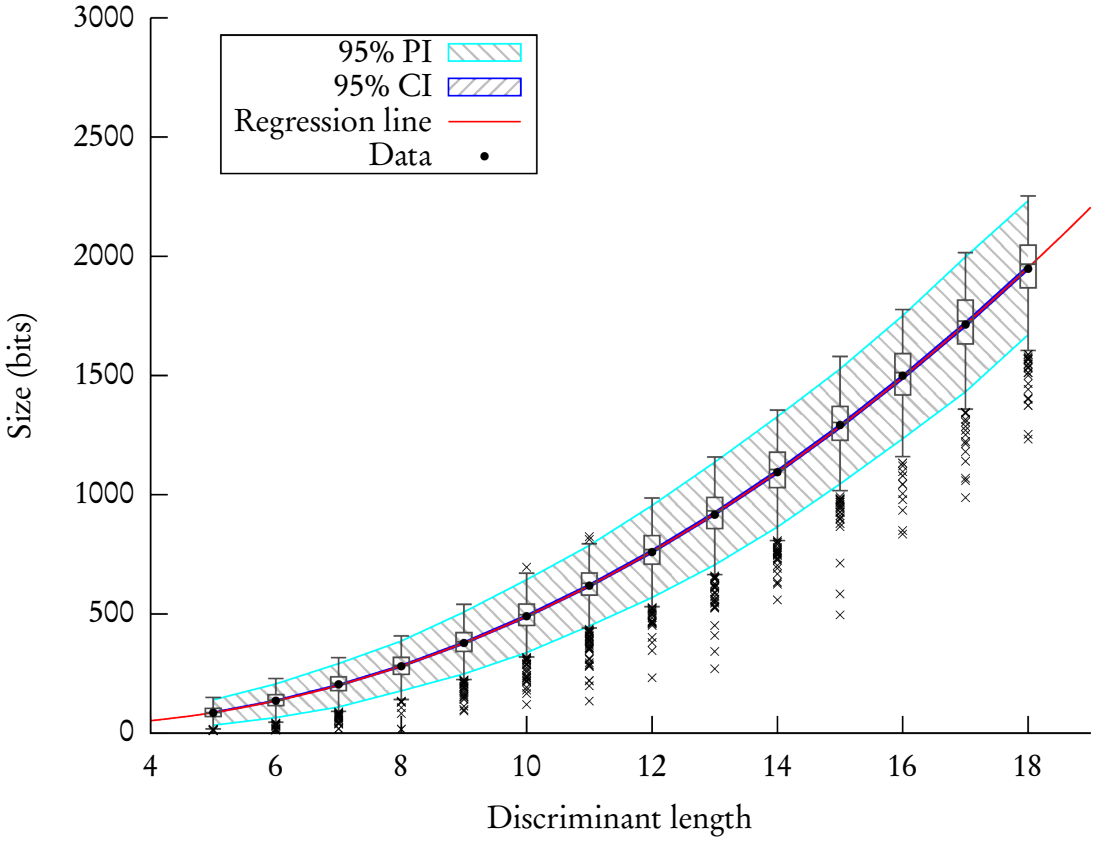


Figure B.2: Statistical analysis of random h-compact representations.

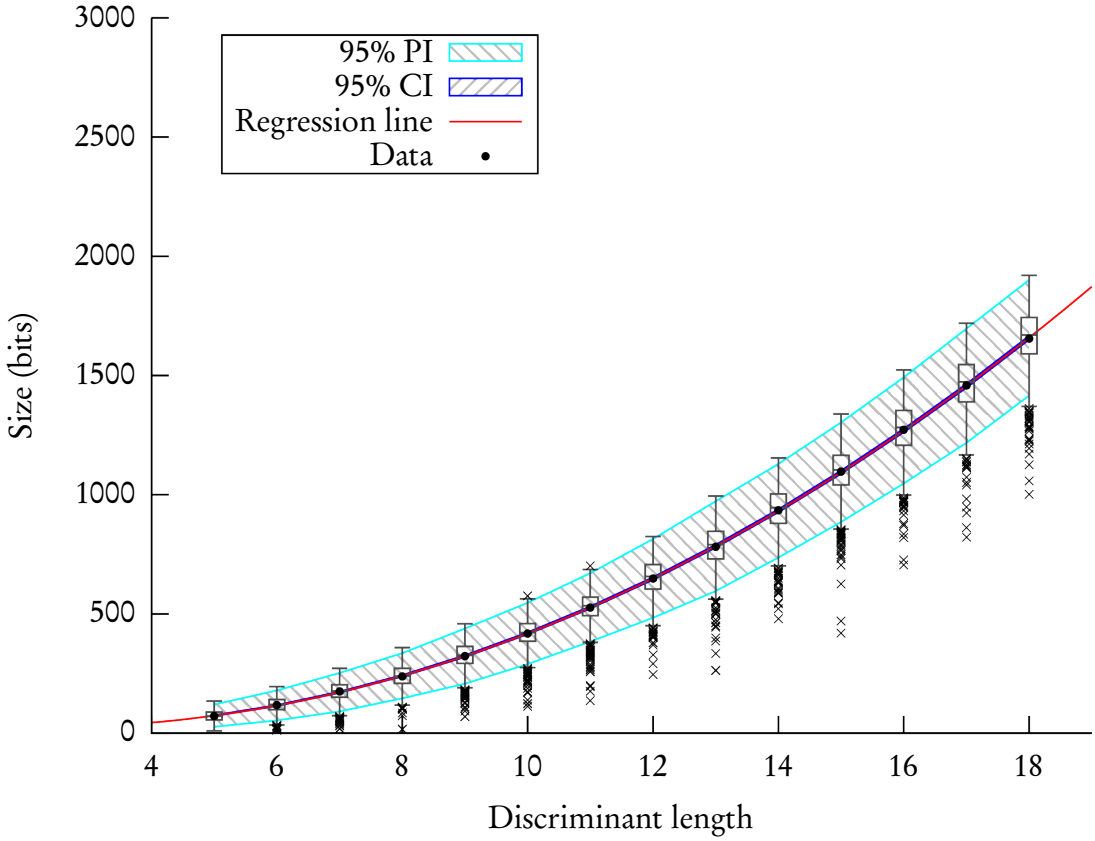


Figure B.3: Statistical analysis of random $3h$-compact representations.

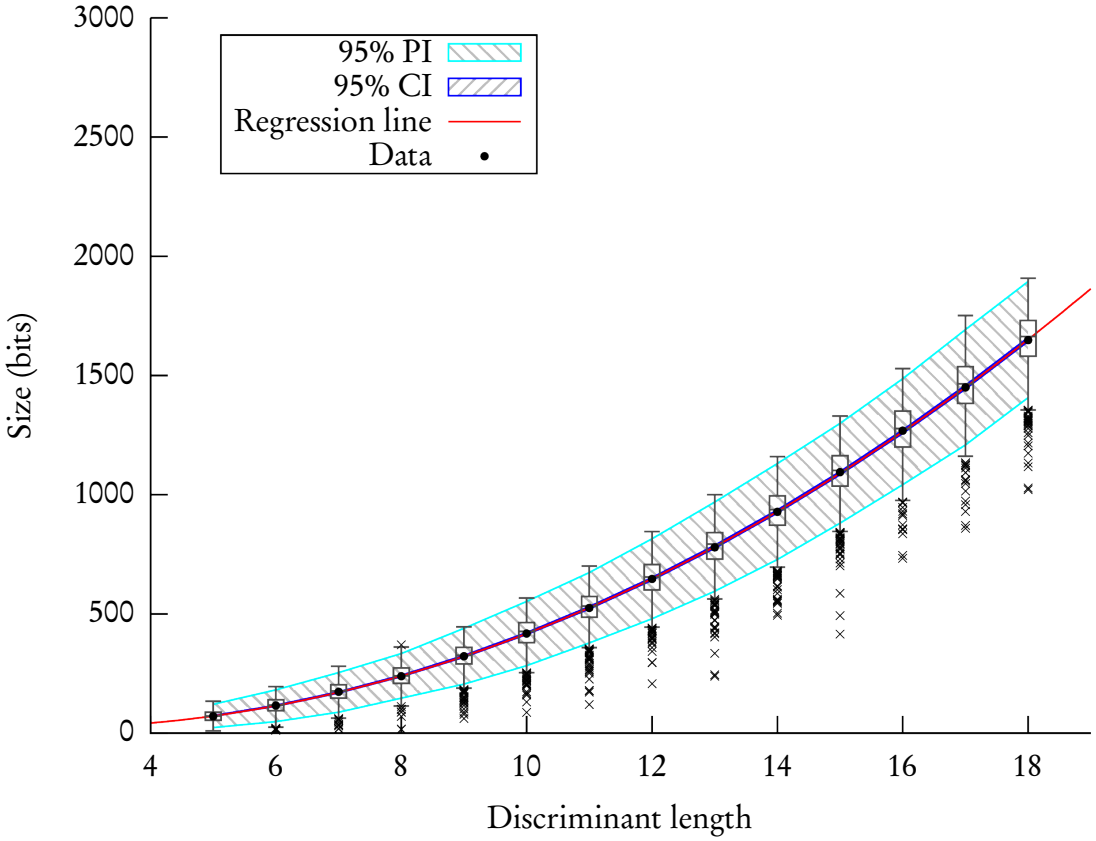


Figure B.4: Statistical analysis of random 4h-compact representations.

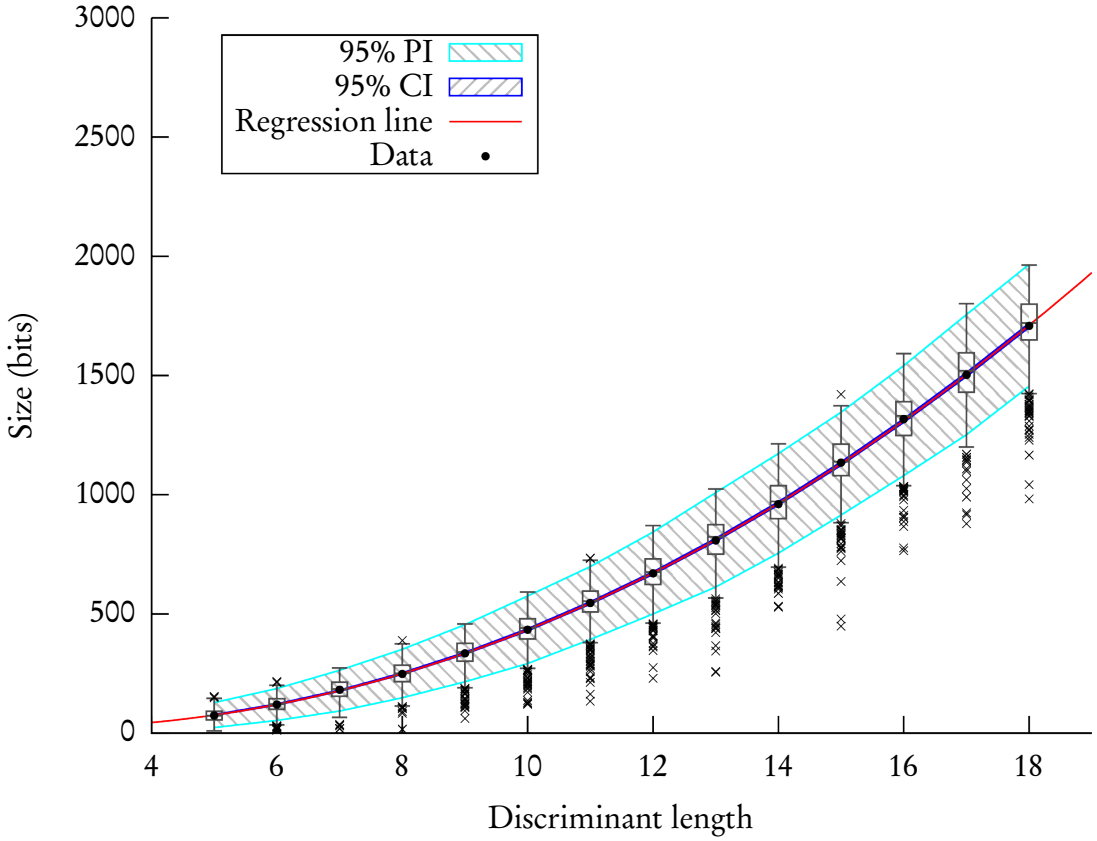


Figure B.5: Statistical analysis of random 5h-compact representations.

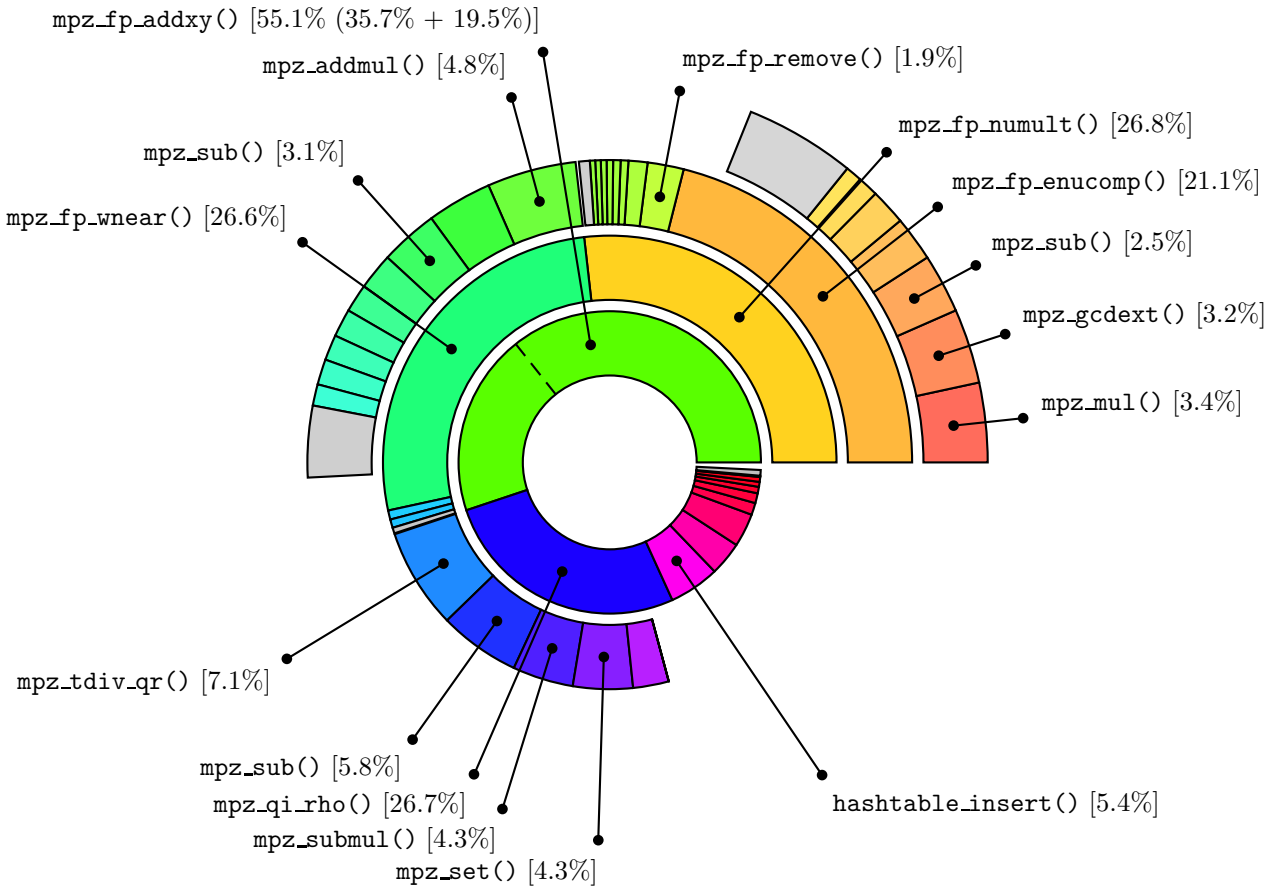


Figure B.6: Estimated cycle costs, with function calling contexts, for the single-processor $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing (LSB-32, 25 decimal digit discriminant).

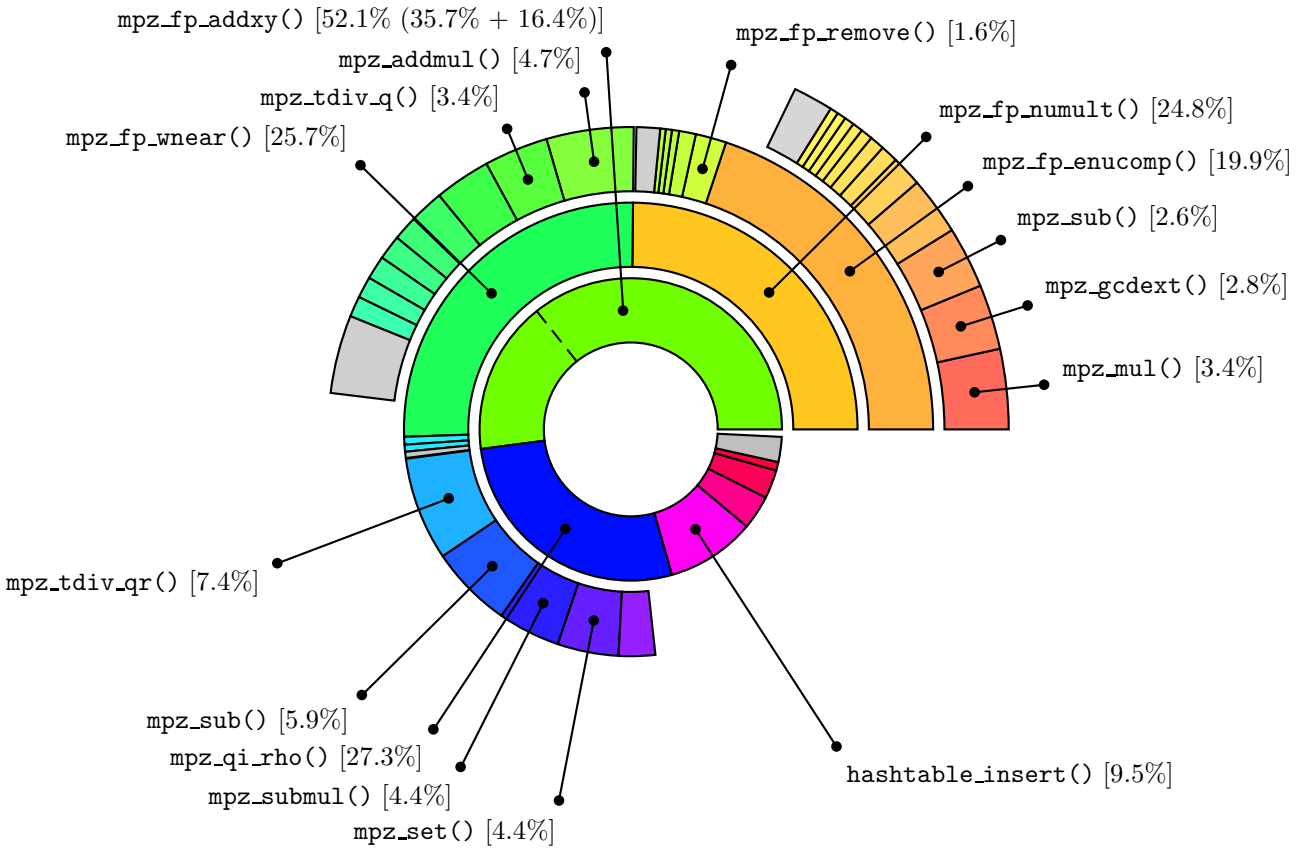


Figure B.7: Estimated cycle costs, with function calling contexts, for the single-processor $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing (LSB-32, 30 decimal digit discriminant).

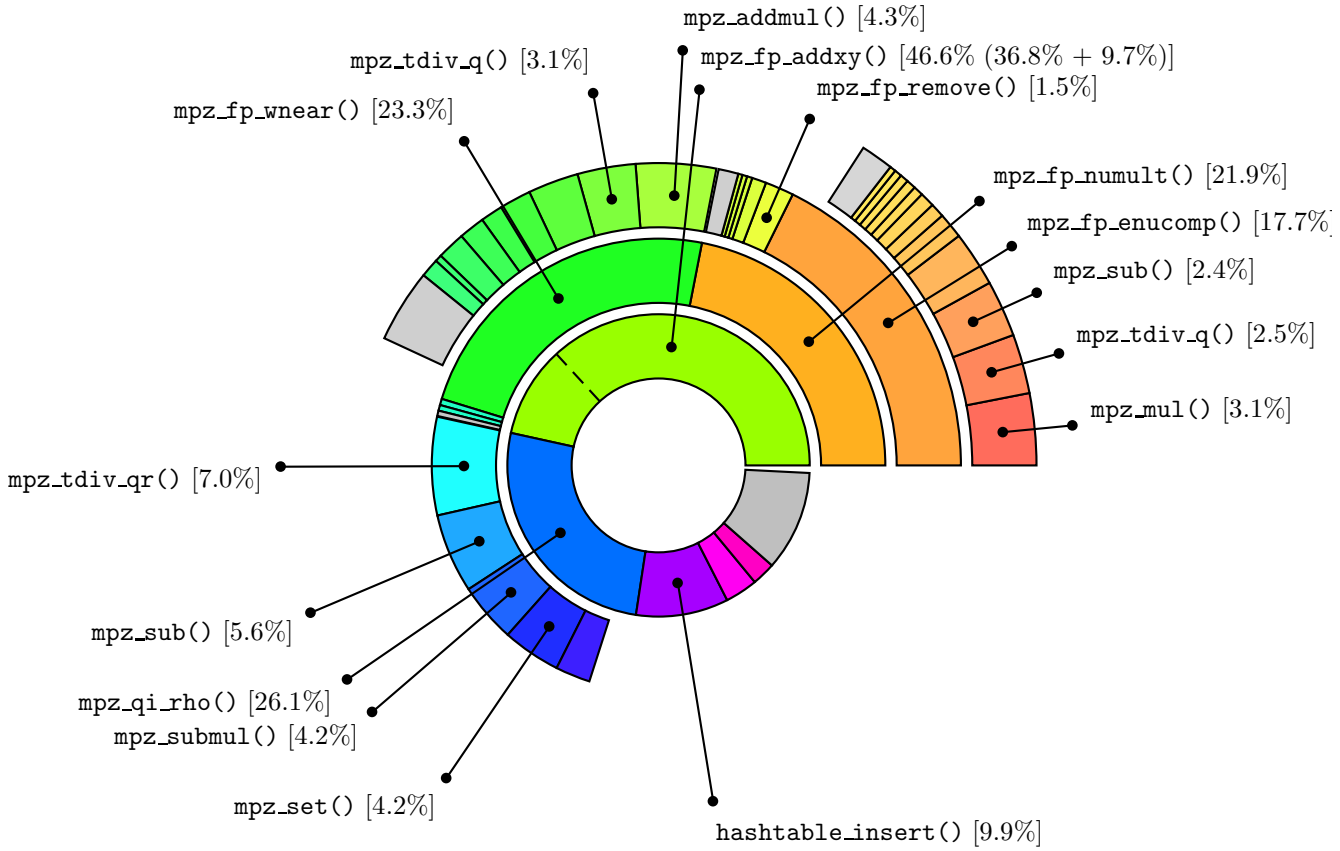


Figure B.8: Estimated cycle costs, with function calling contexts, for the single-processor $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing (LSB-32, 35 decimal digit discriminant).

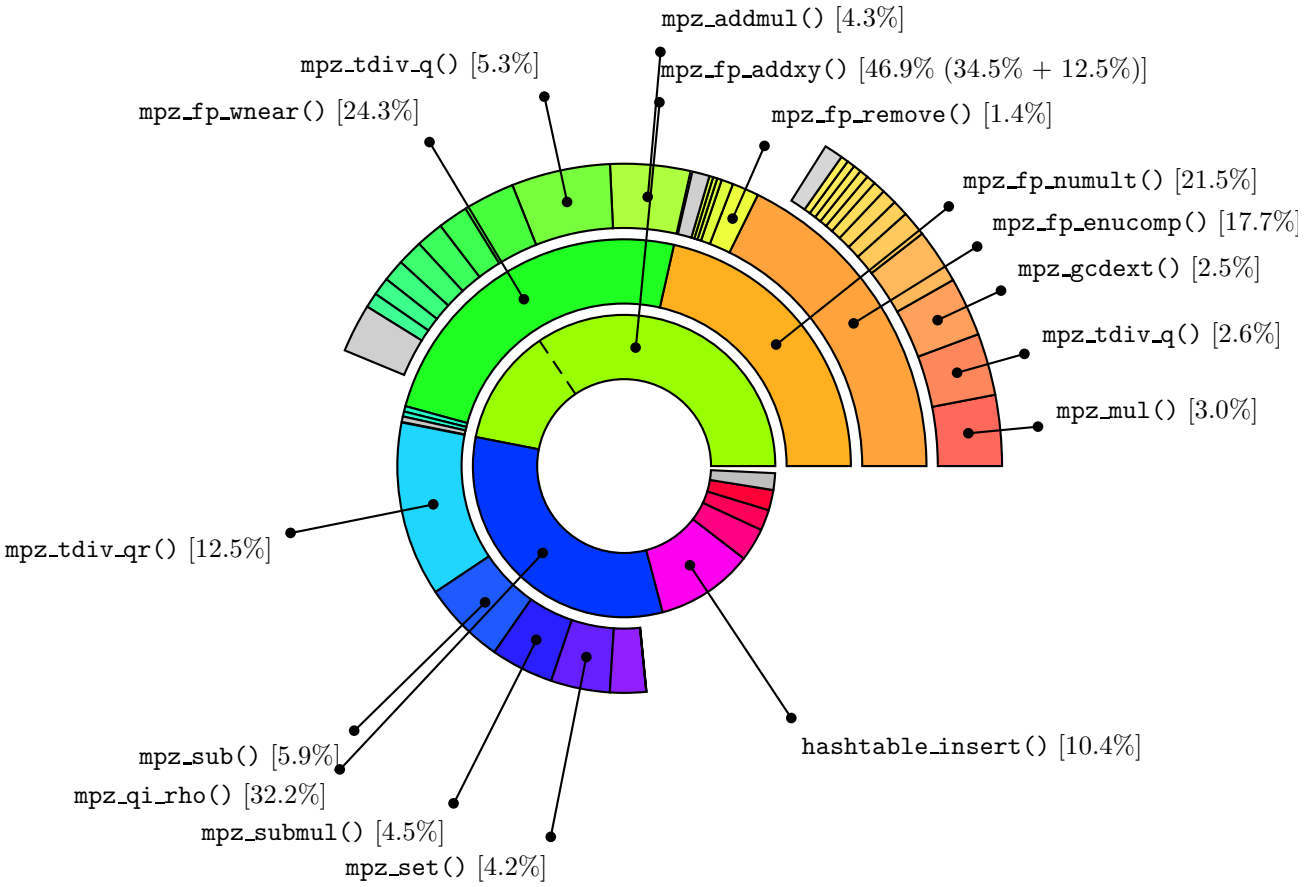


Figure B.9: Estimated cycle costs, with function calling contexts, for the single-processor $O(\Delta^{1/6+\epsilon})$ algorithm using ideal hashing (LSB-32, 40 decimal digit discriminant).