Galois Field computations: Implementation of a library and a study of the discrete logarithm problem

A thesis

submitted for the degree of

Doctor of Philosophy

in the Faculty of Engineering

by

Abhijit Das

Computer Science and Automation

Indian Institute of Science

Bangalore 560 012

September 1999

Contents

Preface

The modern study of finite fields dates back to the seminal work of Evariste Galois in 1830 [33]. It is because of this finite fields are popularly known as *Galois fields*. After Galois' untimely death, many mathematicians have devoted their energy in unleashing the properties of finite fields. At present the theory of finite fields is a vast and rich area of pure mathematics.

With the advent of error control coding theory in the early fifties of this century, finite fields started playing an important role in engineering applications. E. R. Berlekamp was probably the first who realized the need for a formal algorithmic treatment of finite fields [10]. He developed many *modern* algorithms for solving several computational problems associated with finite fields. Almost a decade later (in 1976) Diffe and Hellman's pioneering discovery [31] gave birth to the technology of public-key cryptography, and the theory of finite fields found yet another serious application that demanded further computational developments both for the users of cryptographic protocols and for those who try to break them.

Quite expectedly, the last twenty years saw intense research activities all over the world for designing faster algorithms for finite field problems. At present most of the computational problems on finite fields are reasonably satisfactorily solved, in particular, with extensive applications of randomization techniques. Known deterministic complexities of many such problems are still poor (exponential in the bit-size of the field). In addition, there are problems (like the well-known *discrete logarithm problem*) for which even randomization does not help much, and the best with which we have to be satisfied are the so-called *subexponential* algorithms. Naturally enough, this area will continue to attract many engineers and applied theoreticians at least for the next few decades.

In short this is the setting behind the conception of this thesis. We start with a survey of the known algorithms for solving some important practical problems in finite field computations. Then we talk about a computational library of functions written in C, developed as a part of the research work. This library, known as the *Galois Field Library* (abbreviated G$\mathbb{F}$L), provides built-in routines for many of the computational problems discussed in the survey just mentioned. The rest of the thesis is devoted to a study of the discrete logarithm problem over finite fields of prime cardinality. We report our efficient implementation techniques, some analytic estimates and certain heuristic improvements for some of the well-known algorithms to compute discrete logarithms.

The material in the thesis draws upon many basic results from abstract algebra, elementary number theory and the theory of finite fields. For abstract algebra, one may consult the book of Herstein [53]. For elementary number theory, we refer the reader to the book by Niven, Zuckerman and Montgomery [135]. Lidl and Niederreiter's *bible* [82] should be read by anybody interested in finite fields. For this thesis, the first four chapters of this book should be sufficient. Some knowledge of the programming language C is also strongly recommended.

The chapters of the thesis can be read more or less independently. The only sizable dependency is that of Chapter 4 on Chapter 3. The motivation for Chapter 5 also comes from the description of the cubic sieve method in Chapter 3. Apart from these no particular orders need be strictly adhered to, though reading the material in the way presented here is suggested.

Throughout the thesis we make some abuse of notations and terminology. For example, in the thesis a field is a *finite* field, a prime field is a *field* with prime cardinality, and a field extension is always an *algebraic* extension. Similarly, the term 'cryptography' refers to *public-key* cryptography. An index of notations follows this preface. We tried to be as consistent as possible regarding these notations across the different chapters. A few exceptions are allowed with the hope that these do not lead to confusions.

Acknowledgements

The first in the list of persons whom I should thank would undoubtedly be my research supervisor Prof. C. E. Veni Madhavan. When I joined the Ph.D. program, I had only very little knowledge of algebra and number theory. It was Prof. Veni Madhavan who introduced to me the modern topics of computational number theory and aroused in me the passion for numbers. The deeper I sank, the more I got addicted to the fascinating challenges of algorithmic number theory. I cannot forget the moments of sharing our thoughts, excitements, failures and successes. I am indebted to CEVM for his able guidance, for his friendship, for his care, and for everything.

I thank all the faculty members of the Department of Computer Science and Automation and the Department of Mathematics for teaching me various aspects of computer science and mathematics. Specific thanks go to Prof. Dilip P. Patil, Prof. Basudeb Datta, Prof. Renuka Ravindran, Prof. A. K. Nandakumaran and Prof. S. Sathiya Keerthi. The staff of the above two departments too deserve my sincere gratitude. I particularly thank Mr. B. Gangaraju, Ms. R. Meenakshi, Ms. Lalitha Mohan and Mr. H. S. Mohan for their excellent help. I also thank R. Lakshminarayanan, Ms. Anita R. Singh and Dr. Thomas Chacko of the Foreign Languages Section for providing excellent facilities to learn conversational and scientific German and French.

I have used some of the ideas of Dr. B. B. Amberker for designing the multi-precision integer library of G$\mathbb{F}$L. Mr. Rahul Athale has tested the G$\mathbb{F}$L extensively and provided valuable feedbacks for its improvement. My sincere thanks go to these two colleagues of mine.

I am grateful to Prof. Andrew M. Odlyzko for sending me some of his papers and to S. V. Nagaraj for suggesting me some good references. I also thank Prof. Donald Ervin Knuth, Prof. Leslie Lamport and the LaTeX 2$_\varepsilon$ team for providing nice packages with which this thesis is typeset. The contributions from GNU (Free Software Foundation) and the Linux project are also gratefully acknowledged.

During my tenure in IISc, I have spent some time in designing various software packages and systems which have nothing to do with finite fields. Some of these are Bengali Writer (BW, a set of utilities for computerized typesetting in my mother tongue Bengali), htmltotex (a program that converts html documents to plain TeX), WebScrab (a Java-based implementation of the crossword game Scrabble®), various GUI applications for X Windows, some LaTeX style files for IISc people, html documents for learning German and French. I thank all the people who encouraged me in these endeavors. I also thank users of these packages for providing me valuable feedbacks in various aspects. Special thanks go to Prof. P. A. Meyer for his valuable suggestions regarding the Roman-to-Bengali transliterator of BW, Prof. Probal Sengupta for sharing his views about Bengali fonts, Angshuman Pandey for adding two Assamese characters to BW font files, Jaijeet Roychowdhury for writing the ITRANS interface of BW METAFONT files, Rahul Athale for helping me convert my Bengali text editor to a Devnagari editor, Vladimir Rybnikov for locating a bug in htmltotex and Prof. Dilip P. Patil, Indranath Sengupta and R. Lakshminarayanan for helping me compile a German-English dictionary of mathematical terms.

I thank all of my friends who made my stay at IISc an enjoyable one. The list is long, and with the apprehension that some of them without finding their names in the list would blame me (and not my forgetfulness) for the exclusion, I am refraining from making an attempt to prepare any list. Instead let me say that I am grateful to all of them with whom I love sharing ideas (academic or non-academic), chatting, boozing, cooking, swimming, solving puzzles, playing Scrabble®, . . . I also thank all current and previous members of the working committee of ‘Spandan’, a cultural organization of Bengalis in IISc. Finally I thank the members of the magazine committee of ‘Spandan’ for publishing the Bengali literary magazine ‘আগডুম বাগডুম’.

My parents and my younger brother deserve my sincere thanks for their moral support, love and care.

Part of this research has been supported by project assistantships from the Jawaharlal Nehru Centre for Advanced Scientific Research and from Novell Software Development India Ltd., Bangalore. I express my gratitude to all the persons involved in this regard. Last but not the least, let me take this opportunity to acknowledge the excellent facilities provided by the Department of Computer Science and Automation and by the Indian Institute of Science in general.

Abstract

Computation over finite fields (also called *Galois fields*) is an active area of research in number theory and algebra, and finds many applications in cryptography, error control coding and combinatorial design. In this thesis, we describe our computational experience in this area. Our work consists of two parts. In the first part, we build a comprehensive library for working over finite fields. In the second part, we make a detailed study of the discrete logarithm problem (DLP) over prime fields.

We have developed a computational library of functions written in C for a wide range of problems that are of theoretical and practical interest in finite field computations. We call this library the Galois Field Library or $\mathrm{G}\mathbb{F}\mathrm{L}$ for short. $\mathrm{G}\mathbb{F}\mathrm{L}$ provides routines for field arithmetic and for manipulation of univariate polynomials and matrices over finite fields. It allows the user to work on finite fields of *any* characteristic and *any* cardinality. It is based on a set of routines for doing arbitrary-precision integer arithmetic and is portable, fast and memory-efficient. We have carried out extensive testing and benchmarking of $\mathrm{G}\mathbb{F}\mathrm{L}$. We demonstrate the programming techniques with $\mathrm{G}\mathbb{F}\mathrm{L}$ through some small and simple examples. We also provide an exhaustive list of functions currently provided by $\mathrm{G}\mathbb{F}\mathrm{L}$. We compare the performance of $\mathrm{G}\mathbb{F}\mathrm{L}$ with those of three other libraries, namely, LiDIA, NTL and ZEN.

Computing discrete logarithms over a prime field $\mathbb{F}_p$ is a very difficult problem for which no polynomial time algorithms are known. The best algorithms known till date are based on the index calculus method and take time subexponential in $\log p$. We concentrate on three variants of the index calculus method, namely the basic method, the linear sieve method and the cubic sieve method.

The sieve methods test a set of deterministically generated integers for smoothness over a predetermined set of small primes. The analysis of running times of these methods is based on the heuristic assumption that these deterministically generated integers behave as random integers. We start our study of the DLP by showing that the actual distribution of these integers is not random in the sense that these integers do not follow uniform distribution. To prove our claim we find out the arithmetic mean and the cumulative statistical distribution of these integers. We find that the average bit-length of these test integers is smaller than the expected bit-length of a sample of integers chosen following the uniform distribution.

We then describe our implementation details and heuristic modification schemes for the three methods mentioned above. In the basic method, our heuristic scheme reduces the number of discrete exponentiations. We also make trial divisions faster by adopting two strategies: maintaining a list of remainders and sieving. For the linear sieve method, our heuristic generates a set of integers smaller on an average than the integers checked for smoothness in the original method. This increases the chance of getting smooth integers, but decreases the ratio of the number of relations to the number of elements in the factor base. Finally for the cubic sieve method, we increase the sieving interval by a heuristic strategy. This allows us to build a larger factor base without any significant increase in the running time. We also describe efficient implementation techniques for the sieve methods and establish the superiority of the cubic sieve method over the linear sieve method for a special class of primes.

We conclude our study of the DLP by an analytic study of the congruence $X^3 \equiv Y^2 Z \pmod{p}$ subject to the condition $X^3 \neq Y^2 Z$. This congruence plays an important role in the cubic sieve method. We estimate that the total number of solutions of the congruence for a prime p is $\Theta(p^2)$. We also show that under certain heuristic assumptions, the expected number of solutions of the congruence with $1 \leqslant X, Y, Z \leqslant p^\alpha$ for $1/3 \leqslant \alpha < 1/2$ is $\Omega(p^{3\alpha-1})$. Small scale experiments reveal that apart from constant factors our estimate tallies with the experimental values quite closely.

Index of notations

General

$\mathbb{N}$	The set of natural numbers $\{1, 2, 3, \ldots\}$
$\mathbb{Z}$	The set of integers $\{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
$\mathbb{R}$	The set of real numbers
$\#S$	The cardinality of the set S
$\lvert x\rvert$	The absolute value of the real number x
$[a, b]$	The closed interval consisting of real numbers x satisfying $a \leqslant x \leqslant b$
$\log x$	Logarithm of the positive real number x to the base 10 (or to an unspecified base)
$\ln x$	Logarithm of the positive real number x to the base $e = \sum_{n=0}^{\infty} \frac{1}{n!}$
$\lg x$	Logarithm of the positive real number x to the base 2
$m \mid n$	The integer m divides the integer n
$m \nmid n$	The integer m does not divide the integer n
$p^e \mid\mid n$	For a prime p, a non-zero integer n and a non-negative integer e, $p^e \mid n$ but $p^{e+1} \nmid n$
p	A prime integer
q	A prime or a prime power
q_i	The ith prime ($q_1 = 2$, $q_2 = 3$ and so on)
$d(n)$	Number of (positive integral) divisors of the integer $n \neq 0$
(m, n)	The greatest common divisor (gcd) of the integers m and n
$\lfloor x \rfloor$	Largest integer less than equal to the real x (the floor of x)
$\lceil x \rceil$	Smallest integer greater than equal to the real x (the ceiling of x)
$a \equiv b \pmod m$	The integers a and b are congruent modulo the integer $m > 0$, that is, $m \mid (a - b)$
$\psi(x, y)$	The number of positive integers $\leqslant x$ all of whose prime factors are $\leqslant y$
$L\langle p, \omega, c\rangle$	$\exp\left((c + o(1))(\log p)^{\omega}(\log\log p)^{1-\omega}\right)$
$L(p, c)$	$L\langle p, 1/2, c\rangle = \exp\left((c + o(1))\sqrt{\ln p \ln\ln p}\right)$
$L[c]$	$L(p, c)$ when p is understood from the context
L	$L[1]$
γ	The Euler constant $= 0.57721566\ldots$
$\zeta(s)$	The Riemann zeta function
κ	The exponent of matrix multiplication: $\kappa = 3$ for the "classical" algorithm; for the fastest known algorithm due to Coppersmith and Winograd [29], we can take $\kappa = 2.376$.
$\mathcal{L}(d)$	$\log d \log\log d$
ERH	The Extended Riemann Hypothesis
$f(n) = O(g(n))$	[The Big-O notation] There exist $n_0 \in \mathbb{N}$ and a constant $c > 0$ such that $\lvert f(n)\rvert \leqslant c\lvert g(n)\rvert$ for all $n \geqslant n_0$.
$f(n) = \Omega(g(n))$	[The Big-Ω notation] There exist $n_0 \in \mathbb{N}$ and a constant $c > 0$ such that $\lvert f(n)\rvert \geqslant c\lvert g(n)\rvert$ for all $n \geqslant n_0$.
$f(n) = \Theta(g(n))$	[The Big-Θ notation] $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, that is, there exist $n_0 \in \mathbb{N}$ and constants $c_1, c_2 > 0$ such that $c_1\lvert g(n)\rvert \leqslant \lvert f(n)\rvert \leqslant c_2\lvert g(n)\rvert$ for all $n \geqslant n_0$.
$f(n) = O\tilde{~}(g(n))$	[The soft-O notation] $f(n) = O(g(n)h(\log n))$ where $h(x)$ is a polynomial in x.
$\langle a\rangle$	Ideal generated by a in a ring

Finite fields

$\mathbb{F}_q$	Finite field of cardinality q
$\mathbb{F}_q^*$	The multiplicative group of $\mathbb{F}_q$
$\mathbb{F}_p$	Finite field of prime cardinality p
$\overline{a}\ \ (a \in \mathbb{F}_p)$	The representative of a in the set $\{ 0, 1, \ldots, p-1 \}$
g	A primitive element of $\mathbb{F}_p$, that is, a generator of the cyclic group $\mathbb{F}_p^*$
$\mathrm{char}(K)$	Characteristic of a field K
$\mathrm{Tr}_{E/K}(a)$	Trace of an element $a \in E$ over $K \subseteq E$ (K, E are fields)
$\mathrm{Tr}_E(a)$	Trace of an element $a \in E$ over the prime subfield of E. The suffix E is dropped, when E is understood from the context.
$N_{E/K}(a)$	Norm of an element $a \in E$ over $K \subseteq E$ (K, E are fields)
$N_E(a)$	Norm of an element $a \in E$ over the prime subfield of E. The suffix E is dropped, when E is understood from the context.
$[K : F]$	Degree of the (algebraic) extension of the field K over the field F

Galois field library

$R = 2^b$	Radix for multi-precision integer arithmetic

Discrete logarithm

DLP	The finite field discrete logarithm problem
$\mathrm{ind}_g(a)$	The discrete logarithm (or index) of a with respect to g

Basic index calculus method

B	The factor base = $\{ q_1, q_2, \ldots, q_t \}$
t	The size of the factor base
g^j	A random power of g
v_i	Remainder of division of p by q_i (Heuristic B1)
$\rho_{r,i}$	Remainder of division of $\overline{g^j} + rp$ by q_i (Heuristic B1)
$\mathfrak{L}$	Approximate logarithm (to base 2) of the non-smooth part of $\overline{g^j} + rp$ (Heuristic B2)

Linear sieve method

t	Number of small primes in the factor base
H	$\lfloor \sqrt{p} \rfloor + 1$
J	$H^2 - p$
M	The sieving interval is $-M \ldots M$
$\mathfrak{A}$	The array maintained for storing sums of (approximate) logarithms
$T(c_1, c_2)$	$J + (c_1 + c_2)H + c_1c_2$
$\overline{T}$	The average of $\lvert T(c_1, c_2) \rvert$ over all choices of (c_1, c_2)
$T_{\max}$	The maximum value of $\lvert T(c_1, c_2) \rvert$ over all choices of (c_1, c_2)
$\mathfrak{C}(\eta)$	$\#\{ (c_1, c_2)$ such that $\lvert T(c_1, c_2) \rvert \leqslant \eta T_{\max} \}$
$\mathfrak{c}(\eta)$	$\mathfrak{C}(\eta) / \mathfrak{C}(1)$
ξ	Sieving tolerance for the approximate version of the linear sieve method is set at $\xi \lg q_t$
μ	The sieving interval is $-\mu \ldots \mu$ (Heuristics L1 and L2)
H_r	$\lfloor \sqrt{rp} \rfloor + 1$ (Heuristics L1 and L2)
J_r	$H_r^2 - rp$ (Heuristics L1 and L2)
$T_r(c_1, c_2)$	$J_r + (c_1 + c_2)H_r + c_1c_2$ (Heuristics L1 and L2)
$\overline{T}_{\mathrm{heu}}$	Average value of $\lvert T_r(c_1, c_2) \rvert$ over all choices of r, c_1 and c_2 (Heuristics L1 and L2)
$\mathfrak{r}$	$\overline{T}_{\mathrm{heu}} / \overline{T}$ (Heuristics L1 and L2)

Cubic sieve method

t	Number of small primes in the factor base
X, Y, Z	A solution of $X^3 \equiv Y^2 Z \pmod{p}$, $X^3 \neq Y^2 Z$
M	The sieving interval is $-M \ldots M$
$\mathfrak{A}$	The array maintained for storing sums of (approximate) logarithms
$R(A, B, C)$	$Z + (AB + AC + BC)X + (ABC)Y$
$\overline{R}$	The average of $\lvert R(A, B, C)\rvert$ over all choices of (A, B, C)
$R_{\max}$	The maximum value of $\lvert R(A, B, C)\rvert$ over all choices of (A, B, C)
$\mathfrak{D}(\eta)$	$\#\{\,(A, B, C)$ such that $\lvert R(A, B, C)\rvert \leqslant \eta R_{\max}\,\}$
$\mathfrak{d}(\eta)$	$\mathfrak{D}(\eta)/\mathfrak{D}(1)$
ξ	Sieving tolerance for the approximate version of the cubic sieve method is set at $\xi \lg q_t$
τ	Total number of triples (A, B, C) satisfying $-M \leqslant A \leqslant B \leqslant C \leqslant M$, $A + B + C = 0$
ν	Size of the factor base
λ	For the heuristic variation, $A \geqslant -\lambda M$ $(1 \leqslant \lambda \leqslant 2)$
τ_λ	Total number of triples (A, B, C) satisfying $-\lambda M \leqslant A \leqslant B \leqslant C \leqslant M$, $A + B + C = 0$ (Modified cubic sieve)
ν_λ	Size of the factor base for the modified cubic sieve method
U	$1 + \frac{t}{M}$ (Modified cubic sieve)
λ^*	Optimal value of λ (equals $-U + \sqrt{U^2 + 4U + 1}$) (Modified cubic sieve)
CSC	The congruence $X^3 \equiv Y^2 Z \pmod{p}$
S	The solution set of CSC: $\{(X, Y, Z) \mid X^3 \equiv Y^2 Z \pmod{p},\ 1 \leqslant X, Y, Z < p\}$
$S_=$	$\{(X, Y, Z) \in S \mid X^3 = Y^2 Z\}$
$S_{\neq}$	$\{(X, Y, Z) \in S \mid X^3 \neq Y^2 Z\}$
S_α	$\{(X, Y, Z) \in S_{\neq} \mid 1 \leqslant X, Y, Z \leqslant p^\alpha\}$

1 Introduction

Computation over finite fields (also called *Galois fields*) is an active area of research in number theory and algebra, and finds many applications in cryptography, error control coding and combinatorial design [85, 115]. In this thesis, we describe our computational experience in this area. Our work consists of two parts. In the first part, we build a comprehensive library for working over finite fields. In the second part, we make a detailed study of the discrete logarithm problem over prime fields.

In Section 1.1, we provide a short survey of the algorithms for finite fields, known until recently. In Section 1.2, we introduce our work that the rest of the thesis deals with. We also summarize in this section the organization of the thesis.

1.1 Algorithms for finite fields

The theory of finite fields, originating from the seminal work of Galois [33], continues to be an important and active branch of mathematics. While theoreticians have devoted their effort in extracting properties of finite fields, applied mathematicians and engineers have found it immensely useful to apply these properties to many practical problems – most notably in the areas of error control coding, cryptography and combinatorial design. This has stimulated interest in various computational problems associated with finite fields. In this section we survey some of the recent results on algorithmic aspects of finite fields.

The first encyclopedic treatment of the theory of finite fields is the celebrated book by Lidl and Niederreiter [82], that covers both theoretical and computational results on finite fields, known till early eighties. More recent results can be found in [85] and [115] – the former is a supplement to [82], whereas the latter provides a thorough treatise on finite field algorithms. The recent paper [118] by Shparlinski and Mullen lists many open problems in the areas of theoretical, combinatorial and computational aspects of finite fields. Notwithstanding the usefulness, rather indispensability, of these works, these do not cover the vast research literature of the last few years. This survey aims at filling up this gap and, as expected, focuses mostly on papers that appeared in this decade only. Some older papers are also referred, sometimes for the sake of completeness, sometimes to preserve continuity and sometimes for mere beauty of the results. This survey, by itself, is neither exhaustive nor complete. Also for the sake of brevity, we do not delve into the detailed aspects of the algorithms. We only mention the running times of the algorithms and, whenever possible, some short descriptions of the same. This survey is intended to provide many pointers, hints and references which interested readers and researchers would find worth investigating.

1.1.1 Notations

In what follows, we shall assume, unless otherwise stated, that the following symbols designate the entities as defined below. We shall often use several of these symbols throughout this survey without specific mention of their meanings.

$\mathbb{F}_q$	The finite field with cardinality q
$q = p^m$	p a prime number and m a positive integer
$\mathbb{F}_q[x]$	The ring of univariate polynomials with coefficients from $\mathbb{F}_q$
$\mathbb{F}_q[x_1, \ldots, x_n]$	The ring of multivariate polynomials with coefficients from $\mathbb{F}_q$
n	The number of indeterminates (for multivariate polynomials)
f	A polynomial in $\mathbb{F}_q[x]$ or $\mathbb{F}_q[x_1, \ldots, x_n]$
d	The (total) degree of f
t	A bound on the number of nonzero terms of f
T	A bound on the number of nonzero terms of f or any of its irreducible factors
$O\tilde{}$	The soft O notation (i.e. order notation up to logarithmic factors)
κ	The runtime for multiplying two $r \times r$ matrices is $O(r^\kappa)$: $\kappa = 3$ for the "classical" algorithm; for the fastest known algorithm due to Coppersmith and Winograd [29], we can take $\kappa = 2.376$
$\mathcal{L}(d)$	$\log d \log \log d$
ERH	The Extended Riemann Hypothesis

1.1.2 Arithmetic over finite fields

- The effect of representation of elements of $\mathbb{F}_q$ on the basic operations ($+$,$-$,$\times$ etc.) on $\mathbb{F}_q$ is discussed in Section 6.1.2 of [90].
- If $\mathbb{F}_{q^s}$ is represented as $\mathbb{F}_q/\langle f \rangle$ where f is an irreducible (over $\mathbb{F}_q$) polynomial of degree s in $\mathbb{F}_q[x]$, then addition and subtraction in $\mathbb{F}_{q^s}$ can be performed using $O(s)$ operations in $\mathbb{F}_q$, multiplications with $O(s\mathcal{L}(s))$ operations in $\mathbb{F}_q$ and divisions with $O(s\mathcal{L}(s)\log s)$ operations in $\mathbb{F}_q$.
- Use of *normal basis* for representing elements of $\mathbb{F}_q$ is known to be very convenient for computing the product of two elements of $\mathbb{F}_q$. *Low complexity* normal bases – namely, *optimal* and *near-optimal* normal bases – deserve specific mention in this respect. Chapter 5 of [85] is a good introduction to these topics and provides many references to related works.
- Itoh and Tsujii [61] presented a configuration of parallel multipliers for $\mathbb{F}_{2^m}$ based on *polynomial basis*. They use $O(m^2)$ AND gates and $O(m^2)$ XOR gates and achieve an operation time of about $(\log m)T$ where T is the delay time of an XOR gate.
- Itoh and Tsujii [60] presented an algorithm for computing multiplicative inverses in $\mathbb{F}_{2^m}$ using normal basis. The algorithm uses repeated squaring technique that requires at most $2\lceil \log_2(m-1) \rceil$ multiplications in $\mathbb{F}_{2^m}$ and $(m-1)$ cyclic shifts.
- Efficient sequential and parallel algorithms for exponentiation in a finite field using normal basis are given by Stinson [124] for $q = 2^m$ and by von zur Gathen [37, 39] for general q. Von zur Gathen has also proved that his algorithms are optimal.
- Table of discrete logarithms with respect to a primitive element and Zech's logarithm tables [59, 85] speed up arithmetic in *small* finite fields and their algebraic extensions.
- The papers [3, 130] report various implementation issues for the basic operations in finite fields of characteristic 2.
- **Computation of traces and norms:** Let $\mathbb{F}_q = K \subseteq E = \mathbb{F}_{q^s}$. Von zur Gathen and Shoup's "repeated doubling" algorithm [43] computes the trace $\mathrm{Tr}_{E/K} : E \to K$ of an element in E using $O(\log s)$ additions in E and $O(\log s)$ powering operations

in E of the form $\beta \mapsto \beta^{q^j}$, where $\beta \in E$ and $1 \leq j < s$. The norm $N_{E/K} : E \to K$ of an element in E, on the other hand, can be computed using $O(s\mathcal{L}(s)\log s)$ operations in K, of which only $O(s)$ are divisions [38, 111].

1.1.3 Polynomial arithmetic over finite fields

Here we list the (best-known) running times for the basic arithmetic operations on univariate polynomials over $\mathbb{F}_q$.

- *Evaluation* of a polynomial in $\mathbb{F}_q[x]$ of degree $\leq d$ at a point in $\mathbb{F}_q$ can be performed using $O(d)$ operations ($+$,$-$,$\times$ only) in $\mathbb{F}_q$ (Horner's rule).
- *Evaluation* of a polynomial in $\mathbb{F}_q[x]$ of degree $\leq d$ at a point in $\mathbb{F}_{q^s}$ can be performed using $O(d^{(\kappa-1)/2}s + d^{1/2}s\mathcal{L}(s))$ operations in $\mathbb{F}_q$ [111] (assuming a polynomial representation of $\mathbb{F}_{q^s}$ over $\mathbb{F}_q$).
- *Addition* and *subtraction* of two polynomials in $\mathbb{F}_q[x]$ of degree $\leq d$ can be performed using $O(d)$ operations ($+$,$-$ only) in $\mathbb{F}_q$.
- *Multiplication* of two polynomials in $\mathbb{F}_q[x]$ of degree $\leq d$ can be performed using $O(d\mathcal{L}(d))$ operations ($+$,$-$,$\times$ only) in $\mathbb{F}_q$.

  A lower bound of $2.5d - o(d)$ on the number of multiplications/divisions required to compute the product is shown in [17]. Averbuch et. al. [5] showed that if $d \leq q$, then any optimal algorithm for computing the polynomial product is based on Chinese remainder theorem.
- *Division with remainder* involving two polynomials of degree at most d can be done with $O(d\mathcal{L}(d))$ operations in $\mathbb{F}_q$.
- Let $\alpha_1, \ldots, \alpha_d \in \mathbb{F}_q$. Then the *coefficients* of $(x-\alpha_1)\ldots(x-\alpha_d) \in \mathbb{F}_q[x]$ can be computed using $O(d\mathcal{L}(d)\log d)$ operations ($+$,$-$,$\times$ only) in $\mathbb{F}_q$.
- Let f and g be polynomials in $\mathbb{F}_q[x]$ of degree $\leq d$, $g \neq 0$. Then $f \pmod g$ can be computed using $O(d\mathcal{L}(d))$ operations in $\mathbb{F}_q$.
- Let $f, g_1, \ldots, g_k$ be polynomials in $\mathbb{F}_q[x]$ s.t. $\deg f \leq d$ and $\deg g_1 + \ldots + \deg g_k \leq d$. Then $f(\bmod g_1), \ldots, f(\bmod g_k)$ can be computed using $O(d\mathcal{L}(d)\log k)$ operations in $\mathbb{F}_q$.
- Let f and g be polynomials in $\mathbb{F}_q[x]$ of degree $\leq d$. Then the gcd of f and g can be computed using $O(d\mathcal{L}(d)\log d)$ operations in $\mathbb{F}_q$.

  For the proof of most of these facts see [107].

1.1.4 Finding roots of univariate polynomials

If one factors a univariate polynomial $f \in \mathbb{F}_q[x]$ over $\mathbb{F}_q$, one can read off the roots of f in $\mathbb{F}_q$ from the linear factors of f. On the other hand, the problem of univariate factorization reduces in polynomial time to the problem of root finding over $\mathbb{F}_q$ (see for example [131]).

- Berlekamp proposed a powerful randomized algorithm [11] which can be used when $q = p^m$ for any odd prime p and any integer $m \geq 1$. The expected running time is $O(d^2 \log d \log q)$ $\mathbb{F}_q$ operations. This algorithm is sometimes referred to as the *Berlekamp–Rabin algorithm* for root finding.
- *Berlekamp trace algorithm* [11] is another method for root finding that is useful when q is small and m is large.

- *Berlekamp, Rumsey and Solomon's algorithm* [12] computes the least affine multiple of f and then computes the roots of the affine multiple by solving a linear system of equations.
- *Oorschot and Vanstone's algorithm* [98] also uses the least affine multiple.
- Menezes et. al. [88] present a generalization of Moenck's root finding algorithm over $\mathbb{F}_q$. The generalized algorithm is deterministic, given a primitive element of $\mathbb{F}_q$. If $q-1$ is b-smooth, where $b = (\log q)^{O(1)}$, then the algorithm runs in polynomial time.

  See [85] and [87] for a description and comparison of these methods.

1.1.5 Sparse multivariate polynomial interpolation

The sparse multivariate interpolation problem can be stated as: To reconstruct (i.e. interpolate) a t-sparse polynomial (i.e. a polynomial with at most t terms) in n variables, given a black box which will produce the value of the polynomial for any value of the arguments.

- Clausen et. al. [24] proved a lower bound of $\Omega(n^t)$ for interpolation over a fixed finite field $\mathbb{F}_q$ when the black box can only evaluate points lying in ${\mathbb{F}_q}^n$. Consequently, it is impossible to solve the problem efficiently without enabling the black box to evaluate points over extension fields of $\mathbb{F}_q$.
- The algorithm of Grigoriev et. al. [51] evaluates f at points in a finite field of cardinality $q^{\lceil 2\log_q(nt)\rceil+3}$. The parallel time for the algorithm is $O(\log^3(nt))$ on $O(n^2t^6\log^2(nt))$ number of processors.
- Roth and Benedek [103] proposed an algorithm for the special case $q = 2$.
- The best known algorithm for multivariate interpolation is due to Huang and Rao [57]. It is an effective adaptation of Ben-Or and Tiwari's algorithm [9] to the case of finite fields. The algorithm needs $4t^2d - 2td + 2t$ evaluation points in an extension field of cardinality $q^{\lceil \log_q((8t-2)td^2+1)\rceil}$. The (parallel) running time for the algorithm is $O(\log^4(td)\log^2(tm)log^4p)$ and the processor requirement is polynomial in t, d, m, n and $\log p$.

1.1.6 Univariate polynomial factorization

A brief summary of the univariate factorization algorithms is given below. As par our notations we let $f \in \mathbb{F}_q[x]$ be a polynomial of degree d which we want to factorize.

- Berlekamp's Q-matrix method [10] is the first modern deterministic algorithm for univariate polynomial factorization over finite fields. The running time of this algorithm is $O(d^3q)$.
- Camion proposed a randomized algorithm [20] for computing the primitive idempotents for improving the running time of Berlekamp's Q-matrix algorithm.
- Zassenhaus [131] reduced the factorization problem to the problem of determining roots of certain polynomials over $\mathbb{F}_q$.
- Camion [21] and Cantor and Zassenhaus [22] independently discovered another *randomized* algorithm which does not rely on Berlekamp's subalgebra. This algorithm runs in $O\tilde{}(d^2\log q)$ operations.
- Shoup's *deterministic* algorithm [107] completely factors f in $O(q^{1/2}(\log q)^2d^{2+\epsilon})$ bit operations where d^ϵ denotes a fixed but unspecified polynomial in $\log d$.

- Shparlinski [115] showed that the running time bound of [107] can be improved to $O(q^{1/2}(\log q)d^{2+\epsilon})$ bit operations. This is supposedly the best known running time for a *deterministic* algorithm for univariate factorization over finite fields.
- Niederreiter [95] proposed a new deterministic factorization algorithm for polynomials over finite fields that is based on a new type of linearization of the factorization problem. It uses differential equations in rational function fields and normal bases of field extensions.
- Göttfert's improvement [49] over [95] works for finite fields of characteristic 2. The total cost of calculating all r monic irreducible factors of f by this algorithm is $O(dm^3 + d^\kappa m^\kappa)$ arithmetic operations in $\mathbb{F}_2$ plus $O(r^2 M_q(d) \log d)$ arithmetic operations in $\mathbb{F}_q$, where κ is the exponent of fast matrix multiplication, and $M_q(d)$ is the arithmetic complexity of multiplying two polynomials of degree $\leq d$ in $\mathbb{F}_q[x]$.
- Niederreiter and Göttfert [96] propose another extension of [49] for arbitrary finite fields. It requires $O(qr^2)$ polynomial gcd's, $O(qr^2)$ polynomial multiplications/divisions and $O(qr^2 d)$ arithmetic operations in $\mathbb{F}_q$. Here r is the number of irreducible factors of f.
- Von zur Gathen and Shoup [43] give a new *probabilistic* univariate factorization algorithm that uses $O((d^2 + d\log q)(\log d)^2 \log\log d)$ arithmetic operations over $\mathbb{F}_q$. This algorithm is based on a new way of computing Frobenius maps.
- The best known *probabilistic* algorithm for factorization of univariate polynomials over finite fields is proposed in [66]. It uses the *equal degree factorization* technique of [43] that requires $O(d^{1.688} + d^{1+o(1)} \log q)$ operations in $\mathbb{F}_q$. It solves the *distinct degree factorization* problem by a *baby step/giant step* strategy using $O(d^{1.815} \log q)$ operations in $\mathbb{F}_q$.
- If one assumes the **ERH**, deterministic polynomial time univariate factorization algorithms are known for certain special classes of polynomials. Some references are [35, 55, 100, 101, 102, 109].

Open problems

1. For fixed q, the fastest known deterministic algorithm is [107] that runs in time $O(d^{2+o(1)})$. It remains an open problem to find a subquadratic deterministic algorithm.
2. It is not known whether there exists a deterministic algorithm for factorization of univariate polynomials over finite fields that runs in time polynomial in $\log q$ and d.

1.1.7 Multivariate polynomial factorization

- In [75] Lenstra gives a deterministic multivariate polynomial factorization algorithm that makes use of a basis reduction strategy for lattices over $\mathbb{F}_q[y]$. Let $f \in \mathbb{F}_q[x_1, \ldots, x_n]$ with $\deg_{x_i} f = d_i$. Let $D_j = \prod_{i=j}^{n}(d_i + 1)$. Then Lenstra's algorithm factorizes f completely over $\mathbb{F}_q$ using $O\left((2d_1)^{2n} D_2^2 D_3^4 + (2d_1)^{3n-6} D_2^3 pm\right)$ arithmetic operations in $\mathbb{F}_q$.
- Von zur Gathen [34] describes a polynomial-time probabilistic algorithm for multivariate factorization. It uses an effective version of Hilbert's irreducibility theorem.
- Let $f \in \mathbb{F}_q[x_1, \ldots, x_n]$ ($n \geq 3$) of total degree $d \geq 2$ with r irreducible factors and such that the number of nonzero terms in f or any of its irreducible factors is at

most T. The algorithm in [40] correctly computes the irreducible factorization of f with probability at least $1 - 2^{-d}$ and with an expected number of bit operations $O(k^3(2d^3T)^r + k^{17}T^3)$ where $k = \max(d, n, \log q)$.

- Von zur Gathen and Kaltofen[41] present a probabilistic algorithm that finds the irreducible factors of a bivariate polynomial over $\mathbb{F}_q$ in time $O(d^{11} \log d \log q)$.
- Wan's bivariate factorization algorithm [126] is probabilistic and has running time $O(d^{4.89} \log^2 d \log q)$.
- Shparlinski [116] shows that for $p > d^3$ there exists a deterministic algorithm that factors all except possibly $O(p^{(d+1)(d+2)/2}(\log\log p)^{-2})$ polynomials $f(x, y) \in \mathbb{F}_p[x, y]$ of total degree d, in $O(d^{3.7} \log^{\epsilon} p + d^{2+\epsilon} \log^2 p)$ operations in $\mathbb{F}_p$.
- Huang and Rao [57] showed that their sparse multivariate interpolation algorithm can be combined with the black box Berlekamp algorithm of [67] to give a probabilistic parallel multivariate factorization algorithm with expected running time of $O\left(\log^2 d \log^2(dM) \log^4 p + \log^2(kd) \log^2(kM \log k) \log k \log^4 p\right)$ where $k = \max(T, d)$, $\mathbb{F}_{p^M}$ is a "suitable" extension of $\mathbb{F}_q$. (Other notations are as in the third algorithm [40] of this section.) If δ is a given bound on the probability of failure, then the algorithm runs in expected time which is polylog in d, k, m, n, $\log \frac{1}{\delta}$ and p and the number of processors used is a polynomial in k, d, m, n and $\log p$.

1.1.8 Irreducibility testing

Factoring a univariate or multivariate polynomial allows one to conclude whether the given polynomial is irreducible. This is however not an efficient method for testing irreducibility of a polynomial. In fact, polynomial time deterministic algorithms exist for both univariate and multivariate irreducibility testing.

- **Univariate polynomials:** It is easy to check if a polynomial f is square-free by checking if $\gcd(f, f') = 1$. For a square-free polynomial f the first stage of Berlekamp's Q-matrix method [10] gives the number of irreducible factors of f. This procedure requires a total number of $O(d^3 + d^2 \log q)$ operations in $\mathbb{F}_q$ [90]. For another deterministic check in polynomial time see [85, Theorem 3.28]. Shoup [111] summarizes some more efficient methods for irreducibility testing. These references are tabulated below:

Author	**Reference**	**Complexity**
Butler	[19]	$O\tilde{}(d^{\kappa} + d \log q)$
Rabin	[99]	$O\tilde{}(d^2 \log q)$
Ben-Or	[8]	worst-case: $O\tilde{}(d^2 \log q)$
		average: $O\tilde{}(d \log q)$
Gathen and Shoup	[43]	$O\tilde{}(d^{(\kappa+1)/2} + d \log q)$

- **Multivariate polynomials:** Kaltofen [63, 64] shows that for polynomials in $\mathbb{F}_q[x_1, \ldots, x_n]$ both irreducibility over $\mathbb{F}_q$ and absolute irreducibility can be tested deterministically in polynomial time. This algorithm seemingly takes $O\tilde{}(d^8 \log q)$ operations in $\mathbb{F}_q$ for the bivariate case ($n = 2$).

1.1.9 Construction of irreducible polynomials

Consider the following problem: If a finite field $\mathbb{F}_q$ and a positive integer d are given, how can one efficiently construct an irreducible polynomial of degree d over

$\mathbb{F}_q$? There is presently no deterministic polynomial time algorithm known to solve this problem.

- The construction of irreducible polynomials for many special cases (for example, for special values of q, d etc.) can be found in Sections 3.3 and 3.5 of [82] and in Sections 3.2 through 3.5 of [85].
- Rabin's randomized algorithm [99] is based on the fact that the probability that a random monic polynomial of degree d in $\mathbb{F}_q[x]$ is irreducible, is nearly $\frac{1}{d}$. (See Exercise 3.3 of [85].)
- Shoup [108] gives a deterministic algorithm for prime fields $\mathbb{F}_p[x]$. This algorithm takes $O(\sqrt{p}(\log p)^3 d^{3+\epsilon} + (\log p)^2 d^{4+\epsilon})$ that is, $O\tilde{}(d^3 p^{\frac{1}{2}} + d^4 \log^2 p)$ $\mathbb{F}_p$ operations. If one assumes the **ERH**, the same can be done in deterministic time $O\tilde{}(\log^2 p + d^4 \log p)$. The problem of constructing an irreducible polynomial has been shown to be deterministically reducible in time polynomial in d and $\log p$ to the problem of factoring polynomials over $\mathbb{F}_p$. This algorithm can be modified to work for arbitrary $\mathbb{F}_q$ in which case the running time is $\sqrt{p}(d \log q)^{O(1)}$. Shoup also shows that for any constant $0 < c < \frac{1}{4}$, there exists a randomized algorithm (depending on c) with the following properties: It uses $\lceil d \log p \rceil$ random bits, halts in time polynomial in d and $\log p$, and upon termination, it either outputs an irreducible polynomial of degree d over $\mathbb{F}_p$ or reports failure. Furthermore, the probability that it fails is no more than $1/p^{cd}$.
- The probabilistic algorithm proposed by Shoup [111] uses $O\left((d^2 \log d + d \log q) \log d \log \log d\right)$ $\mathbb{F}_q$ operations.
- The best known probabilistic algorithm is due to Shoup [112] that uses an expected number of $O\tilde{}(d^2 + \log q)$ arithmetic operations in $\mathbb{F}_q$.
- Shparlinski [117] gives a survey on many results associated with the construction of irreducible polynomials over finite fields.
- **Minimal polynomials:** Given an extension $\mathbb{F}_{q^s}$ of degree s over $\mathbb{F}_q$ and an element $\alpha \in \mathbb{F}_{q^s}$, it is possible to compute the minimal polynomial g of α over $\mathbb{F}_q$ deterministically using $O(s^{(\kappa+1)/2})$ operations in $\mathbb{F}_q$. Moreover, if a bound d on the degree of g is given to the algorithm, then it uses only $O(d^{(\kappa-1)/2}s + d^{1/2}s\mathcal{L}(s))$ operations in $\mathbb{F}_q$ [111].

Open problems

1. Does a probabilistic $O\tilde{}(d^k + d \log q)$ algorithm exist for $k < 2$ for the construction of an irreducible polynomial of degree d over $\mathbb{F}_q$?
2. Does an $O\tilde{}(d^2)$ algorithm exist that solves the problem deterministically for $q = 2$? (Shoup [111] suggests a $O\tilde{}(d^3)$ algorithm.)

1.1.10 Construction of primitive polynomials and primitive elements

There are no known polynomial time algorithms for constructing a primitive root (or a primitive polynomial), or even for testing whether a given element is a primitive element. (Note that a primitive polynomial is the minimal polynomial of a primitive element, also called a primitive root.)

- A test for $f \in \mathbb{F}_q[x]$ to be primitive is given in [82, Theorem 3.18].
- Since the product of all primitive polynomials over $\mathbb{F}_q$ of degree d is equal to the cyclotomic polynomial Q_e, with $e = q^d - 1$, factorization of Q_e gives all primitive

polynomials over $\mathbb{F}_q$ of degree d. This fact and another method that is based on the construction of a primitive element of $\mathbb{F}_{q^d}$ are discussed in Section 3.3 of [82]. Also see Section 6.1.3 of [90].

- Shoup [110] considers the problem of deterministically generating in polynomial time a subset of $\mathbb{F}_{p^m}$ that contains a primitive root. A solution to this problem is given for small p, i.e., for $p = m^{O(1)}$. This problem is also solved for large p and $m = 2$ under the assumption of **ERH**.
- Buchmann and Shoup [18] proposed a deterministic polynomial time algorithm for constructing primitive roots in $\mathbb{F}_{p^m}$ assuming the **ERH** and assuming availability of factorization of $p^m - 1$.
- Von zur Gathen and Shparlinski [44] present an algorithm for computing *Gauss periods* of a specific type in polynomial time. These Gauss periods have been shown to have exponentially large multiplicative orders.
- Shparlinski [117] provides a survey on the construction of primitive roots over $\mathbb{F}_q$ and on some related problems. In particular, the paper states that for a field $\mathbb{F}_q$, a primitive root can be found deterministically in time $\tilde{O}(q^{\frac{1}{4}})$ and in time $\tilde{O}(q^{\frac{1}{5}})$ under the **ERH**. This paper also mentions a probabilistic algorithm to find a primitive root of $\mathbb{F}_q$ in the expected time $\exp\Big((1 + o(1))(\log q \log\log q)^{\frac{1}{2}}\Big)$.
- Extensive tables of primitive polynomials over prime fields can be found in [52], [133] and [134].
- **Primitive normal element:** Lenstra and Schoof [79] showed that $\mathbb{F}_{q^m}$ always contains a primitive element that generates a normal basis of $\mathbb{F}_{q^m}$ over $\mathbb{F}_q$. Stepanov and Shparlinski [123] showed that if θ is a primitive element of $\mathbb{F}_{q^m}$ then for $N \geq \max\Big(\exp\exp(c_1 \ln^2(m)), c_2 m \ln(q)\Big)$ there is at least one element in the set $\{\theta, \theta^2, \ldots, \theta^N\}$ which generates a primitive normal basis. Morgan and Mullen [92] provide extensive tables for primitive normal polynomials over prime fields.

1.1.11 Construction of nonresidues

Buchmann and Shoup [18] considered the problem of constructing a kth power nonresidue in $\mathbb{F}_{p^m}$, i.e. an element that is not a perfect kth power of any element in $\mathbb{F}_{p^m}$, where k is a prime divisor of $p^m - 1$. Given $\alpha \in \mathbb{F}_{p^m}$, testing if α is a kth power nonresidue has a trivial solution: just test if $\alpha^{(p^m-1)/k} \neq 1$. Probabilistically, the problem of constructing nonresidues also has a trivial solution: just choose $\alpha \in \mathbb{F}_{p^m}$ at random and test if it is a kth power nonresidue. However, the deterministic complexity of constructing nonresidues is currently unknown, even under the **ERH**. Buchmann and Shoup [18] shows that for any *fixed* m, this problem can be solved in deterministic polynomial time assuming the **ERH**. The research problem 3.2 of [85] states a related problem on computing nonresidues.

1.1.12 Counting number of zeros

In this section we shall consider effective procedures for counting the number of solutions of $f \in \mathbb{F}_q[x_1, \ldots, x_n]$ over ${\mathbb{F}_q}^n$.

- For a treatment of earlier results see Chapter 6 of [82] and the book of Small [122].
- Karpinski and Luby [68] gave an $O(nt^3 \log(\frac{1}{\delta})/\epsilon^2)$ algorithm for counting the number of zeros of $f \in \mathbb{F}_2[x_1, \ldots, x_n]$ with relative error at most ϵ and with probability at least $1 - \delta$, where t is the number of nonzero terms of f.

- Grigoriev and Karpinski [50] generalized the algorithm of [68] for arbitrary q. The generalized algorithm has the running time

$$O\left(nt(t+1)^{(q-1)(1+\log q)}q\log q\, P(q)\log\left(\frac{1}{\delta}\right)/\epsilon^2\right)$$

where t is the number of terms of f and $P(q)$ is the bit cost for multiplication and powering in $\mathbb{F}_q$ (which is $O(\log^2 q\log\log q\log\log\log q)$). This paper also gives an (ϵ-δ)–approximation algorithm for estimating the number of nonzeros of f over $\mathbb{F}_q$ whose complexity is $O(nt^{\log q+2}P(q)\log(\frac{1}{\delta})/\epsilon^2)$.

- Huang and Ierardi [56] consider the problem of counting the number of points on a plane curve given by a homogeneous polynomial $f\in\mathbb{F}_p[x,y,z]$ which is rational over the ground field $\mathbb{F}_p$.

- Some deterministic and probabilistic methods are presented in [42] for counting and estimating the number of points on curves over finite fields and on their projections. Let $f\in\mathbb{F}_q[x,y]$ have degree d, $\mathcal{C}=\{\,f=0\,\}\subseteq\mathbb{F}_{q^l}^2$ and let $A\subseteq\mathbb{F}_{q^m}\subseteq\mathbb{F}_{q^l}$. This algorithm computes the number of points of $\mathcal{C}$ over A in $O\tilde{}(\#A\cdot m\cdot(d\log q+d^{1.7}\log l))$ operations in $\mathbb{F}_q$ (where $\#A$ denotes the cardinality of A).

  See the papers [45, 58] for some related algorithms.

1.1.13 Solution of linear systems

- Kaltofen and Pan [65] showed that the solution set of a system of n linear equations in n unknowns can be computed in parallel with randomization simultaneously in poly-logarithmic time in n and with only as many processors as are necessary to multiply two $n\times n$ matrices.

- Coppersmith [27] proposes a method for solving large sparse systems of homogeneous linear equations over $\mathbb{F}_2$. This algorithm is a modification of an algorithm due to Wiedemann [129].

- See [74] for practical implementation issues regarding solutions of large sparse linear systems over finite fields.

1.1.14 Permutation polynomials and functions

- A polynomial $f\in\mathbb{F}_q[x]$ is called a *permutation polynomial* if the mapping $\mathbb{F}_q\to\mathbb{F}_q$ given by $a\mapsto f(a)$ is bijective. Von zur Gathen [36, 38] gives probabilistic algorithms for checking permutation polynomials in time $O(d\log q)$ (and exceptional polynomials in time $O(\log q\cdot d^{O(1)})$).

- Let $f=g/h\in\mathbb{F}_q[x]$ with $\gcd(g,h)=1$. Then f induces a partial mapping $\mathbb{F}_q\to\mathbb{F}_q$ by $a\mapsto f(a)\ \forall a\in\mathbb{F}_q$ with $h(a)\neq 0$. If f is total and bijective, then f is called a *permutation function* over $\mathbb{F}_q$. (In particular, if $h=1$, then $f=g$ is a permutation polynomial.) Ma and von zur Gathen [83] consider the problem of deciding whether f is a permutation function over $\mathbb{F}_q$. They have shown that this problem is deterministic polynomial time reducible to the problem of factoring univariate polynomials over finite fields. A deterministic test is described that uses $O(q\,M(d)\log d)$ operations in $\mathbb{F}_q$ if $q<64d^4$ and $O(q^{1/2}d^2\,M(d)\log q)$ operations if $q\geq 64d^4$ (where $d=\max(\deg g,\deg h)$ and $M(d)$ is the cost of multiplication in $\mathbb{F}_q$). The algorithm assumes that $d\leq\operatorname{char}\mathbb{F}_q$. A simple probabilistic test is also described for the case $q \geq 64d^4$, which uses $O(dM(d) \log q \log \epsilon^{-1})$ operations in $\mathbb{F}_q$ and $\lceil 2d \log \epsilon^{-1} \rceil$ random choices where ϵ is the probability of failure (when the answer is NO).

- The articles [80, 81] by Lidl and Mullen describe a series of open problems related to permutation polynomials over finite fields. Also see [93, 94, 118, 125] for further open problems and more up-to-date surveys on permutation polynomials.

1.1.15 The discrete logarithm problem

- The old methods, like Shank's baby-step-giant-step method and Pollard's rho heuristic, for the computation of discrete logarithms over $\mathbb{F}_q$ take worst-case expected running time $O(\sqrt{q})$ [85]. The Pohlig-Hellman method solves the problem in time $O(\sqrt{\mathfrak{p}} \log \mathfrak{p})$, where $\mathfrak{p}$ is the largest prime factor of $q-1$. In particular, if $q-1$ has only small factors, the Pohlig-Hellman method is quite efficient. However in the worst case $\mathfrak{p} = O(q)$ and hence this method gives a fully exponential algorithm.
- The index calculus method [85] is currently the best known method for computing discrete logarithms over finite fields. It takes an expected running time of $L\langle q, \omega, c\rangle = O(\exp((c + o(1))(\log q)^{\omega}(\log \log q)^{1-\omega})$ which is *subexponential* in $\log q$, where c and $0 < \omega < 1$ are constants. Various variants of the index calculus method are used in practice.
- Coppersmith, Odlyzko and Schroeppel [28] describe three variants of the index calculus method for prime fields $\mathbb{F}_p$. These methods are called the linear sieve method, the residue list sieve method and the Gaussian integer method. Each of these takes time $L\langle p, 1/2, 1\rangle$. The same paper also proposes a cubic sieve method that can solve the problem in time $L\langle p, 1/2, \sqrt{\alpha/2}\rangle$ for some $1/3 \leqslant \alpha < 1/2$. Also see [77] for a note on the cubic sieve method. LaMacchia and Odlyzko [73] describe an implementation of the linear sieve and the Gaussian integer methods. Also look at the survey article by McCurley [84].
- Gordon [47] uses number field sieves for computing discrete logarithms over prime fields. This algorithm has a heuristic expected running time of $L\langle p, 1/3, c\rangle$. See [78] for a good introduction to number field sieves. Weber et. al. [105, 127, 128] have implemented and proved the practicality of the number field sieve method. Also see Schirokauer's paper [104].
- Odlyzko [97] surveys the algorithms for the fields $\mathbb{F}_{2^m}$. The best algorithm for these fields is Coppersmith's algorithm [26]. This takes time $L\langle q, 1/3, c\rangle$. No analog of this algorithm is known for prime fields. Gordon and McCurley [48] successfully used Coppersmith's algorithm for the computation of discrete logarithms in $\mathbb{F}_{2^{401}}$ and $\mathbb{F}_{2^{503}}$.

1.1.16 Elliptic curves over finite fields

- For elliptic curve group law of addition, see [71, 72, 85, 86].
- Schoof [106] gives an algorithm for counting the number of points on elliptic curves over finite fields.
- **Elliptic curve discrete logarithm problem:** Computation of discrete logarithms in elliptic curves over finite fields seems to be a very difficult problem. A direct adaptation of the index calculus method for computing elliptic curve discrete logarithms is expected to lead to a running time *worse* than that of brute-force search

[120]. Logarithms in a singular elliptic curve defined over $\mathbb{F}_q$ with a cusp can be computed in polynomial time. The discrete logarithm problem for a general elliptic curve over $\mathbb{F}_q$ can be reduced to the discrete logarithm problem in the field $\mathbb{F}_{q^k}$ for a suitable k. However, this k is quite large in general and the reduction takes time exponential in $\log q$ [7]. For supersingular elliptic curves, this reduction can be done in probabilistic polynomial time. Recently, Joseph H. Silverman has proposed a new method, called the *xedni calculus method* [119], which, though originally devised for computing elliptic curve discrete logarithms, can be applied to finite fields. However, this method has been experimentally and heuristically shown to be impractical [62]. Koblitz [70] and Miller [91] pointed out that elliptic curves can be used to build cryptosystems. See the books by Koblitz [71, 72] and Menezes [85, 86] for good surveys on the elliptic curve discrete logarithm problem and its application to cryptography.

1.2 About this thesis

In this section, we outline the work reported in this thesis. Our work can be classified into two major tracks outlined below. We also describe the conventions and organization of the thesis.

1.2.1 Galois Field Library

We have developed a computational library of functions for a wide range of problems that are of theoretical and practical interest in finite field computations. We call this library the Galois Field Library or G$\mathbb{F}$L for short. G$\mathbb{F}$L provides routines for field arithmetic and for manipulation of univariate polynomials and matrices over finite fields. It encompasses most of the topics described in the survey of the last section. To the best of our knowledge, G$\mathbb{F}$L provides the largest variety of built-in routines among the existing symbolic computation packages (like LiDIA, NTL and ZEN) that support computations over finite fields. It allows the user to work on finite fields of *any* characteristic and *any* cardinality. It is based on a set of routines for doing arbitrary-precision integer arithmetic and is portable, fast and memory-efficient. We have carried out extensive testing and benchmarking of G$\mathbb{F}$L. We have used it in our studies of the discrete logarithm problem described next. We have also used it for testing various cryptographic applications.

1.2.2 Study of the discrete logarithm problem

The security of many cryptographic protocols depends on the difficulty of solving the discrete logarithm problem (DLP) over finite fields [28, 70, 73, 84, 97]. We study the DLP over prime fields and report our implementation results and heuristic modification schemes for some methods for solving the DLP. We provide some analytic estimates on certain parameters that arise in connection with these methods.

We concentrate our study on three popular methods for solving the DLP. These are the basic index calculus method, the linear sieve method and the cubic sieve method. We propose heuristic variants of each of these methods. For the basic method, these variants lead to speedup factors between 1.5 and 3. For the sieve methods, our heuristic schemes help us build larger factor bases. The sieve methods generate a set of integers deterministically and check these integers for smoothness over a set of small primes. The analysis of the methods is based on the heuristic assumption that these integers, though generated deterministically, behave as random integers. We show that this behavior is not random in the sense that these integers do not follow uniform distribution. We derive the average and maximum of these integers and plot the distribution of them. Our study shows that the actual behavior of these integers is *better* than that of a sample of integers chosen following the uniform distribution. We also study the effects of our heuristic modification schemes on these average values and distributions. Finally, we find estimates of the number of solutions of a certain congruence that arises in connection with the cubic sieve method.

1.2.3 The organization of the thesis

The rest of the thesis is organized as follows. In Chapter 2, we describe the basic conventions and programming paradigms of $\mathrm{G}\mathbb{F}\mathrm{L}$. We demonstrate the working of the library through some small examples. Running times of many basic $\mathrm{G}\mathbb{F}\mathrm{L}$ routines are also provided and compared with those of analogous routines in some other existing libraries, namely LiDIA, NTL and ZEN.

Chapter 3 starts with a description of the three methods mentioned above for solving the DLP. We then calculate expressions for maximum and average values of the integers checked for smoothness in the sieve methods. We also derive the formulas for the distribution of these numbers.

Chapter 4 is devoted to a description of the implementation details and heuristic modification schemes for the three methods. In the basic method, our heuristic scheme reduces the number of discrete exponentiations. We also make trial divisions faster by adopting two strategies: maintaining a list of remainders and sieving. For the linear sieve method, our heuristic generates a set of integers smaller on an average than the integers checked for smoothness in the original method. This increases the chance of getting smooth integers, but decreases the ratio of the number of relations to the number of elements in the factor base. Finally for the cubic sieve method, we increase the sieving interval by a heuristic strategy. This allows us to build a larger factor base without any significant increase in the running time. In this chapter, we also describe efficient implementation techniques for the sieve methods and establish the superiority of the cubic sieve method over the linear sieve method for a special class of primes.

The congruence $X^3 \equiv Y^2 Z \pmod{p}$ plays a major role in the cubic sieve method. In Chapter 5, we estimate that the total number of solutions of the congruence for a prime p subject to the condition $X^3 \neq Y^2 Z$ is $\Theta(p^2)$. We also show that under certain heuristic assumptions, the expected number of solutions of the congruence with $1 \leqslant X, Y, Z \leqslant p^\alpha$ for $1/3 \leqslant \alpha < 1/2$ is $\Omega(p^{3\alpha-1})$. Small scale experiments reveal that apart from a constant factor our estimate tallies with the experimental values quite closely.

In Chapter 6 we conclude the thesis with a summary of the work done and suggesting the scope for further research in this area.

Each chapter (like this) starts with an introductory note stating the basic theme discussed in that chapter. The main results are also highlighted there. Some of the chapters contain appendix sections after the regular sections. We elaborate the details of certain calculations in these appendices. A quick reference for $\mathrm{G}\mathbb{F}\mathrm{L}$ also appears in the appendix of Chapter 2.

2 Galois Field Library

Galois Field Library (G𝔽L) is a portable general-purpose computational library of functions written in C for working over finite fields. The library provides a comprehensive treatment of operations in prime fields and their arbitrary finite extensions. This chapter illustrates the main features of this library. Running times of many basic G𝔽L routines are also provided. This library should be useful to application programmers for developing programs in the areas of public-key cryptography, error control coding and combinatorial design.

The basic goal for the design of G𝔽L has been to build and make available an easy-to-use and comprehensive library for computer scientists and mathematicians. While implementing the library routines, we have put emphasis on generality and uniform representation of fields and field elements, which most of the other existing libraries are lacking. At the same time we did not want to sacrifice performance at the cost of generality. Unfortunately these two goals are sometimes conflicting. We have tried to make a reasonable trade-off between them. We claim that in spite of the generality and uniformity of G𝔽L library calls, the performance of G𝔽L is comparable to (and, in some cases, better than) that of the other existing libraries.

In Section 2.1, we introduce G𝔽L and highlight the salient features of the library. In Section 2.2, we explain how one can represent various algebraic entities (integers, fields, polynomials, matrices and so on) in G𝔽L. In Section 2.3, we illustrate by two examples the programming techniques with G𝔽L library calls. The first example is a toy one that explains manipulation of multi-precision integers, polynomials and matrices using G𝔽L library calls. In the second example we write three procedures that implement the ElGamal public-key encryption scheme [32]. A high-level listing of the functions currently provided by G𝔽L appears in Section 2.4. In Section 2.5, we tabulate typical timing results for basic field operations and polynomial arithmetic using G𝔽L. We also compare the timings of the G𝔽L routines with those of the corresponding routines provided by some other symbolic computation libraries that support working over finite fields. We conclude this chapter by an appendix that provides a detailed description of the prototypes of G𝔽L library calls.

2.1 Introduction

Galois Field Library (G𝔽L) is a portable general-purpose computational library of functions written in C for working over finite fields (also called *Galois fields*). G𝔽L provides routines for field arithmetic and for manipulation of univariate polynomials and matrices over finite fields. The salient features of G𝔽L are as follows.

1. *Generality:* G𝔽L works on finite fields of any characteristic and any cardinality. It allows one to work both on prime fields and on their finite algebraic extensions obtained by adjoining an arbitrary number of (algebraic) elements. That is, one first creates prime fields and then defines extensions

of these prime fields, extensions of these extensions, and so on. G𝔽L *does not* impose any restriction on the characteristic and extension degree of finite fields, as long as the computer system can provide sufficient memory for storing the relevant data.

2. *Extensiveness:* G𝔽L provides extensive tools for a wide range of problems that are of computational importance in the theory of finite fields. To the best of our knowledge, G𝔽L provides the largest number of built-in functions for working over Galois fields among all the symbolic computation packages.

3. *Performance:* Use of suitable data structures, fine tuning of basic arithmetic operations, and use of several implementation tricks such as table look-up and modularity make G𝔽L a fast and efficient tool.

4. *Efficient memory management:* G𝔽L uses dynamic arrays for representing many algebraic data (for example, polynomials, matrices and even multi-precision integers). The built-in routines of G𝔽L allocate and deallocate memory associated with these arrays as and when needed. This practice leads to an efficient management of system memory and relieves the operating system of garbage collection overheads.

5. *Multi-precision support:* The field arithmetic of G𝔽L is based on a set of routines for carrying out arbitrary precision integer arithmetic. However, the use of these multi-precision routines are much slower compared to the single-precision routines for fields where both types of routines can be used. To alleviate this difficulty, G𝔽L routines have been designed to use the single-precision integer arithmetic routines whenever possible. In particular, for fields of characteristic 2, G𝔽L provides routines that make extensive use of bit operations instead of integer arithmetic operations.

6. *Portability:* G𝔽L can be used on any workstation that has an ANSI C compiler. It is totally self-contained in the sense that it is not built as a library over existing packages. G𝔽L has been built as an easy-to-use tool.

In what follows, we describe the basic conventions and features provided by G𝔽L. We also demonstrate through some examples the basic paradigms that users should follow, when they use G𝔽L routines in their programs. This chapter is by no means a complete reference to G𝔽L. It is intended to give the reader a flavor of the programming techniques using G𝔽L. For a complete reference manual of G𝔽L, we refer the reader to [30]. We do not go into the implementation details of G𝔽L routines (though we sometimes outline the strategy behind them). Nor do we make an attempt to define algebraic terms and concepts that are well-known and can be found in text books on algebra [53], linear algebra [54] or finite fields [82, 85, 90, 115]. We define and/or explain terms that we introduce during the course of the discussion, i.e. those that are specific to G𝔽L. Similarly we assume that the reader is familiar with the programming language C. In what follows we present pertinent material in a manner so as to hold the interest of both mathematicians and computer scientists.

Randomized algorithms play a very important role in computations over finite fields. This is because for many of the common problems, deterministic polynomial time algorithms are not known. (By a polynomial time algorithm, we mean one that runs in time polynomially bounded by the logarithm of the cardinality of the field.) In many cases, even if deterministic algorithms are known, they cannot normally compete in speed with their probabilistic counterparts. G$\mathbb{F}$L implements the probabilistic versions of the algorithms whenever applicable or useful.

Finite field algorithms find immense applications in the areas of public-key cryptography, error control coding, combinatorial designs and so on. We expect that this library would be useful to programmers who develop application packages in these areas. We plan to distribute G$\mathbb{F}$L as a *freeware* for academic and research purposes.

2.2 Basic data structures

In this section, we describe how G$\mathbb{F}$L represents various algebraic entities necessary for computations over finite fields. We explain only the most important data types. The appendix at the end of this chapter gives a complete list of these data structures and the library calls. The reference manual [30] provides all the details left out here.

2.2.1 Multi-precision integers

In typical applications involving finite fields, one uses integers much larger than the maximum integer representable by a long int. For example, a long int typically contains 32 bits in small work-stations or 64 bits in large machines and thus is not sufficient for storing elements of $\mathbb{F}_p$ with p of length 400 bits. A floating point number (say, double), on the other hand, can represent numbers in this range, but not to the full precision. (Typically a 64-bit double has 52 bits precision.) We, therefore, need an alternative representation of large integers. The multi-precision integer library of G$\mathbb{F}$L is designed for this purpose. Multi-precision integers are special data structures that can store an integer value across several long ints. We use dynamic arrays for holding the individual words of a multi-precision integer.

data type mpint

```
typedef struct {   /* Multi-precision integer */
    char sign;     /* '+' for positive integers, '–' for negative integers, ' ' for zero */
    int size;      /* Number of longs needed to represent the integer */
    long *word;    /* link to the array of longs holding the integer */
} mpint;
```

Thus an mpint defines a representation of *signed* integers of *arbitrary* length. The first field of the struct indicates the sign of the integer: '+', '–' or ' ' (space) according as whether the integer it holds is positive, negative or zero. The second field (size) is the exact number of long ints necessary to hold the multi-precision integer and the third field (word) is a pointer to a dynamic array of long int holding the fragments of the (absolute value of the) integer.

The usual arithmetic operations (+, –, *, /, % etc.) for long int can no longer be applied to mpint. G$\mathbb{F}$L provides routines intSum, intDiff and so on to do arithmetic with mpint. For efficient implementation of these routines, we use a 25 bits per long packing (assuming that a long consists of 32 bits and a double has 52 bit precision). That is, each word of an mpint is a *digit* in radix $R = 2^{25} = 33554432$. Thus the integer

$$\begin{aligned} & 112233445566778899009988776655443322 11 \\ = \; & 8853657R^4 + 25051344R^3 + 6227312R^2 + 31737219R + 17261491 \end{aligned}$$

has the following representation as an mpint (n, say).

```
n.sign = '+';
n.size = 5;
n.word[0] = 17261491;
n.word[1] = 31737219;
n.word[2] = 6227312;
n.word[3] = 25051344;
n.word[4] = 8853657;
```

The negative of this integer has the same representation except that

```
n.sign = '–';
```

Finally the mpint n representing the special integer 0 (zero) has the following values for its struct components:

Representation of 0

```
n.sign = ' ';
n.size = 1;
n.word[0] = 0;
```

For efficient memory management, G$\mathbb{F}$L routines never *return* an mpint. An assignment is effected by passing to a routine a pointer to the mpint where we want to store the desired result. For example, the call

```
GFprod(&c, a, b, K);
```

stores in c the mpint obtained by multiplying mpints a and b over the field K.

2.2.2 Fields

G$\mathbb{F}$L maintains a *field descriptor* for every finite field created. This descriptor is of the data type GF_d (which is essentially a short int). All references to the fields created can be done through these descriptors. G$\mathbb{F}$L allows one to work with at most MAX_FIELDS field descriptors.[1]

A finite field of prime cardinality p is represented as an algebraic system where all arithmetic operations are integer operations modulo p. A non-prime field, on the other hand, cannot exist as a stand-alone field. It has to be defined as an algebraic extension of an existing field (which might be a prime field or another non–prime field that has been already defined). Each such algebraic extension is defined by an irreducible polynomial of given degree over the field being extended. Arithmetic in the extension field is carried out as polynomial arithmetic in the subfield modulo the defining irreducible polynomial. In other words, G$\mathbb{F}$L always uses the polynomial basis representations of field extensions.

[1]The header file field.h defines the macro MAX_FIELDS as 64.

There is no limit (other than MAX_FIELDS) on the length of the chain of field extensions $F_1 \subseteq F_2 \subseteq F_3 \subseteq \ldots$ that one can create using G$\mathbb{F}$L. In addition, it is possible to define extensions of F_i other than F_{i+1}, extensions of these extensions, and so on. It is also admissible to define more than one prime field. In short, G$\mathbb{F}$L allows one to work with an arbitrary *directed forest* of fields.

2.2.3 Field elements

Elements of a finite field $\mathbb{F}_q$ of cardinality q are represented as *integers* between 0 and $q-1$. For fields of prime cardinality, this is an obvious representation. For fields of prime power cardinality, this has the following interpretation. First let us assume that $\mathbb{F}_q$ is an extension of the prime field $\mathbb{F}_p$ defined by a polynomial $f(x) \in \mathbb{F}_p[x]$ of degree s (so that $q = p^s$). Let α be a root of $f(x)$ in $\mathbb{F}_q[x]$. Then an element $c \in \mathbb{F}_q$ can be uniquely represented as $c = c_{s-1}\alpha^{s-1} + c_{s-2}\alpha^{s-2} + \cdots + c_1\alpha + c_0$, where $c_{s-1}, c_{s-2}, \cdots, c_1, c_0 \in \mathbb{F}_p$. We may view c as an s-digit integer $c_{s-1}c_{s-2}\cdots c_1c_0$ in base p. Then c is an integer between 0 and $q-1$. Note that $1, \alpha, \alpha^2, \ldots, \alpha^{s-1}$ constitute the polynomial basis of $\mathbb{F}_q$ over $\mathbb{F}_p$. Viewed as integers, these basis elements are respectively $1, p, p^2, \ldots, p^{s-1}$.

Next let us extend $\mathbb{F}_q$ by $g(x) \in \mathbb{F}_q[x]$ of degree t to get the field $\mathbb{F}_{q^t} = \mathbb{F}_q[x]/\langle g \rangle$, where $\langle g \rangle$ represents the ideal in $\mathbb{F}_q[x]$ generated by $g(x)$. If β is a root of $g(x)$ in $\mathbb{F}_{q^t}$, then an element $c \in \mathbb{F}_{q^t}$ has the unique representation $c = c_{t-1}\beta^{t-1} + c_{t-2}\beta^{t-2} + \cdots + c_1\beta + c_0$, where $c_{t-1}, c_{t-2}, \cdots, c_1, c_0 \in \mathbb{F}_q$. We may, therefore, represent c as the t-digit integer $c_{t-1}c_{t-2}\cdots c_1c_0$ in base $q = p^s$. Each c_i, on the other hand, can be represented as an s-digit integer $c_{i,s-1}c_{i,s-2}\cdots c_{i,1}c_{i,0}$ in base p, so that

$$\begin{aligned} c \;=\; & (c_{t-1,s-1}\alpha^{s-1} + c_{t-1,s-2}\alpha^{s-2} + \cdots + c_{t-1,1}\alpha + c_{t-1,0})\beta^{t-1} \\ & + (c_{t-2,s-1}\alpha^{s-1} + c_{t-2,s-2}\alpha^{s-2} + \cdots + c_{t-2,1}\alpha + c_{t-2,0})\beta^{t-2} \\ & + \cdots \\ & + (c_{1,s-1}\alpha^{s-1} + c_{1,s-2}\alpha^{s-2} + \cdots + c_{1,1}\alpha + c_{1,0})\beta \\ & + (c_{0,s-1}\alpha^{s-1} + c_{0,s-2}\alpha^{s-2} + \cdots + c_{0,1}\alpha + c_{0,0}) \end{aligned}$$

Hence we may view c also as the st-digit integer $c_{t-1,s-1}c_{t-1,s-2}\cdots c_{t-1,1}c_{t-1,0}$ $c_{t-2,s-1}c_{t-2,s-2}\cdots c_{t-2,1}c_{t-1,0}\cdots c_{0,s-1}c_{0,s-2}\cdots c_{0,1}c_{0,0}$ in base p. Thus, c is an integer between 0 and $p^{st}-1$. Here $1, \beta, \beta^2, \ldots, \beta^{t-1}$ form the polynomial basis of $\mathbb{F}_{q^t}$ over $\mathbb{F}_q$. G$\mathbb{F}$L represents these basis elements as the integers $1, q, q^2, \ldots, q^{t-1}$. Note also that the elements

$$1, \alpha, \ldots, \alpha^{s-1}, \beta, \beta\alpha, \ldots, \beta\alpha^{s-1}, \ldots, \beta^{t-1}, \beta^{t-1}\alpha, \ldots, \beta^{t-1}\alpha^{s-1}$$

form a basis of $\mathbb{F}_{p^{st}}$ over $\mathbb{F}_p$. This is, in general, not a polynomial basis. We call it a *composed basis* of $\mathbb{F}_{p^{st}}$ over $\mathbb{F}_p$. G$\mathbb{F}$L represents these basis elements as the integers $1, p, p^2, \ldots, p^{st-1}$ respectively.

It is clear that this representation of finite field elements by integers can be similarly extended to extensions of $\mathbb{F}_{q^t}$, to extensions of these extensions, and so on. To sum up, for any q the elements of $\mathbb{F}_q$ are represented as integers between 0 and $q-1$ irrespective of the definition of $\mathbb{F}_q$. The interpretation of the integers is, however, dependent on the definition of $\mathbb{F}_q$. We call this representation of finite field elements the *packed representation* as contrasted with the *unpacked representation* in which elements are represented as tuples or polynomials. Before we proceed further, let us highlight the relative merits and demerits of the packed representation.

1. The packed representation needs less memory than its unpacked counterpart.

2. The packed representation is a uniform representation of a finite field element irrespective of the field to which the element belongs. This means that if an extension K of F is defined where $\mathrm{card}(F) = q$, an element $c \in K$ with $0 \leqslant c \leqslant q - 1$ is automatically an element of F and has the *same* interpretation in both K and F. For example, the integer 0 (resp. 1) represents the additive (resp. multiplicative) identity in any field. There is no overhead of typecasting elements of one field to those of another. This saves time.

3. Indexing arrays etc. by finite field elements or letting a loop variable run over finite field elements become easier with this representation of the elements as integers. This too speeds up computation.

4. Almost all arithmetic operations over finite fields require the individual elements of the unpacked representation. This means that for every such operation the operands should first be unpacked and after the operation the result should be packed and returned. This adds to the cost of arithmetic. This overhead is negligible during computation of products and powers, whereas for sums and differences, we cannot neglect the effect of packing and unpacking. G$\mathbb{F}$L is designed to keep this overhead at a bare minimum.

5. When we are working over fields of characteristic 2, the individual bits of the unpacked representation remain "visible" in the sense that packing and unpacking can be done using only bit operations which are very fast. In addition, the procedure that implements sum (and difference) over these fields need not separate the individual bits of the operands. An XOR operation on a full word processes all the bits in the word simultaneously.

G$\mathbb{F}$L defines the data type GFelement to represent elements of a finite field. As we have seen a GFelement should hold an integer value. Indeed, the multiprecision integer data type defined in the first subsection has been typedef-ed as GFelement.

data type GFelement

```
typedef mpint GFelement;
```

2.2.4 Polynomials

G$\mathbb{F}$L represents a polynomial as a structure of two elements. The first element is the *exact degree* of the polynomial and the second a pointer to the coefficient array. The coefficients are of type GFelement.

data type poly

```
typedef struct {            /* Data structure for polynomial */
    int degree;             /* The exact degree */
    GFelement *coeff;       /* Pointer to the array of coefficients */
} poly;
```

That is, if f is a variable of type poly and f.degree = d ($d \geqslant 0$), the coefficient of x^i of f can be accessed as f.coeff[i] for $0 \leqslant i \leqslant d$. In particular, f.coeff[f.degree] is the *leading coefficient* of f. The *zero polynomial* has the following representation

The zero polynomial

```
f.degree = MINUS_INFINITY; f.coeff = NULL;
```

As in the case of GFelements, the definition of a polynomial does not mention a field to which the polynomial is intended to belong. When one calls a routine that does arithmetic on polynomials, one must specify the field (GF_d) over which the coefficient arithmetic should take place.

2.2.5 Vectors and matrices

The following two data structures define the data types for matrices and vectors over finite fields. As with polynomials, the elements of a matrix or a vector are dynamically managed by pointers. Similarly, the definitions avoid committing to particular fields.

data type vector

```
typedef struct {                /* Data structure vector */
    int size;                   /* Vector size */
    GFelement *element;         /* Pointer to the array of vector elements */
} vector;
```

data type matrix

```
typedef struct {                /* Data structure matrix */
    int row;                    /* Number of rows */
    int col;                    /* Number of columns */
    GFelement **element;        /* Pointer to 2-dimensional array of elements */
} matrix;
```

A vector in this paradigm is neither a row vector nor a column vector. It is just an array of GFelement. It is up to the users how they would like to view it. In some cases one may use vector as an ordered list (tuple) or even as an unordered set. There are certain routines, however, where G𝔽L assumes specific structure on their vector arguments. Most notably, the linear equation solving routines assume that a vector is a column vector. At any rate, we encourage G𝔽L users to treat a vector as a *column vector*.

Note that G𝔽L uses dynamic arrays for representing various algebraic data (mpint, poly, vector and matrix and many other which we do not mention here). The advantage of this representation over the representation by static arrays is that in the former representation passing data to subroutines is much faster than that in the latter. This is because with dynamic arrays only a pointer to the coefficient array need to be passed instead of the entire array. Moreover, use of dynamic arrays leads to more efficient use of memory, since the pointers can be allocated only as much memory as is needed to hold the array. On the other hand, this representation makes programming a little difficult. One has to be careful while allocating and freeing memory associated with such data, in particular, inside one's own subroutines.

2.3 Programming paradigms

In the last section we have seen how we can represent various algebraic entities using G𝔽L. We now demonstrate how we can write programs that use these data structures. We explain the major steps that the programmer should follow while developing his/her own application programs using G𝔽L. We explain the programming process by means of two examples. Before we do so, we make a few general remarks about G𝔽L library calls.

1. Names of the G𝔽L built-in functions have been chosen carefully to make them self-explanatory. For example, we will later see that the G𝔽L routine findRandomIrrPoly finds a random irreducible polynomial of a given degree over a given field.

2. In all G𝔽L functions a uniform convention for sequencing input and output parameters has been adopted. G𝔽L routines often return scalar values like long or char. They never return structures with dynamic arrays.[2] For example, findRandomIrrPoly should compute a poly. Since this data structure contains a dynamic array, it is not returned by the routine. Instead one has to pass a pointer to a poly as the *first* argument to store the irreducible polynomial. In general, the pointers to the data that we need to compute are passed at the beginning of the argument list. Next come the operands followed by relevant field descriptors. Certain flags are sometimes passed at the end of the parameter list. Here is an example. The routine

   ```
   polyDiv(&q, &r, f, g, K);
   ```

   performs polynomial division of f by g over the field K. The quotient polynomial is stored in q and the remainder in r. If one is interested in only one of the output polynomials (say, the remainder), one is allowed to pass the NULL pointer as the other argument. Most other routines do not allow NULL pointers as arguments.

3. In many functions a choice of algorithms is made possible through an input argument. We will see an example later: the routine findRoot finds the roots of a polynomial. The algorithm that it selects is dependent on an input parameter. Certain values of the parameter allow G𝔽L to take the decision by itself.

4. Many G𝔽L routines need a source of random integers. These integers are obtained using the built-in random number generator provided in the C library. In most of the cases, the user is given an option to seed the random number generator. Special flags should be supplied to the routines to effect this. Typical choices of seeding are: do not seed, use current time as seed, or use the value of a specific pre-defined variable. We leave the choice to the programmer as to what is desirable: repeatability or randomness.

5. Every G𝔽L function does automatic memory management. That is, whenever a dynamic array is to be reallocated memory, the routine first frees the memory (if any) allocated to the dynamic array and then reassigns memory to the array. The user need not bother about it. But when one writes one's own subroutines, we encourage one to follow the same strategy. This practice allows G𝔽L programs to hold just the amount of memory they need for the computation, and thereby reduces garbage collection overheads of the operating system.

2.3.1 Example 1

We start with the following example: computation of the characteristic roots of a matrix with proper multiplicities. We develop the detailed program step by step.

[2]The initialization routines are an exception where NULL pointers are returned.

Include header files

One should first include G𝔽L header files to tell the C compiler about the new data structures and external procedure declarations defined in G𝔽L. One may choose only the individual G𝔽L header files that are needed for the particular program. In that case the user should know which data structures and functions are defined in which files. The G𝔽L reference manual [30] describes these in details. At any rate, the easier way to include the necessary files is to include all the files – both the ones we need in our program and the ones that we do not.

Include all header files

```
#include <stdio.h>          /* C standard io header file */
#include <GFL/all.h>        /* Include all GFL header files */
```

Initialize G𝔽L

This is a very important step. Any program that uses G𝔽L must do this before doing anything else. This step carries out certain book-keeping tasks and sets up some tables for later use. If the library is not initialized, one would get bizarre results like unwelcome halts, nasty segmentation faults and so on. Initialization of the G𝔽L kit is rather easy. One should just call

The initialization routine

```
GFLinitialize();
```

Declare variables

We first decide what data we need to represent. We then declare them using the user-defined data types introduced in the last section. In our example, we need a few field descriptors (of type GF_d) for referring to various fields, a matrix whose characteristic roots will be calculated, a poly to hold the characteristic polynomial of this matrix, a vector to store the roots of this polynomial, and some other auxiliary variables of type long.

Variable declarations

```
GF_d F, K, L;          /* The field descriptors */
poly f;                /* Polynomials */
matrix M;              /* Matrices */
vector v;              /* Vectors */
mpint p;               /* Integers */
long i, n;             /* Auxiliary variables */
```

Initialize variables

Any G𝔽L structure that has dynamic arrays must first be initialized before it can be used. In this example, the variables f, M, v, p contain dynamic arrays. These arrays are initialized to NULL as follows:

Variable initialization

```
f = newPoly(); M = newMatrix(); v = newVector(); p = newInt();
```

Alternatively one may explicitly make the pointers in these structures NULL as

Explicit initialization

```
f.coeff = NULL; M.element = NULL; v.element = NULL; p.word = NULL;
```

It is to be noted that this explicit initialization is a little bit more efficient than calling the initialization routines like newInt, newPoly etc. At any rate, this is not really an important issue for the user to ponder too much. Instead we emphasize that whenever a new variable with dynamic arrays is declared (as global variables, inside procedures or loops, or even inside another variable, e.g. in a structure containing polynomials or matrices etc.), it is mandatory to initialize the variable before anything is done with it. Many G$\mathbb{F}$L routines free the memory associated with dynamic arrays (unless they are NULL) before they are assigned new memory. An uninitialized non-NULL value can, therefore, lead to run-time hazards.

Create fields

Let's say that we want to compute the characteristic values of a matrix over $\mathbb{F}_{3^4}$ in the extension field $\mathbb{F}_{3^{12}}$. To this end, we first create the prime field of characteristic 3. Then we find a random irreducible polynomial of degree 4 over $\mathbb{F}_3$ and attach a root of this polynomial to $\mathbb{F}_3$ in order to get the extension field $\mathbb{F}_{3^4}$. In a similar fashion we extend $\mathbb{F}_{3^4}$ by an irreducible polynomial of degree 3 over $\mathbb{F}_{3^4}$. This gives us $\mathbb{F}_{3^{12}}$. All these can be done very simply by a few library calls.

Create fields

```
longToInt(&p, 3);              /* Characteristic of the fields */
F = createPrimeGF(p);          /* Create the prime field of characteristic 3 */
findRandomIrrPoly(&f, F, 4, 1);
                    /* Find a random irreducible polynomial of degree 4 over F */
K = createExtGF(F, f);         /* Extend F by f */
findRandomIrrPoly(&f, K, 3, 1);
                    /* Find a random irreducible polynomial of degree 3 over K */
L = createExtGF(K, f);         /* Extend K by f */
```

Each call to createPrimeGF or createExtGF returns a field descriptor (GF_d) that we shall use for all future references to the respective fields. The last argument of findRandomIrrPoly is a directive to the random irreducible polynomial generator routine on how to seed the random number generator – 0 means "don't seed", 1 means "use current time as seed" and 2 means "read the value of IRR_SEED_VAL as seed". The routine findRandomIrrPoly returns a value (of data type int) that we choose to ignore here. In fact findRandomIrrPoly generates random monic polynomials of the given degree one after another and checks them for irreducibility. As soon as it finds one irreducible polynomial, it returns the total number of polynomials checked before and including this irreducible polynomial. The irreducible polynomial is stored for future use in the poly pointed to by the first argument.

Do computations

We are now ready to carry out the actual computations. First we have to assign a matrix to the variable M. There are many ways in which this can be done. For the time being, we read it from stdin in an interactive fashion.

```
readMatrix(&M);
```

This asks the user for the numbers of rows and columns of M and then the elements in the row-major order. We intend to let M store a matrix over K, i.e., $\mathbb{F}_{3^4}$. As explained in the previous section, the elements of $\mathbb{F}_{3^4}$ are represented as integers between 0 and $3^4 - 1 = 80$ (both inclusive). So we must input a value in this range for every element of M. Since readMatrix does not know in advance how we are going to interpret these values, it does not complain if we supply element values not in the above range. The responsibility of entering meaningful values is, therefore, on the user.

In the next step we compute the characteristic polynomial of M. This is also easy. G$\mathbb{F}$L provides the built-in procedure charPoly to do this.

```
charPoly(&f, M, K);
```

We then find all the roots of f in the extension L. We call another built-in routine findRoot to do this.

```
n = findRoot(&v, f, L, 0);
```

The roots are stored as elements in the vector v. The number of roots can be found from v.size or the value returned by findRoot (assigned to n above). The last argument to findRoot tells findRoot which algorithm to use: (1) the exhaustive search algorithm or (2) the Berlekamp-Rabin algorithm [11] or (3) Berlekamp's trace algorithm [11]. Any other value passed as this argument (say, 0 as in our case) will allow findRoot to take the decision itself. The decision criterion goes like this: If the cardinality of L is less than SMALL_Q_BOUND[3], call the exhaustive search algorithm, else if the characteristic of L is 2, call Berlekamp's trace algorithm, else call the Berlekamp-Rabin algorithm. One may call these routines explicitly as well (findRootES, findRootBR and findRootBT). We won't go into further details of the syntax of these individual calls.

Now we have all the characteristic roots of M in the extension field L. What is left is to compute the multiplicities of these characteristic roots. One can do it in several ways. For example, instead of computing the roots of f, we can factorize f and read the multiplicities from those of the linear factors of f. Another possibility is to divide f successively by $x - a$ for each characteristic root a of f till a non-zero remainder is found. We follow the second approach, because it illustrates how one can do arithmetic on polynomials. We need three auxiliary polynomials g, rem and quot.

[3]SMALL_Q_BOUND is a macro defined in root.h, that has the default value 100.

```
poly g, rem, quot;
mpint pmo;

/* Initialize new variables */
g.coeff = rem.coeff = quot.coeff = NULL; pmo.word = NULL;

/* set pmo to $p - 1$ */
copyInt(&pmo, p); intMM(&pmo);

/* set g to be a polynomial of degree 1 */
g.degree = 1;
g.coeff = (GFelement *)malloc(2 * sizeof(GFelement));
g.coeff[0].word = g.coeff[1].word = NULL;
longToInt(&g.coeff[1], 1);

/* loop for each characteristic root of f */
for (i=0; i<n; i++) {
    int mul;

    /* Set the constant term of g to –a, where a is the ith characteristic root of f */
    /* Note that $-a = (-1)a$, and $-1$ has the representation $p - 1$ in F, K and L */

    GFprod(&g.coeff[0], v.element[i], pmo, L);
    /* GFprod returns in the first argument the product of its second and
       third arguments considered as elements of the field supplied as the
       fourth argument */

    mul = 0;      /* Initialize multiplicity to 0 */

    do {
        polyDiv(&quot, &rem, f, g, L);
        /* Divide f by g and store the quotient in quot and remainder in rem */
        /* L is the field where the coefficient arithmetic takes place */

        if (zeroPoly(rem)) {
            /* If the remainder is zero, that is,
                if (rem.degree == MINUS_INFINITY) */
            mul++;      /* Increase multiplicity by 1 */
            copyPoly(&f, quot);      /* Store the quotient in f */
        }

    } while (zeroPoly(rem));

    printf("Multiplicity of "); writeInt(v.element[i], stdout); printf(" is %d\n", mul);
}
```

The above example clearly illustrates how easy it is to do the desired task by simple library calls. Before we go to the next topic, we mention that an assignment of the form

```
f = quot;
```

should always be avoided, because such an assignment does not make a verbatim copy of the coefficient array of quot to that of f. Instead, it copies the coeff pointer of quot to that of f. This means that after the execution of the statement, the coeff pointers of both quot and f point to the same memory location, so that if quot coefficients are changed (or if quot.coeff is free'd), that change will be reflected in f also (and vice versa). This is undesirable. Moreover, such assignments might lead to fatal run-time errors. We ask the users to use the copying routines (copyPoly for polynomials, copyMatrix for matrices, and so on) in such cases.

Wind up

Now that our program has printed the characteristic roots of the input matrix together with their multiplicities, we may choose to exit from the program. A better approach is first to free the memory allocated to the coeff arrays of poly, the element arrays of matrix and vector and so on. These can be done by explicit calls to free. Alternatively, the following routines can be used.

Free memory

```
destroyPoly(&f);   destroyPoly(&g);   destroyPoly(&quot);   destroyPoly(&rem);
destroyMatrix(&M);   destroyVector(&v);   destroyInt(&p);   destroyInt(&pmo);
```

In general, it is always a good practice to free the memory allocated to dynamic arrays whenever the contents of the memory are no longer needed. This need not be done only at the very end of the programs. In our example, we might call destroyMatrix(&M) immediately after the characteristic polynomial of M is calculated. We don't need the elements of M after this step.

2.3.2 Example 2

In the second example, we illustrate how the user can write his/her own subroutines using G$\mathbb{F}$L library calls. We develop a finite field cryptosystem proposed by ElGamal [32]. We implement three basic subroutines for the cryptosystem using G$\mathbb{F}$L calls. Suppose that A (Alice) wants to send a message to B (Bob) over a channel where a third party C (Carol) may intercept A's messages and read the secret information meant for B only. To befool C, A and B choose a large finite field K ($\mathbb{F}_{2^{401}}$ or $\mathbb{F}_{2^{503}}$, for example) and compute a primitive element g in that field.[4] B then selects a random integer b and computes $l = g^b$. B publishes the public key l and keeps the private key b a secret. This process of key generation can be implemented as follows.

Generate key

```
void makekey ( GFelement *b , GFelement *l , GFelement g , GF_d K ) {
    mpint qmo;

    qmo = newInt();            /* Initialize qmo */
    cardinality(&qmo, K);      /* Store in qmo the cardinality of K */
    intMM(&qmo);               /* qmo – – */
    randRes(b, qmo);           /* Store at *b a random non-negative integer less
                                                                    than qmo */
    GFexp(l, g, *b, K);        /* Calculate l = g^b in K */
    destroyInt(&qmo);          /* Return to system the memory held by qmo */
}
```

[4]A random primitive element in a finite field can be obtained by the call

```
findPrimElement(&g, K, 0);
```

where the last argument is similar to that in findRandomIrrPoly, i.e. it tells findPrimElement how to seed the random number generator.

The call of randRes in the above procedure sets b to a random nonnegative integer less than the cardinality of the field K minus 1. At the next step the power g^b is computed and saved in *l by the call GFexp. Note that in an actual implementation, one should discard trivial values of b, say 0 or 1. We do not indicate this step here for the sake of simplicity. Note also how the temporary mpint qmo is initialized, used and then destroyed in the routine. After B calls this subroutine as

```
makekey(&b, &l, g, K);
```

the mpint b holds B's private key and the mpint l holds his public key.

Now A wants to send an element $m \in K$ to B. She carries out the following three steps to encrypt her message.

1. choose a random integer t and computes $e_1 = g^t$, where $0 < t < q - 1$, q = the cardinality of K.
2. read B's public key (l) and computes $e_2 = ml^t$.
3. send the pair (e_1, e_2) to B.

A can use the following subroutine for the purpose of encryption.

Encryption routine

```
void encrypt ( GFelement *e1 , GFelement *e2 , GFelement g , GFelement m ,
               GFelement l , GF_d K ) {
    GFelement t, qmo;

    t = newInt();   qmo = newInt();               /* Initialize mpint */
    cardinality(&qmo, K);   intMM(&qmo);           /* Set qmo to q – 1 */
    do   randRes(&t, qmo);   while (zeroInt(t));
                     /* Generate a random integer between $1$ and $q-2$ */
    GFexp(e1, g, t, K);         /* First part $e_1 (= g^t)$ of encrypted message */
    GFexp(e2, l, t, K);         /* Compute $l^t$ */
    GFprod(e2, *e2, m, K);      /* Second part $e_2 (= ml^t)$ of encrypted message */
    destroyInt(&t);             /* Free memory */
    destroyInt(&qmo);           /* Free memory */
}
```

A now sends e_1 and e_2 to B. B recovers the message m from these values using his private key b in the following way. We have $e_1 = g^t$ and $e_2 = ml^t = mg^{bt}$, so that $m = e_2 \cdot g^{-bt} = e_2 \cdot e_1^{-b}$. The subroutine for decryption is, thus:

Decryption routine

```
void decrypt ( GFelement *m , GFelement e1 , GFelement e2 , GFelement b ,
               GF_d K ) {
    GFexp(m, e1, b, K);          /* Compute $e_1^b$ */
    GFinv(m, e1, K);             /* Compute $(e_1^b)^{-1} = e_1^{-b}$ */
    GFprod(m, *m, e2, K);        /* Compute $e_2 \cdot e_1^{-b}$ */
}
```

Since $e_1^{q-1} = 1$ in K if $e_1 \neq 0$, we can calculate e_1^{-b} also as $e_1^{-b} = e_1^{q-1-b}$. We leave out the details of the implementation.

Now let's talk about C. She has knowledge of K, g and l, but not of b. Suppose C gets e_1 and e_2 and wants to decipher the original message m. She must first calculate b which is the discrete logarithm of l in K with respect to g. If she goes through the G$\mathbb{F}$L manual carefully, she will be happy to see that G$\mathbb{F}$L provides built-in routines for the computation of discrete logs in finite fields. Similar to the above procedures, it is very easy to write a routine for Carol, that uses these G$\mathbb{F}$L library calls. But the bad news for Carol is that this algorithm does not run in time polynomially bounded by the size of K. Indeed it is *sub-exponential*, so that if K is sufficiently large (say, $\mathbb{F}_{2^{503}}$), C cannot compute discrete logarithms in K in feasible time. But C must not blame the designers of G$\mathbb{F}$L – at present no better algorithms are known for computing discrete logs in finite fields. And this is why ElGamal's scheme of encryption is secure.

2.4 Functions provided

In this section we briefly describe what functions are provided by G$\mathbb{F}$L. We also mention the algorithms implemented for these functions. The details of the syntaxes of the library calls will be listed in Appendix A.

2.4.1 Integer functions

We have demonstrated how G$\mathbb{F}$L represents signed arbitrary-precision integers. We also stated that there are built-in procedures to do arithmetic with these multi-precision integers. Multiplying or dividing a multi-precision integer by another can be coded very efficiently when the second operand is a power of 2. Special G$\mathbb{F}$L routines take care of these situations. These routines find extensive use for fields of characteristic 2. In addition, these functions can be used for doing left and right shift operations on multi-precision integers. Separate (and a tiny bit more efficient) routines are also provided for (in-place) shift operations.

Other integer functions include checking and generating prime numbers, computing integer factorization, integer square root, integer gcd, modular exponentiation etc.

Here we mention the multiplication algorithm we have used. We have stated previously that our 25 bits/long packing of multi-precision integers is motivated by efficiency considerations. We now explain how this helps us write the multiplication routine very efficiently. Let us assume that we are working with long of size 32 bits and double of size 64 bits. We also assume that the data type double has 52 bit precision for the mantissa. These are the default values on a wide range of (small) work-stations available nowadays. For large machines (with say 64 bit long) our strategy has to be modified.

We have seen that each long in the word array of an mpint stores (at most) 25 bits of a multi-precision integer. For the multiplication routine one needs to compute the word-by-word product of these long values. The result can be at most 50 bits long and hence does not fit in a single long. Routines at the assembly-language level can take care of the carry. Our implementation does not use this strategy, because the assembler macros are very much machine-dependent. So the best strategy is to use double multiplication.[5] Since our double has 52 bit precision, multiplying two 25-bit long values (after typecasting to double) does not lead to an overflow in the mantissa. Note that with our assumption of double, we

[5]We have also tried using the data structure long long, which is 64-bit long, but double multiplication seems faster and leads to more efficient codes.

could have opted for 26 bits/long packing. But as we will see now, such a packing renders our multiplication routine erroneous.

Let's say that we want to multiply $a = a_{m-1}R^{m-1} + \ldots + a_1R + a_0$ with $b = b_{n-1}R^{n-1} + \ldots + b_1R + b_0$ (where $R = 2^{25}$ is the radix). The result is to be stored in $c = c_{r-1}R^{r-1} + \ldots + c_1R + c_0$, where $r = m + n$ or $m + n - 1$. In the following code snippet, we denote R by RADIX, $R^2 = 2^{50}$ by RADIX_SQR and $R^{-1} = 2^{-25}$ as RADIX_INV. Note that our double can store each of these three quantities with full precision. For simplicity, we denote a_i by a[i] (rather than by a.word[i]). We assume that a and b are arrays of long, whereas c is an array of double. The variable carry is of data type long.

GF*L's multiplication*

```
/* Initialize c to 0 */
for (i=0; i<m+n; i++) c[i] = 0;

/* Multiplication loop */
for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
        c[i+j] += (double)(a[i]) * (double)(b[j]);
        if (c[i+j] >= RADIX_SQR) {
            c[i+j] −= RADIX_SQR;
            c[i+j+2] += 1;
        }
    }
}

/* Normalize the intermediate result */
carry = 0;
for (i=0; i<m+n; i++) {
    c[i] += (double)(carry);
    carry = (long)(c[i] * RADIX_INV);
    c[i] −= (double)(carry) * (double)RADIX;
}
```

In the multiplication loop of the above code, each c_{i+j} is kept at a value $\leqslant R^2$. With this trick, each word-by-word multiplication is associated with a double multiplication, a double addition and possibly one more double addition and one more double subtraction. We note that if we had 26 bits/long packing, the instruction

c[i+j] += (double)(a[i]) * (double)(b[j]);

might lead to overflow in the 52-bit mantissa of c[i+j]. On the other hand, our 25 bits/long packing leads to no such situation.

Now let's discuss the usual method of multiplication, where one keeps c_{i+j} normalized at values $\leqslant R$. In that case, each word-by-word multiplication would require one long and two double multiplications and several additions and subtractions as shown in the next code snippet. Here all variables used are of data type long (including the array elements c[i]). In the code, we make the assumption that when the product of two long values exceeds the range of long, the carry is neglected and the output long holds the lowest 32 bits of the product. This behavior is true in most modern machines, but there may be exceptions. The operator & used in the code stands for the bitwise 'AND' operation.

It is clear how G𝔽L's strategy speeds up the multiplication loop. But the G𝔽L routine has the additional overhead of normalizing the elements of the array c to

Usual multiplication

```
/* Multiplication loop */
for (i=0; i<m; i++) {
    for (j=0; j<n; j++) {
        low = (a[i] * b[j]) & (RADIX − 1);
        high = (long)(0.25 + RADIX_INV * ((double)(a[i]) * (double)(b[j])
                                                    − (double)(low)));
        c[i+j] += low;
        if (c[i+j] >= RADIX) {
            c[i+j] &= (RADIX − 1);
            c[i+j+1]++;
        }
        c[i+j+1] += high;
        if (c[i+j+1] >= RADIX) {
            c[i+j+1] &= (RADIX − 1);
            c[i+j+2]++;
        }
    }
}
```

the digits in radix R, after the word-wise multiplications are done. This, however, can be neglected, since this normalization process takes time $O(m + n)$ which is smaller (both theoretically and practically) than the $O(mn)$ time taken by the multiplication loop.

G$\mathbb{F}$L's multiplication routine as presented so far can be further optimized. For example, the typecasting (to double) of a[i] and b[j] can be done outside the loop. This saves some time. In addition, we can use three multiplications for computing the four products of a_i and a_{i+1} with b_j and b_{j+1} for even i and j. This can be done using Karatsuba's strategy by computing $a_{i+1}b_{j+1}$, a_ib_j and $a_ib_{j+1} + a_{i+1}b_j = a_{i+1}b_{j+1} + a_ib_j - (a_{i+1} - a_i)(b_{j+1} - b_j)$. The detailed code is shown in the next page, where it is assumed that m and n are even.

We finally note that G$\mathbb{F}$L's multiplication routine with the Karatsuba improvement can be applied *mutatis mutandis* to squaring. The only difference is that for squaring the variable mid can be calculated more efficiently as:

```
mid = (double)(a[i] << 1) * (double)(a[j]);
```

Here << denotes left shift. Note that in the case of squaring $b = a$. The actual implementation of the squaring routine in G$\mathbb{F}$L uses further optimizations. We leave out the details here.

In order to see that our multiplication algorithm is efficient, we here mention the timings of the multiplication of a 2000-bit integer with a 1000-bit one using our routine and using A. K. Lenstra's long integer package LIP [76] (Version 1.1). On a 200 MHz Pentium processor running Linux, our routine takes about 500 μs for the above product, whereas LIP takes about 650 μs. Note that LIP uses Karatsuba multiplication (on the entire integer) which is known to be faster than the quadratic algorithm described above (at least theoretically). We get faster results with the quadratic algorithm at a size of the order of 1000 bits.

For multi-precision division we have implemented the algorithm described in Knuth's book [69, Section 4.3]. G$\mathbb{F}$L provides routines for both ordinary gcd (i.e. gcd by successive division) and binary gcd. It has been observed that for integers of length around 1000 bits, the binary gcd is faster by a factor of around

Karatsuba improvement

```
/* Multiplication loop */
for (i=0; i<m; i+=2) {
    for (j=0; j<n; j+=2) {
        high = (double)(a[i+1]) * (double)(b[j+1]);
        low = (double)(a[i]) * (double)(b[j]);
        mid = high + low − (double)(a[i+1] − a[i]) * (double)(b[j+1] − b[j]);
        c[i+j] += low;
        if (c[i+j] >= RADIX_SQR) {
            c[i+j] −= RADIX_SQR;
            c[i+j+2] += 1;
        }
        if (mid >= RADIX_SQR) {
            mid −= RADIX_SQR;
            c[i+j+3] += 1;
        }
        c[i+j+1] += mid;
        if (c[i+j+1] >= RADIX_SQR) {
            c[i+j+1] −= RADIX_SQR;
            c[i+j+3] += 1;
        }
        c[i+j+2] += high;
        if (c[i+j+2] >= RADIX_SQR) {
            c[i+j+2] −= RADIX_SQR;
            c[i+j+4] += 1;
        }
    }
}
```

6 compared to the other variant. Routines for extended gcd calculations are also provided. For integer square root we have used the algorithm described in Cohen's book [25, Chapter 1]. Miller-Rabin's improved test is implemented for primality testing and the quadratic sieve algorithm is used for integer factorization.

Since we have mentioned A. K. Lenstra's long integer package in connection with the multiplication algorithm, it is worthwhile to compare the performance of $\mathrm{G}\mathbb{F}\mathrm{L}$ routines with that of the LIP routines for other integer operations too. Table 2.1 provides the relevant details. We used the same operands (a random 2000 bit integer and a random 1000 bit integer) as discussed before. In the table 'Add', 'Sub', 'Mul', 'Sqr', 'Div', 'Lsh', 'Rsh' and 'GCD' respectively stand for addition, subtraction, multiplication, squaring, division (with remainder), left shift (by 1000 bits), right shift (by 1000 bits) and binary gcd. The operations Sqr, Lsh and Rsh are applied on the 2000 bit integer.

We note that though our multiplication is faster than that in LIP, all other routines are slower in $\mathrm{G}\mathbb{F}\mathrm{L}$ compared to LIP. This is partly because LIP uses a 30 bits/long packing, whereas $\mathrm{G}\mathbb{F}\mathrm{L}$ packs at 25 bits/long. We are unable to apply the strategy we used for 'Mul' to the other routines (except 'Sqr'). This accounts for a slowing down factor of $30/25 = 1.2$ for additive routines and of $(30/25)^2 = 1.44$ for multiplicative routines.

2.4.2 Field functions

We have seen examples of creating and representing finite fields of arbitrarily large

Table 2.1: Timings (in μs) of basic multi-precision integer operations

Operation	Add	Sub	Mul	Sqr*	Div	Lsh*	Rsh*	GCD
G$\mathbb{F}$L	18	23	490	690	815	18	11	5,004
LIP	7.5	6.7	645	590	440	12	6.8	3,775

* for the 2000 bit integer

cardinalities. We have also seen examples of arithmetic routines for computing sum, difference, product, inverse, power etc. in finite fields. At present, Karatsuba or FFT-based techniques are not employed for field multiplication. The use of logarithm tables to speed up field arithmetic is implemented and is discussed in detail in Section 2.4.5.

In addition to the field arithmetic, G$\mathbb{F}$L provides routines for the following operations on field elements.

1. *Computing traces and norms:* The repeated doubling algorithms proposed by von zur Gathen and Shoup [43] have been implemented.

2. *Computing and checking normal elements:* In order to check if $\alpha \in \mathbb{F}_{q^s}$ generates a normal basis over $\mathbb{F}_q$, we compute the gcd of the polynomials $\alpha^{q^{s-1}}x^{s-1} + \ldots + \alpha^q x + \alpha$ and $x^s - 1$. The element α is normal over $\mathbb{F}_q$ if and only if the above gcd is 1 (See [85, Theorem 4.5]). In order to construct (random) normal elements in $\mathbb{F}_{q^s}$ over $\mathbb{F}_q$, three algorithms have been implemented. The first algorithm generates $\alpha \in \mathbb{F}_{q^s}$ randomly and checks if α is normal over $\mathbb{F}_q$. The second algorithm is also a randomized one and is based on a lemma due to Artin [85, Theorem 4.23]. This algorithm is effective, if $q > 2s(s-1)$. The last algorithm implemented is Bach, Driscoll and Shallit's deterministic factor refinement algorithm [6].

3. *Computing and checking primitive elements:* In order to check if an element $\alpha \in \mathbb{F}_q^*$ is primitive, we use factorization of the integer $q-1$. If $q-1$ is prime, then α is primitive. Otherwise, let $q - 1 = p_1^{\beta_1} \ldots p_r^{\beta_r}$ be the prime factorization of $q-1$. Then α is primitive, if and only if $\alpha^{\frac{q-1}{p_i}} \neq 1$ for all $i = 1, \ldots, r$. In order to find a primitive element of $\mathbb{F}_q^*$, we generate elements of $\mathbb{F}_q^*$ randomly and check them for primitivity.

4. *Computing transformation matrices between different bases of finite fields:* Let $\alpha_0, \ldots, \alpha_{s-1}$ constitute a basis of $\mathbb{F}_{q^s}$ over $\mathbb{F}_q$. We first express the α_i in the polynomial (or composed) basis of $\mathbb{F}_{q^s}$ over $\mathbb{F}_q$. We then use linear algebra techniques to compute the transformation matrix.

5. *Computing isomorphism between two fields of the same cardinality (and of different representation):* Let K_1 and K_2 be two representations of the finite field of cardinality p^s. In order to compute the matrix that transforms coordinates of an element of $\alpha \in K_1$ in the composed basis of $\mathbb{F}_{p^s}$ over $\mathbb{F}_p$ to those of an isomorphic image of α in K_2, we find out a polynomial basis of K_1 over $\mathbb{F}_p$ and then compute the transformation matrices between this polynomial basis and the composed bases of K_1 and K_2 over $\mathbb{F}_p$.

6. *Computing discrete logarithms with respect to primitive elements:* Currently only the basic index calculus method [85] has been fully implemented. For

prime fields, heuristic B1 (discussed in Chapter 4) is employed. For fields $\mathbb{F}_{2^s}$, the improvement due to Blake et. al. [14] has been incorporated. The linear and cubic sieve methods [28] for prime fields have been partially implemented.

2.4.3 Polynomial functions

G$\mathbb{F}$L's polynomial arithmetic is based on the standard high-school techniques. Karatsuba and FFT-based techniques are not yet incorporated. Apart from standard arithmetic functions, G$\mathbb{F}$L provides many utilities for univariate polynomials over finite fields. These include

1. *Computing minimal polynomials:* We compute the minimal polynomial of $\alpha \in \mathbb{F}_{q^s}$ over $\mathbb{F}_q$ as $(x-\alpha)(x-\alpha^q)\dots(x-\alpha^{q^{d-1}})$, where d is the least positive integer for which $\alpha^{q^d} = \alpha$. (Note that $d|s$.)

2. *Computing irreducible polynomials and checking polynomials for irreducibility:* The check of [85, Theorem 3.28] is used for testing irreducibility of polynomials. In order to compute random irreducible polynomials over a finite field $\mathbb{F}_q$, we generate monic polynomials with coefficients randomly chosen from $\mathbb{F}_q$ and check if these polynomials are irreducible over $\mathbb{F}_q$.

3. *Computing characteristic polynomials of matrices, companion matrices of polynomials, resultants and discriminants of polynomials:* Standard techniques from linear algebra are used. (See, for example, [25, Chapters 2,3].)

4. *Factorizing polynomials:* The well-known route comprising of square-free factorization, distinct-degree factorization, equal-degree factorization in succession is followed as in [43]. However, the latest development in this area, namely Kaltofen and Shoup's range decomposition strategy [66], has not been implemented.

5. *Finding roots of polynomials:* Three algorithms are implemented for finding roots of a polynomial $f(x)$ over a finite field $\mathbb{F}_q$. The exhaustive search algorithm computes $f(\alpha)$ for each $\alpha \in \mathbb{F}_q$ and returns those values of α for which $f(\alpha) = 0$. This is a reasonable strategy, if q is *small*. The Berlekamp-Rabin algorithm (applicable for fields of odd characteristics only) and Berlekamp's trace algorithm are two powerful randomized algorithms [11] for root finding and have been incorporated in G$\mathbb{F}$L.

6. *Checking permutation polynomials:* The von zur Gathen test for permutation polynomials [36, 38] is applied.

7. *Computing affine multiples of polynomials:* The coefficients of the least affine multiple of a polynomial are calculated by solving a homogeneous over-specified system of linear equations [85, Section 2.9].

2.4.4 Linear algebra functions

G$\mathbb{F}$L provides all basic arithmetic routines on matrices and vectors over finite fields. In addition it provides routines for computing ranks, determinants and

LUP decompositions of square matrices. Routines for solving systems of linear equations (including cases of over- and under-specified systems) are also available. Well-known algorithms of linear algebra have been implemented. Fast matrix multiplication techniques are not used. Special routines for solving sparse linear systems [74] have also not been implemented yet.

2.4.5 Speeding up field arithmetic

Primitive elements are very useful for speeding up arithmetic in finite fields. To see how let's assume that the fields $\mathbb{F}_p = F \subseteq K = \mathbb{F}_{p^s}$ are defined. Let g be a primitive element of K. For arbitrary elements $a, b \in K, a, b \neq 0$, let u and v be the discrete logarithms of a and b respectively with respect to g, that is $a = g^u$ and $b = g^v$. Then u and v are integers unique modulo $p^s - 1$. Then the product $a \cdot b \in K$ is g^w where $w \equiv u + v \pmod{p^s - 1}$. Now let's assume that $v > u$. Then $a + b = g^u(1 + g^{v-u})$. If we know the discrete logarithm of $1 + g^{v-u} = g^{v'}$ (say), then we can calculate $a + b = g^{w'}$ where $w' \equiv u + v' \pmod{p^s - 1}$.

In general, it is computationally very difficult to find discrete logarithms in a finite field. Though G$\mathbb{F}$L provides routines for computing discrete logarithms in finite fields, use of these routines does not speed up finite field arithmetic. Instead G$\mathbb{F}$L provides facilities for creating and using tables of powers of a primitive element and discrete logs with respect to the same primitive element. These tables are used for computing products and inverses in K. If one wants to use primitive elements to accelerate sums (and differences) too, one needs another table called *Zech's logarithm table* [59] that stores for each u the discrete log of $1 + g^u$ (with respect to g, of course).

Each of these three tables must reside in the main memory and therefore their sizes are limited by the amount of RAM provided by the system. With current-day technology it is possible even with small personal computers to store tables for fields as large as 2^{16}. For a larger field that has a subfield of *small* cardinality, we recommend the following strategy. Suppose we want to work over $\mathbb{F}_{3^{100}}$. We create the tower of field extensions $F \subseteq K \subseteq L$, where $F = \mathbb{F}_3$, $K = \mathbb{F}_{3^{10}}$ and $L = \mathbb{F}_{3^{100}}$. We then create primitive power, discrete log and Zech logarithm tables for the intermediate field $K = \mathbb{F}_{3^{10}}$. This speeds up arithmetic considerably in both K and L.

2.4.6 Fields of small characteristics

It is observed that the built-in arithmetic routines of C for single-precision integers are much faster than the multi-precision routines for the same integers. Therefore, when the characteristic of the field (over which we are working) is small, it is preferable to use the single-precision routines to the multi-precision ones. This may lead to speed-ups of the order of as high as 10. Special G$\mathbb{F}$L routines have been written to exploit this phenomenon. In addition to this, G$\mathbb{F}$L provides routines for certain field operations, that work nicely for $\mathbb{F}_{p^s}$ (p odd), if $s(p-1)^2$ is less than the largest positive integer representable by a signed long. If, on the other hand, $s(p-1)^2$ is larger than this value, the routines might cause long overflow. Hence these routines are termed *unsafe*. When unsafe routines apply, they are reasonably faster than their *safe* counterparts. The user should turn on a flag in order to invoke the unsafe routines. Otherwise the safe routines are called by default.

Fields of characteristic 2 deserve specific mention in this subsection. These fields are probably the ones that are most useful in practice. Almost all basic operations on elements of fields of characteristic 2 can be performed using bit operations only, whereas those for fields of odd characteristic require integer arithmetic. Since bit operations are much faster than integer arithmetic operations, G$\mathbb{F}$L uses special routines for fields of characteristic 2.

In spite of the existence of different sets of routines for fields of different characteristics, the user need not bother about them and call the appropriate routines explicitly. The G$\mathbb{F}$L routines at the outermost level make suitable branchings depending on the characteristic of the underlying field.

2.5 Performance measure

In this section we tabulate the timings for basic integer, field and polynomial operations as achieved by G$\mathbb{F}$L routines. We obtained these figures on a 200 MHz Pentium machine running Linux version 2.0.34. GNU's C compiler version 2.7 was used. We use the tricks of speeding up finite field arithmetic by maintaining multiplication and Zech tables whenever possible.

2.5.1 Multi-precision integer arithmetic

In Table 2.1 we have listed typical timing figures for arithmetic operations on two multi-precision integer operands. The first one is a random 2000 bit integer and the second one a random 1000 bit integer. Squaring and shift operations are applied on the 2000 bit integer.

2.5.2 Field arithmetic

In Table 2.2 we give timings for operations in the fields: $\mathbb{F}_{2^{400}}$, $\mathbb{F}_{2^{401}}$, $\mathbb{F}_{3^{250}}$ and $\mathbb{F}_{3^{251}}$. We maintain multiplication and Zech's logarithm tables for $\mathbb{F}_{2^{16}}$ (a subfield of $\mathbb{F}_{2^{400}}$) and $\mathbb{F}_{3^{10}}$ (a subfield of $\mathbb{F}_{3^{250}}$). This strategy does not work for the other two fields, since 401 and 251 are primes. We also show timing results for $\mathbb{F}_{2^{400}+181}$ which is a prime field. Note that the cardinalities of these five fields are of nearly equal sizes (nearly 400 bits or 120 digits).

Table 2.2: Times (in μs) for basic field operations

Operation	**Field**				
	$\mathbb{F}_{2^{400}}$	$\mathbb{F}_{2^{401}}$	$\mathbb{F}_{3^{250}}$	$\mathbb{F}_{3^{251}}$	$\mathbb{F}_{2^{400}+181}$
Addition	5.0	4.7	803	7,710	9.6
Subtraction	5.2	5.0	805	7,720	13.8
Multiplication	880	960	2,610	19,700	218
Inverse	900	1,400	2,520	26,000	2,620

It is evident that the routines for fields of characteristic 2 are two to three orders of magnitude more efficient than those of odd characteristics. This table also illustrates the speedup due to multiplication tables and Zech's logarithm table. For characteristic 2 fields, use of multiplication tables speed up multiplication and inverse by a small factor. This behavior tallies closely with the observations reported in [130]. For fields of odd characteristic, the speedup obtained using lookup tables is of the order of 10. For the field $\mathbb{F}_{3^{251}}$, we have used the unsafe mode of multiplication, since $251 \cdot (3-1)^2 = 1004$ is much less than the capacity of a long. With the safe multiplication, computing product and inverse in this field takes time 37,700 μs and 38,300 μs respectively. These figures clearly illustrate the benefit of using unsafe multiplication mode when it applies.

2.5.3 Polynomial arithmetic

We choose two random polynomials of degrees 200 and 100 respectively over each of the five fields of Table 2.2. We tabulate in Table 2.3 the time taken by $\mathrm{G}\mathbb{F}\mathrm{L}$ routines for doing arithmetic on these two polynomials. We maintained multiplication and Zech's tables for the fields $\mathbb{F}_{2^{16}}$ and $\mathbb{F}_{3^{10}}$ to accelerate computations in $\mathbb{F}_{2^{400}}$ and $\mathbb{F}_{3^{250}}$ respectively.

Table 2.3: Times (in seconds) for basic polynomial operations

Operation	**Field**				
	$\mathbb{F}_{2^{400}}$	$\mathbb{F}_{2^{401}}$	$\mathbb{F}_{3^{250}}$	$\mathbb{F}_{3^{251}}$	$\mathbb{F}_{2^{400+181}}$
Addition	0.0015	0.0014	0.087	0.774	0.0020
Subtraction	0.0015	0.0014	0.085	0.772	0.0022
Multiplication	6.79	20.0	41.12	243.9	4.49
Division	3.59	10.1	20.93	124.3	2.39

For polynomial arithmetic we see similar patterns in the timings as we described in connection with the field operations. We point out an important observation here for characteristic 2 fields. Though the field multiplication (and inverse) become nominally faster with multiplication tables, the polynomial multiplication and division routines run about 3 times faster when these tables are maintained. For polynomials over $\mathbb{F}_{3^{251}}$, we once again used the unsafe multiplication mode. In the safe mode, the above multiplication and division take time 594.3 and 302.3 seconds respectively.

2.5.4 Comparison with other libraries

Here we list a set of symbolic computation libraries other than $\mathrm{G}\mathbb{F}\mathrm{L}$, that support computation over finite fields.

LiDIA A C++ library for Computational Number Theory, The LiDIA Group, TU Darmstadt [13]

NTL A C++ Library for doing Number Theory, V. Shoup [113]

ZEN A toolbox for computations in finite extensions of finite rings, F. Chabaud and R. Lercier [23]

(MAGMA [15] and SIMATH [132] are two computer algebra *systems* that provide routines for finite field computations.)

In this subsection, we compare the timings of the $\mathrm{G}\mathbb{F}\mathrm{L}$ routines for field arithmetic with the analogous routines provided by the above libraries. The timings are listed in Table 2.4. We used same compilers (gcc and g++) for building the libraries and for compiling the test programs on the same machine (a 200 MHz

Table 2.4: Comparison of timings of basic field operations in various symbolic computation packages

Field	**Operation**	**Time (in μs)**			
		GFL	**LiDIA**	**NTL**	**ZEN**
$\mathbb{F}_{2^{400}+181}$	Addition	9.6	4	3.2	1.9
	Subtraction	13.8	5	3.6	2.5
	Multiplication	218	68	101	57
	Inverse	2,620	17,510	820	378
$\mathbb{F}_{2^{401}}$	Addition	4.7	1.3	1.1	1.6
	Subtraction	5.0	1.4	1.2	1.7
	Multiplication	960	1,240	230	434
	Inverse	1,400	13,840	960	5,740
$\mathbb{F}_{3^{251}}$	Addition	7,710	447	37	56
	Subtraction	7,720	385	35	33
	Multiplication	19,700	18,060	8,920	46,920
	Inverse	26,000	214,800	48,000	82,000

Pentium-based Linux PC mentioned before). As a result, these timing data are directly comparable (at least for the Pentium architecture). We used Version 1.3.1 of LiDIA, Version 3.0e of NTL and Version 1.0b of ZEN. $\mathrm{G}\mathbb{F}\mathrm{L}$ is yet to be released publicly (and thus get a version number).

Each library uses the polynomial basis representation for extension fields. We used the *same* irreducible polynomials for defining $\mathbb{F}_{2^{401}}$ and $\mathbb{F}_{3^{251}}$. We intended to work in a general situation and therefore we chose *dense* irreducible polynomials for extending the respective prime fields.

The most obvious conclusion from the above table is that ZEN is the fastest library for prime fields and NTL is the fastest one for extension fields. $\mathrm{G}\mathbb{F}\mathrm{L}$ and LiDIA are slower in general than NTL and ZEN.

The packed representation of field elements in $\mathrm{G}\mathbb{F}\mathrm{L}$ (Section 2.2.3) adds to the running time for additive routines over extension fields. For characteristic 2 fields the overhead is minimal, whereas for fields of odd characteristics, the overhead is significant. $\mathrm{G}\mathbb{F}\mathrm{L}$'s multiplication is slower by a factor of around 5 compared to the best timing tabulated. Our implementation of field inverse is relatively slow for prime fields, but quite close to the best for extension fields. Indeed, for $\mathbb{F}_{3^{251}}$, $\mathrm{G}\mathbb{F}\mathrm{L}$'s inverse routine is the fastest.

There are several other issues that lead to slower performance of $\mathrm{G}\mathbb{F}\mathrm{L}$. First of all, $\mathrm{G}\mathbb{F}\mathrm{L}$ is very *general* in the sense that it provides a uniform treatment of all fields, irrespective of whether the characteristic is small (i.e. 2) or large, whether the field is a prime field or a simply or multiply represented extension etc. The outermost routines call appropriate lower level routines depending on the fields. Thus a user's program written for characteristic 2 fields will work equally well for a field of characteristic 3 or 101 or $2^{400} + 181$, only if the value of the argument in createPrimeGF is changed. No other library discussed here achieves this generality. Indeed a user has to write separate (albeit similar) programs for doing the same thing over fields of different characteristics. Most notably, the other libraries provide a set of routines for fields of characteristic 2, a set for fields of single-precision characteristics and yet another set for fields of multi-precision characteristics. In addition, most of these libraries other than G$\mathbb{F}$L do not provide facility to work with multiply represented extension fields or with more than one prime or extension fields simultaneously. G$\mathbb{F}$L allows one to create and work with as many fields as one wants. In spite of that, the outer prototype of a call is same for all fields. It is apparent that maintaining different data structures and using different sets of library calls for fields of different *types* of characteristics speeds up individual operations. We mentioned that the other libraries do this. We did not, because we did not want to sacrifice generality.

G$\mathbb{F}$L uses dynamic memory for storing data whose sizes are not known *a priori*. Some test implementations carried out by us show that the same routines built with static arrays can speed up the running time by 10 to 20%. However, the use of dynamic arrays appears better to us because of efficient memory management.

The packing overhead for addition and subtraction can be minimized by the following strategy. First define a prime field $\mathbb{F}_p$ and then an extension $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/\langle f(x)\rangle$. Then an element $a \in \mathbb{F}_{p^n}$ has the representation $a = a_0 + a_1x + \ldots + a_{n-1}x^{n-1}, a_i \in \mathbb{F}_p$. G$\mathbb{F}$L substitutes p for x and represents a as a non-negative integer. A packed representation would still be possible if one substitutes b for x, where b is a power of 2 and $b \geqslant p$. This strategy is, however, somewhat less memory efficient and counterintuitive. It also increases the running time of some other operations like generation of random elements of a field. So we did not opt for this.

We finally note that in spite of the packing overhead, generality and dynamic memory management schemes of G$\mathbb{F}$L, it can be made faster by further tuning in the codes and using and/or devising better algorithms for various operations. For example, Karatsuba or FFT-based multiplication techniques can be applied for fields of high extension degree. It is also useful to define field extensions by *sparse* irreducible polynomials. Last but not the least, we have seen in Section 2.4 that for other operations on field elements, polynomials and matrices, it is possible to implement some of the *practically better* algorithms. We plan to enhance continually both the capability and the performance of G$\mathbb{F}$L.

Appendix A Reference manual for Galois Field Library

In this section, we provide a detailed listing of all data structures and prototypes of all library calls provided by G𝔽L at present.

A.1 Include and compile directives

Include header file	`#include <GFL/all.h>`
Compile option	`-lGFL`

A.2 Data Structures

Many of the data structures provided by G𝔽L have already been discussed. For the sake of completeness, we repeat those definitions here.

```
typedef struct {               /* Multi-precision integer */
      char sign;               /* ‘+’ for positive numbers, ‘-’ for negative numbers, ‘ ’ for zero */
      int size;                /* Number of longs needed to represent the number */
      long *word;              /* link to the array of longs holding the integer */
} mpint;

typedef struct {               /* Data structure to store complete or partial factorization of an integer */
      int nf;                  /* Number of factors */
      int *multiplicity;       /* Pointer to the array holding the multiplicities of the factors */
      mpint *factor;           /* Pointer to the array of factors */
} intFactor;

typedef mpint GFelement;       /* Element of a Galois field */

typedef short GF_d;            /* Galois field descriptor */

struct {
      GFelement p;             /* characteristic */
      GFelement q;             /* cardinality */
      int extlev;              /* Extension level */
      GF_d extof;              /* Extension of */
      long extdeg;             /* Extension degree (over immediate subfield) */
      long totextdeg;          /* Total extension degree (over prime subfield) */
      GFelement *defpoly;      /* Pointer to coefficients of defining polynomial */
      long *primpower;         /* Table of powers of a primitive element */
      long *disclog;           /* Table of discrete logs with respect to a primitive element */
      long *zechlt;            /* Zech’s logarithm table */
} GF_info[MAX_FIELDS];

typedef struct {               /* The data structure poly */
      long degree;             /* The exact degree */
      GFelement *coeff;        /* Pointer to the array of coefficients */
} poly;

typedef struct {              /* Data structure matrix */
      int row;                /* Number of rows */
      int col;                /* Number of columns */
      GFelement **element;    /* Pointer to 2-dimensional array of matrix elements */
} matrix;

typedef struct {            /* Data structure vector */
    int size;               /* Vector size */
    GFelement *element;     /* Pointer to the array of vector elements */
} vector;

typedef struct {            /* Structure holding the partial or complete factorization of a polynomial */
    long nf;                /* Number of factors */
    long *multiplicity;     /* Pointer to the array holding the multiplicities of the factors */
    poly *factor;           /* Pointer to the array of factors */
} polyFactor;

/* Factor base for prime fields */
typedef struct {
    GFelement p;            /* Characteristic of the field in which discrete log is taken */
    GFelement gen;          /* Base to which discrete log is taken */
    int fbsize;             /* Number of primes in the factor base */
    long *base;             /* Elements of factor base */
    GFelement *baselog;     /* Discrete logs of factor base elements */
} factorBase1;

/* Factor base for non-prime fields of extension level = 1 */
typedef struct {
    GFelement p;            /* Characteristic of the field in which discrete log is taken */
    GFelement gen;          /* Base to which discrete log is taken */
    int maxdeg;             /* Maximum degree of irreducible polynomials in the factor base */
    int fbsize;             /* Number of elements in the factor base */
    int nprime;             /* Number of primes between 2 and p-1 */
    long *base;             /* Elements of factor base */
    GFelement *baselog;     /* Discrete logs of factor base elements */
} factorBase2;

/* Factor base for non-prime fields of extension level > 1 */
typedef struct {
    GFelement p;            /* Characteristic of the field in which discrete log is taken */
    matrix ctop;            /* Matrix for composed-to-polynomial basis transformation */
    GFelement gen;          /* Base to which discrete log is taken */
    int maxdeg;             /* Maximum degree of irreducible polynomials in the factor base */
    int fbsize;             /* Number of elements in the factor base */
    int nprime;             /* Number of primes between 2 and p-1 */
    long *base;             /* Elements of factor base */
    GFelement *baselog;     /* Discrete logs of factor base elements */
} factorBase3;
```

A.3 Built-in routines

Initialization routines

void GFLinitialize()	This routine initializes the G𝔽L library and must be called at the very beginning of any program that involves G𝔽L library calls.
void readSmallPrimes(int n)	Read the first n ($\leqslant 10^6$) primes from database and store them in the array SMALL_PRIME. The array element SMALL_PRIME[$i-1$] holds the ith prime for $i \geqslant 1$.

Multiprecision integer arithmetic

mpint newInt()	Initialize an mpint before use

void assignInt(mpint *n, char c[])	Convert a numeric character string c to the mpint n.
void longToInt(mpint *n, long m)	Convert the long m to the mpint format and store it in n.
long intToLong(mpint a)	Return the value of the mpint a as long. No error check for overflow.
void readInt(mpint *n, char *msg)	Read the mpint n from stdin. msg is the prompt to display for the input.
int writeInt(mpint n, FILE *fp)	Print the mpint n as a decimal integer to the file pointer fp. If fp is NULL, output is directed to stdout. Returns the number of characters printed.
int showInt(mpint n, FILE *fp)	Print the words of the mpint n to the file pointer fp. If fp is NULL, output is directed to stdout. Returns the number of characters printed.
void destroyInt (mpint *n)	Free memory currently allocated to the mpint n.
void copyInt(mpint *n, mpint m)	Copy the contents of m to n.
int compInt(mpint n, mpint m)	Return 1, 0 or –1 depending on whether $n > m$, $n = m$ and $n < m$ respectively.
int zeroInt(mpint n)	Check if $n = 0$
int positiveInt(mpint n)	Check if $n > 0$
int negativeInt(mpint n)	Check if $n < 0$
int unityInt(mpint n)	Check if n =1
int negUnityInt(mpint n)	Check if $n = -1$
int intTwo(mpint n)	Check if $n = 2$
void twoPowerToInt(mpint *n, long e)	$n = 2^e$
long logTwo(mpint n)	Return $\lfloor \log_2(n) \rfloor$
void intMinus(mpint *n, mpint m)	Assign $n = -m$
void intSum(mpint *t, mpint n, mpint m)	$t = n + m$
void intDiff(mpint *t, mpint n, mpint m)	$t = n - m$
void intProd(mpint *t, mpint n, mpint m)	$t = nm$
void intProdTwo(mpint *t, mpint n, long e)	$t = n \cdot 2^e$
void intSqr(mpint *t, mpint n)	$t = n^2$
void intExp (mpint *t, mpint n, mpint e)	$t = n^e$
void intExpTwo(mpint *t, mpint n, long e)	$t = n^{2^e}$
void intDiv(mpint *t, mpint *s, mpint n, mpint m)	$t = n/m$ (quotient), $s = n\%m$ (remainder). If only one of t and s is needed, the NULL pointer can be passed as the other output argument.
void intDivTwo(mpint *t, mpint *s, mpint n, long e)	$t = n/2^e$ (quotient), $s = n\%2^e$ (remainder). If only one of t and s is needed, the NULL pointer can be passed as the other output argument.
void intModProd(mpint *t, mpint n, mpint m, mpint r)	$t = nm \ \% \ r$ (Modular product).
void intModExp(mpint *t, mpint n, mpint e, mpint r)	$t = n^e \ \% \ r$ (Modular exponentiation).
void modpInv(GFelement *t, GFelement n, GFelement m)	$t = n^{-1} \pmod m$ (Modular inverse). This routine assumes $(n, m) = 1$.
void intPP(mpint *n)	n++
void intMM(mpint *n)	n––
void intInc(mpint *n, long a)	n += a (Increment n by the long a)
void intDec(mpint *n, long a)	n –= a (Decrement n by the long a)
void intOR(mpint *t, mpint n, mpint m)	t is assigned the bitwise OR of n and m.
void intAND(mpint *t, mpint n, mpint m)	t is assigned the bitwise AND of n and m.
void intXOR(mpint *t, mpint n, mpint m)	t is assigned the bitwise XOR of n and m.
void intLeftShift (mpint *n, long e)	Left shift n by e bits.
void intRightShift (mpint *n, long e)	Right shift n by e bits.
void randInt(mpint *n, int len, short seedInfo)	Assign to n a random integer of bit length len. The third argument specifies how to seed the random number generator. Admissible values are: 0 (don't seed), 1 (current time), 2 (use the value of unsigned int INT_SEED_VAL as seed).
void randRes (mpint *n, mpint m)	Assign to n a random number between 0 and $\|m\| - 1$.
int prime(mpint n)	Check if n is prime.
int randPrime(mpint *n, int len, short seedInfo)	Set n to a random prime of bit length len. The third argument has the same interpretation as in randInt. randPrime returns the number of iterations that was necessary to get the first random prime number.
void nextPrime(mpint *n, mpint m)	Assign to n the smallest odd prime larger than or equal to m.

void intSqrt(mpint *n, mpint m)	$n = \lfloor\sqrt{m}\rfloor$ (m must not be negative)
void intGCD(mpint *n, mpint a, mpint b)	$n = \gcd(a, b)$
void intBGCD(mpint *n, mpint a, mpint b)	$n = \gcd(a, b)$ (The binary GCD algorithm)
void intEGCD(mpint *n, mpint *u, mpint *v, mpint a, mpint b)	$n = \gcd(a, b) = au + bv$ (The extended GCD algorithm)
intFactor newIntFactor()	Initialize an intFactor
void destroyIntFactor(intFactor *ff)	Free memory associated with the intFactor ff
int printIntFactors(intFactor ff, FILE *fp)	Print the factorization stored in the intFactor ff to the file pointer fp. If fp is NULL, output goes to stdout.
void factorizeInt(intFactor *ff, mpint n)	Assign to ff the complete factorization of n.

Galois fields

GF_d createPrimeGF(mpint p)	Create a Galois field of prime characteristic p. The field descriptor (of type GF_d) returned can be used for all later references to this field. The value of p must be prime.
GF_d createExtGF(GF_d K, poly f)	Create an algebraic extension of the existing field K by attaching a root of the polynomial f. The GF_d returned is to be used to access the extension field created. The irreducibility of f is not checked.
GF_d primeSubGF(GF_d K)	Return the field descriptor of the prime subfield of K.
GF_d subGF(GF_d K)	Return the field descriptor of the field of which K is represented as an extension. –1 is returned if K is a prime field.
void defPoly(GF_d K, poly *f)	Assign to f the defining polynomial of K.
void characteristic(mpint *p, GF_d K)	Assign to p the characteristic of K.
void cardinality(mpint *q, GF_d K)	Assign to q the cardinality of K.
long extDeg (GF_d K, GF_d F)	Return the extension degree of K over F.
long totExtDeg (GF_d K)	Return the extension degree of K over its prime subfield.
int extLevel(GF_d K)	Level of extension of K over its prime subfield, i.e. number of polynomials attached to the prime subfield to have a representation of K.
void printGFInfo(GF_d K)	Print to stdout information on the Galois field K.

Galois field arithmetic

void GFsum(GFelement *t, GFelement a, GFelement b, GF_d K)	$t = a + b$ (over K)
void GFdiff(GFelement *t, GFelement a, GFelement b, GF_d K)	$t = a - b$ (over K)
void GFprod(GFelement *t, GFelement a, GFelement b, GF_d K)	$t = a \cdot b$ (over K)
void GFinv(GFelement *t, GFelement a, GF_d K)	$t = a^{-1}$ (over K)
void GFqt(GFelement *t, GFelement a, GFelement b, GF_d K)	$t = a \cdot b^{-1}$ (over K)
void GFexp(GFelement *t, GFelement a, mpint e, GF_d K)	$t = a^e$ (over K)
void trace(GFelement *t, GFelement a, GF_d K, GF_d F)	$t = \mathrm{Tr}_{K\|F}(a)$ (Trace)
void absTrace(GFelement *t, GFelement a, GF_d K)	$t = \mathrm{Tr}_{K\|F}(a)$ (absolute trace) where F is the prime subfield of K
void norm(GFelement *t, GFelement a, GF_d K, GF_d F)	$t = \mathrm{N}_{K\|F}(a)$ (Norm)
void absNorm(GFelement *t, GFelement a, GF_d K)	$t = \mathrm{N}_{K\|F}(a)$ (absolute norm) where F is the prime subfield of K
int printGFElement(GFelement a, GF_d K, FILE *fp, short flag)	Print a as an element of K to the file pointer fp (stdout if fp is NULL). The flag specifies the format of printing. The admissible values and the corresponding formats are: 0 (Single integer), 1 (Vector over its immediate subfield), 2 (Vector of vectors of ... over prime subfield), 3 (Flattened form of 2), 4 (Same as 3 except without parentheses), 5 (Polynomial in last extending element), 6 (Polynomial in extending elements [default]). printGFElement returns the number of characters printed.

Arithmetic of polynomials over finite fields

poly newPoly()	Initialize a data structure poly before use
void destroyPoly(poly *f)	Free memory associated with the poly f
void readPoly(poly *f)	Read the polynomial f interactively from stdin
void readPolyFromArray(poly *f, long d, GFelement *ca)	Read the coefficients of a polynomial f of degree d from the array ca. ca[i] should store the coefficient of x^i in $f(x)$ for $i = 0, \ldots, d$.

int writePoly(poly f, GF_d K, FILE *fp, short flag)	Print the polynomial $f(x)$ as a polynomial over K. The output goes to the file pointer fp (or to stdout if fp == NULL). The flag specifies the format for printing the coefficients. See printGFElement above for a meaning of this. writePoly returns the number of characters printed.
void copyPoly(poly *f, poly g)	Assign $f(x) = g(x)$
void lc(GFelement *a, poly f)	Assign to a the leading coefficient of $f(x)$.
int monic(poly f)	Check if $f(x)$ is a monic polynomial.
int zeroPoly(poly f)	Check if $f(x)$ is the zero polynomial.
int equalPoly(poly f, poly g)	Check if $f(x) = g(x)$ (as polynomials).
void evalPoly(GFelement *b, poly f, GFelement a, GF_d K)	Set $b = f(a)$. The field arithmetic is that of K.
void monicize(poly *f, GF_d K)	Monicize a polynomial f by multiplying it with the inverse of its leading coefficient. (over field K)
void polySum(poly *h, poly f, poly g, GF_d K)	$h(x) = f(x) + g(x)$ (in $K[x]$)
void polyDiff(poly *h, poly f, poly g, GF_d K)	$h(x) = f(x) - g(x)$ (in $K[x]$)
void polyProd(poly *h, poly f, poly g, GF_d K)	$h(x) = f(x) \cdot g(x)$ (in $K[x]$)
void polyDiv(poly *h, poly *r, poly f, poly g, GF_d K)	Perform polynomial division: $h(x) = f(x)/g(x)$ (quotient), $r(x) = f(x) \% g(x)$ (remainder) (in $K[x]$). One of h or r can be NULL.
void polyExp(poly *h, poly f, mpint e, GF_d K)	$h(x) = f(x)^e$ (in $K[x]$)
void polyModProd(poly *h, poly f, poly g, poly m, GF_d K)	$h(x) = (f(x) \cdot g(x)) \% m(x)$ (in $K[x]$)
void polyModExp(poly *h, poly f, poly m, mpint e, GF_d K)	$h(x) = f(x)^e \% m(x)$ (in $K[x]$)
void polyGcd(poly *h, poly f, poly g, GF_d K)	$h(x) = \gcd(f(x), g(x))$ (in $K[x]$)
void polyDerivative(poly *h, poly f, GF_d K)	Assign to h the formal derivative of $f(x) \in K[x]$.
void minimalPoly(poly *f, GFelement a, GF_d K, GF_d F)	Assign to f the minimal polynomial of $a \in K$ over F, where $F \subseteq K$ are fields.
void FrobeniusOrder(poly *f, GFelement a, GF_d K, GF_d F)	Store in f the order of $a \in K$ with respect to the Frobenius automorphism of K over F.
void charPoly(poly *f, matrix A, GF_d K)	Assign to f the characteristic polynomial of the matrix A (with elements from K).
void compMatrix(matrix *M, poly f, GF_d K)	Assign to M the companion matrix of $f(x) \in K[x]$.
void polyRes(GFelement *a, poly f, poly g, GF_d K)	Assign to a the resultant of the polynomials $f(x)$ and $g(x)$ in $K[x]$.
void polyDisc(GFelement *a, poly f, GF_d K)	Assign to a the discriminant of $f(x) \in K[x]$.

Matrices and vectors over finite fields

matrix newMatrix()	Initialize a data of type matrix.
matrix newVector()	Initialize a data of type vector.
void destroyMatrix(matrix *M)	Free memory associated with the matrix M.
void destroyVector(vector *v)	Free memory associated with the vector v.
void readMatrix(matrix *M)	Read matrix M interactively from stdin.
void readMatrixFromArray(matrix *M, int row, int col, GFelement **src)	Read a matrix M with row rows and col columns from a 2-dimensional array src.
void readVector(vector *v)	Read vector v interactively from stdin.
void readVectorFromArray(vector *v, int size, GFelement *src)	Read a vector v of dimension size from an array src.
void writeMatrix(matrix M, GF_d K, FILE *fp, short flag)	Print a matrix M to a file pointer fp (The NULL value of fp implies output to stdout). The elements of M are formatted as elements of K depending upon the value of flag (see printGFElement).
void writeVector(vector v, GF_d K, FILE *fp, short flag)	The analogous output routine for vectors.
void setZeroMatrix(matrix *M, int n, int m)	Set M to the $n \times m$ null matrix.
void setIdentityMatrix(matrix *M, int n)	Set M to the $n \times n$ identity matrix.
void setZeroVector(vector *v, int n)	Set v to the zero vector of dimension n.
void copyMatrix(matrix *M, matrix A)	Assign $M = A$.
void copyVector(vector *v, vector u)	Assign $v = u$.
void mtocv(vector *v, matrix M)	Copy a $n \times 1$ matrix M to a vector v.
void mtorv(vector *v, matrix M)	Copy a $1 \times n$ matrix M to a vector v.
void mctov(vector *v, matrix M, int j)	Store in v the jth column of M.

void mrtov(vector *v, matrix M, int i)	Store in v the ith row of M.
void cvtom(matrix *M, vector v)	Store in M a vector v of dimension n as a $n \times 1$ matrix.
void rvtom(matrix *M, vector v)	Store in M a vector v of dimension n as a $1 \times n$ matrix.
void vtomc(matrix M, vector v, int j)	Set the jth column of M to the vector v.
void vtomr(matrix M, vector v, int i)	Set the ith row of M to the vector v.
int equalMatrix(matrix A, matrix B)	Check if $A = B$.
int equalVector(vector v, vector u)	Check if $v = u$.
int zeroMatrix(matrix A)	Check if A is a null matrix.
int zeroVector(vector v)	Check if v is a null vector.
int identityMatrix(matrix A)	Check if A is an identity matrix.
int symmetricMatrix(matrix A)	Check if A is a symmetric matrix.
void matrixTranspose(matrix *M, matrix A)	Assign $M = A^t$.
void matrixSum(matrix *M, matrix A, matrix B, GF_d K)	$M = A + B$ (using arithmetic of the field K)
void vectorSum(vector *w, vector v, vector u, GF_d K)	$w = v + u$ (using arithmetic of the field K)
void matrixDiff(matrix *M, matrix A, matrix B, GF_d K)	$M = A - B$ (using arithmetic of the field K)
void vectorDiff(vector *w, vector v, vector u, GF_d K)	$w = v - u$ (using arithmetic of the field K)
void matrixProd(matrix *M, matrix A, matrix B, GF_d K)	$M = A \cdot B$ (using arithmetic of the field K)
void matrixVectorProd(vector *w, matrix M, vector v, GF_d K)	$w = Mv$ where v is treated as a column vector (over the field K)
void vectorMatrixProd(vector *w , vector v, matrix M, GF_d K)	$w = vM$ where v is treated as a row vector (over the field K)
void scalarMatrixProd(matrix *M, GFelement c, matrix A, GF_d K)	Scalar multiplication $M = cA$ (arithmetic in field K)
void scalarVectorProd(vector *w, GFelement c, vector v, GF_d K)	Scalar multiplication $w = cv$ (arithmetic in field K)
void matrixExp(matrix *M, matrix A, mpint e, GF_d K)	$M = A^e$ (using arithmetic of the field K). Negative values of e are allowed, if A is invertible.
void matrixDet(GFelement *a, matrix A, GF_d K)	Assign to a the determinant of the square matrix A (arithmetic in field K).
int singular(matrix A, GF_d K)	Check if A is a singular matrix over K.
void matrixInv(matrix *M, matrix A, GF_d K)	Set $M = A^{-1}$ (using arithmetic of the field K).
int matrixRank(matrix A, GF_d K)	Returns the rank of the square matrix A over K.
int linIndep(vector *va, int n, GF_d K)	Check if the n vectors (over K) stored in the array va are linearly independent.
int linIndepRows(vector *v, matrix A, GF_d K)	Compute a maximal set of linearly independent rows of the matrix A. The rank (say, r) is returned and the first r entries of the vector v hold the indices of a set of linearly independent rows of A.
int linIndepCols(vector *v, matrix A, GF_d K)	Compute a maximal set of linearly independent columns of the matrix A. The rank (say, r) is returned and the first r entries of the vector v hold the indices of a set of linearly independent columns of A.
int LUPD(matrix *M, vector *v, matrix A, GF_d K)	Compute the LUP decomposition of the invertible matrix A. Let $PA = LU$ where P is a permutation matrix, L a lower-triangular matrix with 1's at the diagonal, and U an upper-triangular matrix. After the call the entries in M that are above and on the main diagonal are elements of U whereas those below the diagonal are elements of L. The (i, j)-th element in P is 1 if and only if $k = i$ and $v_k = j$ for some k.
void linSysSolve(vector *w, matrix A, vector v, GF_d K)	Solve the linear system $Aw = v$. A is assumed invertible. (arithmetic over K)
void overspLinSysSolve(vector *w, matrix A, vector v, GF_d K)	Solve the overspecified linear system $Aw = v$. Here A is an $n \times m$ matrix with $n \geq m$. The routine assumes that the rank of A is m.
void underspLinSysSolve(matrix *M, vector *w, vector *u, matrix A, vector v, GF_d K)	Solve an underspecified system of linear equations. Here the coefficient matrix A is an $n \times m$ matrix with $n \leq m$ and rank n. For details see the G$\mathbb{F}$L reference manual.
int sqLinSysSolve(matrix *M, vector *w, vector *u, matrix A, vector v, GF_d K)	Solve a square linear system. Here the coefficient matrix A is an $n \times n$ matrix whose rank is (possibly) less than n. For details see the G$\mathbb{F}$L reference manual.

Irreducible polynomials

int irreducible(poly f, GF_d K)	Check if $f(x)$ is irreducible over the field K.

int findRandomIrrPoly(poly *f, GF_d K, long d, short seedInfo)	Assign to f a random polynomial of degree d irreducible over K. The flag seedInfo specifies how to seed the random number generator. It can take the following values: 0 (don't seed), 1 (use local time as seed), 2 (use the value of unsigned int SEED_VAL as seed). The function returns the number of random polynomials checked to find the irreducible polynomial.
void findFirstIrrPoly(poly *f, GF_d K, long d)	Assign to f the lexicographically first irreducible polynomial of degree d over K.
void listAllIrrPoly(GFelement *count, GF_d K, long d, FILE *fp, short flag)	List all irreducible polynomials of degree d over K. The file pointer fp should be supplied for directing the output (NULL means stdout). The flag specifies the formatting option (See writePoly and printGFElement). After the routine returns, count holds the total number of irreducible polynomials found.

Roots of polynomials

int findRootES(vector *v, poly f, GF_d K)	Find roots of $f(x)$ in K and return the roots as elements of the vector v. The routine returns the number of roots found. The exhaustive search algorithm is used.
int findRootBR(vector *v, poly f, GF_d K)	Find roots of $f(x)$ in K and return the roots as elements of the vector v. The routine returns the number of roots found. Berlekamp-Rabin algorithm is used. It applies only to fields of odd characteristic.
int findRootBT(vector *v, poly f ,GF_d K, GF_d F)	Find roots of $f(x)$ in K and return the roots as elements of the vector v. The routine returns the number of roots found. Berlekamp's trace algorithm is used. This requires a sub-field F of K. Typically F is the prime sub-field of K. This algorithm is suitable when F has small cardinality.
int findRoot(vector *v, poly f, GF_d K, short flag)	This routine calls one of the above routines depending on the last argument (flag). The actions corresponding to the different values are: 1 – (Call findRootES), 2 – (Call findRootBR), 3 – (call findRootBT with $F =$ the prime sub-field of K), anything else – (if cardinality of K is $<$ SMALL_Q_BOUND call findRootES, else if characteristic of K is 2, call findRootBT, else call findRootBR).

Polynomial factorization

polyFactor newPolyFactor()	Initialize a data of type polyFactor.
void destroyPolyFactor(polyFactor *pf)	Free memory associated with pf.
long printFactors(polyFactor pf, GF_d K, FILE *fp, short flag)	Print the factorization stored in pf to the file pointer fp (NULL implies stdout). flag specifies how to format the coefficients (as elements of the field K); see writePoly and printGFElement for details. The routine printFactors returns the total number of characters printed.
void squareFreeFactorization(polyFactor *pf, poly f, GF_d K)	Store in pf the square-free factorization of $f(x)$ over the field K.
void distinctDegreeFactorization(polyFactor *pf, poly f, GF_d K)	This routine takes as input a square-free polynomial $f(x)$ and computes the distinct degree factorization of $f(x)$ over K and stores this factorization in pf.
void equalDegreeFactorization(polyFactor *pf, poly f, int d, GF_d K)	The polynomial $f(x)$ input to the routine must be a square-free polynomial of which all the irreducible factors are of degree d. The equal degree factorization of $f(x)$ over the field K is computed and stored in pf.
void factorizePoly(polyFactor *pf, poly f, GF_d K)	This routine factorizes a polynomial $f(x)$ over the field K and stores this factorization in pf. This routine in turn calls the square-free, distinct-degree and equal-degree factorization routines described above.

Permutation polynomials

int permPoly(poly f, GF_d K)	Check if f is a permutation polynomial over K

Primitive elements

int primitive(GFelement a, GF_d K)	Check if a is a primitive element of K.
int primitive2(GFelement a, GF_d K, intFactor ff)	Same as primitive except that the integer factorization of $q-1$ is supplied through ff, where q is the cardinality of K.
void findPrimElement(GFelement *a, GF_d K, short seedInfo)	Assign to a a random primitive element in K. The last argument (seedInfo) specifies how to seed the random number generator: 0 means 'don't seed', 1 means 'use current time as seed', and 2 means use the value of the unsigned int PRIMITIVE_SEED_VAL as seed.
void findPrimElement2(GFelement *a, GF_d K, intFactor ff, short seedInfo)	Same as findPrimElement except that the integer factorization of $q-1$ is supplied through ff, where q is the cardinality of K.

void createPrimTable(GFelement g, GF_d K)	Create tables of powers of the primitive element $g \in K$. Maintaining this table speeds up computation (products and inverse) over K. However, the cardinality of K should not be large so that these tables can reside in main memory.
void destroyPrimTable(GF_d K)	Free memory associated with the primitive power tables
void savePrimTable(GF_d K, char *fname)	Save primitive power tables of the field K in hard disk file fname.
void readPrimTable(GF_d K, char *fname)	Read primitive power tables for the field K from hard disk file fname.
void createZechTable(GF_d K)	Create Zech's logarithm table for the field K. This table speeds up addition and subtraction in K. Before this function is called, the primitive power and discrete logarithm tables must be created. Zech's logarithm table resides in main memory, so the cardinality of K should be small if this table is to be maintained.
void destroyZechTable(GF_d K)	Free memory associated with the Zech's logarithm table for the field K.
void saveZechTable(GF_d K, char *fname)	Save Zech's logarithm table of the field K in hard disk file fname.
void readZechTable(GF_d K, char *fname)	Read Zech's logarithm table for the field K from hard disk file fname.

Normal elements

int normal(GFelement a, GF_d K, GF_d F)	Check if a is a normal element for the extension $K\|F$.
int NPoly(poly f, GF_d F)	Check if $f(x) \in F[x]$ is an N-polynomial for the extension $K\|F$ where $K = F[x]/<f(x)>$.
void listAllNormalElements(GF_d K, GF_d F, FILE *fp, short flag)	List all normal elements for the extension $K\|F$ to the file pointer fp (or to stdout if fp is NULL). The flag specifies the output format of elements in K (see printGFElement).
void listAllNPolys(GF_d F, long d, FILE *fp, short flag)	List all N-polynomials in $F[x]$ of degree d. fp and flag has the same significance as in listAllNormalElements.
void findNormalElementR(GFelement *a, GF_d K, GF_d F, short seedInfo)	Find a random normal element for the extension $K\|F$ and store it in a. The flag seedInfo is a directive to seed the random number generator: 0 means 'don't seed', 1 means 'use current time as seed' and 2 means 'use the value of the unsigned int NORMAL_SEED_VAL as seed.
void findNormalElementA(GFelement *a, GF_d K, GF_d F, short seedInfo)	Find a normal element for the extension $K\|F$ and store it in a. This uses the algorithm that follows from Artin's lemma [85, Theorem 4.23]. seedInfo has same meaning as in the routine findNormalElementR.
void findNormalElementBDS(GFelement *a, GF_d K, GF_d F)	Find a normal element for the extension $K\|F$ and store it in a. This routine uses Bach, Driscoll and Shallit's factor refinement algorithm [6].
void findNormalElement(GFelement *a, GF_d K, GF_d F, short flag)	Find a normal element for the extension $K\|F$ and store it in a. It calls one of the above routines depending on the value of flag. For the values 1, 2 and 3 of flag, the routines findNormalElementR, findNormalElementA and findNormalElementBDS are called respectively. For any other value of flag, findNormalElementA is called if $q > 2s(s-1)$, otherwise findNormalElementBDS is called (where q is the cardinality of F and s is the degree of the extension $K\|F$).

Basis utilities

int basis(vector v, GF_d K, GF_d F)	Check if the elements of the vector v form a basis of K over F.
void gtopTransMatrix(matrix *M, vector v, GF_d K)	Assign to M the transformation matrix from a general basis to the polynomial basis of K over its immediate subfield. The elements of the general basis are those of v.
void ptogTransMatrix(matrix *M, vector v, GF_d K)	Assign to M the transformation matrix from the polynomial basis to a general basis of K over its immediate subfield. The elements of the general basis are those of v.
void gtogTransMatrix(matrix *M, vector u, vector v, GF_d K)	Assign to M the transformation matrix from a general basis (defined by u) to another general basis (defined by v) of K over its immediate subfield.
void ntopTransMatrix(matrix *M, GFelement a, GF_d K)	Assign to M the transformation matrix from the normal basis $a, a^q, \cdots, a^{q^{s-1}}$ to the polynomial basis of K over its immediate subfield F where $\|F\| = q$ and $[K : F] = s$.
void ptonTransMatrix(matrix *M, GFelement a, GF_d K)	Assign to M the transformation matrix from the polynomial basis to the normal basis $a, a^q, \cdots, a^{q^{s-1}}$ of K over its immediate subfield F where $\|F\| = q$ and $[K : F] = s$.
void polyBasis(vector *v, poly *f, GF_d K, GF_d F, short seedInfo)	This routine assigns to v a polynomial basis of the field K over a subfield F (not necessarily immediate). That is, suppose that the tower of extensions $F = F_0 \subseteq F_1 \cdots \subseteq F_t = K$ is represented. The default basis of K over F that G𝔽L works with, is called the *composed basis*. This is not, in general, a polynomial basis of K over F. polyBasis returns a polynomial basis for the extension $K\|F$.

void gtocTransMatrix(matrix *M, vector v, GF_d K, GF_d F)	Assign to M the transformation matrix from a general basis to the composed basis of K over a subfield F. The elements of the general basis are those of v.
void ctogTransMatrix(matrix *M, vector v, GF_d K, GF_d F)	Assign to M the transformation matrix from the composed basis to a general basis of K over a subfield F. The elements of the general basis are those of v.
void gtogTransMatrix2(matrix *M, vector u, vector v, GF_d d K, GF_F)	Assign to M the transformation matrix from a general basis (defined by u) to another general basis (defined by v) of K over a subfield F.
void ntocTransMatrix(matrix *M, GFelement a, GF_d K, GF_d F)	Assign to M the transformation matrix from the normal basis $a, a^q, \cdots, a^{q^{s-1}}$ to the composed basis of K over a subfield F where $\|F\| = q$ and $[K : F] = s$.
void ctonTransMatrix(matrix *M, GFelement a, GF_d K, GF_d F)	Assign to M the transformation matrix from the composed basis to the normal basis $a, a^q, \cdots, a^{q^{s-1}}$ of K over a subfield F where $\|F\| = q$ and $[K : F] = s$.

Isomorphism between finite fields

void findIsoMatrix(matrix *M, GF_d K1, GF_d K2)	This routine computes the transformation matrix M between two fields K_1 and K_2 of the same cardinality. Suppose that F is the prime subfield of K_1 and K_2 with $[K_1 : F] = [K_2 : F] = s$. Let $c_1 \in K_1$ have the isomorphic image $c_2 \in K_2$. If the coordinates of c_1 in the composed basis of K_1 over F are $(a_0, \cdots, a_{s-1})$ and those of c_2 in the composed basis of K_2 over F are $(b_0, \cdots, b_{s-1})$, then the relation between the a_i and b_j is given by $(b_0, \cdots, b_{s-1})^t = M(a_0, \cdots, a_{s-1})^t$.

Discrete logarithms

Discrete logarithms in finite fields can be computed with respect to primitive elements using the index calculus method. In the first stage one should compute logarithms of elements in a factor base. At the second stage one computes individual logarithms with the help of the factor base. At present only the basic index calculus method has been implemented. Sieve methods will come up shortly.

The data type for the storage of factor bases and the corresponding routines are dependent on the type of the field over which one intends to calculate discrete logarithms. They have generic names. One has to append 1, 2 or 3 to these names depending on whether the field is a prime field, a simply represented extension of a prime field or an extension field of representation level more than 1. Thus for a field K represented as $K = \mathbb{F}_p[x]/<f(x)>= \mathbb{F}_{p^d}$ (where $\deg(f(x)) = d$ and p a prime), one should use the data type factorBase2 and the routines newFactorBase2, destroyFactorBase2, createFactorBase2, saveFactorBase2, readFactorBase2 and dlog2.

In what follows, we explain the generic description of these routines.

factorBase newFactorBase()	Initialize a factorBase
void destroyFactorBase(factorBase *fb)	Free memory associated with a factorBase
void saveFactorBase(factorBase fb, char *fname)	Save contents of a factorBase in hard disk file fname
void readFactorBase(factorBase *fb, char *fname)	Read factorBase from a hard disk file fname
void createFactorBase(factorBase *fb, GF_d K, GFelement g, long or int N)	Create a factor base for the field K with respect to the primitive element g and store the data in fb. The last argument (N) is of type long for Type 1 factor bases (it signifies a bound on the value of the primes in the factor base) and of type int for Type 2 and Type 3 factor bases (here it signifies the maximum degree of an irreducible polynomial in the factor base).
void dlog(GFelement *l, GFelement a, GF_d K, factorBase fb)	Assign to l the discrete logarithm of $a \in K$ using the data stored in factorBase fb. The primitive element with respect to which the discrete logarithm is taken is stored in fb; one doesn't have to specify it as input to the routine.

3 Algorithms for computing discrete logarithms over prime fields

Computation of discrete logarithms over a finite field $\mathbb{F}_q$ is a difficult problem. No algorithms are known to solve the problem in time bounded by a polynomial in $\log q$. For practical applications, one typically uses prime fields or fields of characteristic 2. In this chapter, we concentrate on prime fields only. We describe three variants of the index calculus method for the computation of discrete logarithms over finite fields of prime cardinality. The first one called the basic method is not a practical method for discrete logarithm computation. It can be applied only to fields of *small* cardinality. The other two methods known as the linear sieve method and the cubic sieve method are practical methods for medium-sized primes.

In Section 3.1 we formally define the discrete logarithm problem (DLP) and provide a generic description of the index calculus method to solve DLP. In Section 3.2, we describe the three methods mentioned above for solving DLP. The analysis of the sieve methods [28] are based on the heuristic assumption that the integers checked for smoothness over a set of primes are randomly distributed. In Section 3.3, we prove that this behavior is not random in the sense that these integers do not follow uniform distribution. Indeed we establish that from the consideration of bit-size, the actual distribution is *better* than the uniform distribution for both the linear sieve and the cubic sieve methods. We give the details of the calculations of Section 3.3 in the appendix at the end of this chapter.

3.1 The discrete logarithm problem

Let $\mathbb{F}_p$ be a prime field of cardinality p. For an element $a \in \mathbb{F}_p$, we denote by $\overline{a}$ the representative of a in the set $\{ 0, 1, \ldots, p-1 \}$. Let g be a primitive element of $\mathbb{F}_p$ (i.e. a generator of the cyclic multiplicative group $\mathbb{F}_p^*$). Given an element $a \in \mathbb{F}_p^*$, there exists a unique integer $0 \leqslant x \leqslant p-2$ such that $a = g^x$ in $\mathbb{F}_p$. This integer x is called the *discrete logarithm* or *index* of a in $\mathbb{F}_p$ with respect to g and is denoted by $\mathrm{ind}_g(a)$. The determination of x from the knowledge of p, g and a is referred to as the *discrete logarithm problem* (DLP).

In general, one need not assume g to be a primitive element and one is supposed to compute x from a and g, if such an x exists (i.e. if a belongs to the cyclic subgroup of $\mathbb{F}_p^*$ generated by g). In this article, we always assume for simplicity that g is a primitive element of $\mathbb{F}_p^*$.

We note that the DLP is the inverse of discrete exponentiation. Discrete exponentiation is *easy* to compute in the sense that there exist algorithms to solve this problem in time bounded by a polynomial in $\log p$. The DLP, on the other hand, is a *difficult* computational problem. No algorithms are known to solve the DLP in time bounded by a polynomial in $\log p$. The intractability of the DLP is exploited for designing various public-key cryptosystems, for example, the ElGamal scheme [32]. It is, therefore, of great interest to obtain practical performance improvements and rigorous analysis of algorithms for the DLP, typically for primes of size $\leqslant 1000$ bits.

The older methods for solving the DLP, namely, Shank's Baby-Step Giant-Step method, Pollard's rho method and Pohlig-Hellman method [85, Sections 6.4, 6.5], have worst-case running times exponential in $\log p$ and hence cannot be used except for *small* fields. They, however, have the advantage that they work for any arbitrary cyclic group. The index calculus method [28, 73, 84, 85, 97] is currently the best known method for solving DLP over fields of both prime and prime-power cardinalities and has a *sub-exponential* expected running time. It, however, does not apply to any arbitrary group. For example, a direct adaptation of the index calculus method for computing discrete logarithms in elliptic curves over finite fields is expected to lead to a running time *worse* than that of brute-force search [120]. Recently, Joseph H. Silverman has proposed a new algorithm, called the *xedni calculus method* [119], which, though originally devised for computing elliptic curve discrete logarithms, can be applied to finite fields. However, this algorithm has been experimentally and heuristically shown to be impractical [62].

3.1.1 The index calculus method

Suppose that we want to compute $\operatorname{ind}_g(a)$ in $\mathbb{F}_p^*$. In the index calculus method, we start by choosing a *factor base* B which is a *suitable* subset of $\mathbb{F}_p^*$ of *small* cardinality. Let us denote $B = \{b_1, b_2, \ldots, b_s\}$. We then search for *relations* of the form

$$g^\alpha a^\beta \prod_{i=1}^{s} b_i^{\gamma_i} \equiv \prod_{i=1}^{s} b_i^{\delta_i} \pmod p$$

This gives us a linear congruence

$$\alpha + \beta \operatorname{ind}_g a + \sum_{i=1}^{s} \gamma_i \operatorname{ind}_g b_i \equiv \sum_{i=1}^{s} \delta_i \operatorname{ind}_g b_i \pmod{p-1}$$

The index calculus method proceeds in two stages. In the first stage, we search for relations with $\beta = 0$. When sufficiently many relations are available, the resulting system of linear congruences is solved mod $p-1$ for the unknowns $\operatorname{ind}_g b_i$. In the second stage, a single relation involving $(\beta, p-1) = 1$ is found. Substituting the *precomputed* values of $\operatorname{ind}_g b_i$ yields $\operatorname{ind}_g a$.

The running time of the index-calculus method is of the form

$$L\langle p, \omega, c\rangle = \exp\Big((c + o(1))(\log p)^\omega (\log\log p)^{1-\omega}\Big) \tag{3.1}$$

for some positive constant c and for some real number $0 < \omega < 1$. By an abuse of notation, we denote by $L(p, c)$ any quantity that satisfies

$$L(p, c) = \exp\Big((c + o(1))\sqrt{\ln p \ln\ln p}\Big)$$

This corresponds to $\omega = \frac{1}{2}$ in Eqn. 3.1. When p is understood from the context, we write $L[c]$ for $L(p, c)$. In particular, $L[1]$ is denoted simply by L.

The basic index calculus method [85, Section 6.6.2] for the computation of discrete logarithms over prime fields and the adaptations of this method take time

$L[c]$ for c between 1.5 and 2 and are not useful in practice for prime fields $\mathbb{F}_p$ with $p > 2^{100}$. Coppersmith, Odlyzko and Schroeppel [28] proposed three variants of the index calculus method that run in time $L[1]$ and are practical for $p \leqslant 2^{250}$. A subsequent paper [73] by LaMacchia and Odlyzko reports implementation of two of these three variants, namely the linear sieve method and the Gaussian integer method. They were able to compute discrete logarithms in $\mathbb{F}_p$ with p of about 200 bits.

The paper [28] also describes a cubic sieve method due to Reyneri for the computation of discrete logarithms over prime fields. The cubic sieve method has a heuristic running time of $L[\sqrt{2\alpha}]$ for some $\frac{1}{3} \leqslant \alpha < \frac{1}{2}$ and is, therefore, asymptotically faster than the linear sieve method (and the other $L[1]$ methods described in [28]). However, the authors of [28] conjectured that the theoretical asymptotics do not appear to take over for p in the range of practical interest (a few hundred bits). A second problem associated with the cubic sieve method is that it requires a solution of a certain Diophantine equation. It is not known how to find a solution of this Diophantine equation in the general case. For certain special primes p a solution arises naturally, for example, when p is *close* to a whole cube.

Recently, a new variant of the index calculus method based on general number field sieves has been proposed and has a conjectured heuristic run time of

$$L\langle p, 1/3, c\rangle = \exp\left((c + o(1))(\log p)^{\frac{1}{3}}(\log\log p)^{\frac{2}{3}}\right)$$

(See [78] for a good reference on this topic). Weber et. al. [105, 127, 128] have implemented and proved the practicality of this method.

3.2 Three variants of the index calculus method for DLP

In this section, we describe the details of the basic method, the linear sieve method and the cubic sieve method. They differ in the choice of the factor base and in the way the relations are generated. We concentrate on the first stage only. (This is typically the more time-consuming stage.) The description of the second stage is similar and can be found in [28, 84, 85].

3.2.1 The basic method

For this method, the factor base is $B = \{q_1, q_2, \ldots, q_t\}$, where q_i is the ith prime ($q_1 = 2$, $q_2 = 3$ and so on). With a harmless abuse of convention, we call an integer n to be B-smooth, if all the prime factors of n are in B, that is, if n factorizes completely over the factor base B. The first stage of the method computes the discrete logarithms of the elements of B (with respect to g). In order to do that, one raises g to a random power j, $2 \leqslant j \leqslant p-2$ and checks if $\overline{g^j}$ factorizes smoothly over the factor base B as an integer. Thus, if $\overline{g^j} = \prod_{i=1}^{t} q_i^{\alpha_i}$, then

$$j \equiv \sum_{i=1}^{t} \alpha_i d_i \pmod{p-1}$$

where d_i is the discrete logarithm of q_i. This gives us a linear congruence in the unknowns d_i. After t such linearly independent relations are found, the resulting system is solved modulo $p-1$. Every search step for a relation involves the following two computations:

1. Computation of the discrete exponentiation g^j.
2. A check if $\overline{g^j}$ factorizes completely over B.

The second step is carried out by trial divisions by the elements of B.

3.2.2 The linear sieve method

Let $H = \lfloor\sqrt{p}\rfloor + 1$ and $J = H^2 - p$. Then $J \leqslant 2\sqrt{p}$. Let's consider the congruence

$$(H + c_1)(H + c_2) \equiv J + (c_1 + c_2)H + c_1c_2 \pmod p \tag{3.2}$$

For *small* integers c_1, c_2, the right side of the above congruence, henceforth denoted as

$$T(c_1, c_2) = J + (c_1 + c_2)H + c_1c_2 \tag{3.3}$$

is of the order of $\sqrt{p}$. If the integer $T(c_1, c_2)$ is smooth with respect to the first t primes $q_1, q_2, \ldots, q_t$, that is, if we have a factorization like $J + (c_1 + c_2)H + c_1c_2 = \prod_{i=1}^{t} q_i^{\alpha_i}$, then we have a relation

$$\mathrm{ind}_g(H + c_1) + \mathrm{ind}_g(H + c_2) = \sum_{i=1}^{t} \alpha_i \,\mathrm{ind}_g(q_i)$$

For the linear sieve method, the factor base comprises of primes less than $L[1/2]$ (so that $t \approx L[1/2]/\ln(L[1/2])$ by the prime number theorem) and integers $H + c$ for $-M \leqslant c \leqslant M$. The bound M on c is chosen such that $2M \approx L[1/2 + \epsilon]$ for some small positive real ϵ. Once we check the factorization of $T(c_1, c_2)$ for all values of c_1 and c_2 in the indicated range, we expect to get $L[1/2 + 3\epsilon]$ relations like (3.2) involving the unknown indices of the factor base elements. If we further assume that the primitive element g is a small prime which itself is in the factor base, then we get a relation $\mathrm{ind}_g(g) = 1$. The resulting system with asymptotically more equations than unknowns is expected to be of full rank and is solved to compute the discrete logarithms of elements in the factor base.

In order to check for the smoothness of the integers $T(c_1, c_2)$ for c_1, c_2 in the range $-M, \ldots, M$, sieving techniques are used. First one fixes a c_1 and initializes to zero an array $\mathfrak{A}$ indexed $-M, \ldots, M$. One then computes for each prime power q^h (q is a small prime in the factor base and h is a small positive exponent), a solution for c_2 of the congruence $(H + c_1)c_2 + (J + c_1H) \equiv 0 \pmod{q^h}$. If the gcd $(H + c_1, q) = 1$, i.e. if $H + c_1$ is not a multiple of q, then the solution is given by $d \equiv -(J + c_1H)(H + c_1)^{-1} \pmod{q^h}$. The inverse in the last equation can be calculated by running the extended gcd algorithm on $H + c_1$ and q^h. Then for each value of c_2 ($-M \leqslant c_2 \leqslant M$) that is congruent to $d \pmod{q^h}$, $\lg q$ is added[1] to the array location $\mathfrak{A}_{c_2}$. On the other hand, if $q^{h_1}||(H + c_1)$ with $h_1 > 0$, we compute $h_2 \geqslant 0$ such that $q^{h_2}||(J + c_1H)$. If $h_1 > h_2$, then for each value of c_2, the expression $T(c_1, c_2)$ is divisible by q^{h_2} and by no higher powers of q. So we add the quantity $h_2 \ln q$ to $\mathfrak{A}_{c_2}$ for all $-M \leqslant c_2 \leqslant M$. Finally, if $h_1 \leqslant h_2$, then we add $h_1 \ln q$ to $\mathfrak{A}_{c_2}$ for all $-M \leqslant c_2 \leqslant M$ and for $h > h_1$ solve the congruence as $d \equiv -\left(\frac{J+c_1H}{q^{h_1}}\right)\left(\frac{H+c_1}{q^{h_1}}\right)^{-1} \pmod{q^{h-h_1}}$.

[1]More precisely, some approximate value of $\lg q$, say, for example, the integer $\lfloor 1000 \lg q \rfloor$.

Once the above procedure is carried out for each small prime q in the factor base and for each small exponent h,[2] we check for which values of c_2, the entry of $\mathfrak{A}$ at index c_2 is *sufficiently close* to the value $\lg(T(c_1, c_2))$. These are precisely the values of c_2 such that for the given c_1, the integer $T(c_1, c_2)$ factorizes smoothly over the small primes in the factor base.

In an actual implementation, one might choose to vary c_1 in the sequence $-M, -M+1, -M+2, \ldots$ and, for each c_1, consider only the values of c_2 in the range $c_1 \leqslant c_2 \leqslant M$. The criterion for 'sufficient closeness' of the array element $\mathfrak{A}_{c_2}$ and $\lg(T(c_1, c_2))$ goes like this. If $T(c_1, c_2)$ factorizes smoothly over the small primes in the factor base, then it should differ from $\mathfrak{A}_{c_2}$ by a small positive or negative value. On the other hand, if the former is not smooth, it would have a factor at least as small as p_{t+1}, and hence the difference between $\lg(T(c_1, c_2))$ and $\mathfrak{A}_{c_2}$ would not be less than $\lg p_{t+1}$. In other words, this means that the values of the difference $\lg(T(c_1, c_2)) - \mathfrak{A}_{c_2}$ for smooth values of $T(c_1, c_2)$ are well-separated from those for non-smooth values and one might choose for the criterion a check whether the absolute value of the above difference is less than 1.

This completes the description of the equation collecting phase of the first stage of the linear sieve method. This is followed by the solution of the linear system modulo $p - 1$.

3.2.3 The cubic sieve method

Let us assume that we know a solution of the Diophantine equation

$$\begin{aligned} X^3 &\equiv Y^2 Z \pmod{p} \\ X^3 &\neq Y^2 Z \end{aligned} \tag{3.4}$$

with X, Y, Z of the order of p^α for some $\frac{1}{3} \leqslant \alpha < \frac{1}{2}$. Then we have the congruence

$$\begin{aligned} &(X + AY)(X + BY)(X + CY) \equiv \\ &\qquad Y^2 \Big[Z + (AB + AC + BC)X + (ABC)Y \Big] \pmod{p} \end{aligned} \tag{3.5}$$

for all triples (A, B, C) with $A + B + C = 0$. If the bracketed expression on the right side of the above congruence, namely,

$$R(A, B, C) = \Big[Z + (AB + AC + BC)X + (ABC)Y \Big] \tag{3.6}$$

is smooth with respect to the first t primes $q_1, q_2, \ldots, q_t$, that is, if we have a factorization $R(A, B, C) = \prod_{i=1}^{t} q_i^{\beta_i}$, then we have a relation like

$$\begin{aligned} &\operatorname{ind}_g(X + AY) + \operatorname{ind}_g(X + BY) + \operatorname{ind}_g(X + CY) \equiv \\ &\qquad\qquad \operatorname{ind}_g(Y^2) + \sum_{i=1}^{t} \beta_i \operatorname{ind}_g(q_i) \pmod{p-1} \end{aligned}$$

If A, B, C are *small* integers, then $R(A, B, C)$ is of the order of p^α, since each of X, Y and Z is of the same order. This means that we are now checking integers

[2]The exponent h can be chosen in the sequence $1, 2, 3, \ldots$ until one finds an h for which none of the integers between $-M$ and M is congruent to d.

smaller than $O(p^{\frac{1}{2}})$ for smoothness over first t primes. As a result, we expect to get relations like (3.5) more *easily* than relations like (3.2) as in the linear sieve method.

This observation leads to the formulation of the cubic sieve method as follows. The factor base comprises of primes less than $L[\sqrt{\alpha/2}]$ (so that by prime number theorem $t \approx L[\sqrt{\alpha/2}]/\ln\left(L[\sqrt{\alpha/2}]\right)$), the integer Y^2 and the integers $X + AY$ for $0 \leqslant |A| \leqslant M$, where M is of the order of $L[\sqrt{\alpha/2}]$. The integer $R(A, B, C)$ is, therefore, of the order of $p^\alpha L[\sqrt{3\alpha/2}]$ and hence the probability that it is smooth over the first t primes selected as above, is about $L[-\sqrt{\alpha/2}]$. As we check the smoothness for $L[\sqrt{2\alpha}]$ triples (A, B, C) (with $A + B + C = 0$), we expect to obtain $L[\sqrt{\alpha/2}]$ relations like (3.5).

In order to check for the smoothness of $R(A, B, C) = Z + (AB + AC + BC)X + (ABC)Y$ over the first t primes, sieving techniques are employed. We maintain an array $\mathfrak{A}$ indexed $-M \ldots + M$ as in the linear sieve method. At the beginning of each sieving step, we fix C, initialize the array $\mathfrak{A}$ to zero and let B vary. The relation $A + B + C = 0$ allows us to eliminate A from $R(A, B, C)$ as $R(A, B, C) = -B(B + C)(X + CY) + (Z - C^2X)$. For a fixed C, we try to solve the congruence

$$-B(B + C)(X + CY) + (Z - C^2X) \equiv 0 \pmod{q^h} \tag{3.7}$$

where q is a small prime in the factor base and h is a small positive exponent. This is a quadratic congruence in B. If $X + CY$ is invertible modulo q^h (i.e. modulo q), then the solution for B is given by

$$B \equiv -\frac{C}{2} + \sqrt{(X + CY)^{-1}(Z - C^2X) + \frac{C^2}{4}} \pmod{q^h} \tag{3.8}$$

where the square root is modulo q^h. If the expression inside the radical is a quadratic residue modulo q^h, then for each solution d of B in (3.8), $\lg q$ is added to those indices of $\mathfrak{A}$ which are congruent to d modulo q^h. On the other hand, if the expression under the radical is a quadratic non-residue modulo q^h, we have no solutions for B in (3.7). Finally, if $X + CY$ is non-invertible modulo q, we compute $h_1 > 0$ and $h_2 \geqslant 0$ such that $q^{h_1}||(X + CY)$ and $q^{h_2}||(Z - C^2X)$. If $h_1 > h_2$, then $R(A, B, C)$ is divisible by q^{h_2} and by no higher powers of q for each value of B (and for the fixed C). We add $h_2 \lg q$ to $\mathfrak{A}_i$ for each $-M \leqslant i \leqslant M$. On the other hand, if $h_1 \leqslant h_2$, we add $h_1 \lg q$ to $\mathfrak{A}_i$ for each $-M \leqslant i \leqslant M$ and try to solve the congruence $-B(B + C)\left(\frac{X+CY}{q^{h_1}}\right) + \left(\frac{Z-C^2X}{q^{h_1}}\right) \equiv 0 \pmod{q^{h-h_1}}$ for $h > h_1$. Since $\frac{X+CY}{q^{h_1}}$ is invertible modulo q^{h-h_1}, this congruence can be solved similar to (3.8).

Once the above procedure is carried out for each small prime q in the factor base and for each small exponent h, we check for which values of B, the entry of $\mathfrak{A}$ at index B is *sufficiently close* to the value $\lg(R(A, B, C))$. These are precisely the values of B for which $R(A, B, C)$ is smooth over the first t primes for the given C. The criterion of 'sufficient closeness' of $\mathfrak{A}_B$ and $\lg(R(A, B, C))$ is the same as described in connection with the linear sieve method.

In order to avoid duplication of effort, we should examine the smoothness of $R(A, B, C)$ for $-M \leqslant A \leqslant B \leqslant C \leqslant M$. The ranges over which A, B and C vary are described in Lemma 3.5.

After sufficiently many relations are available, the resulting system is solved modulo $p-1$ and the discrete logarithms of the factor base elements are stored for computation of individual discrete logarithms.

Attractive as it looks, the cubic sieve method has several drawbacks which impair its usability in practical situations.

1. It is currently not known how to solve the congruence (3.4) for a general p. And even when it is solvable, how large can α be? For practical purposes α should be as close to $\frac{1}{3}$ as possible. No non-trivial results are known to the authors, that can classify primes p according as the smallest possible values of α they are associated with. (See Chapter 5 for some estimates of the expected number of solutions of the congruence.)

2. Because of the quadratic and cubic expressions in A, B and C as coefficients of X and Y in $R(A, B, C)$, the integers $R(A, B, C)$ tend to be as large as $p^{\frac{1}{2}}$ even when α is equal to $1/3$. If we compare this scenario with that for $T(c_1, c_2)$ (See Equation 3.3), we see that the coefficient of H is a linear function of c_1 and c_2 and as such, the integers $T(c_1, c_2)$ are larger than $p^{\frac{1}{2}}$ by a small multiplicative factor. This shows that though the integers $R(A, B, C)$ are asymptotically smaller than the integers $T(c_1, c_2)$, the formers are, in practice, around 10^4–10^6 times smaller than latter ones, even when α assumes the most favorable value (namely, $1/3$). In other words, when one wants to use the cubic sieve method, one should use values of t (i.e. the number of small primes in the factor base) much larger than the values prescribed by the asymptotic formula for t.

3. The second stage of the cubic sieve method, i.e. the stage that involves computation of individual logarithms, is asymptotically as slow as the equation collection stage. For the linear sieve method, on the other hand, individual logarithms can be computed much faster than the equation collecting phase.

3.3 Average behavior and distribution of $T(c_1, c_2)$ and $R(A, B, C)$

Central to the running time analysis of the index calculus methods described in the previous section is the concept of smoothness and probabilistic density expression of smooth integers.

DEFINITION 3.1 An integer x is said to be *y-smooth* for a positive integer y, if all prime factors of x are $\leqslant y$. When y is understood from the context, we speak of x being *smooth*. We denote by $\psi(x, y)$ the number of positive integers $\leqslant x$ that are y-smooth.

The following theorem gives an asymptotic estimate for $\psi(x, y)$ [28, 77, 84].

THEOREM 3.2 If $u = \log x / \log y$, then for $u \to \infty$ and $y \geqslant \log^2 x$,

$$\psi(x, y) = x u^{-u+o(u)}$$

In particular, if we have $x = p^{\alpha}$ and $y = L[\beta] = \exp(\beta\sqrt{\log p \log\log p})$, then $\psi(x, y)/x = L\left[-\dfrac{\alpha}{2\beta}\right]$.

The quantity $\psi(x, y)/x$ measures the likelihood that an integer chosen randomly between 1 and x is y-smooth. The analysis of the running times of the linear and cubic sieve methods makes the *heuristic* assumption that the numbers $T(c_1, c_2)$ and $R(A, B, C)$ are randomly distributed between 1 and a bound x. Therefore, the expected number of relations obtained is governed by the probability $\psi(x, y)/x$.

In this section, we prove that this assumption does not strictly hold. To this end, we first define the following quantities:

DEFINITION 3.3 For the linear sieve method, we define $\overline{T}$ to be the average value of $|T(c_1, c_2)|$ and $T_{\max}$ to be the maximum value of $|T(c_1, c_2)|$, as c_1 and c_2 range over all permissible values, namely $-M \leqslant c_1 \leqslant c_2 \leqslant M$. For the cubic sieve method, let $\overline{R}$ and $R_{\max}$ denote the average and maximum values of $|R(A, B, C)|$ over all triples (A, B, C) satisfying $-M \leqslant A \leqslant B \leqslant C \leqslant M$, $A + B + C = 0$.

If $T(c_1, c_2)$ (resp. $R(A, B, C)$) were truly random, we expect that $\overline{T}$ (resp. $\overline{R}$) would be close to $T_{\max}/2$ (resp. $R_{\max}/2$). However, we calculate that $\overline{T}$ and $\overline{R}$ are significantly smaller than $T_{\max}/2$ and $R_{\max}/2$ respectively. Though these differences are asymptotically unimportant, they reveal that in practical situations the sieve methods tend to produce more relations than predicted by the theoretical estimate.

3.3.1 Average value of $T(c_1, c_2)$

From Eqn. 3.3, it is clear that for most of the values of c_1 and c_2, the term $(c_1 + c_2)H$ dominates in the expression for $T(c_1, c_2)$. In view of this, we can write the approximate value of $\overline{T}$ as

$$\overline{T} \approx H \cdot \left(\sum_{\substack{c_1, c_2 = -M \\ c_1 \leqslant c_2}}^{M} |c_1 + c_2| \right) \Bigg/ \left(\sum_{\substack{c_1, c_2 = -M \\ c_1 \leqslant c_2}}^{M} 1 \right) \tag{3.9}$$

The denominator on the right side of this equation counts the number of pairs of integers (c_1, c_2) subject to the condition $-M \leqslant c_1 \leqslant c_2 \leqslant M$, and can be shown to have a value of $(M + 1)(2M + 1) = 2M^2 + O(M)$. The sum in the numerator of (3.9) can be broken as $\sum\limits_{\substack{c_1, c_2 = -M \\ c_1 \leqslant c_2, c_1 + c_2 > 0}}^{M} (c_1 + c_2) + \sum\limits_{\substack{c_1, c_2 = -M \\ c_1 \leqslant c_2, c_1 + c_2 < 0}}^{M} -(c_1 + c_2)$. It is easy to see that these two sub-sums are identical. Therefore, evaluating the first sum gives the value of the numerator of the right side of (3.9) to be $(\frac{4}{3}M^3 + O(M^2))H$. Thus we get

$$\overline{T} \approx H \left(\frac{2}{3} M + O(1) \right) \tag{3.10}$$

Finally it is easy to see that the maximum value of $T(c_1, c_2)$ is attained approximately at $c_1 = c_2 = \pm M$. To sum up, we have

RESULT 3.4 $\overline{T} \approx H(\frac{2}{3}M + O(1))$ and $T_{\max} \approx 2MH$, so that $\overline{T}/T_{\max} \approx 1/3$.

3.3.2 Average value of $R(A, B, C)$

We first study the range over which A, B and C can vary. The next lemma provides the complete description of this.

LEMMA 3.5 With the conditions $-M \leqslant A \leqslant B \leqslant C \leqslant M$ and $A+B+C=0$ in the cubic sieve method,
i) C varies from 0 to M. In particular, C is always positive.
ii) For a given C, B varies from $-C/2$ to $\min(C, M-C)$.
iii) For a given C, A varies from $\max(-2C, -M)$ to $-C/2$. In particular, A is always negative.

Proof i) If $C < 0$ then $A+B+C=0$ implies that $A+B>0$ and hence at least one of A and B is greater than 0. This contradicts the fact that $A \leqslant C$ and $B \leqslant C$.

ii) We have $2B \geqslant A+B = -C$, so $B \geqslant -C/2$. Again $B \leqslant C$ and $B = -A-C \leqslant M-C$ since $A \geqslant -M$. Thus $B \leqslant \min(C, M-C)$. It is easy to see that all values of B in the range $-C/2 \leqslant B \leqslant \min(C, M-C)$ correspond to some triple (A, B, C).

iii) $2A \leqslant A+B = -C$, $A \geqslant -M$ and $A = -B-C \geqslant -2C$, since $B \leqslant C$. ■

In what follows we assume that Z is small in the sense that it can be neglected in the expression for $R(A, B, C)$. We consider two cases. In the first case we assume $Y \ll X/M$. For example, we study a specific instance of this case with $Y=1$ in the next chapter. It is evident from Eqn. 3.6 that in this case the term $(AB+AC+BC)X$ is the dominant one in the expression for $R(A, B, C)$. Therefore, we can write

$$\overline{R} \approx X \cdot \left(\sum_{\substack{-M \leqslant A \leqslant B \leqslant C \leqslant M \\ A+B+C=0}} |AB+AC+BC| \right) \Bigg/ \left(\sum_{\substack{-M \leqslant A \leqslant B \leqslant C \leqslant M \\ A+B+C=0}} 1 \right) \tag{3.11}$$

The denominator is the number of triples (A, B, C) for which the smoothness of $R(A, B, C)$ is checked and evaluates to $\frac{1}{2}M^2 + O(M)$. In order to evaluate the numerator, we note that $AB+AC+BC = -(B^2+BC+C^2) = -\frac{1}{2}[(B+C)^2+B^2+C^2] \leqslant 0$ for all values of B and C. Therefore, the lemma 3.5 allows us to write the numerator as $X \cdot \sum_{C=0}^{M} \sum_{B=-C/2}^{\min(C,M-C)} (B^2+BC+C^2)$. This sum evaluates to $\frac{329}{1536}M^4 + O(M^3)$. We thus get

RESULT 3.6 For $Y \ll X/M$, $\overline{R} \approx X(\frac{329}{768}M^2 + O(M))$ and $R_{\max} \approx M^2 X$, so that $\overline{R}/R_{\max} \approx 329/768 \approx 0.43$.

The value of $R_{\max}$ in the last theorem can be calculated in the following way. Since $|R/X| \approx B^2+BC+C^2 = (B+C/2)^2 + 3/4C^2$, then by lemma 3.5, $R(A, B, C)$ increases monotonically for a fixed C in the range of admissible variation of B. In particular, if for a particular C, R_C denotes the maximum of $|R(A, B, C)/X|$, then we have

$$R_C = \begin{cases} 3C^2 & \text{for } C \leqslant M/2 \\ (M-C)^2 + (M-C)C + C^2 = M^2 - MC + C^2 & \text{for } C \geqslant M/2 \end{cases}$$

Since $M^2 - MC + C^2 = (C - M/2)^2 + 3/4M^2$ increases monotonically with C for $C \geqslant M/2$, R_C reaches maximum at $C = M$, the maximum value being equal to M^2.

Next we consider the more general case, namely, when X and Y are of the same order of magnitude. In this case $MY \gg X$ and hence the term involving Y dominates in the expression for $R(A, B, C)$. Since $A = -(B + C)$ is always non-positive and C is always non-negative, (3.6) allows us to write $\overline{R}$ as

$$\overline{R} \approx Y \cdot \left(\sum_{\substack{-M \leqslant A \leqslant B \leqslant C \leqslant M \\ A+B+C=0}} |B|C(B + C) \right) \Big/ \left(\sum_{\substack{-M \leqslant A \leqslant B \leqslant C \leqslant M \\ A+B+C=0}} 1 \right) \qquad (3.12)$$

The denominator is $\frac{1}{2}M^2 + O(M)$ as before. The numerator evaluates to $\frac{31}{960}M^5 + O(M^4)$. Finally, it is also easy to check that the maximum value of $|R(A, B, C)|$ is obtained at $A = B = -M/2, C = M$. Thus we have

RESULT 3.7 When $MY \gg X$, $\overline{R} \approx Y(\frac{31}{480}M^3 + O(M^2))$, $R_{\max} \approx YM^3/4$ and therefore, $\overline{R}/R_{\max} \approx 31/120 \approx 0.26$.

3.3.3 Distribution of $T(c_1, c_2)$

The average value $\overline{T}$ does not portray a complete picture of the distribution of $T(c_1, c_2)$. In order to have a better insight of the integers $T(c_1, c_2)$, we define the following:

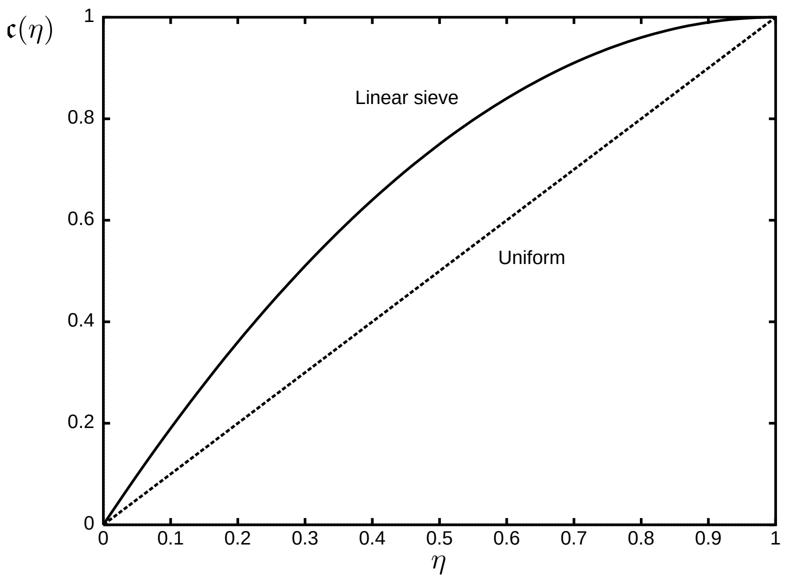
DEFINITION 3.8 For a real $0 \leqslant \eta \leqslant 1$, let us denote $\mathfrak{C}(\eta) = \#\{\, (c_1, c_2) \mid -M \leqslant c_1 \leqslant c_2 \leqslant M, |T(c_1, c_2)| \leqslant \eta T_{\max} \,\}$. We also define $\mathfrak{c}(\eta) = \mathfrak{C}(\eta)/\mathfrak{C}(1)$

By the above definition, $\mathfrak{C}(1)$ is the total number of pairs (c_1, c_2) for which $T(c_1, c_2)$ is checked for smoothness. We have calculated $\mathfrak{C}(1) = 2M^2 + O(M)$. In order to calculate $\mathfrak{C}(\eta)$ for $0 \leqslant \eta < 1$, we proceed as follows. The inequality $|T(c_1, c_2)| \leqslant \eta T_{\max}$ implies $-2\eta M - c_1 \leqslant c_2 \leqslant 2\eta M - c_1$. We also have $c_1 \leqslant c_2 \leqslant M$. Therefore,

$$\mathfrak{C}(\eta) = \sum_{c_1=-M}^{M} \max\Big(0, \min(M, 2\eta M - c_1) - \max(c_1, -2\eta M - c_1) + 1\Big)$$

The closed form expression for $\mathfrak{C}(\eta)$ can be obtained by evaluating the sum on the right side of the last equation and we get $\mathfrak{C}(\eta) = 2\eta(2 - \eta)M^2 + O(M)$. Normalizing by $\mathfrak{C}(1)$ gives $\mathfrak{c}(\eta) \approx \eta(2 - \eta)$. The variation of $\mathfrak{c}(\eta)$ is shown in Fig 3.1. The dotted line corresponds to the variation of $\mathfrak{c}(\eta)$, if $|T(c_1, c_2)|$ *were* uniformly distributed between 0 and $T_{\max}$. The graphs show that for a given $0 < \eta < 1$, there are more integers $|T(c_1, c_2)| \leqslant \eta T_{\max}$ than for the uniform distribution. For example, $\mathfrak{c}(1/2) \approx 3/4$, that is, about 75% of the integers $T(c_1, c_2)$ have absolute value no larger than $\frac{1}{2}T_{\max}$. If the distribution were uniform, this percentage would be 50%. Since smaller integers have higher chance of being smooth, the result shows that the actual distribution of $|T(c_1, c_2)|$ is better than the uniform distribution.

Figure 3.1: Variation of $\mathfrak{c}(\eta)$ for the linear sieve method



3.3.4 Distribution of $R(A, B, C)$

Similar to the case with $T(c_1, c_2)$ we define the following analogous quantities for the cubic sieve method:

DEFINITION 3.9 For a real $0 \leqslant \eta \leqslant 1$, let us denote $\mathfrak{D}(\eta) = \#\{\,(A, B, C) \mid -M \leqslant A \leqslant B \leqslant C \leqslant M, A + B + C = 0, |R(A, B, C)| \leqslant \eta R_{\max}\,\}$. We also define $\mathfrak{d}(\eta) = \mathfrak{D}(\eta)/\mathfrak{D}(1)$

For the cubic sieve method, we consider the two cases: (i) $Y \lll X/M$ and (ii) $Y \ggg X/M$. As told before we assume Z to be *small*. In the former case, we approximate $|R(A, B, C)| \approx (B^2 + BC + C^2)X$. Therefore, $|R(A, B, C)| \leqslant \eta R_{\max}$ leads to the inequality $B \leqslant -C/2 + \sqrt{\eta M^2 - 3/4C^2}$, if the quantity inside the radical is positive. We also have from Lemma 3.5 that $-C/2 \leqslant B \leqslant \min(C, M - C)$. Therefore, $\mathfrak{D}(\eta)$ evaluates to the sum

$$\mathfrak{D}(\eta) \approx \sum_{C=0}^{\min\left(M, \left\lfloor 2M\sqrt{\eta/3}\right\rfloor\right)} \left[1 + \lfloor C/2 \rfloor + \min\left(C, M - C, \left\lfloor -C/2 + \sqrt{\eta M^2 - 3C^2/4} \right\rfloor\right)\right]$$

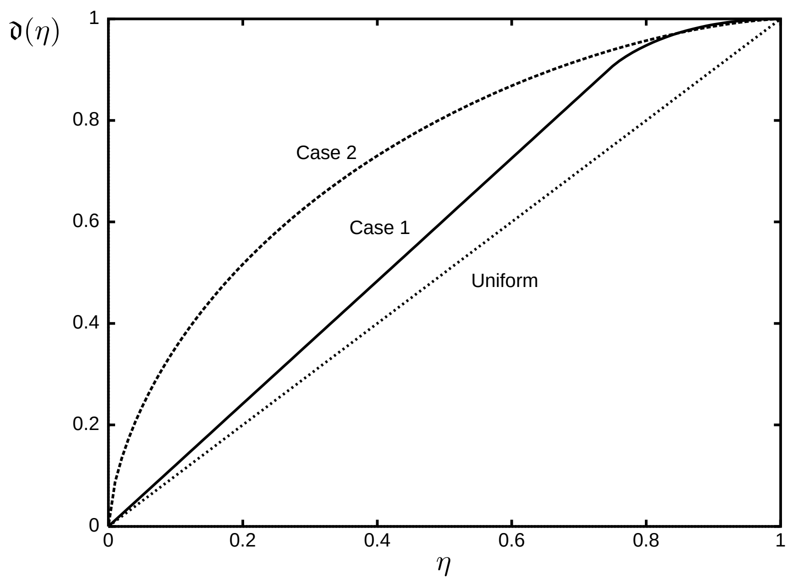
Fig 3.2 shows the variation of $\mathfrak{d}(\eta) = \mathfrak{D}(\eta)/\mathfrak{D}(1)$ with η. Here also we see that the curve for $\mathfrak{d}(\eta)$ lies *above* the curve for the uniform distribution implying that the situation with the $R(A, B, C)$ is better than that with a uniformly distributed sample of integers.

In the second case $Y \ggg X/M$ and we approximate $R(A, B, C) \approx |B|C(B + C)Y$. In this case, $\mathfrak{D}(\eta)$ can be computed from the sum

$$\begin{aligned}\mathfrak{D}(\eta) &= 1 + \sum_{C=1}^{M} \left[1 + \min\left(C, M - C, \left\lfloor -C/2 + \frac{1}{2}\sqrt{C^2 + \eta M^3/C} \right\rfloor\right)\right] \\ &\quad + \sum_{C=1}^{\lfloor \eta^{1/3} M \rfloor} \lfloor C/2 \rfloor + \sum_{\lfloor \eta^{1/3} M \rfloor + 1}^{M} \left\lfloor C/2 - \frac{1}{2}\sqrt{C^2 - \eta M^3/C} \right\rfloor\end{aligned}$$

The corresponding variation of $\mathfrak{d}(\eta) = \mathfrak{D}(\eta)/\mathfrak{D}(1)$ is also shown in Fig 3.2. In this case, the distribution of $R(A, B, C)$ is even better than that in the previous case.

Figure 3.2: Variation of $\mathfrak{d}(\eta)$ for the cubic sieve method
Case 1: $Y \ll X/M$, Case 2: $Y \gg X/M$



To sum up, we see that for both the linear sieve and the cubic sieve methods, the numbers $T(c_1, c_2)$ and $R(A, B, C)$ do not behave exactly as a random sample of integers between 0 and $T_{\max}$ or $R_{\max}$.

Appendix A Detailed calculations

In the last section, we have outlined the procedure to derive the expressions for $\overline{T}$, $T_{\max}$, $\overline{R}$, $R_{\max}$, $\mathfrak{C}(\eta)$ and $\mathfrak{D}(\eta)$. In this appendix, we provide complete details of the calculations. Result 3.6 and the expression for $\mathfrak{D}(\eta)$ for the case $Y \lll X/M$ are not derived in this chapter. In the appendix of the next chapter, we prove them in a more general setting.

A.1 Calculation of $\overline{T}$

We recall from Eqn 3.9 that $\overline{T}$ can be approximated as

$$\overline{T} \approx H \cdot \left(\sum_{\substack{c_1,c_2=-M \\ c_1 \leqslant c_2}}^{M} |c_1+c_2| \right) \Bigg/ \left(\sum_{\substack{c_1,c_2=-M \\ c_1 \leqslant c_2}}^{M} 1 \right)$$

The denominator is equal to the sum $(2M+1)+(2M)+\ldots+1$ which equals $(2M+1)(2M+2)/2 = 2M^2 + O(M)$. The sum in the numerator can be written as

$$\sum_{\substack{c_1,c_2=-M \\ c_1 \leqslant c_2}}^{M} |c_1+c_2| = \sum_{\substack{c_1,c_2=-M \\ c_1 \leqslant c_2 \\ c_1+c_2>0}}^{M} (c_1+c_2) \;\; + \sum_{\substack{c_1,c_2=-M \\ c_1 \leqslant c_2 \\ c_1+c_2<0}}^{M} -(c_1+c_2)$$

The second subsum can be written as $\sum_{\substack{-c_2,-c_1=-M \\ -c_2 \leqslant -c_1 \\ (-c_2)+(-c_1)>0}}^{M} ((-c_2)+(-c_1))$ and is, therefore, equal to the first subsum. Hence the sum in the numerator of Eqn 3.9 evaluates to

$$\begin{aligned}
& 2 \sum_{\substack{c_1,c_2=-M \\ c_1 \leqslant c_2 \\ c_1+c_2>0}}^{M} (c_1+c_2) \\
= \;& 2 \left[\sum_{c_1=-M}^{0} \sum_{c_2=-c_1+1}^{M} (c_1+c_2) \;\; + \sum_{c_1=1}^{M} \sum_{c_2=c_1}^{M} (c_1+c_2) \right] \\
= \;& 2 \left[\sum_{c_1=-M}^{0} (1+2+\ldots+(c_1+M)) + \right. \\
& \qquad\qquad \left. \sum_{c_1=1}^{M} (2c_1 + (2c_1+1) + \ldots + (c_1+M)) \right] \\
= \;& 2 \left[\sum_{c_1=-M}^{0} \frac{(c_1+M)(c_1+M+1)}{2} + \sum_{c_1=1}^{M} \frac{(M-c_1+1)(3c_1+M)}{2} \right] \\
= \;& \frac{M(M+1)(4M+5)}{3} \\
\approx \;& 4M^3/3
\end{aligned}$$

Therefore, $\overline{T} \approx H(\frac{2}{3}M + O(1))$.

A.2 Calculation of $T_{\max}$

Since $T \approx |c_1 + c_2|H$ and $-M \leqslant c_1 \leqslant c_2 \leqslant M$, it is clear that $T(c_1, c_2)$ attains the maximum value at $c_1 = c_2 = \pm M$. This maximum value is $T_{\max} \approx 2MH$.

A.3 Calculation of $\mathfrak{C}(\eta)$ and $\mathfrak{c}(\eta)$

The inequality $|T(c_1, c_2)| \leqslant \eta T_{\max} \approx 2\eta MH$ implies $-2\eta M \leqslant c_1 + c_2 \leqslant 2\eta M$, that is, $-2\eta M - c_1 \leqslant c_2 \leqslant 2\eta M - c_1$. We also have $c_1 \leqslant c_2 \leqslant M$. Therefore, for a given value of c_1, the values of c_2 corresponding to $|T(c_1, c_2)| \leqslant \eta T_{\max}$ are

$$\max(c_1, -2\eta M - c_1) \leqslant c_2 \leqslant \min(M, 2\eta M - c_1)$$

Summing over all values of c_1 gives

$$\mathfrak{C}(\eta) = \sum_{c_1=-M}^{M} \max(0, 1 + \min(M, 2\eta M - c_1) - \max(c_1, -2\eta M - c_1))$$

Now $M \leqslant 2\eta M - c_1$ if and only if $c_1 \leqslant (2\eta - 1)M$, and $c_1 \leqslant -2\eta M - c_1$ if and only if $c_1 \leqslant -\eta M$. Finally, $(2\eta - 1)M \leqslant -\eta M$ if and only if $\eta \leqslant 1/3$. Therefore, we consider two cases $\eta \leqslant 1/3$ and $\eta \geqslant 1/3$. We derive the expression for $\mathfrak{C}(\eta)$ only for the former case; the derivation for the other case is very similar. Note that in the former case, we have $-M \leqslant (2\eta - 1)M \leqslant -\eta M \leqslant M$ and thus the sum for $\mathfrak{C}(\eta)$ can be written as

$$\begin{aligned} \mathfrak{C}(\eta) &= \sum_{c_1=-M}^{\lfloor(2\eta-1)M\rfloor} \max(0, M + 2\eta M + c_1 + 1) \\ &+ \sum_{c_1=\lfloor(2\eta-1)M\rfloor+1}^{\lfloor-\eta M\rfloor} \max(0, 2\eta M - c_1 + 2\eta M + c_1 + 1) \\ &+ \sum_{c_1=\lfloor-\eta M\rfloor+1}^{M} \max(0, 2\eta M - c_1 - c_1 + 1) \end{aligned} \tag{3.13}$$

Since $M + c_1 \geqslant 0$ and $2\eta M + 1$ is a positive quantity, the first sum on the right side of the last equation evaluates to

$$\begin{aligned} & \sum_{c_1=-M}^{\lfloor(2\eta-1)M\rfloor} (M + 2\eta M + 1 + c_1) \\ \approx\ & \frac{1}{2}\left[(1 + 2\eta - 1)M + 1\right]\left[M + 2\eta M + 1 - M + M + 2\eta M + 1 + 2\eta M - M\right] \\ =\ & 6\eta^2 M^2 + O(M) \end{aligned}$$

The second sum on the right side of (3.13) is simply seen to be approximately equal to

$$(4\eta M + 1)(-\eta M - (2\eta - 1)M) = 4\eta(1 - 3\eta)M^2 + O(M)$$

Finally note that $2\eta M - 2c_1 + 1 \geqslant 0$ if and only if $c_1 \leqslant \eta M - \frac{1}{2}$. Therefore, the third sum on the right side of (3.13) is

$$\begin{aligned} & \sum_{c_1=\lfloor-\eta M\rfloor+1}^{\lfloor\eta M-\frac{1}{2}\rfloor} (2\eta M - 2c_1 + 1) \\ \approx\ & (2\eta M + 1)2\eta M + O(M) \\ =\ & 4\eta^2 M^2 + O(M) \end{aligned}$$

Summing up the values of these three subsums gives the value $\mathfrak{C}(\eta) \approx 2\eta(2-\eta)M^2$. For the other case, namely, $\eta \geqslant 1/3$, $\mathfrak{C}(\eta)$ evaluates to the same expression. Since $\mathfrak{C}(1) \approx 2M^2$, we have $\mathfrak{c}(\eta) = \mathfrak{C}(\eta)/\mathfrak{C}(1) \approx \eta(2-\eta)$.

This completes the derivation of the average and distribution of the integers $T(c_1, c_2)$ for the linear sieve method. We next concentrate on the behavior of $R(A, B, C)$ for the cubic sieve method. As told at the beginning of this appendix, we concentrate only on the case $MY \gg X$. For the other case that we studied, namely $Y \ll X/M$, we refer the reader to the Appendix B of the next chapter.

A.4 Calculation of $\overline{R}$

In the case $MY \gg X$, we approximate $|R(A,B,C)|$ as $|R(A,B,C)| \approx |ABC|Y = |-BC(B+C)|Y = |B|C(B+C)Y$, since by Lemma 3.5, $B + C = -A$ is always non-negative. We also see from Lemma 3.5 that for a given C, B varies from $-C/2$ to $\min(C, M-C)$. Consequently, we can write $\overline{R}$ as

$$\overline{R} \approx Y \cdot \left(\sum_{C=0}^{M} \sum_{B=\lceil -C/2 \rceil}^{\min(C,M-C)} |B|C(B+C) \right) \Bigg/ \left(\sum_{C=0}^{M} \sum_{B=\lceil -C/2 \rceil}^{\min(C,M-C)} 1 \right) \tag{3.14}$$

Now $C \leqslant M - C$ if and only if $C \leqslant M/2$. Therefore, the denominator of the above equation is

$$\begin{aligned}
& \sum_{C=0}^{M} \left[1 + \left\lceil \frac{C}{2} \right\rceil + \min(C, M-C) \right] \\
= \;& M + 1 + \sum_{C=0}^{\lfloor M/2 \rfloor} \left[\left\lceil \frac{C}{2} \right\rceil + C \right] + \sum_{C=\lfloor M/2 \rfloor + 1}^{M} \left[\left\lceil \frac{C}{2} \right\rceil + M - C \right] \\
\approx \;& \frac{1}{4} M(2M+5) \;=\; M^2/2 + O(M)
\end{aligned}$$

The sum in the numerator can, on the other hand, be written as

$$\sum_{C=0}^{M} C \sum_{B=\lceil -C/2 \rceil}^{\min(C,M-C)} |B|B \;+\; \sum_{C=0}^{M} C^2 \sum_{B=\lceil -C/2 \rceil}^{\min(C,M-C)} |B| \tag{3.15}$$

The first subsum evaluates to

$$\begin{aligned}
& \sum_{C=0}^{\lfloor M/2 \rfloor} C \sum_{B=\lceil -C/2 \rceil}^{C} |B|B \;+\; \sum_{C=\lfloor M/2 \rfloor + 1}^{M} C \sum_{B=\lceil -C/2 \rceil}^{M-C} |B|B \\
= \;& \sum_{C=0}^{\lfloor M/2 \rfloor} C[-\lfloor C/2 \rfloor^2 - \ldots - 2^2 - 1^2 + 0^2 + 1^2 + 2^2 + \ldots + C^2] \\
& + \sum_{C=\lfloor M/2 \rfloor + 1}^{M} C[-\lfloor C/2 \rfloor^2 - \ldots - 2^2 - 1^2 + 0^2 + 1^2 + 2^2 + \ldots + (M-C)^2] \\
\approx \;& -\frac{1}{24} \sum_{C=0}^{M} C^4 + \frac{2}{3} \sum_{C=0}^{\lfloor M/2 \rfloor} C^4 + O(M^4) \\
\approx \;& -\frac{1}{240} M^5 + O(M^4)
\end{aligned}$$

and the second sum to

$$\begin{aligned}
& \sum_{C=0}^{\lfloor M/2 \rfloor} C^2 \sum_{B=\lceil -C/2 \rceil}^{C} |B| \;+\; \sum_{C=\lfloor M/2 \rfloor+1}^{M} C^2 \sum_{B=\lceil -C/2 \rceil}^{M-C} |B| \\
= \;& \sum_{C=0}^{\lfloor M/2 \rfloor} C^2[\lfloor C/2 \rfloor + \ldots + 2 + 1 + 0 + 1 + 2 + \ldots + C] \\
& + \sum_{C=\lfloor M/2 \rfloor+1}^{M} C^2[\lfloor C/2 \rfloor + \ldots + 2 + 1 + 0 + 1 + 2 + \ldots + M - C] \\
\approx \;& \frac{5}{8}\sum_{C=0}^{M} C^4 - M \sum_{C=\lfloor M/2 \rfloor+1}^{M} C^3 + \frac{1}{2}M^2 \sum_{C=\lfloor M/2 \rfloor+1}^{M} C^2 + O(M^4) \\
\approx \;& \frac{7}{192}M^5 + O(M^4)
\end{aligned}$$

Therefore, the numerator of (3.14) is approximately equal to $((-\frac{1}{240} + \frac{7}{192})M^5 + O(M^4))Y = (\frac{31}{960}M^5 + O(M^4))Y$, so that $\overline{R} \approx (\frac{31}{480}M^3 + O(M^2))Y$.

A.5 Calculation of $R_{\max}$

In order to compute the value of $R_{\max}$, we consider the following cases separately.

Case 1: $B \leqslant 0$

In this case, B varies from $-\lfloor C/2 \rfloor$ to 0. Writing $\beta = -B$, we see that $|R(A,B,C)|/Y \approx |B|C(B+C) = \beta C(C-\beta) = C(C^2/4 - (C/2-\beta)^2)$. For a given C, this expression attains the maximum value at $\beta = C/2$ and this maximum value is $C^3/4$ (neglecting the possible inequality of $\lfloor C/2 \rfloor$ and $C/2$). As C ranges over all possible values, the maximum of this maximum value becomes $M^3/4$ attained when $C = M$.

Case 2: $B \geqslant 0$

In this case, B varies from 0 to $\min(C, M-C)$ and we have the approximation $|R(A,B,C)|/Y \approx BC(B+C) = C((B+C/2)^2 - C^2/4)$, which increases monotonically in the range of variation of B and, therefore, reaches maximum at $B = \min(C, M-C)$.

Case 2a: $B \geqslant 0$ **and** $0 \leqslant C \leqslant M/2$

We have $\min(C, M-C) = C$ and hence the maximum value of $|R(A,B,C)|/Y$ for a given C is approximately $C((C+C/2)^2 - C^2/4) = 2C^3$. This approximate value is maximized at $C = M/2$, the maximum being equal to $M^3/4$.

Case 2b: $B \geqslant 0$ **and** $M/2 \leqslant C \leqslant M$

Here $\min(C, M-C) = M - C$ and for a given C, the maximum value of $|R(A,B,C)|/Y$ is approximately $C((M-C+C/2)^2 - C^2/4) = M(M^2/4 - (C - M/2)^2)$. The last expression attains the maximum value of $M^3/4$ for $C = M/2$.

Thus we see that in all the cases $|R(A,B,C)|/Y$ has the approximate maximum value of $M^3/4$ and, therefore, $R_{\max} \approx YM^3/4$.

A.6 Calculation of $\mathfrak{D}(\eta)$ and $\mathfrak{d}(\eta)$

By definition, $\mathfrak{D}(\eta)$ equals the number of triples (A, B, C) for which $|R(A,B,C)| \leqslant \eta R_{\max}$. This inequality, in turn, leads to the approximate condition

$$|B|C(B+C) \leqslant \eta M^3/4 \tag{3.16}$$

For $C = 0$, there is only one value of B, namely $B = 0$, that satisfies (3.16) and Lemma 3.5 for all values of η. For $C > 0$, we consider two cases:

Case 1: $B \geqslant 0$

In this case (3.16) reduces to $(B + C/2)^2 \leqslant \frac{\eta M^3}{4C} + \frac{C^2}{4}$, or equivalently, $B \leqslant -C/2 + \sqrt{\frac{\eta M^3}{4C} + \frac{C^2}{4}}$. Since we also have $B \leqslant \min(C, M - C)$, we see that for a given C, non-negative values of B that satisfy (3.16) are

$$0 \leqslant B \leqslant \min\left(C, M-C, -C/2 + \sqrt{\frac{\eta M^3}{4C} + \frac{C^2}{4}}\right)$$

Case 2: $B < 0$

In this case, the condition (3.16) demands

$$(B + C/2)^2 \geqslant \frac{C^2}{4} - \frac{\eta M^3}{4C} \tag{3.17}$$

If the right side of this inequality is negative, that is, if $C < \eta^{1/3}M$, then the inequality is satisfied by all B in the range $-\lfloor C/2 \rfloor \leqslant B < 0$. On the other hand, for $C \geqslant \eta^{1/3}M$, the right side of (3.17) is non-negative, and the negative values of B that satisfy (3.16) are

$$-C/2 + \sqrt{\frac{C^2}{4} - \frac{\eta M^3}{4C}} \leqslant B < 0$$

This is certainly a more restrictive condition than $-C/2 \leqslant B < 0$.

Considering the above two cases, we can write the approximate value of $\mathfrak{D}(\eta)$ as

$$\begin{aligned}\mathfrak{D}(\eta) \quad &\approx \quad 1 + \sum_{C=1}^{M}\left[1 + \min\left(C, M-C, -C/2 + \sqrt{\frac{\eta M^3}{4C} + \frac{C^2}{4}}\right)\right] \\ &\quad + \sum_{C=1}^{\lfloor \eta^{1/3}M \rfloor} \lfloor C/2 \rfloor + \sum_{C=\lfloor \eta^{1/3}M \rfloor + 1}^{M} \left\lfloor -C/2 + \sqrt{\frac{C^2}{4} - \frac{\eta M^3}{4C}} \right\rfloor\end{aligned}$$

We have plotted the value of $\mathfrak{d}(\eta) = \mathfrak{D}(\eta)/\mathfrak{D}(1)$ in Fig 3.2 for $M = 1000$.

4 Heuristic modifications of discrete logarithm algorithms

In the last chapter, we discussed three variants of the index calculus method for the computation of discrete logarithms over finite fields of prime cardinality. In this chapter, we propose some heuristic modifications of these methods.

In Section 4.1 we start with the description of two heuristic variants of the basic method. These variants reduce the number of discrete exponentiations and make the trial division procedure more efficient. This leads to a speed-up of about 2 over the basic method. In Section 4.2, we discuss efficient implementation schemes for the linear and cubic sieve methods. A heuristic modification of the linear sieve method follows in Section 4.3. This modification checks smaller numbers for smoothness over the chosen factor base and is, thereby, expected to produce more relations than the original method. A heuristic improvement of the cubic sieve method is described in Section 4.4. Our heuristic helps generate a larger factor base at almost no extra cost. We also study the effect of the heuristic on the average, the maximum and the distribution of the integers that are checked for smoothness.

In this chapter, we use terms and notations introduced in the previous chapter – often without specific mention. All experiments reported in this chapter are carried out using the Galois Field Library described in Chapter 2, on a 200 MHz Pentium machine running Linux version 2.0.34 and having 64 Mb RAM. The GNU C Compiler version 2.7 is used. The timings correspond to an older (and somewhat slower) version of $\mathrm{G}\mathbb{F}\mathrm{L}$.

4.1 Heuristic modification of the basic method

We recall from Section 3.2.1 that in the basic method, a relation is set up by raising g to a random power and by checking if that power factorizes completely over the chosen factor base as an integer. This process involves two time-consuming operations: a discrete exponentiation in the field and a check whether the power of the primitive element factorizes smoothly over the factor base. This check is usually carried out by trial divisions by elements of the factor base. Here we propose some heuristic variants of this basic method. These methods try to factorize several integers which are identical as elements of $\mathbb{F}_p$, before a new exponent is tried. This decreases the total number of discrete exponentiations. In order to reduce the cost of trial divisions we apply certain tricks that help us avoid all unnecessary trial divisions. We concentrate only on the first stage of the method. Our modification can be applied directly to the second stage of the index calculus method as well.

4.1.1 The first heuristic

For all integers r, $\overline{g^j} + rp = \overline{g^j}$ as elements of $\mathbb{F}_p$. (In this section '$+$' denotes integer addition unless otherwise specified.) Therefore, if for some non-negative integer r, we have a factorization $\overline{g^j} + rp = \prod_{i=1}^{t} q_i^{\alpha_i}$, we have a relation. On the other hand, if for some negative integer r, we have $\overline{g^j} + rp = -\prod_{i=1}^{t} q_i^{\alpha_i}$, then

$$j + \frac{p-1}{2} \equiv \sum_{i=1}^{t} \alpha_i \operatorname{ind}_g q_i \pmod{p-1}$$

since $g^{\frac{p-1}{2}} = -1$ in $\mathbb{F}_p$ (g being a primitive element of $\mathbb{F}_p$). This motivates us to reframe the search procedure for relations in the following way:

HEURISTIC B1

1. Choose a random integer j, $2 \leqslant j \leqslant p-2$.
2. Check if $\overline{g^j}$ factorizes completely over the factor base B. If yes, a relation is found, store it and proceed to Step 1.
3. Check if $\overline{g^j} + rp$ factorizes completely over B in succession for $r = 1, 2, \ldots$ If a relation is found for some value of r, store it and proceed to Step 1. Else if r exceeds a predefined value k (the choice of k will be explained later), proceed to Step 4.
4. Check if $-\left(\overline{g^j} + rp\right)$ factorizes completely over B in succession for $r = -1, -2, \ldots$ If a relation is found for some value of r, store it and proceed to Step 1. Else if $|r|$ exceeds k, go to Step 1.

Note that $\overline{g^j}$ is a number with less than or equal to $\lceil \lg p \rceil$ bits. On the other hand, $\overline{g^j} + rp$ is a number with at most $\lceil \lg |rp| \rceil$ bits. The bound k of r in Steps 3 and 4 should be small enough so that the bit-length of $\overline{g^j} + rp$ is not too large compared with $\lceil \lg p \rceil$. This is because if this bit-length is large compared to $\lceil \lg p \rceil$, the probability that $\overline{g^j} + rp$ factorizes smoothly over B is sufficiently small compared to the corresponding probability for an integer less than p (See Theorem 3.2). As a result, the search procedure loses effectiveness if large values of r are chosen. Later we discuss the variation of the performance of our heuristic for different values of k for some small example problems.

Since we check the factorization of $\overline{g^j} + rp$ in succession for $r = 0, 1, 2, \ldots$ in Step 3, we can make trial divisions more efficient in the following way. Let v_i be the integer between 0 and $q_i - 1$, that is congruent to p modulo q_i. The t integers $v_1, v_2, \ldots, v_t$ can be pre-computed. Let us denote by $\rho_{r,1}, \rho_{r,2}, \ldots, \rho_{r,t}$ the remainders of divisions of $\overline{g^j} + rp$ by $q_1, q_2, \ldots, q_t$ respectively. After $\overline{g^j}$ is computed for some random j, $\rho_{0,1}, \rho_{0,2}, \ldots, \rho_{0,t}$ are computed by performing actual divisions of $\overline{g^j}$ by the primes q_i in the factor base. The remainders $\rho_{1,1}, \rho_{1,2}, \ldots, \rho_{1,t}$ are computed from $\rho_{0,1}, \rho_{0,2}, \ldots, \rho_{0,t}$ and, in general, the remainders $\rho_{r+1,1}, \rho_{r+1,2}, \ldots, \rho_{r+1,t}$ are calculated from $\rho_{r,1}, \rho_{r,2}, \ldots, \rho_{r,t}$ using the relation $\rho_{r+1,i} \equiv \rho_{r,i} + v_i \pmod{q_i}$ for all $i = 1, 2, \ldots, t$. Now a $q_i \in B$ divides $\overline{g^j} + rp$ if and only if $\rho_{r,i} = 0$. Hence trial divisions of $\overline{g^j} + rp$ by q_i is carried out if and only if $\rho_{r,i} = 0$. Since in typical situations only a few of $\rho_{r,i}$ are 0 irrespective of whether $\overline{g^j} + rp$ factorizes completely over B or not, we save many unnecessary divisions of $\overline{g^j} + rp$ by q_i.

In Step 4, the incremental procedure involves the following operations. $p - \overline{g^j}$ is computed by subtracting $\overline{g^j}$ from p. Subsequently, $(|r|+1)p - \overline{g^j}$ is computed by adding p to $|r|p - \overline{g^j}$. Here we designate by $\rho_{-|r|,i}$ the remainder of division of $|r|p - \overline{g^j}$ by q_i. The remainders $\rho_{-1,1}, \rho_{-1,2}, \ldots, \rho_{-1,t}$ are obtained using the relations $\rho_{-1,i} \equiv v_i - \rho_{0,i} \pmod{q_i}$. For $|r| \geqslant 1$, we have the relation $\rho_{-|r|-1,i} \equiv$

$v_i + \rho_{-|r|,i} \pmod{q_i}$. Therefore, the incremental procedure in Step 4 is essentially the same as in Step 3 (except for the case $r = -1$).

4.1.2 The second heuristic

As explained in the last section, we cannot take large values of k (the bound of $|r|$ in $\overline{g^j} + rp$). In typical applications one should be satisfied with $k \leqslant 10$. If we want to use larger values of k (say, values of the order of 100), we should have an even faster method of checking integers for B-smoothness. Our second heuristic achieves that by using sieving techniques similar to those used in connection with the quadratic sieve method for factoring integers [16, 46]. This method calls for more additional storage than what is needed in the previous heuristic, and incurs a costlier pre-computation stage. In this case, the basic steps are as follows:

HEURISTIC B2

1. Choose a random integer j, $2 \leqslant j \leqslant p-2$.
2. For each $r = -k, -k+1, \ldots, -1, 0, 1, \ldots, k$, check if $|\overline{g^j} + rp|$ factorizes completely over the factor base B. Store relations corresponding to all B-smooth values of $\overline{g^j} + rp$. Go to Step 1.

The basic difference between heuristics B1 and B2 is that in B1, an exponent j is discarded when a relation is found or when all values $|r| \leqslant k$ are checked. In B2, on the other hand, all values $|r| \leqslant k$ are considered for a given exponent irrespective of whether we get relations for some values of r. This is justifiable because we are checking a larger range of values of r, so that we might expect to get more than one r for which $|\overline{g^j} + rp|$ is B-smooth for a given j. The second difference between B1 and B2 is the way in which the check for B-smoothness of $|\overline{g^j} + rp|$ is implemented. We now describe this check procedure for B2.

Before the search for relations is started, we precompute and store the following quantities:

1. For each $i = 1, 2, \ldots, t$, the exponent β_i such that $q_i^{\beta_i} < (k+1)p \leqslant q_i^{\beta_i + 1}$.
2. The powers q_i^l for all $i = 1, 2, \ldots, t$ and for all $l = 1, 2, \ldots, \beta_i$.
3. The approximate values of the logarithms $\lg q_i^l$ for all $i = 1, 2, \ldots, t$ and for all $l = 1, 2, \ldots, \beta_i$. For a multi-precision integer n represented in base $R = 2^b$ as

$$n = a_s R^s + a_{s-1} R^{s-1} + \ldots + a_1 R + a_0$$

with $a_s \neq 0$, the approximate logarithm is set equal to the float $\log_2(a_0)$, if $s = 0$, and to the float $\log_2(a_s R + a_{s-1}) + (s-1)b$, if $s > 1$.

4. The (positive) remainders of division of $-p^{-1}$ by q_i^l for all $i = 1, 2, \ldots, t$ and for all $l = 1, 2, \ldots, \beta_i$. Note that $p^{-1} \equiv p^{q_i^l - q_i^{l-1} - 1} \pmod{q_i^l}$.

Our sieving procedure is little different from the traditional one used in the quadratic sieve method for integer factorization. Before we discuss the modified sieving procedure, let us introduce the following terminology. For a given r, $-k \leqslant r \leqslant k$, the least integer $r + u$ with $u \geqslant 0$ satisfying $(\overline{g^j} + rp) + up \equiv 0 \pmod{q_i^l}$

is defined to be the *next divisibility index* of q_i^l at r. If $r + u$ is less than or equal to k, we say that the prime power q_i^l is *active* in the range $r, r + 1, \ldots, k$.

After a random j is selected and the discrete exponentiation g^j is carried out, we first calculate $\overline{g^j} - kp$. Then for each $i = 1, 2, \ldots, t$ and for each $l = 1, 2, \ldots, \beta_i$, the next divisibility index for q_i^l at $-k$ is calculated. By definition, this is the smallest integer $u - k$ with $u \geqslant 0$ satisfying $(\overline{g^j} - kp) + up \equiv 0 \pmod{q_i^l}$, i.e., $u \equiv (-p^{-1})(\overline{g^j} - kp) \pmod{q_i^l}$. This is where we use the pre-computed values of $-p^{-1}$ modulo q_i^l. We also calculate for each $i = 1, 2, \ldots, t$ the value δ_i such that $q_i^{\delta_i}$ is active in the range $-k, \ldots, k$, but $q_i^{\delta_i+1}$ is not. We have $\delta_i \leqslant \beta_i$.

Now we repeat the following procedure in succession for $r = -k, -k + 1, \ldots, -1, 0, 1, \ldots, k$. If the next divisibility index of q_i at r is equal to r itself, then q_i divides $(\overline{g^j} + rp)$ and we calculate the largest integer $\epsilon_i \leqslant \delta_i$ such that the next divisibility index of $q_i^{\epsilon_i}$ is r but that of $q_i^{\epsilon_i+1}$ is greater than r. Then $q_i^{\epsilon_i} || (\overline{g^j} + rp)$. On the other hand, if the next divisibility index of q_i at r is greater than r, we take $\epsilon_i = 0$. We then calculate the quantity

$$\mathfrak{L} = \lg(\overline{g^j} + rp) - \sum_{i=1}^{t} \lg(q_i^{\epsilon_i}) \tag{4.1}$$

If $\mathfrak{L}$ is close to 0 or negative, then we have a relation.

Before, we proceed with $r + 1$, we compute the next divisibility indices for q_i^l at $r + 1$ in the following way (for all $1 \leqslant i \leqslant t$ and $1 \leqslant l \leqslant \delta_i$). If $l > \epsilon_i$, the next divisibility index of q_i^l at $r + 1$ is the same as that at r. If $l \leqslant \epsilon_i$, then the next divisibility index of q_i^l at $r + 1$ is q_i^l plus the next divisibility index of q_i^l at r.

This completes the description of the details of the heuristic B2. Before we end this section, two further comments are in order. First we note that we can implement the next divisibility indices as single-precision integers, though their definition demands them to be multiple-precision ones. This is because, if some q_i^l is not active in the range $r, \ldots, k$, then this prime power divides none of the integers $\overline{g^j} + rp, \ldots, \overline{g^j} + kp$. Our heuristic never tries to check the integers $\overline{g^j} + rp$ for $r > k$ and hence whenever q_i^l gets non-active, we may set its next divisibility index to $k + 1$. Also note that instead of defining the next divisibility indices as integers $\geqslant -k$, one can define them as integers $\geqslant 0$ by adding k to the values following from the current definition.

Finally note that if the logarithms calculated are *exact*, then $\overline{g^j} + rp$ is B-smooth if and only if $\mathfrak{L} = 0$. But in practical situations we work with approximate logarithms and as such the smoothness criterion should be different from the check $\mathfrak{L} = 0$. If $\overline{g^j} + rp$ is B-smooth, then $\mathfrak{L}$ should be a small real number (positive or negative). On the other hand, if $\overline{g^j} + rp$ is not B-smooth, it will have at least one prime factor not in B and hence $\mathfrak{L}$ must not be too less than $\lg(q_{t+1})$, q_{t+1} being the smallest prime not in B. This implies that the values of $\mathfrak{L}$ for B-smooth integers are well-separated from those for non-B-smooth integers and the selection criterion might be taken to be the check whether $\mathfrak{L} < 1$.

4.1.3 Performance analysis

In this section, we argue heuristically why our modifications should run faster than the basic method. Suppose that multi-precision integers are represented in the computer memory as an array of words in radix 2^b. Therefore the storage of an l-bit number needs $\lceil l/b \rceil$ words. In particular, each of the numbers $\overline{g}$, p, j, $\overline{g^j}$, $\overline{g^j} + rp$ (for small r) occupies nearly $w = \lceil \frac{\lg p}{b} \rceil$ words. The discrete exponentiation g^j requires $O(w^3)$ multiplications and an equal number of additions of machine words. A summation $(\overline{g^j} + rp) + p$ to get $\overline{g^j} + (r+1)p$ from $\overline{g^j} + rp$, on the other hand, needs $O(w)$ additions of machine words. Trial divisions of $\overline{g^j} + rp$ by a prime of B takes $O(w)$ multiplications, divisions, additions and subtractions of machine words (since B consists only of single-precision primes in typical applications).

In the basic method, only a single integer is checked for B-smoothness after every discrete exponentiation. In our heuristics, several other integers are checked for B-smoothness, before the next exponentiation is carried out. These integers are obtained by successively adding p and so the costs of obtaining these integers are much less than those for obtaining integers by discrete exponentiation.

For the heuristic B1, computation of $\rho_{0,i}$ for all $i = 1, 2, \ldots, t$ requires $O(tw)$ operations (additions, subtractions, multiplications and divisions) of machine words. Subsequently, $\rho_{r,i}$ for $r \neq 0$ and for all $i = 1, 2, \ldots, t$ can be computed using $O(t)$ additions and subtractions of machine words (since they are obtained by adding $\rho_{r-1,i}$ or $\rho_{r+1,i}$ to v_i each of which is a single-precision integer). If s is the average number of i for which $\rho_{r,i} = 0$, our procedure avoids $O(t-s)$ unnecessary divisions of $|\overline{g^j} + rp|$ by factor base primes at the cost of $O(t)$ single-precision additions and subtractions mentioned above. Since s is usually much smaller compared to t, this leads to a significant decrease in the cost associated with the trial division procedure.

For the other heuristic B2, trial divisions are completely dispensed with at the cost of maintaining the next divisibility indices for factor base primes and for suitable powers of them. Let T denote the total number of primes and prime powers monitored by the algorithm. Using the terminology of the previous section, $T = O(\sum_{i=1}^{t} \beta_i)$, where $\beta_i = \lceil \log_{q_i}((k+1)p) \rceil$. In particular, $T = O(t \lg(kp))$. If s primes of B divide $|\overline{g^j} + rp|$, then we require $O(s \lg(kp))$ integer comparisons to compute the exponents ϵ_i and $O(s)$ floating-point subtractions to compute $\mathfrak{L}$ in Eqn 4.1. Subsequently we update the next divisibility indices of the primes and the associated powers only for those $O(s)$ primes that divide $|\overline{g^j} + rp|$. This requires $O(s \lg(kp))$ single-precision additions. Once again, since s is usually much smaller than t, our heuristic speeds up the trial division procedure.

4.1.4 Experimental results

In this section, we compare typical timings for the first stage of the basic index calculus method with those of our modified methods. In the following tables, we list timing results for $p = 19196459099$, $p = 781487259479$ and $p = 29438018625539$. For each of these values of p, $\frac{p-1}{2}$ is a prime. These primes are obtained by searching random integers of given bit-lengths. The following notations are used in the tables:

Table 4.1: Timings for heuristic B1 with $p = 19196459099$ (A 35-bit prime)

	Basic method				Heuristic B1							
t	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	k	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	P_0	P_+	P_-
50	0.348	0.122	0.471	41.043	1	0.138	0.076	0.214	16.796	42.8	39.3	17.7
					2	0.111	0.083	0.194	15.346	34.8	38.2	26.9
					3	0.091	0.084	0.175	12.836	25.7	45.0	29.2
					4	0.090	0.100	0.191	16.015	24.2	41.1	34.5
					5	0.082	0.106	0.189	17.005	22.8	44.8	32.2
					7	0.064	0.106	0.171	14.439	15.9	47.4	36.5
					10	0.066	0.143	0.21	13.569	15.9	41.3	42.7
60	0.239	0.099	0.338	34.973	1	0.096	0.062	0.159	19.631	41.5	37.9	20.5
					2	0.082	0.070	0.153	16.862	32.9	38.9	28.0
					3	0.067	0.073	0.140	19.039	25.4	45.0	29.4
					4	0.062	0.080	0.143	15.841	23.3	39.3	37.2
					5	0.056	0.083	0.139	14.562	19.6	40.9	39.4
					7	0.047	0.088	0.136	18.421	17.5	42.0	40.3
					10	0.048	0.115	0.164	19.574	16.6	38.6	44.7
75	0.164	0.083	0.247	45.911	1	0.074	0.057	0.132	25.713	40.4	37.6	21.8
					2	0.058	0.061	0.120	21.566	30.4	38.4	31.0
					3	0.046	0.060	0.106	20.692	26.1	42.1	31.7
					4	0.045	0.067	0.112	18.113	26.1	39.3	34.4
					5	0.036	0.066	0.102	19.017	19.9	42.6	37.4
					7	0.027	0.064	0.092	18.632	14.1	45.6	40.1
					10	0.029	0.086	0.116	19.229	14.2	37.9	47.8

t	$=$	Number of primes in the factor base
k	$=$	The bound on $\lvert r\rvert$ such that the integers $\overline{g^j} + rp$ are tested for B-smoothness
$\mathfrak{t}_1$	$=$	Average time in seconds (per relation generated) taken for computing integers for trial divisions (i.e. for the computation of $\overline{g^j}$ and (in our heuristics) $\overline{g^j} + rp$)
$\mathfrak{t}_2$	$=$	Average time in seconds (per relation generated) taken by trial divisions
$\mathfrak{t}$	$=$	Average time to generate a relation $= \mathfrak{t}_1 + \mathfrak{t}_2$
$\mathfrak{T}_{\text{av}}$	$=$	Average total time in seconds taken by the first stage of the index calculus method
P_0	$=$	Percentage of the occurrences when $\overline{g^j}$ is B-smooth
P_+	$=$	Percentage of the occurrences when $\overline{g^j} + rp$ is B-smooth for some $r > 0$
P_-	$=$	Percentage of the occurrences when $\overline{g^j} + rp$ is B-smooth for some $r < 0$

These data are average ones obtained over a set of 10 random runs of the first stage of the index calculus method corresponding to each set of values of the various parameters (p, t and k).

The values $\mathfrak{t}_1$ and $\mathfrak{t}_2$ represent the average times spent by exponentiations and by trial divisions respectively for generating each relation. Their sum ($\mathfrak{t} = \mathfrak{t}_1 + \mathfrak{t}_2$) is the total average time spent for generating a relation (dependent or independent). This number $\mathfrak{t}$ seems to be the best metric to assess the effectiveness of our heuris-

Table 4.2: Timings for heuristic B1 with $p = 781487259479$ (A 40-bit prime)

	Basic method				Heuristic B1							
t	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	k	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	P_0	P_+	P_-
50	2.308	0.710	3.018	216.32	1	0.952	0.419	1.371	95.811	43.4	38.9	17.6
					2	0.859	0.515	1.374	102.35	33.6	41.0	25.3
					3	0.675	0.509	1.184	85.367	26.0	43.7	30.1
					4	0.650	0.591	1.241	90.977	26.0	41.8	32.0
					5	0.523	0.555	1.079	63.85	20.2	44.7	35.0
					7	0.482	0.650	1.132	67.252	18.4	49.0	32.5
					10	0.476	0.829	1.305	81.419	17.7	38.6	43.6
60	1.475	0.535	2.011	173.40	1	0.644	0.332	0.976	85.133	40.4	38.7	20.7
					2	0.550	0.382	0.932	83.47	35.7	36.3	27.9
					3	0.448	0.391	0.839	74.695	27.8	41.3	30.7
					4	0.386	0.403	0.790	57.112	24.5	38.7	36.6
					5	0.341	0.419	0.761	58.729	22.0	43.8	34.1
					7	0.280	0.439	0.720	61.04	19.1	47.9	32.8
					10	0.311	0.620	0.932	89.156	17.7	37.7	44.5
75	0.921	0.410	1.332	199.92	1	0.388	0.238	0.627	71.707	42.0	38.2	19.7
					2	0.331	0.277	0.609	83.923	32.6	38.7	28.5
					3	0.247	0.260	0.507	67.547	26.6	42.1	31.1
					4	0.219	0.278	0.498	61.649	23.1	42.5	34.2
					5	0.198	0.288	0.486	59.805	20.1	42.6	37.2
					7	0.163	0.301	0.464	53.713	15.8	45.3	38.7
					10	0.159	0.383	0.542	74.955	15.7	38.4	45.7

tics as compared with the basic method. $\mathfrak{T}_{\text{av}}$ represents the total time taken by each execution of the first stage of the index calculus method. This includes the time to do the necessary pre-computations, the time to generate t linearly independent relations and the time to solve the resulting system modulo $p - 1$. $\mathfrak{T}_{\text{av}}$ depends on $\mathfrak{t}$ and also on the number of relations that need to be generated before a full-rank system is obtained. This number of relations generated varies widely from run to run (typically from t to $3t$). Only 10 random runs that we carried out for each set of parameter values, seem insufficient to smooth out the variation. So we do not take $\mathfrak{T}_{\text{av}}$ as an effective measure of the performance of the algorithms, though this quantity reflects the scenario on the whole.

The first three tables (Tables 4.1 through 4.3) correspond to the heuristic B1. We have taken runs for some values of k in the range $1 \ldots 10$. It is evident from the tables that for a given p and t, the quantity $\mathfrak{t}_1$ decreases with increasing k, whereas $\mathfrak{t}_2$ increases with k. Their sum $\mathfrak{t}$ decreases with increasing k when k is small, reaches a minimum at some optimal value of k, and increases with increasing k for values of k larger than the optimal value. In our experiments, we get the optimal value at around $k = 7$. The next three tables (4.4 through 4.6) represent average data for the heuristic B2. In this case, $\mathfrak{t}_1$, $\mathfrak{t}_2$ and $\mathfrak{t}$ vary as $\mathfrak{t}$ does in case of B1. That is, when we increase k, each of these three quantities decreases for small values of k, reaches a minimum at some optimal value of k and then increases with k. The optimal $\mathfrak{t}$ is obtained at around $k = 75$. Typical speed-ups obtained using our modifications over the basic method range from 2.5 to 2.9 for the heuristic B1 and from 1.3 to 1.5 for B2. B1, therefore, seems to perform better than B2. It

Table 4.3: Timings for heuristic B1 with $p = 29438018625539$ (A 45-bit prime)

	Basic method				Heuristic B1							
t	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	k	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	P_0	P_+	P_-
60	9.321	3.004	12.32	1042.0	1	4.165	1.864	6.030	464.41	39.7	41.5	18.7
					2	3.312	1.986	5.298	370.84	30.9	41.0	27.9
					3	2.684	2.045	4.729	388.66	27.8	42.3	29.7
					4	2.451	2.235	4.687	395.35	24.9	40.7	34.2
					5	2.312	2.478	4.790	349.30	23.6	41.3	35.0
					7	1.967	2.683	4.650	355.27	15.7	48.1	36.0
					10	2.130	3.749	5.880	461.11	17.9	38.8	43.1
75	5.123	2.042	7.165	712.14	1	2.077	1.132	3.210	392.66	39.0	40.9	19.9
					2	1.752	1.275	3.027	377.06	36.1	37.5	26.2
					3	1.486	1.364	2.850	309.73	28.0	42.4	29.5
					4	1.376	1.502	2.879	317.24	24.2	40.4	35.3
					5	1.143	1.463	2.607	334.16	19.7	45.2	34.9
					7	0.997	1.593	2.590	258.53	15.3	47.6	36.9
					10	0.969	2.041	3.010	315.68	17.9	34.6	47.3
90	3.121	1.472	4.593	695.79	1	1.396	0.890	2.287	318.05	37.9	42.3	19.6
					2	1.238	1.043	2.282	337.74	31.7	39.9	28.2
					3	0.912	0.970	1.883	273.60	26.0	41.9	31.9
					4	0.870	1.109	1.979	315.42	25.6	38.5	35.8
					5	0.734	1.086	1.821	254.70	19.4	44.7	35.7
					7	0.570	1.080	1.651	216.39	15.0	48.8	36.1
					10	0.641	1.565	2.206	323.70	16.2	38.6	45.1

remains undecided which heuristic yields better (i.e. faster) results when applied to large-scale problems.

We now heuristically justify the behavior of $\mathfrak{t}_1$ and $\mathfrak{t}_2$ as functions of k. $\mathfrak{t}_1$ counts the average time (per relation) needed to generate integers for checking B-smoothness. This involves discrete exponentiations (computations of g^j) and multi-precision additions or subtractions (computations of $\overline{g^j} + rp$ for non-zero r). Each such discrete exponentiation is much costlier than such an addition or a subtraction. As k is increased, the ratio of the number of discrete exponentiations to the number of additions or subtractions decreases. This leads to smaller values of $\mathfrak{t}_1$ for larger values of k. The other quantity $\mathfrak{t}_2$ counts the cost of the following: in case of B1,

(a) computation of $\rho_{0,i}$
(b) updation of the remainders $\rho_{r,i}$ for $r \neq 0$
(c) trial divisions by q_i for which $\rho_{r,i} = 0$

and in case of B2,

(d) computation of next divisibility indices at $-k$
(e) computation of $\mathfrak{L}$ in Eqn 4.1
(f) updation of the next divisibility indices

For the heuristic B1, (a) is costlier compared to (b) and (c), whereas for B2, (d) is costlier than (e) and (f). As k is increased, the operation (a) (resp. (d)) is carried out less frequently in comparison with the operations (b) and (c) (resp. (e) and (f)). As a result, we expect $\mathfrak{t}_2$ to decrease whenever we increase k. However, we

Table 4.4: Timings for heuristic B2 with $p = 19196459099$ (A 35-bit prime)

	Basic method				Heuristic B2							
t	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	k	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	P_0	P_+	P_-
50	0.348	0.122	0.471	41.043	25	0.039	0.320	0.360	30.899	9.03	40.9	50
					50	0.033	0.312	0.346	23.958	6.82	42.0	51.1
					75	0.031	0.292	0.323	26.267	5.04	47.1	47.8
					100	0.030	0.326	0.357	23.001	3.67	48.4	47.8
					200	0.039	0.392	0.431	29.39	3.80	47.1	49.0
					300	0.039	0.408	0.448	34.015	1.99	47.8	50.1
					500	0.048	0.502	0.550	40.032	2.22	44.5	53.1
60	0.239	0.099	0.338	34.973	25	0.027	0.259	0.286	27.591	8.83	43.6	47.5
					50	0.021	0.231	0.252	23.86	6.08	43.2	50.6
					75	0.020	0.225	0.246	21.79	3.78	46.2	50
					100	0.019	0.234	0.254	23.408	3.16	45.0	51.7
					200	0.023	0.270	0.293	26.251	1.99	46.5	51.4
					300	0.026	0.314	0.341	29.703	1.77	48.6	49.6
					500	0.033	0.399	0.433	33.159	1.28	51.1	47.5
75	0.164	0.083	0.247	45.911	25	0.016	0.190	0.206	32.881	7.69	42.7	49.5
					50	0.013	0.168	0.181	28.769	5.27	45.7	48.9
					75	0.013	0.175	0.188	28.028	3.12	46.1	50.7
					100	0.013	0.181	0.194	24.031	4.03	48.7	47.2
					200	0.013	0.197	0.211	27.005	1.66	47.2	51.1
					300	0.013	0.197	0.210	27.711	1.71	48.6	49.5
					500	0.017	0.262	0.280	35.372	1.28	49.0	49.6

see a different pattern of variation of $\mathfrak{t}_2$ in connection with both the heuristics. In particular, $\mathfrak{t}_2$ exhibits the expected pattern only in case of B2 and for small values of k. The unexpected behavior of $\mathfrak{t}_2$ can be accounted for from the following consideration. As we increase k, the bit-size of the integers $\overline{g^j} + rp$ (for $|r| \leqslant k$) increases. This leads to a smaller probability of finding B-smooth integers among the ones $\overline{g^j} + rp$. As a result more integers are checked to find a single relation. Since $\mathfrak{t}_2$ measures the cost of trial divisions for generating a relation, this too increases with the decrease in the above probability. The same argument holds for $\mathfrak{t}_1$ also and is corroborated by the behavior of $\mathfrak{t}_1$ for large values of k, typically $k > 300$ (See the tables for B2). Another quantity that tallies with this decrease of probability with increasing bit-size of the integers $\overline{g^j} + rp$ is P_0. If all integers in the set $\{\overline{g^j} + rp \mid -k \leqslant r \leqslant k\}$ had the same probability of being B-smooth, P_0 would be approximately $100/(2k+1)$. The tables show much larger values than this.

At any rate, our heuristics are motivated by the need to decrease the number of discrete exponentiations carried out during the generation of relations. This leads to integers with absolute value larger than p being subject to trial division. So we adopted certain tricks to bring down the cost associated with trial divisions. Our heuristics (in particular, B2) are useful when the cost of each discrete exponentiation is comparable with or more than the cost of a trial division by all primes in the factor base. This typically happens if the factor base size t is $\leqslant w^2 \lg p$ (which is $O(\lg^3 p)$), where w is the number of machine words needed to represent an integer having $\lg p$ bits.

Table 4.5: Timings for heuristic B2 with $p = 781487259479$ (A 40-bit prime)

t	Basic method				Heuristic B2							
	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	k	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	P_0	P_+	P_-
50	2.308	0.710	3.018	216.32	25	0.277	2.239	2.517	153.70	8.36	40.8	50.8
					50	0.244	2.112	2.357	159.17	6.55	42.6	50.7
					75	0.248	2.246	2.495	156.10	6.10	48.1	45.7
					100	0.231	2.165	2.396	151.83	3.92	45.8	50.2
					200	0.260	2.561	2.821	172.12	4.61	47.5	47.8
					300	0.283	2.873	3.157	184.45	1.61	47.7	50.6
					500	0.321	3.302	3.624	218.25	3.31	48.2	48.4
60	1.475	0.535	2.011	173.40	25	0.169	1.553	1.723	140.25	8.19	44.6	47.1
					50	0.127	1.281	1.409	126.42	6.24	42.5	51.2
					75	0.132	1.359	1.492	137.80	4.50	44.5	50.9
					100	0.140	1.540	1.681	124.39	2.94	43.9	53.0
					200	0.144	1.696	1.841	138.70	2.91	48.8	48.1
					300	0.156	1.815	1.971	160.40	2.19	49.7	48.0
					500	0.195	2.364	2.559	180.78	1.92	48.1	49.9
75	0.921	0.410	1.332	199.92	25	0.093	1.030	1.123	137.61	7.67	43.9	48.3
					50	0.079	0.955	1.035	114.44	6.74	43.9	49.2
					75	0.072	0.905	0.977	115.53	4.11	46.7	49.1
					100	0.073	0.962	1.035	110.61	4.16	48.2	47.5
					200	0.081	1.118	1.199	159.71	2.46	48.8	48.6
					300	0.084	1.205	1.290	129.6	2.84	50.9	46.2
					500	0.093	1.381	1.475	150.13	1.03	48.2	50.6

4.1.5 Open questions

We have shown both heuristically and experimentally that our heuristic ideas speed up the basic method considerably. Before we end, we raise some important theoretical questions that, if answered, would give better explanation of the performance of our modifications.

- Given that $\overline{g^j}$ does not factorize completely over the factor base B, what is the probability that at least one of $\overline{g^j} + rp$ for $r = \pm 1, \pm 2, \ldots, \pm k$ does for some pre-determined k?
- Can one find a j easily such that for a given k, the set $\{\overline{g^j} + rp \mid r = 0, \pm 1, \pm 2, \ldots, \pm k\}$ contains with a high probability at least one element that factorizes smoothly over the factor base?
- Can one find an expression for the optimum value of k for the heuristic methods B1 and B2 (i.e. the values of k that minimize the running times of the methods for given p and t)?

If we assume that the integers $\overline{g^j} + rp$ behave as *random* integers of absolute value $O(p)$, and if the factor base B comprises of primes less than $L[\beta]$, then Theorem 3.2 suggests that the probability of finding B-smooth integers among $\overline{g^j} + rp$ is of the order of $L[-1/2\beta]$. For practical situations, this probability is rather *low* and demands values of k higher than the optimal range found out experimentally.

Table 4.6: Timings for heuristic B2 with $p = 29438018625539$ (A 45-bit prime)

	Basic method				Heuristic B2							
t	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	k	$\mathfrak{t}_1$	$\mathfrak{t}_2$	$\mathfrak{t}$	$\mathfrak{T}_{\text{av}}$	P_0	P_+	P_-
60	9.321	3.004	12.32	1042.0	25	1.097	10.12	11.22	825.36	9.64	38.4	51.9
					50	0.950	9.651	10.60	724.68	6.85	43.5	49.6
					75	0.865	9.089	9.954	819.64	4.96	47.4	47.5
					100	0.854	9.423	10.27	783.33	7.13	42.7	50.0
					200	0.933	10.59	11.52	776.69	3.52	45.6	50.8
					300	0.987	11.50	12.49	884.44	2.33	43.8	53.8
					500	1.242	14.65	15.89	1151.5	2.29	47.4	50.2
75	5.123	2.042	7.165	712.14	25	0.562	6.070	6.632	627.40	10.5	42.0	47.4
					50	0.485	5.858	6.343	592.00	6.44	42.7	50.8
					75	0.415	5.207	5.623	504.61	5.59	44.8	49.5
					100	0.407	5.318	5.726	543.98	5.29	43.5	51.1
					200	0.424	5.715	6.140	646.06	2.94	48.7	48.2
					300	0.501	7.071	7.572	741.76	2.32	48.4	49.2
					500	0.541	7.859	8.401	839.67	1.86	50.0	48.0
90	3.121	1.472	4.593	695.79	25	0.332	4.139	4.472	611.08	6.87	42.8	50.2
					50	0.301	4.232	4.533	596.87	5.52	42.5	51.9
					75	0.274	4.083	4.358	491.58	4.09	46.0	49.8
					100	0.240	3.708	3.948	404.00	3.31	47.0	49.5
					200	0.273	4.451	4.725	586.19	2.58	47.8	49.5
					300	0.303	4.827	5.131	694.37	2.14	47.5	50.3
					500	0.318	5.545	5.864	628.55	2.12	47.2	50.5

4.2 Efficient implementation of the linear and cubic sieve methods

In this section, we delve into the details of our implementation of the linear sieve and the cubic sieve methods. The tricks that help us speed up the equation collecting phase of the sieve methods are very similar to those employed in the quadratic sieve method for integer factorization (See [16, 46, 121] for details).

4.2.1 Implementation of the linear sieve method

We start our discussion with the linear sieve method. We first recall from Section 3.2.2 that at the beginning of each sieving step, we find a solution for c_2 modulo q^h in the congruence $T(c_1, c_2) \equiv 0 \pmod{q^h}$ for every small prime q in the factor base and for a set of small exponents h. The costliest operation that need be carried out for each such solution is the computation of a modular inverse (namely, that of $H + c_1$ modulo q^h). As described in [73] and as is evident from our experiments too, calculations of these inverses take more than half of the CPU time needed for the entire equation collecting stage. Any trick that reduces the number of computations of the inverses, speeds up the algorithm.

One way to achieve this is to solve the congruence every time only for $h = 1$ and ignore all higher powers of q. That is, for every q (and c_1), we check which of the integers $T(c_1, c_2)$ are divisible by q and then add $\lg q$ to the corresponding indices of the array $\mathfrak{A}$. If some $T(c_1, c_2)$ is divisible by a higher power of q, this strategy fails to add $\lg q$ the required number of times. As a result, this $T(c_1, c_2)$, even if smooth, may fail to pass the 'closeness criterion' described in Section 3.2.2. This is, however, not a serious problem, because we may increase the cut-off from a value smaller than $\lg q_t$ to a value $\xi \lg q_t$ for some $\xi \geqslant 1$. This means that some non-smooth $T(c_1, c_2)$ will pass through the selection criterion in addition to some smooth ones that could not, otherwise, be detected. This is reasonable, because the non-smooth ones can be later filtered out from the smooth ones and one might use even trial divisions to do so. For primes p of less than 200 bits, values of $\xi \leqslant 2.5$ work quite well in practice [16, 121].

The reason why this strategy performs well in practice is as follows. If q is small, for example $q = 2$, we should add *only* 1 to $\mathfrak{A}_{c_2}$ for every power of 2 dividing $T(c_1, c_2)$. On the other hand, if q is much larger, say $q = 1299709$ (the 10^5th prime), then $\lg q \approx 20.31$ is *large*. But $T(c_1, c_2)$ would not be, in general, divisible by a *high* power of this q. The approximate calculation of logarithm of the smooth part of $T(c_1, c_2)$, therefore, leads to a situation where the probability that a smooth $T(c_1, c_2)$ is actually detected as smooth is quite high. A few relations would be still missed out even with the modified 'closeness criterion', but that is more than compensated by the speed-up gained by the method.

The above strategy helps us in a way other than by reducing the number of modular inverses. We note that for practical values of p, the small primes in the factor base are usually single-precision ones. As a result, the computation of d can be carried out using single-precision operations only.

We now compare the performance of the modified strategy with that of the original strategy for a value of p of length around 150 bits. This prime is chosen as a random one satisfying the conditions (i) $(p-1)/2$ is also a prime, and (ii) p is close to a whole cube. This second condition is necessary, because for such a prime, the cubic sieve method is also applicable, so that we can compare the performance of the two sieve methods for this prime.

Table 4.7: Performance of the linear sieve method
$p = 1320245474656309183513988729373583242842871683$
$t = 7000,\ M = 30000$

Algorithm	ξ	No. of Relations ($\bar{\rho}$)	No. of Variables ($\bar{\nu}$)	$\bar{\rho}/\bar{\nu}$	CPU Time (seconds)
Exact	0.1	108637	67001	1.6214	225590
Approximate	1.0	108215	67001	1.6151	101712
	1.5	108624	67001	1.6212	101818
	2.0	108636	67001	1.6214	102253
	2.5	108637	67001	1.6214	102250

In Table 4.7 we compare the performance of the 'exact' version of the algorithm (where all relations are made available by choosing values of $h \geqslant 1$) with that of the 'approximate' version of the algorithm (in which powers $h > 1$ are neglected). The CPU times listed in the table do not include the time for filtering out the 'spurious' relations obtained in the approximate version. It is evident from the table that the performance gain obtained using the heuristic variant is more than 2. It is also clear that values of ξ between 1.5 and 2 suffice for fields of this size.

4.2.2 Implementation of the cubic sieve method

For the cubic sieve method (Section 3.2.3), we employ similar strategies. That is, we solve the congruence $R(A,B,C) \equiv 0 \pmod{q}$ for each small prime q in the factor base and ignore higher powers of q that might divide $R(A,B,C)$. As before, we set the cut-off at $\xi \lg q_t$ for some $\xi \geqslant 1$. We are not going to elaborate the details of this strategy and the expected benefits once again. Instead we focus on the performance figures available from our experiments. As in the linear sieve, we work in the prime field $\mathbb{F}_p$ with

$$p = 1320245474656309183513988729373583242842871683$$

For this prime, we have $X = \lfloor \sqrt[3]{p} \rfloor + 1 = 1097029305312372$, $Y = 1$, $Z = 31165$ as a solution of (3.4).

We did not implement the 'exact' version of this algorithm in which one tries to solve (3.7) for exponents $h > 1$ of q. Table 4.8 lists the experimental details for the 'approximate' algorithm. (The meaning of the parameter λ will be explained in Section 4.4.) As in Table 4.7, the CPU times do not include the time for filtering out the spurious relations available by the more generous closeness criterion for the approximate algorithm. For the cubic sieve method, the values of ξ around 1.5 works quite well for our prime p.

Table 4.8: Performance of the cubic sieve method for various values of ξ
$p = 1320245474656309183513988729373583242842871683$
$t = 10000,\ M = 10000,\ \lambda = 1.5$

ξ	No. of Relations ($\bar{\rho}$)	No. of Variables ($\bar{\nu}$)	$\bar{\rho}/\bar{\nu}$	CPU Time (seconds)
1.0	54805	35001	1.5658	43508
1.5	54865	35001	1.5675	43336
2.0	54868	35001	1.5676	43492

4.2.3 Performance comparison between linear and cubic sieve methods

The speed-up obtained by the cubic sieve method over the linear sieve method is about 2.5 for the field of size around 150 bits. For larger fields, this speed-up is expected to be more. It is, therefore, evident that the cubic sieve method, at least for the case $\alpha = 1/3$, runs faster than the linear sieve counterpart for the practical range of sizes of prime fields.

4.3 Heuristic modification of the linear sieve method

We now describe a heuristic way of modifying the first stage of the linear sieve method for the computation of discrete logarithms over prime fields. Our heuristic allows us to build a factor base consisting of integers around square roots of several small multiples of p. The strategy reduces the average of the absolute value of the integers that are checked for smoothness with respect to the small primes in the factor base. This, in turn, leads to a larger density of smooth integers compared to the original method. On the other hand, our heuristic decreases the ratio of the number of relations to the number of variables and may lead to failure to get a full-rank system of linear congruences.

4.3.1 The heuristics

In the linear sieve method, we work with the quantities $H = \lfloor\sqrt{p}\rfloor + 1$, $J = H^2 - p$ and the bound M in the sieving interval (See Section 3.2.2). Let us now define, for any integer $r \geqslant 1$, the quantities: $H_r = \lfloor\sqrt{rp}\rfloor + 1$, $J_r = H_r^2 - rp$ (so that $H = H_1$ and $J = J_1$). The linear sieve method works exactly the way we described in Section 3.2.2 independent of the value of r we choose. In this case, however, $H_r \approx \sqrt{rp}$ and $J_r \leqslant 2\sqrt{rp}$. Therefore, if a value $r > 1$ is chosen, both H_r and J_r are $\sqrt{r}$ times the values H and J respectively. This increases the value of $T(c_1, c_2)$ by a factor of $\sqrt{r}$ and, thereby, reduces the chance of smooth factorization of this integer. As a result, we have to select a larger value of M in order to get sufficient number of relations.

To work around with this difficulty and at the same time to use the possibility of using different values of r, we propose the following heuristic variations of the linear sieve method. To start with, we select a *small* positive integer s and compute for each r, $1 \leqslant r \leqslant s$, the values of H_r and J_r as defined above. The factor base now comprises of primes less than $L[1/2]$ (as in the original version of the method) and integers $H_r + c$ for each $1 \leqslant r \leqslant s$ and $-\mu \leqslant c \leqslant \mu$, where μ is the bound on $|c|$ for each r in the modified method. Now for each value of r, we repeat the sieving procedure, that is, we collect relations involving the indices $\operatorname{ind}_g(q_i)$, $\operatorname{ind}_g(H_r + c_1)$ and $\operatorname{ind}_g(H_r + c_2)$ for $-\mu \leqslant c_1 \leqslant c_2 \leqslant \mu$.

In the original method we work with a factor base of size $2M + 1 + t$ and check the smoothness of $T(c_1, c_2)$ for approximately $2M^2$ pairs (c_1, c_2) with $c_1 \leqslant c_2$. If we apply our heuristic modification, the factor base size becomes $s(2\mu + 1) + t$ and the number of integers of the form

$$T_r(c_1, c_2) = J_r + (c_1 + c_2)H_r + c_1c_2$$

checked for smoothness (for all $1 \leqslant r \leqslant s$) becomes approximately $2\mu^2 s$.

HEURISTIC L1

Define the integer μ as $\mu = \lfloor\frac{M}{\sqrt{s}}\rfloor$, where M is chosen as in the original method described in Section 3.2.2. With this choice the total number of integers checked for smoothness remains the same as in the original method (viz. $2M^2$), whereas the factor base size increases from $2M + 1 + t$ to approximately $2M\sqrt{s} + s + t$.

HEURISTIC L2

The second alternative is to keep the factor base size same as in the original method. This can be achieved by taking $\mu = \lfloor\frac{M}{s}\rfloor$. With this choice of μ, the number of integers reduces approximately to $2M^2/s$.

We show in the next section that with both these choices of μ, the average of the absolute value of $T_r(c_1, c_2)$ decreases compared with the average of $|T(c_1, c_2)|$ in the original method. As a result, the probability that $T_r(c_1, c_2)$ factorizes smoothly over the first t primes is more than that for $T(c_1, c_2)$. We, therefore, expect to get more relations for a given number of (c_1, c_2) pairs.

All these do not come free. For the first heuristic, the number of variables (i.e. the number of factor base elements) increases by approximately a factor of $\sqrt{s}$. For the second, on the other hand, the number of pairs (c_1, c_2) decreases by a factor of s. These variations seem unimportant asymptotically at least for small values of s. In practice, however, one might fail to get a system with more equations than unknowns for values of $s > 1$, while the original strategy with corresponding values of t and M produces a full-rank linear system of congruences. In particular, one must not choose s to be quite large.

4.3.2 Analysis of the heuristics

In this section, we prove that the average value of $|T(c_1, c_2)|$ over all possible combinations of c_1 and c_2 is larger than the average value of $|T_r(c_1, c_2)|$ over all possible combinations of r, c_1 and c_2. For the original method, we have calculated the average value of $|T(c_1, c_2)|$ in Section 3.3.1. From Result 3.4, we write this average value as

$$\overline{T} \approx \frac{2MH}{3} \approx \frac{2M\sqrt{p}}{3}$$

For the heuristic modifications, we can proceed similarly and prove that the average of $|T_r(c_1, c_2)|$ over all choices of r, c_1 and c_2 is

$$\overline{T}_{\text{heu}} \approx \frac{2\mu(H_1 + H_2 + \ldots + H_s)}{3s} \approx \frac{2\mu(\sqrt{1} + \sqrt{2} + \ldots + \sqrt{s})\sqrt{p}}{3s}$$

The proportion of these average values is

$$\mathfrak{r} = \frac{\overline{T}_{\text{heu}}}{\overline{T}} \approx \frac{\mu}{M} \frac{\sqrt{1} + \sqrt{2} + \ldots + \sqrt{s}}{s}$$

For the heuristic L1, we have $\mu \approx \frac{M}{\sqrt{s}}$, so that $\mathfrak{r} \approx \frac{\sqrt{1}+\sqrt{2}+\ldots+\sqrt{s}}{s\sqrt{s}}$. Clearly, $\mathfrak{r} < 1$. In fact, $\mathfrak{r}$ approaches $\frac{2}{3}$ as s tends to ∞. For the heuristic L2, on the other hand, $\mu \approx \frac{M}{s}$, so that $\mathfrak{r} \approx \frac{\sqrt{1}+\sqrt{2}+\ldots+\sqrt{s}}{s^2} < \frac{1}{\sqrt{s}}$ and approaches to zero as s tends to ∞.

4.3.3 Experimental results

In this section, we compare typical timings and number of relations obtained in the first stage of the linear sieve method with those obtained from our heuristic modifications. We report the results available from the 'exact' version of the algorithm. (See Section 4.2.1 for the meaning of 'exact' in the last sentence.)

We experimented in the prime field $\mathbb{F}_p$ with

$$p = 38275450020766122418475251523827352087$$

This is a randomly generated prime of length 125 bits, for which $(p - 1)/2$ is a prime. The parameters t and M are selected slightly larger than the optimal values so that the number of relations available is about twice the size of the factor base (for the original method). In the following tables we illustrate how the number of relations generated by our heuristic schemes varies with the additional parameter s introduced at the beginning of this section. The case $s = 1$ corresponds to the original method. We did not try to solve the resulting systems, neither did we make an attempt to check the ranks of the systems. We allowed s to increase as long as we get sufficiently more relations than the number of variables (size of the factor base). The tables also list the total CPU time taken by the execution of the relation-collecting stage of the method.

Table 4.9: Performance of heuristic L1
$p = 38275450020766122418475251523827352087$, $t = 3000$, $M = 12500$

	Size of the	No. of relations						Total time
s	factor base	$r = 1$	$r = 2$	$r = 3$	$r = 4$	$r = 5$	Total	(Seconds)
1	28001	62244					62244	36632
2	38354	34380	36347				70727	46742
3	46299	24194	25726	23968			73888	53725
4	53004	18908	19996	18828	19659		77391	59422
5	58905	15585	16441	15554	16162	14251	77993	64284

Table 4.9 shows the performance of the first heuristic. We see the expected increase in the total number of relations generated, as we increase s. However, this increase in the number of relations seems to reach a saturation for $s \geqslant 4$. It is also evident that the ratio of the number of relations to the number of variables decreases with increasing s. For larger values of s, say, for $s = 10$ (not shown in the table), our heuristic fails to generate more relations than the size of the factor base. The value $s = 3$ appears to be an optimal choice. As we increase s, the running time increases too, but at a rate smaller than the increase in the size of the factor base. For the case $s = 5$, for example, we generate relations for a factor base whose size is 2.104 times that in the original case ($s = 1$), whereas the time we spend to achieve this is about 1.755 times that for the original method.

Table 4.10: Performance of heuristic L2
$p = 38275450020766122418475251523827352087$, $t = 3000$, $M = 12500$

	Size of the	No. of relations				Total time
s	factor base	$r = 1$	$r = 2$	$r = 3$	Total	(Seconds)
1	28001	62244			62244	36632
2	28002	18908	19996		38904	29937
3	27999	9358	9908	9400	28666	26321

For the second heuristic (see Table 4.10), the problem of not having a system with more equations than unknowns becomes more acute, as we increase s. As a result, the largest values of s allowed by the second heuristic are smaller than those allowed by the first.

4.4 Heuristic modification of the cubic sieve method

We recall from Lemma 3.5 that we check the smoothness of $R(A, B, C)$ for $-M \leqslant A \leqslant B \leqslant C \leqslant M$. With this condition, C varies from 0 to M. We note that for each value of C, we have to execute the entire sieving operation once. For each such sieving operation (that is, for a fixed C), the sieving interval for B is (i.e. the admissible values of B are) $-C/2 \leqslant B \leqslant \min(C, M - C)$. Correspondingly $A = -(B + C)$ can vary from $\max(-2C, -M)$ to $-C/2$. It is easy to see that in this case total number of triples (A, B, C) for which the smoothness of $R(A, B, C)$

is examined is $\tau = \sum_{C=0}^{M} \Big(C/2 + \min(C, M-C) \Big) \approx M^2/2$. The number of unknowns, that is, the size of the factor base, on the other hand, is $\nu \approx 2M + t$.

4.4.1 The heuristic

If we remove the restriction $A \geqslant -M$ and allow A to be as negative as $-\lambda M$ for some $1 < \lambda \leqslant 2$, then we benefit in the following way. As before, we allow C to vary from 0 to M keeping the number of sieving operations fixed. Since A can now assume values smaller than $-M$, the sieving interval increases to $-C/2 \leqslant B \leqslant \min(C, \lambda M - C)$. As a result, the total number of triples (A, B, C) becomes $\tau_\lambda = \sum_{C=0}^{M} \Big(C/2 + \min(C, \lambda M - C) \Big) \approx \frac{M^2}{4}(4\lambda - \lambda^2 - 1)$, whereas the size of the factor base increases to $\nu_\lambda \approx (\lambda + 1)M + t$. (Note that with this notation the value $\lambda = 1$ corresponds to the original method and $\tau = \tau_1$ and $\nu = \nu_1$.) The ratio τ_λ/ν_λ is approximately proportional to the number of smooth integers $R(A, B, C)$ generated by the method divided by the number of unknowns. Therefore, λ should be set at a value for which this ratio is maximum. If one treats t and M as constants, then the maximum is attained at $\lambda^* = -U + \sqrt{U^2 + 4U + 1}$, where $U = \frac{M+t}{M} = 1 + \frac{t}{M}$. As we increase U from 1 to ∞ (or, equivalently the ratio t/M from 0 to ∞), the value of λ^* increases monotonically from $\sqrt{6} - 1 \approx 1.4495$ to 2. (See Appendix A for detailed calculations.) In Table 4.11, we summarize the variation of τ_λ/ν_λ for some values of U. These values of U correspond from left to right to $t \ll M$, $t \approx M/2$, $t \approx M$ and $t \approx 2M$ respectively. The corresponding values of λ^* are respectively 1.4495, 1.5414, 1.6056 and 1.6904. It is clear from the table, that for practical ranges of values of U, the choice $\lambda = 1.5$ gives performance quite close to the optimal.

Table 4.11: Variation of τ_λ/ν_λ with λ

	τ_λ/ν_λ (approx)			
λ	$U = 1$	$U = 1.5$	$U = 2$	$U = 3$
1	$0.2500\,M$	$0.2000\,M$	$0.1667\,M$	$0.1250\,M$
1.5	$0.2750\,M$	$0.2292\,M$	$0.1964\,M$	$0.1527\,M$
2	$0.2500\,M$	$0.2143\,M$	$0.1875\,M$	$0.1500\,M$
λ^*	$0.2753\,M$	$0.2293\,M$	$0.1972\,M$	$0.1548\,M$

We note that this scheme keeps M and the range of variation of C constant and hence does not increase the number of sieving steps and, in particular, the number of modular inverses and square roots. It is, therefore, advisable to apply the trick (with, say, $\lambda = 1.5$) instead of increasing M. With that one is expected to get a speed-up of about 10 to 20%.

4.4.2 Experimental results

We work in $\mathbb{F}_p$ with the 150-bit prime p of Section 4.2. In Table 4.12, we fix ξ at 1.5 and tabulate the variation of the performance of the cubic sieve method for some values of λ. (The parameter ξ is defined in Section 4.2.) It is clear from the table that among the cases observed, the largest value of the ratio $\bar{\rho}/\bar{\nu}$ is obtained at $\lambda = 1.5$. (The theoretical maximum is attained at $\lambda \approx 1.6$) We also note that changing the value of λ incurs variation in the running time by at most 1%. Thus our heuristic allows us to build a larger database at approximately no extra cost.

Table 4.12: Performance of the cubic sieve method for various values of λ
$p = 132024547465630918351398872937358324284287l683$
$t = 10000,\ M = 10000,\ \xi = 1.5$

	No. of	No. of		CPU Time
λ	Relations ($\bar{\rho}$)	Variables ($\bar{\nu}$)	$\bar{\rho}/\bar{\nu}$	(seconds)
1.0	43434	30001	1.4478	43047
1.5	54865	35001	1.5675	43336
1.6	56147	36001	1.5596	43347
2.0	58234	40001	1.4558	43499

4.4.3 Effect of the heuristic on $\overline{R}$, $R_{\max}$ and $\mathfrak{d}(\eta)$

We recall from Section 3.3 that $\overline{R}$ and $R_{\max}$ denote the average and maximum values of $|R(A,B,C)|$ as A, B, C run over all possible triples with $A+B+C=0$, $A \leqslant B \leqslant C$. The distribution function $\mathfrak{d}(\eta)$ is introduced in Definition 3.9. We now investigate the effect of our heuristic modification, namely $-\lambda M \leqslant A$, on these quantities. Since we are experimenting with a prime *close* to a cube, so that $Y = 1$, we consider the case $Y \ll X/M$ and write the approximate value of $\overline{R}$ as

$$\overline{R} \approx X \cdot \left(\sum_{C=0}^{M} \sum_{B=-C/2}^{\min(C,\lambda M - C)} (B^2 + BC + C^2) \right) \Bigg/ \left(\sum_{C=0}^{M} \sum_{B=-C/2}^{\min(C,\lambda M - C)} 1 \right) \quad (4.2)$$

The denominator equals τ_λ and is shown in Section 4.4.1 to be approximately equal to $\frac{M^2}{4}(4\lambda - \lambda^2 - 1)$. The numerator evaluates to $\frac{M^4}{1536}(-151\lambda^4 + 512\lambda^3 - 384\lambda^2 + 512\lambda - 160)$. The maximum of $|R(A,B,C)|$ can be easily shown to be obtained at $C = M, B = (\lambda - 1)M, A = -\lambda M$. We, therefore, have

RESULT 4.1 For the heuristic modification of the cubic sieve method, $\overline{R}$ and $R_{\max}$ can be written in terms of λ as

$$\overline{R} \approx \left[\frac{-151\lambda^4 + 512\lambda^3 - 384\lambda^2 + 512\lambda - 160}{384(4\lambda - \lambda^2 - 1)} \right] \cdot M^2 X$$

$$R_{\max} \approx (\lambda^2 - \lambda + 1) M^2 X$$

In Table 4.13, we list the values of $\overline{R}$, $R_{\max}$ and $\overline{R}/R_{\max}$ for $\lambda = 1$ (the original method), $\lambda = 1.5$ (the recommended value) and $\lambda = 2$. We see that increasing λ increases both the average value $\overline{R}$ and the maximum value $R_{\max}$. However, the increase in $R_{\max}$ is more than the increase in $\overline{R}$. Therefore, the ratio $\overline{R}/R_{\max}$ decreases with increasing λ.

We end this section by a graphic description of $\mathfrak{d}(\eta)$. Similar to the original method, we compute $\mathfrak{D}(\eta)$ from the sum

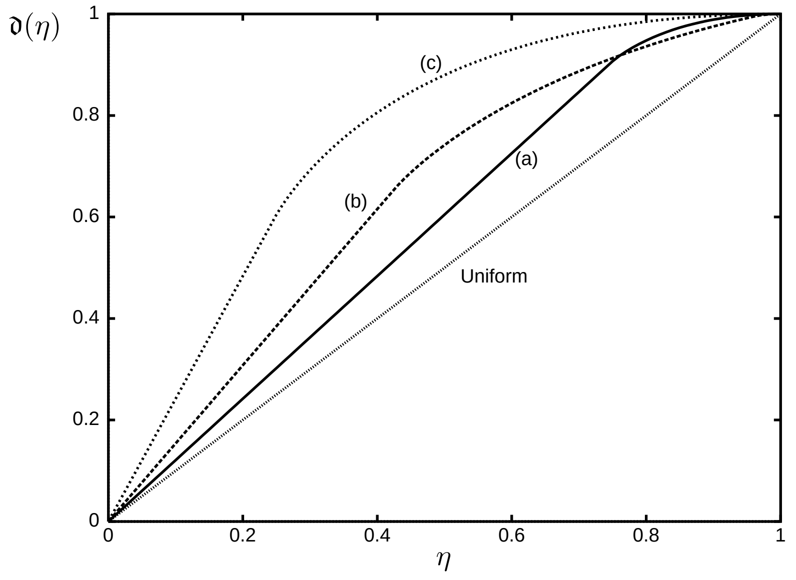
Table 4.13: Variation of $\overline{R}$, $R_{\max}$ and $\overline{R}/R_{\max}$ with λ

λ	$\overline{R}$	$R_{\max}$	$\overline{R}/R_{\max}$
1.0	$0.43M^2X$	M^2X	0.43
1.5	$0.67M^2X$	$1.75M^2X$	0.38
2.0	$0.88M^2X$	$3M^2X$	0.29

$$\begin{aligned}\mathfrak{D}(\eta) \;\approx\; & \sum_{C=0}^{\min\left(M,\left\lfloor M\sqrt{4\eta(\lambda^2-\lambda+1)/3}\right\rfloor\right)} \Bigg[1+\lfloor C/2\rfloor + \\ & \min\left(C, M-C, \left\lfloor -C/2+\sqrt{\eta(\lambda^2-\lambda+1)M^2-3C^2/4}\right\rfloor\right)\Bigg]\end{aligned}$$

and $\mathfrak{d}(\eta)=\mathfrak{D}(\eta)/\mathfrak{D}(1)$, where $\mathfrak{D}(1)\approx(4\lambda-\lambda^2-1)M^2/4$.

Figure 4.1: Variation of $\mathfrak{d}(\eta)$ for the modified cubic sieve method
(a) $\lambda=1.0$ (b) $\lambda=1.5$ (c) $\lambda=2.0$



From the above figure we see that as we increase λ, the curve for $\mathfrak{d}(\eta)$ shifts upwards. This phenomenon is corroborated by the decrease of $\overline{R}/R_{\max}$ with increasing λ.

The details of the calculations of this section are given in Appendix B.

Appendix A Determination and properties of λ^*

We recall from Section 4.4.1 that for our heuristic modification of the cubic sieve method, we have

$$\begin{aligned} \tau_\lambda &= \text{Number of triples for which } R(A,B,C) \text{ is tested for smoothness} \\ &\approx \frac{M^2}{4}(4\lambda - \lambda^2 - 1) \\ \nu_\lambda &= \text{Size of the factor base} \\ &\approx (\lambda+1)M + t \end{aligned}$$

We want to maximize

$$f(\lambda) = \tau_\lambda / \nu_\lambda = \frac{M^2}{4}\left[\frac{4\lambda - \lambda^2 - 1}{(\lambda+1)M + t}\right]$$

In order to do so, we compute $f'(\lambda)$:

$$f'(\lambda) = -\frac{M^3}{4}\left[\frac{\lambda^2 + 2\left(\frac{M+t}{M}\right)\lambda - 4\left(\frac{M+t}{M}\right) - 1}{((\lambda+1)M + t + 1)^2}\right]$$

If we write $U = \frac{M+t}{M}$, we see that $f'(\lambda)$ has two zeros at $-U \pm \sqrt{U^2 + 4U + 1}$. Since λ is positive in the region of our interest, we have $\lambda^* = -U + \sqrt{U^2 + 4U + 1}$. It is not difficult to see that at this value of λ^*, we have $f''(\lambda) < 0$, so that $f(\lambda)$ is maximum at $\lambda = \lambda^* = -U + \sqrt{U^2 + 4U + 1}$.

We now deduce some properties of λ^*. First we note that t can vary from 0 to $+\infty$ and, therefore, $U = \frac{M+t}{M}$ varies from 1 to $+\infty$. We now prove the following

LEMMA 4.2 As U varies from 1 to $+\infty$, $\lambda^* = -U + \sqrt{U^2 + 4U + 1}$ increases monotonically from $\sqrt{6} - 1$ to 2. In particular, for all $1 \leqslant U \leqslant \infty$, we have $1 \leqslant \lambda^* \leqslant 2$.

Proof We have

$$\frac{\mathrm{d}\lambda^*}{\mathrm{d}U} = -1 + \sqrt{\frac{U^2 + 4U + 4}{U^2 + 4U + 1}} > 0$$

for all $U \geqslant 1$. Therefore, λ^* increases monotonically with U for $U \geqslant 1$. For $U = 1$, $\lambda^* = \sqrt{6} - 1$. As U increases, the quantity $\sqrt{U^2 + 4U + 1}$ tends to the quantity $\sqrt{U^2 + U + 4} = U + 2$. Hence $\lim_{U \to \infty} = 2$. ■

The above lemma guarantees that for all values of U, we get the optimal value λ^* in the region where λ is defined, namely, $1 \leqslant \lambda \leqslant 2$. However, note that increasing λ increases $\overline{R}$ and thereby reduces the fraction of smooth integers among $R(A,B,C)$. (See Result 4.1 and Table 4.13.) We have experimentally verified that taking $\lambda = 1.5$ works quite well in practical situations.

Appendix B Calculations of $\overline{R}$, $R_{\max}$ and $\mathfrak{D}(\eta)$

In this section, we derive the expressions for $\overline{R}$, $R_{\max}$ and $\mathfrak{D}(\eta)$ for our heuristic modification of the cubic sieve method. We consider the case $Y \ll X/M$ only. In particular, we have experimented with $Y = 1$, so that this case applies to the results and observations we have reported. Finally note that putting $\lambda = 1$ gives the expressions for the original cubic sieve method – the ones which we purposefully omitted in the appendix to the previous chapter.

B.1 Calculation of $\overline{R}$

For the modified cubic sieve method, C varies from 0 to M. For a given C, B varies from $-\lfloor C/2 \rfloor$ to $\min(C, \lambda M - C)$. Since we have assumed $Y \ll X/M$, we approximate $R(A,B,C)$ as $|R(A,B,C)| \approx |AB + AC + BC|X = (B^2 + BC + C^2)X$. This leads to the following approximate value of $\overline{R}$.

$$\overline{R} = X \cdot \left(\sum_{C=0}^{M} \sum_{B=-\lfloor C/2 \rfloor}^{\min(C,\lambda M - C)} B^2 + BC + C^2 \right) \Big/ \left(\sum_{C=0}^{M} \sum_{B=-\lfloor C/2 \rfloor}^{\min(C,\lambda M - C)} 1 \right) \quad (4.3)$$

The denominator is equal to τ_λ in Appendix A and can be evaluated as follows. With the observation that $C \leqslant \lambda M - C$ if and only if $C \leqslant \lambda M/2$, we have

$$\begin{aligned} & \sum_{C=0}^{M} \sum_{B=-\lfloor C/2 \rfloor}^{\min(C,\lambda M - C)} 1 \\ = \; & \sum_{C=0}^{\lfloor \lambda M/2 \rfloor} (\lfloor C/2 \rfloor + C) \; + \sum_{C=\lfloor \lambda M/2 \rfloor + 1}^{M} (\lfloor C/2 \rfloor + \lambda M - C) \\ \approx \; & \frac{M^2}{4}(4\lambda - \lambda^2 - 1) \end{aligned}$$

The sum in the numerator of (4.3), on the other hand, can be written as

$$\sum_{C=0}^{\lfloor \lambda M/2 \rfloor} \sum_{B=-\lfloor C/2 \rfloor}^{C} (B^2 + BC + C^2) \; + \sum_{C=\lfloor \lambda M/2 \rfloor + 1}^{M} \sum_{B=-\lfloor C/2 \rfloor}^{\lambda M - C} (B^2 + BC + C^2) \quad (4.4)$$

The former sum in the last expression equals

$$\begin{aligned} & \sum_{C=0}^{\lfloor \lambda M/2 \rfloor} \Big[\Big(1^2 + 2^2 + \ldots (\lfloor C/2 \rfloor)^2\Big) + \Big(1^2 + 2^2 + \ldots + C^2\Big) + \\ & \qquad C\left((\lfloor C/2 \rfloor + 1) + (\lfloor C/2 \rfloor + 2) + \ldots + C \right) + C^2(C + \lfloor C/2 \rfloor + 1) \Big] \\ \approx \; & \frac{21}{512}\lambda^4 M^4 + O(M^3) \end{aligned}$$

and the second sum in (4.4) equals

$$\begin{aligned} & \sum_{C=\lfloor \lambda M/2 \rfloor + 1}^{M} \Big[\Big(1^2 + 2^2 + \ldots + (\lfloor C/2 \rfloor)^2\Big) + \Big(1^2 + 2^2 + \ldots + (\lambda M - C)^2\Big) \\ & + \; C[(-\lfloor C/2 \rfloor) + (-\lfloor C/2 \rfloor + 1) + \ldots + (-1) + 1 + 2 + \ldots + (\lambda M - C)] \\ & + \; C^2(\lfloor C/2 \rfloor + 1 + \lambda M - C) \Big] \end{aligned}$$

$$\approx \frac{1}{768}\left(-107\lambda^4 + 256\lambda^3 - 192\lambda^2 + 256\lambda - 80\right) M^4 + O(M^3)$$

Adding these two sums gives the value of the numerator of (4.3) as

$$\left(\frac{1}{1536}\left[-151\lambda^4 + 512\lambda^3 - 384\lambda^2 + 512\lambda - 160\right] M^4 + O(M^3)\right) X$$

Therefore,

$$\overline{R} \approx \left(\frac{-151\lambda^4 + 512\lambda^3 - 384\lambda^2 + 512\lambda - 160}{384(4\lambda - \lambda^2 - 1)}\right) M^2 X$$

B.2 Calculation of $R_{\max}$

For a fixed C, the expression

$$|R(A,B,C)| \approx (B^2 + BC + C^2)X = ((B + C/2)^2 + 3C^2/4)X$$

increases with B and thus attains the maximum value of $3C^2X$ for $0 \leqslant C \leqslant \lambda M/2$ and $((\lambda M - C)^2 + (\lambda M - C)C + C^2)X = (\lambda^2 M^2 - \lambda MC + C^2)X = ((C - \lambda M/2)^2 + 3\lambda^2/4)X$ for $\lambda M/2 \leqslant C \leqslant M$. Now if we vary C, we see that the first expression reaches the maximum value of $\frac{3}{4}\lambda^2 M^2 X$ at $C = \lambda M/2$, whereas the second expression reaches the maximum value of $(\lambda^2 - \lambda + 1)M^2 X$ at $C = M$. Now $\frac{3}{4}\lambda^2 M^2 X > (\lambda^2 - \lambda + 1)M^2 X$ if $(\lambda - 2)^2 < 0$ which is impossible for any real λ. Therefore,

$$R_{\max} \approx (\lambda^2 - \lambda + 1)M^2 X$$

B.3 Calculation of $\mathfrak{D}(\eta)$

The condition $|R(A,B,C)| \leqslant \eta R_{\max}$ demands

$$(B + C/2)^2 \leqslant \eta(\lambda^2 - \lambda + 1)M^2 - 3C^2/4 \tag{4.5}$$

If the right side of the inequality (4.5) is negative, that is, if C is larger than $M\sqrt{4\eta(\lambda^2 - \lambda + 1)/3}$, then no values of B satisfy (4.5). On the other hand, if $C \leqslant M\sqrt{4\eta(\lambda^2 - \lambda + 1)/3}$, then (4.5) is satisfied by all B satisfying $0 \leqslant B + C/2 \leqslant \sqrt{\eta(\lambda^2 - \lambda + 1)M^2 - 3C^2/4}$, that is, $-C/2 \leqslant B \leqslant -C/2 + \sqrt{\eta(\lambda^2 - \lambda + 1)M^2 - 3C^2/4}$. In addition B satisfies $-C/2 \leqslant B \leqslant \min(C, \lambda M - C)$. Combining these results gives the value of $\mathfrak{D}(\eta)$ as

$$\mathfrak{D}(\eta) \approx \sum_{C=0}^{\min\left(M, \left\lfloor M\sqrt{4\eta(\lambda^2-\lambda+1)/3}\right\rfloor\right)} \Bigg[1 + \lfloor C/2 \rfloor + \min\left(C, M - C, \left\lfloor -C/2 + \sqrt{\eta(\lambda^2 - \lambda + 1)M^2 - 3C^2/4}\right\rfloor\right)\Bigg]$$

In Figure 4.1, we plot $\mathfrak{d}(\eta) = \mathfrak{D}(\eta)/\mathfrak{D}(1)$ for $\lambda = 1, 1.5, 2$ and $M = 1000$.

5 On the congruence $X^3 \equiv Y^2 Z \pmod{p}$

In Section 3.2.3, we introduced the cubic sieve method for the computation of discrete logarithms over a prime field $\mathbb{F}_p$. The working of this method is based on the availability of a solution of the congruence

$$X^3 \equiv Y^2 Z \pmod{p} \tag{5.1}$$

with X, Y, Z of the order of p^α for some $\frac{1}{3} \leqslant \alpha < \frac{1}{2}$. We are interested in solutions with $X^3 \neq Y^2 Z$. Henceforth, we denote by 'the cubic sieve congruence' or by CSC for brevity, the congruence specified by (5.1).

This chapter is devoted to a study of the solutions of the CSC. In the Section 5.1, we introduce some notations and results from analytic number theory, that we use throughout the chapter. In Section 5.2, we deduce that the number of solutions of the CSC (with or without the inequality $X^3 \neq Y^2 Z$) is $\Theta(p^2)$. In Section 5.3, we provide a heuristic estimate of the number of solutions of the CSC subject to the condition $X, Y, Z \leqslant p^\alpha, X^3 \neq Y^2 Z$. We show that for sufficiently large p, a value of α, $1/3 < \alpha < 1/2$, is expected to give at least a solution of the CSC with $X, Y, Z \leqslant p^\alpha$. Our argument is not to be taken as a proof for the existence of a solution. It heuristically justifies that for sufficiently large primes, one is *expected* to have a desired solution of the CSC. Some small-scale experiments carried out by us provide evidence in favor of our claim regarding this asymptotic expected value. Indeed our experimental results tally quite closely with the theoretical estimates up to a constant factor. We finally emphasize that our demonstration is *not* procedural in the sense that it does not lead to an algorithm for finding a solution when one exists.

5.1 Some results from analytic number theory

In this section, we introduce some notions and results from analytic number theory, that we use throughout the chapter. For details we refer the reader to any introductory text book on analytic number theory, for example [4, Chapter 3].

DEFINITION 5.1 A real- or complex-valued function defined on the set $\mathbb{N}$ of natural numbers is called an *arithmetic function*.

If f is an arithmetic function, it is often possible to extend the domain of definition of f to the set of all positive real numbers such that the restriction of f to $\mathbb{N}$ is the given arithmetic function. Certain results require f to have a *continuous derivative* $f'(x)$ for all $x \in \mathbb{R}$ or at least for $x \in [a, b]$ for some $0 < a < b$. In such a case, we can evaluate the sum $\sum_{a<n\leqslant b} f(n)$ by evaluating an integral as follows: (Here the sum extends over all integers n, $a < n \leqslant b$.)

THEOREM 5.2 [Euler's summation formula] If f has a continuous derivative f' in the closed interval $[a, b]$, where $0 < a < b$, then

$$\begin{aligned}\sum_{a<n\leqslant b} f(n) &= \int_a^b f(t)\mathrm{d}t + \int_a^b (t - \lfloor t \rfloor) f'(t)\mathrm{d}t \\ &\quad + f(b)(\lfloor b \rfloor - b) - f(a)(\lfloor a \rfloor - a)\end{aligned}$$

The following results are easy consequences of Euler's summation formula:

THEOREM 5.3 For $x \geqslant 1$ and for real values of s, we have:

(a) $\displaystyle\sum_{1\leqslant n\leqslant x} \frac{1}{n} = \ln x + \gamma + O\left(\frac{1}{x}\right)$

(b) $\displaystyle\sum_{1\leqslant n\leqslant x} \frac{1}{n^s} = \frac{x^{1-s}}{1-s} + \zeta(s) + O(x^{-s}) \quad \text{if } s > 0, s \neq 1$

(c) $\displaystyle\sum_{n>x} \frac{1}{n^s} = O(x^{1-s}) \quad \text{if } s > 1$

(d) $\displaystyle\sum_{1\leqslant n\leqslant x} n^s = \frac{x^{s+1}}{s+1} + O(x^s) \quad \text{if } s \geqslant 0$

In the above theorem, γ is the Euler constant defined by

$$\gamma = \lim_{n\to\infty} \left(\frac{1}{1} + \frac{1}{2} + \ldots + \frac{1}{n} - \ln n\right) = 0.57721566\ldots$$

and $\zeta(s)$ is the Riemann zeta function defined for all real $s > 0, s \neq 1$ as

$$\zeta(s) = \begin{cases} \displaystyle\sum_{n=1}^{\infty} \frac{1}{n^s} & \text{if } s > 1 \\ \displaystyle\lim_{x\to\infty} \left(\sum_{n=1}^{\infty} \frac{1}{n^s} - \frac{x^{1-s}}{1-s}\right) & \text{if } 0 < s < 1 \end{cases}$$

We now define for an integer $n \in \mathbb{N}$ the integer $d(n)$ to be the total number of (positive integral) divisors of n. Then Theorem 5.3 gives the following:

THEOREM 5.4 For all real $x \geqslant 1$, we have

$$\sum_{n\leqslant x} d(n) = x \ln x + (2\gamma - 1)x + O(\sqrt{x})$$

5.2 Total number of solutions of the CSC

To start with let us introduce a few notations related to the set of solutions of the CSC.

$$S = \{(X, Y, Z) \mid X^3 \equiv Y^2 Z \pmod{p},\ 1 \leqslant X, Y, Z < p\}$$

$$\begin{aligned}
S_= &= \{(X,Y,Z) \in S \mid X^3 = Y^2 Z\} \\
S_{\neq} &= \{(X,Y,Z) \in S \mid X^3 \neq Y^2 Z\} \\
S_\alpha &= \{(X,Y,Z) \in S_{\neq} \mid 1 \leqslant X,Y,Z \leqslant p^\alpha\}
\end{aligned}$$

For the cubic sieve method, we are not interested in solutions of the CSC in $S_=$. However, it is easy to estimate the cardinality of $S_=$. This, in turn, gives the cardinality of $S_{\neq}$. We also remark that the sets S_α for $1/3 \leqslant \alpha < 1/2$ are extremely important for the cubic sieve method. In fact, the smallest possible value of α for which S_α is non-empty, determines the running time of the cubic sieve method.

It turns out that the set S under coordinate-wise multiplication modulo p is a group with identity $(1,1,1)$ and $(X,Y,Z)^{-1} = (X^{-1}, Y^{-1}, Z^{-1})$, where the inverses of X, Y and Z are modulo p. Since $(1,1,1) \notin S_{\neq}$, $S_{\neq}$ is never a subgroup of S. The same argument holds for the sets S_α. For $X^3 = Y^2 Z$, it is not necessary that $(X^{-1})^3 = (Y^{-1})^2 Z^{-1}$ and thus $S_=$ is also not a subgroup of S. At any rate, these facts do not seem to have a bearing on the material that follows in this chapter.

In this section, we derive the cardinalities of S, $S_=$ and $S_{\neq}$. We will discuss about the cardinalities of the sets S_α in the next section.

5.2.1 Cardinality of S

For each value of $X, Y \in \mathbb{F}_p^*$, we have a unique solution for $Z \in \mathbb{F}_p^*$ satisfying the CSC. Therefore

$$\#S = (p-1)^2 = \Theta(p^2) \tag{5.2}$$

5.2.2 Cardinality of $S_=$

Choose $1 \leqslant X < p$ and a solution $(X,Y,Z) \in S_=$. Let the prime factorization of X be $X = p_1^{\beta_1} p_2^{\beta_2} \dots p_r^{\beta_r}$, where p_i are distinct primes and $\beta_i > 0$. Therefore, $Y^2 Z = X^3 = p_1^{3\beta_1} p_2^{3\beta_2} \dots p_r^{3\beta_r}$. Since $Y^2 | X^3$, for each $i = 1, \dots, r$, the power of p_i dividing Y must be one of $0, 1, 2, \dots, \lfloor 3\beta_i/2 \rfloor$. Some choices of these powers lead to $Y > p$. We neglect this for the time being and see that for the given X, total number of choices for Y (and hence for Z) is

$$\begin{aligned}
&\leqslant \prod_{i=1}^{r} (1 + \lfloor 3\beta_i/2 \rfloor) \\
&\leqslant \prod_{i=1}^{r} (1 + 3\beta_i/2) \\
&\leqslant \frac{3}{2} \prod_{i=1}^{r} (1 + \beta_i) \\
&= \frac{3}{2} d(X)
\end{aligned}$$

If we sum this quantity over all $1 \leqslant X < p$ and use Theorem 5.4, we get

$$\begin{aligned}
\#S_= &\leqslant \frac{3}{2} \sum_{1 \leqslant X < p} d(X) \\
&= \frac{3}{2}(p-1)\ln(p-1) + (3\gamma - \frac{3}{2})(p-1) + O(\sqrt{p}) \qquad (5.3) \\
&= O(p \ln p)
\end{aligned}$$

Next we derive a lower bound for $S_=$. First note that each $X = Y = Z \in \mathbb{F}_p^*$ is in $S_=$ and hence $\#S_= \geqslant p - 1$. We can determine a bound slightly better than this. To do so, we first fix Y. Then $Y^2 \leqslant X^3 < Y^2 p$, since $1 \leqslant Z < p$. Let the values of X that satisfy $Y^2 \leqslant X^3 < Y^2 p$ be $X_1, X_2, \ldots, X_s$ where $s = (Y^2 p)^{1/3} - (Y^2)^{1/3} + O(1)$ and $X_i = X_1 + i - 1$. Since $Y < p$, it's clear that each X_i above is less than p. We consider only those values of X_i for which $Y^2 | X_i^3$. We see that if $Y | X_i$, then $Y^2 | X_i^3$. Hence for a fixed Y, total number of solutions $(X, Y, Z) \in S_=$ is greater than or equal to $((Y^2 p)^{1/3} - (Y^2)^{1/3} + O(1))/Y$. If we sum this over all Y, we get applying the formula (a) and (b) of Theorem 5.3

$$\begin{aligned}
\#S_= &\geqslant \sum_{1 \leqslant Y < p} \frac{(Y^2 p)^{1/3} - (Y^2)^{1/3} + O(1)}{Y} \\
&= (p^{1/3} - 1) \sum_{1 \leqslant Y < p} \frac{1}{Y^{1/3}} + O(\ln p) \\
&= (p^{1/3} - 1) \left[\frac{(p-1)^{1-1/3}}{1 - 1/3} + \zeta(1/3) + O(p^{-1/3}) \right] + O(\ln p) \\
&= \frac{3}{2} p + O(p^{2/3}) \qquad (5.4)
\end{aligned}$$

where $\zeta(1/3) = -0.97336024\ldots$ In particular, $\#S_= = \Omega(p)$.

5.2.3 Cardinality of $S_{\neq}$

Since S is the disjoint union of $S_=$ and $S_{\neq}$, Eqns 5.2, 5.3 and 5.4 give

$$(p-1)^2 - \frac{3}{2}(p-1)\ln(p-1) + O(p) \leqslant \#S_{\neq} \leqslant (p-1)^2 - \frac{3}{2}p + O(p^{2/3}) \qquad (5.5)$$

In particular, $\#S_{\neq} = \Theta(p^2)$.

5.3 Heuristic estimate of $\#S_\alpha$

In this section, we count the number of solutions of the CSC with $X, Y, Z \leqslant p^\alpha$, $X^3 \neq Y^2 Z$. Since the cubic sieve method demands $1/3 \leqslant \alpha < 1/2$, we consider α only in this range, though our argument is valid for any $0 \leqslant \alpha \leqslant 1$.

We first fix Y and write $X^3 = Y^2 Z + kp$ for some $k \in \mathbb{Z} \setminus \{0\}$. We then see that $X^3 \equiv kp \pmod{Y^2}$. This implies that k must be chosen such that kp is a cubic residue modulo Y^2. We are interested only in the cubic residues $1^3, 2^3, \ldots, \lfloor p^\alpha \rfloor^3$ modulo Y^2.

CLAIM 5.5 $\boxed{\text{Irrespective of whether the } \lfloor p^\alpha \rfloor \text{ cubic residues } 1^3, 2^3, \ldots, \lfloor p^\alpha \rfloor^3 \text{ are distinct modulo } Y^2 \text{ or not, for any } n \text{ distinct random values of } kp, \text{ we expect } n\lfloor p^\alpha \rfloor / Y^2 \text{ distinct solutions for } (X, Y, Z) \text{ with } X \leqslant p^\alpha.}$

Proof This is because if $X_1^3 \equiv X_2^3 \equiv kp \pmod{Y^2}$ for some k with $X_1 \neq X_2$, then we get two solutions (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2). In particular, if the *good* cubic residues $1^3, 2^3, \ldots, \lfloor p^\alpha \rfloor^3$ assume m distinct values modulo Y^2, then from the n given values of kp, we expect nm/Y^2 values of k to correspond to the set of these cubic residues. Each such residue, on the other hand, is associated, on an average, with $\lfloor p^\alpha \rfloor / m$ solutions with $X \leqslant p^\alpha$. Hence the expected number of solutions (X, Y, Z) corresponding to the given n random values of kp is $(nm/Y^2)(\lfloor p^\alpha \rfloor / m) = n\lfloor p^\alpha \rfloor / Y^2$. ■

Now we allow k to vary in the range

$$\frac{1}{p} - p^{\alpha-1}Y^2 \leqslant k \leqslant p^{3\alpha-1} - \frac{Y^2}{p} \tag{5.6}$$

This corresponds to a total of

$$p^{3\alpha-1} - \frac{Y^2}{p} + p^{\alpha-1}Y^2 + O(1)$$

values of $k \neq 0$. Since $kp = X^3 - Y^2 Z$ and $1 \leqslant X, Z \leqslant p^\alpha$, we have for the fixed value of Y chosen above, $1 - p^\alpha Y^2 \leqslant kp \leqslant p^{3\alpha} - Y^2$ which implies (5.6). Note, however, that the converse is not true, that is, all values of k prescribed by (5.6) do not lead to values of $1 \leqslant X, Z \leqslant p^\alpha$. We will force $1 \leqslant X \leqslant p^\alpha$ and consider only those solutions for which $1 \leqslant Z \leqslant p^\alpha$.

Now we make the following *heuristic assumption*:

ASSUMPTION 5.6 As k varies in the range given by (5.6), the integers kp behave as random integers modulo Y^2.

This is a reasonable assumption since the gcd $(Y^2, p) = 1$. This assumption together with Claim 5.5 guarantees an expected number of approximately

$$\left(p^{3\alpha-1} + O(1) + \left(p^{\alpha-1} - \frac{1}{p}\right) Y^2\right) \frac{p^\alpha}{Y^2} \tag{5.7}$$

solutions (X, Y, Z) with the given Y. All these solutions correspond to $1 \leqslant X \leqslant p^\alpha$, but not necessarily to $1 \leqslant Z \leqslant p^\alpha$ as told before. The inequalities (5.6) together with $X^3 = Y^2 Z + kp$ show that the range of variation of Z is

$$1 - \frac{p^{3\alpha} - X^3}{Y^2} \leqslant Z \leqslant p^\alpha + \frac{X^3 - 1}{Y^2} \tag{5.8}$$

At this point we make the *second heuristic assumption*:

ASSUMPTION 5.7 All these values of Z are equally likely to occur.

For any $1 \leqslant X \leqslant p^\alpha$, (5.8) prescribes $p^\alpha - 1 + \frac{p^{3\alpha}-1}{Y^2} + O(1)$ non-zero values for Z including the values $1 \leqslant Z \leqslant p^\alpha$. Therefore, by assumption 5.7, the probability that Z lies in the range $1 \ldots \lfloor p^\alpha \rfloor$ is

$$\begin{aligned} &\approx \frac{p^\alpha}{p^\alpha - 1 + \frac{p^{3\alpha}-1}{Y^2}} \\ &= \frac{p^\alpha Y^2}{(p^\alpha - 1)Y^2 + p^{3\alpha} - 1} \\ &> \frac{Y^2}{Y^2 + p^{2\alpha}} \\ &\geqslant \frac{Y^2}{2p^{2\alpha}} \end{aligned} \tag{5.9}$$

This probability multiplied by (5.7) gives the expected number of solutions in S_α with the given fixed Y, as greater than or equal to

$$\frac{1}{2p^\alpha}\left(p^{3\alpha-1}+O(1)+\left(p^{\alpha-1}-\frac{1}{p}\right)Y^2\right).$$

We finally vary Y with $1 \leqslant Y \leqslant p^\alpha$ and use (d) of Theorem 5.3 to obtain:

Expected cardinality of S_α

$$\begin{aligned}
&\geqslant \frac{1}{2p^\alpha}\left(p^{3\alpha-1}p^\alpha+O(p^\alpha)+\left(p^{\alpha-1}-\frac{1}{p}\right)\left(\frac{(p^\alpha)^3}{3}+O((p^\alpha)^2)\right)\right)\\
&= \frac{2}{3}p^{3\alpha-1}+O(\max(1,p^{2\alpha-1}))\\
&= \Omega(p^{3\alpha-1}) \qquad (5.10)
\end{aligned}$$

For sufficiently large p, the term $\frac{2}{3}p^{3\alpha-1}$ dominates and one might expect to get a solution if $\frac{2}{3}p^{3\alpha-1} \gg 1$, say, for example, if $\frac{2}{3}p^{3\alpha-1} \geqslant 1000$, i.e., if

$$\alpha \geqslant \frac{1}{3}+\frac{\ln(1500)}{3\ln p}$$

For example, if $p \approx 2^{500}$, then $\alpha = 0.34037$ is expected to make S_α non-empty.

We have noted that assumption 5.6 is reasonable and gives a good picture of the average situation. Assumption 5.7, on the other hand, is difficult to justify mathematically. Indeed this assumption is equivalent to the question of existence of a suitable solution. We assumed an average scenario to get an estimate of $\#S_\alpha$. As we pointed out earlier, our aim is not to *prove* the non-emptiness or otherwise of S_α, but to compute an approximate value of its cardinality with the hope that this behavior is general enough to portray the average situation. In the next section, we show that up to a constant factor our estimates are quite close to the experimental values we obtained from a set of small scale experiments. These experimental results together with our theoretical estimate tempt us to make the following conjecture:

CONJECTURE 5.8 The expected cardinality of S_α is asymptotically equal to $\chi p^{3\alpha-1}$ for all $0 \leqslant \alpha \leqslant 1$ and for some constant $\chi \approx 1$. (Note that (5.5) demands $\chi = 1$.)

Few primes of special forms might not obey the conjectured estimates. But we do not see any such special form – both experimentally and theoretically. The bulk of the derivation of (5.10) is based on the cubic residues module Y^2 for integers $Y = 1, 2, 3, \ldots$ The prime p does not seem to play an important role in connection with assumption 5.6. The second assumption, however, can be influenced by the choice of p and may lead to situations we failed to visualize.

5.3.1 Experimental verification

We experimented with randomly generated primes of size around 30 bits. We actually enumerated all the solutions of the CSC for various values of α in the range $0.33 \leqslant \alpha \leqslant 0.50$. We tabulate these experimental values together with the theoretical estimates obtained as $\#S_\alpha = \lfloor \frac{2}{3}p^{3\alpha-1} \rfloor$. We also list the conjectured values given by $\#S_\alpha = \lfloor \chi p^{3\alpha-1} \rfloor$ with $\chi = 1$.

Table 5.1: $\#S_\alpha$ for $p = 32263723$ (A 25-bit prime)

α	Values of $\#S_\alpha$ (a)	(b)	(c)	(b)/(a)	(c)/(a)
0.33	0	0	0	_	_
0.34	0	0	1	_	_
0.35	1	1	2	1.00	2.00
0.36	1	2	3	2.00	3.00
0.37	1	4	6	4.00	6.00
0.38	4	7	11	1.75	2.75
0.39	8	12	18	1.50	2.25
0.40	23	21	31	0.91	1.35
0.41	38	35	53	0.92	1.39
0.42	62	59	89	0.95	1.44
0.43	105	100	150	0.95	1.43
0.44	191	168	252	0.88	1.32
0.45	356	283	424	0.79	1.19
0.46	623	475	713	0.76	1.14
0.47	1060	798	1198	0.75	1.13
0.48	1785	1341	2012	0.75	1.13
0.49	3043	2254	3381	0.74	1.11
0.50	5225	3786	5680	0.72	1.09

(a) experimental, (b) estimated, (c) conjectured

Table 5.2: $\#S_\alpha$ for $p = 1034302223$ (A 30-bit prime)

α	Values of $\#S_\alpha$ (a)	(b)	(c)	(b)/(a)	(c)/(a)
0.33	0	0	0	_	_
0.34	1	1	1	1.00	1.00
0.35	1	1	2	1.00	2.00
0.36	2	3	5	1.50	2.50
0.37	5	6	9	1.20	1.80
0.38	9	12	18	1.33	2.00
0.39	23	22	34	0.96	1.48
0.40	53	42	63	0.79	1.19
0.41	98	78	118	0.80	1.20
0.42	185	147	220	0.79	1.19
0.43	368	274	411	0.74	1.17
0.44	695	511	766	0.74	1.10
0.45	1363	952	1429	0.70	1.05
0.46	2475	1776	2664	0.72	1.08
0.47	4646	3310	4965	0.71	1.07
0.48	8815	6170	9256	0.70	1.05
0.49	16615	11502	17253	0.69	1.04
0.50	31451	21440	32160	0.68	1.02

(a) experimental, (b) estimated, (c) conjectured

Table 5.1 gives data for $p = 32263723$, a random 25 bit prime. Table 5.2 gives the same for a random 30-bit prime $p = 1034302223$. Though we have experimented with many primes of this size, we give the values of $\#S_\alpha$ only for these two values. This is because we get exactly similar pattern of variation of $\#S_\alpha$ with α for all of our test primes. Thus a few representatives are sufficient to reflect the scenario.

The tables clearly show that apart from constant factors the experimental, estimated and conjectured values exhibit the same pattern of variation of $\#S_\alpha$ with α. For α close to 0.33, the relation between these values is little erratic. As α increases, say $\alpha \geqslant 0.40$, the ratio of the estimated value to the experimental value and the ratio of the conjectured value to the experimental value tend to approach constant values. In particular, the conjectured values are quite close to the experimental values. It remains unsettled if this pattern continues to hold for general primes of larger sizes, say for primes of size $\leqslant$ 1000 bits. Since at present no algorithms are known to solve the CSC in polynomial time of $\log p$, we cannot experiment with higher values of p. In addition, even if such an algorithm exists, one should spend $O(p^{2\alpha})$ time for enumerating all the solutions in S_α. This makes it infeasible to continue the experimental study with primes of practical interest. These small-scale experiments give us some confidence about the theoretical estimates derived in this section.

In spite of all these theoretical and experimental exercises, the question of existence or otherwise of a solution of the CSC for some $1/3 \leqslant \alpha < 1/2$ continues to remain unanswered. It is believed that a solution exists [28, 77]. Our analysis only strengthens the belief in favor of a solution and to that effect is much stronger than the argument presented in [77].

6 Conclusion

In this chapter we summarize the work reported in the thesis. We also describe the possibilities and need for further research in this area.

6.1 Summary of work done

This thesis has been devoted to a study of the computational aspects of finite fields. We started with a brief survey on the main computational problems of theoretical and practical concern to applied mathematicians and computer scientists. We provided a list of the state-of-the-art algorithms to solve these problems and the running times of these algorithms.

In the second chapter, we described our computational library of functions for working over finite fields. This library developed by us is termed the Galois Field Library ($\mathrm{G}\mathbb{F}\mathrm{L}$). $\mathrm{G}\mathbb{F}\mathrm{L}$ consists of built-in routines for solving many computational problems discussed in the survey of Chapter 1. It provides arithmetic over finite fields of arbitrary characteristic and cardinality. It also provides routines for univariate polynomials and matrices over finite fields. Our library allows the user to work with prime fields of any characteristic and with their algebraic extensions obtained by adjoining roots of any number of irreducible polynomials. Our library introduces and makes extensive use of what we call the packed representation of finite field elements. This packed representation helps us provide a uniform treatment of all finite fields. To the best of our knowledge, no other library for computation over Galois fields provides this generality. Another important feature of $\mathrm{G}\mathbb{F}\mathrm{L}$ is its dynamic memory management policy which eliminates garbage collection overheads. $\mathrm{G}\mathbb{F}\mathrm{L}$ seems to provide the largest set of built-in routines as far as computation over finite fields is concerned.

We demonstrated the programming techniques with $\mathrm{G}\mathbb{F}\mathrm{L}$ through some small and simple examples. We also provided an exhaustive list of functions currently provided by $\mathrm{G}\mathbb{F}\mathrm{L}$. We compared the performance of $\mathrm{G}\mathbb{F}\mathrm{L}$ with those of three other libraries, namely, LiDIA, NTL and ZEN.

The rest of the thesis (Chapters 3 through 5) has been devoted to a study of the discrete logarithm problem (DLP) over finite fields of prime cardinality p. The DLP is a very difficult computational problem for which no polynomial time algorithms are known. It is not even known if this problem can be solved in polynomial time. The best algorithms known till date are based on the index calculus method and take time subexponential in $\log p$. We concentrated on three variants of the index calculus method, namely the basic method, the linear sieve method and the cubic sieve method.

The sieve methods test a set of deterministically generated integers (the integers $T(c_1, c_2)$ and $R(A, B, C)$ introduced in Chapter 3) for smoothness over a predetermined set of small primes. The analysis of running times of these methods is based on the heuristic assumption that these deterministically generated integers behave as if they have been chosen following a random distribution. We started our study of the DLP by showing that the actual distribution of these integers is not really random in the sense that these integers do not follow uniform distribution. To prove our claim we found out the arithmetic mean and the cumulative statistical distribution of these integers. We found that the average bit-length of these test integers is smaller than the expected bit-length of a sample of integers chosen following the uniform distribution. Since smaller integers have higher chance of being smooth, we concluded that the actual distribution of the test integers is better than the uniform distribution.

In Chapter 4, we proposed heuristic modification schemes for the three variants of the index calculus method stated above. We analytically and experimentally found out the effectiveness of our heuristics. For the basic method, our heuristic schemes reduce the total number of discrete exponentiations carried out in the field. We also bring down the cost of trial divisions by factor base primes using two strategies: maintaining a list of remainders and sieving. All these help us get a speedup of between 1.5 and 3 over the original method. We, however, note that the index calculus method in the basic form is very slow and achieving a speed-up of this order does not make it usable in practical situations. Therefore, our study of the basic method is mostly of theoretical interest.

The linear sieve and the cubic sieve methods are practical methods for primes of medium size ($\leqslant$ 250 bits). Our heuristic modifications of the linear sieve method decrease the running time per relation generated. This is because our heuristics test for smoothness a set of integers that are on an average smaller than the integers tested for smoothness in the original method. At the same time the heuristics reduce the ratio of the number of relations to the size of the factor base and may lead to a situation where one fails to get a full-rank system of linear congruences.

Although the cubic sieve method proposed in 1986 [28] is asymptotically faster than the linear sieve method, it drew very little attention by the research community. The most probable reason for this is that the applicability of this method banks on a solution of the congruence $X^3 \equiv Y^2Z \pmod{p}$ with $X^3 \neq Y^2Z$. Given a solution of this congruence, one can, however, readily use the cubic sieve method. We studied a case when a solution of the congruence is easily available, namely the case when the cardinality p of the prime field is close to a whole cube. We showed that in this case the cubic sieve method runs faster than the linear sieve method by a factor of 2.5, even when p is small (of length around 150 bits). For larger fields, the speed-up of the cubic sieve method over the linear sieve method is expected to be more. In order to prove the superiority of the cubic sieve method, we implemented an efficient version of the two sieve methods. Our implementation speeds up the equation collecting phase by a reasonable amount. Finally we proposed a heuristic modification of the cubic sieve method, that allows us to build a larger factor base without any significant increase in the running time. We also determined, theoretically and experimentally, the optimal value of a parameter which plays the central role in this heuristic scheme.

We conclude our study of the DLP by an analytic study of the congruence $X^3 \equiv Y^2Z \pmod{p}$. A solution of this congruence with X, Y, Z of the order of p^α with $1/3 \leqslant \alpha < 1/2$ is needed for the cubic sieve method. The smallest possible value of α determines the best running time of the cubic sieve method. It is, however, not known how one can find a solution of the congruence. Moreover, it is not even known if a solution with α in the above range exists. Our heuristic analytic arguments show that on an average one can expect to have a solution of the congruence with α close to $1/3$. More precisely, we showed that under two (reasonable) heuristic assumptions, the expected number of solutions of the above congruence with $1 \leqslant X, Y, Z \leqslant p^\alpha$ is $\Omega(p^{3\alpha-1})$. We carried out some small scale experiments to enumerate all the solutions of the congruence and found that our heuristic estimate tallies quite closely with the experimental values. Our analysis, however, does not lead to an algorithmic determination of a solution.

6.2 Directions for further research

The theory of finite fields finds many applications in various areas like cryptography, error control coding, combinatorial design. As a result, design, analysis and implementation of algorithms for computation over finite fields are getting more and more popular among mathematicians and engineers. Many tools are coming up to meet the practical needs of users. We have developed $\mathrm{G}\mathbb{F}\mathrm{L}$ as a general-purpose easy-to-use library. There are many ways in which the library can be enhanced. We mention a few important possibilities.

1. *Improving performance of* $\mathrm{G}\mathbb{F}\mathrm{L}$ *routines:* This involves devising and/or implementing algorithms that run more efficiently compared to the routines currently implemented in $\mathrm{G}\mathbb{F}\mathrm{L}$. A comparative study of $\mathrm{G}\mathbb{F}\mathrm{L}$ with other existing libraries (See Section 2.5.4) shows that there are many scopes for improvement. Though the generality of $\mathrm{G}\mathbb{F}\mathrm{L}$ is partially responsible for its slower relative performance, it is not the only source of inefficiency. It requires considerable additional effort for finding out and removing possible loop-holes in the implementation.
2. *Adding new features:* $\mathrm{G}\mathbb{F}\mathrm{L}$ can be made to address a wider range of computational problems over finite fields. For example, data structures and routines for manipulating multi-variate polynomials and polynomial functions can be added to $\mathrm{G}\mathbb{F}\mathrm{L}$. Routines for elliptic curves over finite fields can also be added.
3. *Designing a front-end:* An interpreter that runs on top of $\mathrm{G}\mathbb{F}\mathrm{L}$ can make programmer's task much easier and user-friendly.
4. *Parallelization:* The $\mathrm{G}\mathbb{F}\mathrm{L}$ routines can be parallelized and run in a distributed fashion on a network of processors. One possible way to achieve this is to implement a client-server application with the help of Unix domain sockets.

As we discussed in the survey of Chapter 1, many computational problems over finite fields do not have deterministic polynomial-time solutions. This is, in general, not a problem, because randomized algorithms solve these problems reasonably efficiently and are sufficient for all practical purposes. The discrete logarithm problem, on the other hand, continues to remain an outstanding open problem. The advent of cryptography to exploit human inability to solve the problem efficiently (even with randomization) only intensifies the intellectual challenge. Cryptography is a negative application in a broad sense, but it has practical usefulness. It is rather debatable if an efficient solution of the DLP does any good to mankind. But then as A. K. Lenstra and H. W. Lenstra, Jr. say [77]: *"Most number theorists considered the small group of colleagues that occupied themselves with these problems as being inflicted with an incurable but harmless obsession."*

An obsession or scientific inquisitiveness, the need for further research to solve the DLP can hardly be denied. Our study of the DLP in this thesis is motivated by this need. The question of solvability or otherwise of the DLP in polynomial time is a hard one to answer. We point out a few easier and more down-to-earth questions, the solutions of which can augment our study:

- *Randomness of $T(c_1, c_2)$ and $R(A, B, C)$:* We showed that the integers $T(c_1, c_2)$ and $R(A, B, C)$ do not follow uniform distribution. The question that remains unanswered is whether these integers behave randomly as a sample of integers drawn according to the distributions they follow.
- *Comparison with the number field sieve method:* The number field sieve (NFS) method is currently known to be the fastest method to solve the DLP, both theoretically and experimentally. Our study reveals that the cubic sieve method holds promise. It is, therefore, necessary to calibrate the performance of the cubic sieve method against the NFS method. The NFS method is asymptotically faster than the cubic sieve method. But it demands experimentation to settle from which sizes of p, the NFS method starts performing better than the cubic sieve. Similarly, albeit somewhat less importantly, we need to compare the performance of the cubic sieve method with the Gaussian integer method (though the Gaussian integer method is asymptotically slower than the other two methods discussed in this paragraph). It is imortant to note here that for primes close to a cube, the number field sieve method also gets efficient, that is, one should use the special number field sieve method instead of the general number field sieve method.
- *More about the cubic sieve congruence:* We need a proof for the existence of suitable solutions of the congruence $X^3 \equiv Y^2 Z \pmod{p}$. More importantly, we need 'good' algorithms for calculating a solution.
- *Further enhancements of the algorithms:* We need further improvements (heuristic or otherwise) over the existing methods for solving the DLP. In particular, the second stage of the cubic sieve method is known to be quite slow. Any improvement in the running time of this stage makes the cubic sieve method more usable.

The integer factorization problem is known to be yet another difficult open problem and is widely believed to be equivalent to the DLP. There are evidences in favor of the equivalence; see, for example, [85, Section 6.9]. Indeed, save a few exceptions, most algorithms we use nowadays for solving the DLP are direct adaptations of the algorithms for solving the integer factorization problem. Any new algorithm to solve one of these problems is expected to apply to the other problem as well. We end this section with the following sobering quote by A. K. Lenstra and H. W. Lenstra, Jr. [77], which though meant to address the integer factorization problem is equally applicable for the DLP.

"It is important to point out that there is only historical evidence that factorization is an intrinsically hard problem. Generations of number theorists, a small army of computer scientists, and legions of cryptologists spent a considerable amount of energy on it, and the best they came up with are (the) relatively poor algorithms . . . Of course, as long as the widely believed P≠NP-conjecture remains unproved, complexity theory will not have fulfilled its originally intended mission of proving certain algorithmic problems to be intrinsically hard; but with factorization the situation is worse, since even the celebrated conjecture just mentioned has no implications about its intractability. Factorization is considered easier than NP-complete and although the optimistic conjecture that it might be doable in polynomial time is only rarely publicly voiced, it is not an illegitimate hope to foster."

Publications from this thesis

[1] A. Das and C. E. Veni Madhavan, ‘Galois field library: Reference manual’, Technical report No. IISc-CSA-98-05, Department of Computer Science and Automation, Indian Institute of Science, Feb 1998.

[2] A. Das and C. E. Veni Madhavan, ‘Performance Comparison of linear sieve and cubic sieve algorithms for discrete logarithms over prime fields’, Proc. 10th International Symposium on Algorithms and Computation (ISAAC’99), in LNCS #1741 (Ed. A. Aggarwal and C. Pandu Rangan), 1999, pp 295–306.

References

[1] L. M. ADLEMAN AND H. W. LENSTRA, ‘Finding irreducible polynomials over finite fields’, Proc. 18th. annual ACM Symposium on Theory of Computing (1986), 350–355.

[2] B. B. AMBERKER, ‘Large-scale integer and polynomial computations: Efficient implementation and applications’, Ph.D. Thesis, Computer Science and Automation, Indian Institute of Science, Bangalore, 1996.

[3] K. AOKI AND K. OHTA, ‘Fast arithmetic operations over $\mathbb{F}_{2^n}$ for software implementation’, Proceedings of the 4th Annual Workshop on Selected Areas in Cryptography, 1997.

[4] T. M. APOSTOL, ‘Introduction to analytic number theory’, Undergraduate text in Mathematics, Springer Verlag, 1976.

[5] A. AVERBUCH, N. H. BSHOUTY AND M. KAMINSKI, ‘A classification of algorithms for multiplying polynomials of small degree over finite fields’, Journal of Algorithms, 13 (1992), 577–588.

[6] E. BACH, J. DRISCOLL AND J. SHALLIT, ‘Factor refinement’, Journal of Algorithms, 15 (1993), 199–222.

[7] R. BALASUBRAMANIAN AND N. KOBLITZ, ‘The improbability that an elliptic curve has subexponential discrete log problem under the Menezes-Okamoto Vanstone algorithm’, Journal of Cryptology, 11 (1998), 141–145.

[8] M. BEN-OR, ‘Probabilistic algorithms in finite fields’, Proc. IEEE 22nd. annual symposium on Foundations of Computer Science (1981), 394–398.

[9] M. BEN-OR AND P. TIWARI, ‘A deterministic algorithm for sparse multivariate polynomial interpolation’, Proc. 20th. annual ACM Symposium on Theory of Computing (1988), 301–309.

[10] E. R. BERLEKAMP, ‘Algebraic Coding Theory’, McGraw-Hill, New York, 1968.

[11] E. R. BERLEKAMP, ‘Factoring polynomials over large finite fields’, Mathematics of Computation, 24 (1970), 713–735.

[12] E. R. BERLEKAMP, H. RUMSEY AND G. SOLOMON, ‘On the solution of algebraic equations over finite fields’, Information and Control, 10 (1967), 553–564.

[13] I. BIEHL, J. BUCHMANN, T. PAPANIKOLAOU, ‘LiDIA – A library for computational number theory’, Technical Report, SFB 04/95, Universität des Saarlandes, 1994. (Available for downloading from http://www.informatik.tu-darmstadt.de/TI/Mitarbeiter/papanik/ps/TR_02_95.ps.gz)

[14] I. F. Blake, R. Fuji-Hara, R. C. Mullin, and S. A. Vanstone Computing logarithms in finite fields of characteristic twoSIAM Journal of Algebraic and Discrete Methods, 5 (1984), 276-285

[15] W. BOSMA, J. J. CANON AND C. PLAYOUST, ‘The Magma Algebra System I: The user language’, J. Symbolic Computation, 24 (1997), 351–369.

[16] D. M. BRESSOUD, ‘Factorization and Primality Testing’, Undergraduate Text in Mathematics, Springer Verlag, 1989.

[17] N. H. BSHOUTY AND M. KAMINSKI, ‘Multiplication of polynomials over finite fields’, SIAM Journal of Computing, 19 (1990), 452–456.

[18] J. BUCHMANN AND V. SHOUP, ‘Constructing nonresidues in finite fields and the extended Riemann hypothesis’, Proc. 23rd. annual ACM Symposium on Theory of Computing (1991), 72–79.

[19] M. C. R. BUTLER, ‘On the irreducibility of polynomials over a finite field’, Quart. J. Math., Oxford Ser.(2), 5 (1954), 102–107.

[20] P. CAMION, ‘A deterministic algorithm for factoring polynomials of $\mathbb{F}_q[x]$’, Annals of Discrete Mathematics, 17 (1983), 149–157.

[21] P. CAMION, ‘Improving an algorithm for factoring polynomials over a finite field and constructing large irreducible polynomials’, IEEE transaction on Information Theory, 29 (1983), 378–385.

[22] D. CANTOR AND H. ZASSENHAUS, ‘A new algorithm for factoring polynomials over finite fields’, Mathematics of Computation, 36 (1981), 587–592.

[23] F. CHABAUD, ‘Recherche de performance dans l’algorithmique des corps finis. Applications à la cryptographie’, thesis, Ecole Polytechnique, Oct 1996 (in French). (Available for downloading from http://www.dmi.ens.fr/˜chabaud/data/these.ps.gz)

[24] M. CLAUSEN, A. DRESS, J. GRABMEIER AND M. KARPINSKI, ‘On zero-testing and interpolation of k-sparse multivariate polynomials over finite fields’, Theoretical Computer Science, 84(2) (1991), 151–164.

[25] H. COHEN, ‘A course in computational algebraic number theory’, Graduate Text in Mathematics #138, Springer Verlag, 1993.

[26] D. COPPERSMITH, ‘Fast evaluation of logarithms in fields of characteristic two’, IEEE transaction on Information Theory, 30 (1984), 587–594.

[27] D. COPPERSMITH, ‘Solving homogeneous equations over GF[2] via block Wiedemann algorithm’, Mathematics of Computation, 62 (1994), 333–350.

[28] D. COPPERSMITH, A. M. ODLYZKO AND R. SCHROEPPEL, ‘Discrete logarithms in $GF(p)$’, Algorithmica, 1 (1986), 1–15.

[29] D. COPPERSMITH AND S. WINOGRAD, ‘Matrix multiplication via arithmetic progressions’, Journal of Symbolic Computation, 9(3) (1990), 23–52.

[30] A. DAS AND C. E. VENI MADHAVAN, ‘Galois field library: Reference manual’, Technical report No. IISc-CSA-98-05, Department of Computer Science and Automation, Indian Institute of Science, Feb 1998.

[31] W. DIFFE AND M. HELLMAN, ‘New directions in cryptography’, IEEE transaction on Information Theory, 22 (1976), 644–654.

[32] T. ELGAMAL, ‘A public-key cryptosystem and a signature scheme based on discrete logarithms’, IEEE transaction on Information Theory, 31 (1985), 469–472.

[33] E. GALOIS, ‘Sur la théorie des nombres’, In R. Bourgne and J.-P. Arza, editors, Écrits et Mémoires Mathématoques d’Évariste Galois, Gauthier-Villars, 1830, 112–128.

[34] J. VON ZUR GATHEN, ‘Irreducibility of multivariate polynomials’, Journal of Computer and System Sciences, 31 (1985), 225–264.

[35] J. VON ZUR GATHEN, ‘Factoring polynomials and primitive elements for special primes’, Theoretical Computer Science, 52 (1987), 77–89.

[36] J. VON ZUR GATHEN, ‘Testing permutation polynomials’, Proc. IEEE 30th. annual symposium on Foundations of Computer Science (1989), 88–92.

[37] J. VON ZUR GATHEN, ‘Efficient exponentiation in finite fields’, Proc. IEEE 32nd. annual symposium on Foundations of Computer Science (1991), 384–391.

[38] J. VON ZUR GATHEN, ‘Tests for permutation polynomials’, SIAM Journal of Computing, 20 (1991), 591–602.

[39] J. VON ZUR GATHEN, ‘Processor-efficient exponentiation in finite fields’, Information Processing Letters, 41 (1992), 81–86.

[40] J. VON ZUR GATHEN AND E. KALTOFEN, ‘Factoring sparse multivariate polynomials’, Journal of Computer and System Sciences, 31 (1985), 265–287.

[41] J. VON ZUR GATHEN AND E. KALTOFEN, ‘Factorization of multivariate polynomials over finite fields’, Mathematics of Computation, 45 (1985), 251–261.

[42] J. VON ZUR GATHEN, M. KARPINSKI AND I. E. SHPARLINSKI, ‘Counting curves and their projections’, Proc. 25th. annual ACM Symposium on Theory of Computing (1993), 805–812.

[43] J. VON ZUR GATHEN AND V. SHOUP, ‘Computing Frobenius maps and factoring polynomials’, Proc. 24th. annual ACM Symposium on Theory of Computing (1992), 97–105.

[44] J. VON ZUR GATHEN AND I. E. SHPARLINSKI, ‘Orders of Gauss periods in finite fields’, Proceedings ISAAC ’95, Lecture Notes in Computer Science, 1004 (1995), 208-215.

[45] J. VON ZUR GATHEN AND I. E. SHPARLINSKI, ‘Finding points on curves over finite fields’, Proc. IEEE 36th annual symposium on Foundations of Computer Science (1995), 284–292.

[46] J. GERVER, ‘Factoring large numbers with a quadratic sieve’, Mathematics of Computation, 41 (1983), 287–294.

[47] D. M. GORDON, ‘Discrete logarithms in $GF(p)$ using the number field sieve’, SIAM Journal of Discrete Mathematics, 6 (1993), 124–138.

[48] D. M. GORDON AND K. S. MCCURLEY, ‘Massively Parallel Computation of Discrete Logarithms’, Proc. Crypto ’92, Lecture Notes in Computer Science, 740 (1992), Springer-Verlag, 312–323.

[49] R. GÖTTFERT, ‘An acceleration of the Niederreiter factorization algorithm algorithms in characteristic 2’, Mathematics of Computation, 62 (1994), 831–839.

[50] D. Y. GRIGORIYEV AND M. KARPINSKI, ‘An approximation algorithm for the number of zeros of arbitrary polynomials over GF$[q]$’, Proc. IEEE 32nd. annual symposium on Foundations of Computer Science (1991), 662–669.

[51] D. Y. GRIGORIYEV, M. KARPINSKI AND M. F. SINGER, ‘Fast parallel algorithms for sparse multivariate polynomial interpolation over finite fields’, SIAM Journal of Computing, 19 (1990), 1059–63.

[52] T. HANSEN AND G. L. MULLEN, ‘Primitive polynomials over finite fields’, Mathematics of Computation, 59 (1992), 639–643.

[53] I. N. HERSTEIN, ‘Topics in Algebra’, 2nd. Edition, John Wiley & Sons, 1975.

[54] K. HOFFMAN AND R. KUNZE, ‘Linear Algebra’, 2nd. Edition, Prentice-Hall, 1971.

[55] M. A. HUANG, ‘Generalized Riemann hypothesis and factoring polynomials over finite fields’, Journal of Algorithms, 12 (1991), 464–481.

[56] M. A. HUANG AND D. IERARDI, ‘Counting rational points on curves over finite fields’, Proc. IEEE 34th. annual symposium on Foundations of Computer Science (1993), 616–625.

[57] M. A. HUANG AND A. J. RAO, ‘Interpolation of sparse multivariate polynomials over large finite fields with applications’, Proc. 7th. annual ACM–SIAM Symposium on Discrete Algorithms (1996), 508–517.

[58] M. A. HUANG AND Y. WONG, ‘Solving systems of polynomial congruences modulo a large prime’, Proc. IEEE 37th annual symposium on Foundations of Computer Science (1996), 115–124.

[59] K. HUBER, ‘Some comments on Zech’s logarithms’, IEEE transaction on Information Theory, 36 (1990), 946–950.

[60] T. ITOH AND S. TSUJII, ‘A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases’, Information and Computation, 78 (1988), 171–177.

[61] T. ITOH AND S. TSUJII, ‘Structure of parallel multipliers for a class of fields $\mathbb{F}_{2^m}$’, Information and Computation, 83 (1989), 21–40.

[62] M. J. JACOBSON, N. KOBLITZ, J. H. SILVERMAN, A. STEIN AND E. TESKE, ‘Analysis of the xedni calculus attack’, Technical report, CORR 99-06, University of Waterloo, 1999.

[63] E. KALTOFEN, ‘Fast parallel absolute irreducibility testing’, Journal of Symbolic Computation, 1 (1985), 57–67.

[64] E. KALTOFEN, ‘Deterministic irreducibility testing of polynomials over large finite fields’, Journal of Symbolic Computation, 4 (1987), 77–82.

[65] E. KALTOFEN AND V. PAN, 'Processor-efficient parallel solution of linear systems II, The positive characteristic and singular cases', Proc. IEEE 33rd. annual symposium on Foundations of Computer Science (1992), 714–723.

[66] E. KALTOFEN AND V. SHOUP, 'Subquadratic–time factoring of polynomials over finite fields', Proc. 27th. annual ACM Symposium on Theory of Computing (1995), 398–406.

[67] E. KALTOFEN AND B. M. TRAGER, 'Computing with polynomials given by black boxes for their evaluations: greatest common divisors, factorization, separation of numerators and denominators', Journal of Symbolic Computation, 9 (1990), 301–320.

[68] M. KARPINSKI AND M. LUBY, 'Approximating the number of zeros of a GF$[2]$ polynomial', Journal of Algorithms, 14 (1993), 280–287.

[69] D. E. KNUTH, 'The Art of Computer Programming', Vol. 2 : 'Seminumerical Algorithms', 3rd edition, Addison Wesley, 1997.

[70] N. KOBLITZ, 'Elliptic curve cryptosystems', Mathematics of Computation, 48 (1987), 203–209..

[71] N. KOBLITZ, 'A course in number theory and cryptography', 2nd edition, Springer Verlag, 1994.

[72] N. KOBLITZ, 'Algebraic aspects of cryptography', Springer Verlag, 1998.

[73] B. A. LAMACCHIA AND A. M. ODLYZKO, 'Computation of discrete logarithms in prime fields', Designs, Codes, and Cryptography, 1 (1991), 46–62.

[74] B. A. LAMACCHIA AND A. M. ODLYZKO, 'Solving large sparse linear systems over finite fields', Advances in Cryptology – CRYPTO '90, A. J. Menezes and S. A. Vanstone (eds.), Lecture Notes in Computer Science #537 (1991), Springer Verlag, 109–133.

[75] A. K. LENSTRA, 'Factoring multivariate polynomials over finite fields', Journal of Computer and System Sciences, 30 (1985), 235–248.

[76] A. K. LENSTRA, 'LIP', Source-code and documentation are available by anonymous ftp from `ftp://ftp.ox.ac.uk/pub/math/freelip/`.

[77] A. K. LENSTRA AND H. W. LENSTRA, 'Algorithms in number theory', in Handbook of Theoretical Computer Science, ed. J. van Leeuwen, 1990, 675–715.

[78] A. K. LENSTRA AND H. W. LENSTRA, eds., 'The development of the number field sieve', Lecture notes in mathematics #1554, Springer Verlag, 1993.

[79] H. W. LENSTRA AND R. J. SCHOOF, 'Primitive normal bases over finite fields', Mathematics of Computation, 48 (1987), 217–231.

[80] R. LIDL AND G. L. MULLEN, 'When does a polynomial over a finite field permute the elements of the field?', American Mathematical Monthly, 95 (1988), 243–246.

[81] R. LIDL AND G. L. MULLEN, 'When does a polynomial over a finite field permute the elements of the field?, II', American Mathematical Monthly, 100 (1993), 71–74.

[82] R. LIDL AND H. NIEDERREITER, 'Finite Fields', Encyclopedia of Mathematics and its Applications, 20, Cambridge University Press, 1984.

[83] K. MA AND J. VON ZUR GATHEN, 'The computational complexity of recognizing permutation functions', Proc. 26th. annual ACM Symposium on Theory of Computing (1994), 392–401.

[84] K. MCCURLEY, 'The discrete logarithm problem', Cryptology and Computational Number Theory, Proceedings of the Symposium in Applied Mathematics, 42 (1990), 49–74.

[85] A. J. MENEZES ed., 'Applications of finite fields', Kluwer Academic Publishers, 1993.

[86] A. J. MENEZES, 'Elliptic curve public key cryptosystems', Kluwer Academic Publishers, 1993.

[87] A. J. MENEZES, P. VAN OORSCHOT AND S. VANSTONE, 'Some computational aspects of root finding in GF(q^m)', in Symbolic and Algebraic Computation, Lecture Notes in Computer Science, 358 (1989), 256–270.

[88] A. J. MENEZES, P. VAN OORSCHOT AND S. VANSTONE, ‘Subgroup refinement algorithms for root finding in GF(q)’, SIAM Journal of Computing, 21 (1992), 228–239.

[89] A. J. MENEZES, P. VAN OORSCHOT AND S. VANSTONE, ‘Handbook of applied cryptography’, CRC Press, 1997.

[90] M. MIGNOTTE, ‘Mathematics for Computer Algebra’, Springer Verlag, 1992.

[91] V. MILLER, ‘Uses of elliptic curves in cryptography’, Advances in cryptology, CRYPTO’85, Lecture notes in computer science, 18 (1986), Springer Verlag, 417–426.

[92] I. H. MORGAN AND G. L. MULLEN, ‘Primitive normal polynomials over finite fields’, Mathematics of Computation, 63 (1994), 759–765.

[93] G. L. MULLEN, ‘Permutation polynomials over finite fields’, In: Finite fields, coding theory and advances in comm. and computing, (G. L. Mullen and P. J.-S. Shiue, Eds.), Lecture notes in pure and applied mathematics, 141 (1993), Marcel Dekker, 131–151.

[94] G. L. MULLEN, ‘Permutation polynomials: A matrix analogue of Schur’s conjecture and a survey of recent results’, Finite Fields Applications, 1 (1995), 242–258.

[95] H. NIEDERREITER, ‘Factoring polynomials over finite fields using differential equations and normal bases’, Mathematics of Computation, 62 (1994), 819–830.

[96] H. NIEDERREITER AND R. GÖTTFERT, ‘On a new factorization algorithm for polynomials over finite fields’, Mathematics of Computation, 64 (1995), 347–353.

[97] A. M. ODLYZKO, ‘Discrete logarithms and their cryptographic significance’, Advances in Cryptology: Proceedings of Eurocrypt ’84, Lecture Notes in Computer Science, 209 (1985), Springer-Verlag, 224–314.

[98] P. VAN OORSCHOT AND S. VANSTONE, ‘A geometric approach to root finding in GF(q^m)’, IEEE transaction on Information Theory, 35 (1989), 444–453.

[99] M. RABIN, ‘Probabilistic algorithms in finite fields’, SIAM Journal of Computing, 9 (1980), 273–280.

[100] L. RÓNYAI, ‘Factoring polynomials over finite fields’, Journal of Algorithms, 9 (1988), 391–400.

[101] L. RÓNYAI, ‘Galois groups and factoring polynomials over finite fields’, Proc. IEEE 30th. annual symposium on Foundations of Computer Science (1989), 99–104.

[102] L. RÓNYAI, ‘Factoring polynomials modulo special primes’, Combinatorica, 9 (1989), 199–206.

[103] R. M. ROTH AND G. M. BENEDEK, ‘Interpolation and approximation of sparse multivariate polynomials over GF(2)’, SIAM Journal of Computing, 20 (1991), 291–314.

[104] O. SCHIROKAUER, ‘Discrete logarithms and local units’, Philosophical transactions of the Royal Society of London, Series A, 345 (1993), 409–423.

[105] O. SCHIROKAUER, D. WEBER, AND T.DENNY, ‘Discrete logarithms: the effectiveness of the index calculus method’, Proc. ANTS II, Lecture notes in Computer Science, 1122 (1996), Springer-Verlag, 337–361.

[106] R. J. SCHOOF, ‘Elliptic curves over finite fields and the computation of square roots mod p’, Mathematics of Computation, 44 (1985), 483–494.

[107] V. SHOUP, ‘On the deterministic complexity of factoring polynomials over finite fields’, Information Processing Letters, 33 (1990), 261–267.

[108] V. SHOUP, ‘New algorithms for finding irreducible polynomials over finite fields’, Mathematics of Computation, 54 (1990), 435–447.

[109] V. SHOUP, ‘Smoothness and factoring polynomials over finite fields’, Information Processing Letters, 38 (1991), 39–42.

[110] V. SHOUP, ‘Searching for primitive roots in finite fields’, Mathematics of Computation, 58 (1992), 369–380.

[111] V. SHOUP, ‘Fast construction of irreducible polynomials over finite fields’, Proc. 4th. annual ACM–SIAM Symposium on Discrete Algorithms (1993), 484–492.

[112] V. SHOUP, ‘Fast construction of irreducible polynomials over finite fields’, Journal of Symbolic Computation, 17 (1994), 371–391.

[113] V. SHOUP, ‘NTL: A Library for doing Number Theory’, http://www.cs.wisc.edu/~shoup/ntl/.

[114] I. E. SHPARLINSKI, ‘On some problems in the theory of finite fields’, Russian Mathematical Surveys, 46:1 (1991), 199–240.

[115] I. E. SHPARLINSKI, ‘Computational problems in finite fields’, Kluwer Academic Publishers, 1992.

[116] I. E. SHPARLINSKI, ‘On bivariate polynomial factorization over finite fields’, Mathematics of Computation, 60 (1993), 787–791.

[117] I. E. SHPARLINSKI, ‘Approximate constructions in finite fields’, Proceedings of the 3rd Conference of finite fields and their applications, London Math Soc, 1996, 313–332.

[118] I. E. SHPARLINSKI AND G. L. MULLEN, ‘Open problems in finite fields’, Proceedings of the 3rd Conference of finite fields and their applications, London Math Soc, 1996, 243–268.

[119] J. H. SILVERMAN, ‘The Xedni Calculus and the Elliptic Curve Discrete Logarithm Problem’, Design, Codes, and Cryptography, to appear.

[120] J. H. SILVERMAN AND J. SUZUKI, ‘Elliptic curve discrete logarithms and the index calculus’, Asia-Crypt ’98.

[121] R. D. SILVERMAN, ‘The multiple polynomial quadratic sieve’, Mathematics of Computation, 48 (1987), 329–339.

[122] C. SMALL, ‘Arithmetic of finite fields’, Marcel Dekker Inc., 1991.

[123] S. A. STEPANOV AND I. E. SHPARLINSKI, ‘On the construction of a primitive normal basis in a finite field’, Math. USSR Sbornik, 67 (1990), 527–533.

[124] D. R. STINSON, ‘Some observations on parallel algorithms for fast exponentiation in GF(2^n)’, SIAM Journal of Computing, 19 (1990), 711–717.

[125] G. TURNWALD, ‘A new criterion for permutation polynomials’, Finite Fields Applications, 1 (1995), 64–82.

[126] D. WAN, ‘Factoring multivariate polynomials over large finite fields’, Mathematics of Computation, 54 (1990), 755–770.

[127] D. WEBER, ‘Computing discrete logarithms with the general number field sieve’, Proc. ANTS II, Lecture notes in Computer Science, 1122 (1996), Springer-Verlag, 337–361.

[128] D. WEBER AND T. DENNY, ‘The solution of McCurley’s discrete log challenge’, Advances in Cryptography – Crypto’98, Lecture Notes in Computer Science, 1462 (1998), Springer-Verlag, 458–471.

[129] D. H. WIEDEMANN, ‘Solving sparse linear equations over finite fields’, IEEE transaction on Information Theory, 32 (1986), 54–62.

[130] E. D. WIN, A. BOSSALEARS, S. VANDENBERGHE, P. D. GERSEM AND J. VANDEWALLE, ‘A fast software implementation for arithmetic operations in GF(2^n)’, Advances in Cryptology – Asia-Crypt’96, Lecture Notes in Computer Science, 1163 (1996), Springer-Verlag, 65–76.

[131] H. ZASSENHAUS, ‘On Hensel factorization I’, Journal of Number Theory, 1 (1969), 291–311.

[132] H. G. ZIMMER, ‘SIMATH – a computer algebra system for number theoretic applications’, http://emmy.math.uni-sb.de/ simath/doc/paper1.dvi.

[133] M. ŽIVKOVIĆ, ‘A table of primitive binary polynomials’, Mathematics of Computation, 62 (1994), 385–386.

[134] M. ŽIVKOVIĆ, ‘Tables of primitive binary polynomials II’, Mathematics of Computation, 63 (1994), 301–306.

[135] H. S. ZUCKERMAN, H. L. MONTGOMERY, I. M. NIVEN AND A. NIVEN, ‘An introduction to the theory of numbers’, John Wiley & Sons, 5th. edition, 1991.

Index