THE UNIVERSITY OF CALGARY

Nonstandard Factoring Methods for Cryptographic Applications

by

Raymond Ball

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER

ENGINEERING

CALGARY, ALBERTA

January, 2009

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "NONSTANDARD FACTORING METHODS FOR CRYPTOGRAPHIC APPLICATIONS" submitted by Raymond Ball in partial fulfillment of the requirements for the degree of M.Sc. IN COMPUTER ENGINEERING.

VS Dmitrov

Supervisor, Dr. Vassil Dimitrov
Department of Electrical and Computer Engineering

Dr. Svetlana Yanushkevich
Department of Electrical and Computer Engineering

Dr. Laleh Behjat
Department of Electrical and Computer Engineering

R. Safavi-Naini

Dr. Rei Safavi-Naini
Department of Computer Science

January 13, 2009

Date

Abstract

In the information age, cryptography is becoming an important part of efficiently transmitting data securely. The RSA cryptosystem is a popular system used to transmit secure information over insecure channels. To allow secure communication over these insecure channels, analysis regarding the strength of the RSA cryptosystem is important to stay one step ahead of those who wish to break it. At the heart of the RSA cryptosystem is the intractable problem of factoring. State-of-the-art methods of factoring, such as the General Number Field Sieve (GNFS) show good potential at factoring; however, few consider the problem as a general optimization problem. Several optimization models will be covered and how the models lead to a well known difficulty in factoring, solving the Diophantine equation. This thesis gives evidence that some chosen nonstandard algorithms, such as optimization, cannot be used as a viable method for efficiently factoring large numbers.

Acknowledgements

I'd like to thank my mom for all her support, pátience, and wisdom. I'd like to thank my family for all their loving support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Background for Factoring in Cryptography

1.1 Basic Introduction and Motivation

Cryptography is used all over the world to keep private data secret. It is becoming a necessity for the new age of information and without it, many virtual infrastructures would not be possible. For example, e-businesses and online shopping require that billing information be encrypted and verified to ensure the person requesting the purchase is the proper person. If the shopper were to send their billing information in the clear, it would be easy to forge another request using the same information, resulting in possible theft of funds and drastically increasing the costs of shopping online.

Figure 1.1 shows a popular communication paradigm when considering traffic over the Internet.

In this setup, we consider Alice and Bob as the people that wish to securely communicate with each other, perhaps to make a purchase. Unfortunately, the nefarious Eve wishes to listen in on Alice and Bob, recording any billing information that may come up in the communication. If Alice and Bob wish to communicate with each other, they must encrypt their traffic using an algorithm called a cipher, thereby making the message unintelligible to Eve. When Alice uses the chosen cipher on her

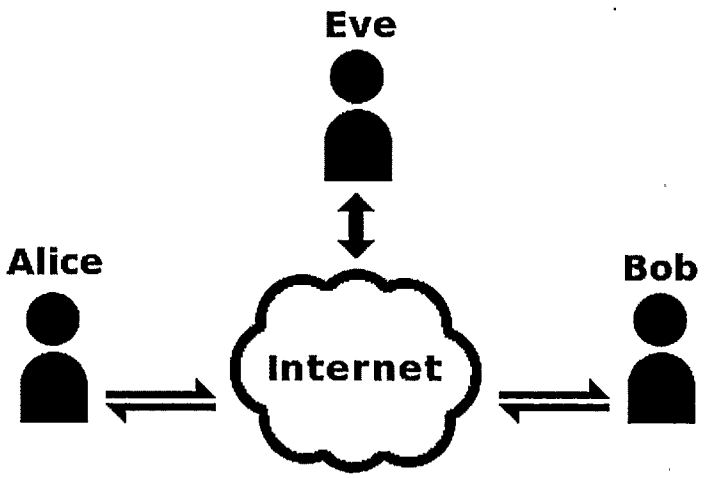


Figure 1.1: The Internet communication paradigm

plaintext message, it is transformed into ciphertext which Eve cannot understand. She may then send the ciphertext over the channel to Bob, provided the cipher she used is reasonably secure.

While encrypting the message, the algorithm performs operations on the plaintext based on a seed called an encryption key. A decryption key can be used to reverse the operations applied to the plaintext to retrieve the message. In some ciphers, the encryption and decryption keys are the same and are called symmetric ciphers. Ciphers that have different encryption and decryption keys are said to be asymmetric. By having a different decryption and encryption key, Alice needs only send her encryption key to Bob who can then use the key to encrypt a message and send the ciphertext to Alice. Alice is then confident that only she can decrypt the ciphertext and read the message because the decryption key was never sent over the insecure channel.

The advantage of an asymmetric cipher is the ability to use the insecure channel to transmit key information without compromising the security of the system; however, in practice asymmetric ciphers are slower than the symmetric ciphers. In the paradigm discussed above, if Alice chooses a symmetric cipher for speed, there is no way to send the key to Bob without Eve potentially intercepting the key and using it to decrypt all the traffic. Her only alternative is to use a special secure channel to send the key. However, in practice, secure channels are very expensive or infeasible. This problem was solved by Whitfield Diffie and Martin Hellman using the first asymmetric algorithm called the Diffie-Hellman key exchange algorithm [24]. Later, this algorithm was replaced by a more versatile cipher called the RSA cipher. In the RSA cipher, the algorithm is secured by the difficulty of factoring large integers and will be the main focus of this thesis.

While asymmetric ciphers are more versatile, for most real world applications, both cipher types are use. When both cipher are used, an asymmetric cipher encrypts a set of randomly chosen symmetric keys. The set of symmetric keys are then used to encrypt the body of the message. The use of both cipher types allows the versatility of asymmetric ciphers with the speed of symmetric ciphers.

The purpose of this thesis is to investigate how the RSA cipher withstands to opimization techniques and contrasts those techniques against with current state-of-the-art methods of breaking the cipher. The thesis will cover several experiments attempting to break the RSA cipher using optimization in both the integer and real number sets.

The thesis is broken into four chapters. In the first chapter, background in the number theory and optimization fields of mathematics are given. The second chapter

covers some current state of the art. Chapter three contains the experiments and briefly discusses the steps taken to perform the computation. Chapter four is the final chapter and it provides a discussion and conclusion from the experiements. Future work is also given in the fourth chapter.

1.2 Number Theory

To understand the RSA cipher, a bit of number theory is required. This section will cover background needed to understand the RSA cipher.

Number theory is one of the oldest mathematics known to man. Its origins date back as far as 800 BC. In number theory, the focus is on the properties of numbers and how they relate to each other. In particular, the integer numbers are focused on. This section will very briefly cover some basic number theory required to understand the upcoming concepts.

1.2.1 Divisibility

The first concept usually covered in number theory is the idea of divisibility [6, 8]. A number n is divisible by d if d divides evenly in n so that there is no remainder. It is denoted $d|n$ and is read "d divides n". For example, $3|15$ and $6|42$ but $12 \not| 25$. If a number $p > 1$ can not be divided by any number except itself and 1 it is called prime. If a number does have a divisor other than 1 or itself, it is called composite. For example, all even numbers greater than 2 are composite because they are all divisible by 2. A biprime or semiprime is a product of two primes. Furthermore, if a prime p can divide n, k times, it is written $p^k||n$ and is read "p^k fully divides n".

1.2.2 Fundamental Theorem of Arithmetic

The fundamental theorem of arithmetic (FTA) says that all numbers can be uniquely represent as a product of primes [8]. Mathematically, this can be written as

$$n = p_1^{\alpha_1}...p_k^{\alpha_k} \forall n \in \mathbb{Z}, k \geq 1, \alpha \geq 1.$$

Representing n as a product of primes is called the canonical factorization (or simply factorization) of n. With the FTA, it becomes clear that when $n = ab$ and $p|n$ where p is prime, p divides either a or b or both. Also, if both a and b are prime, either $p = a$ or $p = b$ or both (in the case that n is a perfect square).

1.2.3 Greatest Common Divisor

The concept of greatest common divisor (or GCD) is central to the study of number theory [11]. As the name suggests, the GCD finds the largest common factors between two numbers. Common notation for the GCD function is $\gcd(m, n) = d$, where d is the largest common factor between m and n. If d is 1, it is said that m and n are relatively prime, or coprime. Some properties of the GCD function are given below.

1. If p is prime, then $\gcd(p, n) = 1$ or $\gcd(p, n) = p$.
2. If $\gcd(m, n) = d, m = dt, n = ds$ then $\gcd(s, t) = 1$.
3. If d is a common divisor of m and n then $d|\gcd(m, n)$.
4. If $m = nq + r$, then $\gcd(m, n) = \gcd(n, r)$.

To efficiently compute the GCD of two numbers, the Euclidean algorithm can be used. Another way of finding the GCD is looking at the canonical factorization of

the two numbers; however, factoring large numbers can be a difficult. The Euclidean algorithm is performed by dividing the reminder until the remainder is zero. It is shown below how these divisions are done:

$$\begin{aligned} m &= nq_1 + r_1, 1 \geq r_1 < n, \\ n &= r_1 q_2 + r_2, 1 \geq r_2 < r_1, \\ &\vdots \\ r_{k-2} &= r_{k-1} q_k + r_k, 1 \geq r_k < r_{k-1}, \\ r_{l-1} &= r_k q_{k+1} + r_{k+1}, r_{k+1} = 0. \end{aligned}$$

The last nonzero remainder r_k is the GCD of m and n.

1.2.4 Modular Arithmetic

Modular arithmetic is another central concept to number theory. It allows the computation on a finite subset of integer numbers in a circular manner. In this system the numbers "wrap around" so that all computation is limited in this subset. This wrap around is achieved by considering the remainder in division. A relation called a congruence relation is used in the notation of modular arithmetic and is expressed as

$$a \equiv r \pmod{d}$$

and is read "a is congruent to r modulo d". This expression says that after removing (or adding) multiples of d in a, we are left with r. Note that neither a nor r have to be less than d. The congruence relation says that two number are equivalent if they were mapped onto a circle of d points. For example, $3 \equiv 10 \equiv 17 \equiv \cdots 3 + 7k$

(mod 7). In practice, when doing modular arithmetic only the numbers less than d are considered because smaller numbers are typically easier to work with. Some properties of congruences are given below:

1. $a \equiv a \pmod{d}$.

2. If $a \equiv b \pmod{d}$ and $b \equiv c \pmod{d}$ then $a \equiv c \pmod{d}$.

3. If $a \equiv b \pmod{d}$ then $b \equiv a \pmod{d}$.

4. If $a_1 \equiv b_1 \pmod{d}$ and $a_2 \equiv b_2 \pmod{d}$ then $a_1 + a_2 \equiv b_1 + b_2 \pmod{d}$.

5. If $a_1 \equiv b_1 \pmod{d}$ and $a_2 \equiv b_2 \pmod{d}$ then $a_1 * a_2 \equiv b_1 * b_2 \pmod{d}$.

6. If $a \equiv b \pmod{d}$ then for any integer k, $ka \equiv kb \pmod{d}$.

One interesting property of this arithmetic is the use of modular exponentiation. In standard arithmetic, computing large exponentials results in large numbers. However, in modular arithmetic numbers can be reduced to be more manageable because of the circular nature of the arithmetic. By considering property 5, it follows that $c \equiv a * b \pmod{d}$ is equivalent to $c \equiv (a \pmod{d})(b \pmod{d}) \pmod{d}$. For example,

$$117649 = 7^6 = (7^3)(7^3) \equiv (2)(2) \pmod{31}$$

because

$$7^3 = 343 \equiv 2 \pmod{31}.$$

To further speed up the process, considering the binary representation of the exponent tells when to multiply and reduce. In the above example, the exponent of

$6 = 110_2$ is read from right to left and the result is multiplied by $7^{2^i} \pmod{31}$ and reduced by 31 every time there is a 1. This results in

$$(7^{2^0} \pmod{31})^0 (7^{2^1} \pmod{31})^1 (7^{2^2} \pmod{31})^1 \equiv (1)(18)(14) \equiv 4 \pmod{31}.$$

In general, modular exponentiation then takes less than $O(\log n)$ multiplications.

1.2.5 Euler's Theorem

One of the most famous results of number theory is Euler's theorem [20]. It states that

$$a^{\phi(m)} \equiv 1 \pmod{m}.$$

In this theorem, a and m are relatively prime positive integers, and $\phi(m)$ is Euler's totient function which computes the number of positive integers less than and relatively prime to m. Clearly, if m is some prime p, then $\phi(p) = p - 1$ because all numbers are relatively prime to a prime number. Another famous theorem that is related to Euler's Theorem is Fermat's little theorem [20]. It states that

$$a^p \equiv a \pmod{p}$$

where p is prime. By multiplying both sides by a^{-1}, it becomes clear how this is a special case of Euler's theorem.

1.2.6 The RSA Cryptosystem

One of the most popular cryptosystem used today is the RSA cryptosystem. The system is named RSA after the last names of the authors Ron Rivest, Adi Shamir, and Len Addleman [17]. In this system, the security relies on the computational

intractability of two problems: factoring a large composite integer into its primes and the RSA number problem. The intractability is achieved with the use of a one-way function. In a one-way function like multiplication, it is easy to multiply two prime numbers together and return the result, but difficult to factor the result returning the prime numbers. If the primes could be factored fast enough, then the cryptosystem would no longer become intractable and the system would fail to keep the information safe. Motivations of research into factoring algorithms test the strength of the intractable problems such as factoring and allow us to stay ahead of people who wish to break the system for malicious reasons.

An interesting aspect of the cryptosystem is that its security is entirely based on the assumption that factoring and the RSA number problem has no efficient solution. There is no known unclassified proof regarding the security of the cryptosystem. Thus, it is unknown whether or not the system is truly secure. The only proofs that are available are proofs that show factoring is equivalently secure to other unproven intractable problems, such as the Diffie-Hellman key exchange algorithm [13]. This means that the security of the system is entirely based off the continuous failure of any algorithm to factor a large biprime and is merely a universally accepted hunch.

As described above, the RSA cipher is an asymmetric cipher. As an asymmetric cipher, two keys are generated: a public encryption key, and a private decryption key. The keys themselves consist of a few integers. How these keys are generated are given below, followed by a more detailed discussion.

1. Choose two large prime numbers p and q.

2. Compute the product of p and q, $n = pq$.

3. Compute Euler's totient function on n as $\phi(n) = (p-1)(q-1)$.

4. Choose a value e such that $1 < e < \phi(n)$ and such that e is relatively prime to $\phi(n)$.

5. Compute d so that it satisfies the congruence

$$de \equiv 1 \pmod{\phi(n)}.$$

The encryption key is then the pair (n, e) and the decryption key is the remaining information (p, q, d). While some steps seem trivial, when computed at the scale of 2^{512} the task is not trivial from a computation perspective.

In the first step, two prime numbers are required. However, guaranteeing a prime number is a difficult task and would reduce the system to unusable at a practical level of security. With current algorithms, generating prime numbers in the 2^{512} range would take too long for practical use. The fastest known primality test to date is the Elliptic Curve Primality Proving algorithm [1]. It has an approximate complexity of $O((\ln n)^5)$. Its worse-case complexity is not yet known. Computing a prime with this method would be impractical for a general encryption suite. To get around this problem, probabilistic prime testing is used to find large primes. Probabilistic primality testing is much faster than deterministic methods. This speed increase comes at the cost of returning a pseudoprime. A pseudoprime is a number that passes a probabilistic primality testing algorithm but is not necessarily truly prime. One such algorithm that is used to find pseudoprimes is Fermat's primality test. While this does not guarantee that p and q will be prime, the probability of the number not being prime can be reduced to a quantity that is negligible without

much effort. The last two steps make use of the Euclidean and extend Euclidean algorithms to quickly satisfy the conditions. Once the keys have been generated, (n, e) is published and used to encrypt the message.

To encrypt the message the sender does the following.

1. Breaks the message into blocks such that a numerical representation of the block of data m, is such that $m < n$.

2. Compute

$$c = m^e \mod n$$

where c is the ciphertext.

3. Stores all the ciphered c values so that the receiver knows how to apply the decryption algorithm.

The exponential in the second step can use modular exponentiation properties to quickly find c.

To decrypt the message, the receiver does the following.

1. Retrieve c from the storage structure of the sender.

2. Compute

$$m = c^d \mod n.$$

3. Reassembles the message with the computed m values.

To see that this cipher works, we put the encryption and decryption computations together:

$$c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}$$

But we know that

$$ed \equiv 1 \pmod{(p-1)(q-1)}$$

so by Fermat's little theorem

$$m^{ed} \equiv m \pmod{p}$$

and

$$m^{ed} \equiv m \pmod{q}.$$

By applying the Chinese Remainder Theorem

$$m^{ed} \equiv m \pmod{pq}.$$

Thus,

$$c^d \equiv m \pmod{n}$$

With the private key (p, q, d) never transmitted and kept safe, there is no way for Eve, the eavesdropper, to directly decrypt the message. To get the private key indirectly, Eve would have to calculate the private key from the public key via factoring n into p and q. This computation is where the importance of factoring comes in. If an efficient algorithm for factoring exists, then the cipher no longer would be considered secure. State-of-the-art factoring methods do not factor quick enough to consider them efficient. These algorithm take more time to complete than the message is considered valid for. Thus, a better factoring algorithm is required for a direct attack on this cipher. More discussion on the state-of-the-art factoring techniques follows in the next chapter.

It is worth mentioning that while computing the private key from the public is one way to break the cipher, it is not the only way to attack the cryptosystem in

general. There are many other ways to attack this cryptosystem that are beyond the scope of this thesis. Some methods include attacking the random number generator used to find p, q, and e. By using a weak random number generator, Eve can make good guesses on what the next number would be and circumvent the cryptosystem. Other methods include attacking how the RSA cryptosystem is implemented. For example, if the primes chosen are of a certain form, n can be easily factored.

From a mathematical and computational perspective, factoring n is the most direct approach to defeating the cryptosystem. By factoring n, p and q become known and the equation $de \equiv 1 \pmod{\phi(n)}$ can be solved in the same manner as the key generator. With the congruence solved, the decryption key (d, p, q) is available for decryption.

1.2.7 An Example of the RSA Cryptosystem

An example of the RSA cryptosystem is provided below. Suppose that Alice wants to send Bob the message

`I'd rather be researching.`

On your standard PC, this would be represented in ASCII as

`73 39 100 32 114 97 116 104 101 114 32 98 101 32 114 101 115 101 97 115 99 104 105 110 103 46.`

To keep the calculations simple, the message will be encrypted byte by byte. Doing byte-by-byte encryption means $n > 255$ to ensure $m < n$, but because all of our values are less than 127, we will find an n such that $127 < n < 256$. The biprime $11 * 13 = 143$ satisfies this condition nicely. To obtain d, e is chosen at random

such that $1 < e < \phi(n)$ or $1 < e < 120$ in this case. Recall that e must also be relatively prime to $\phi(n)$, so $e = 7$ will be sufficient. To compute d, $7d \equiv 1 \pmod{120}$ is solved by $d = 103$. The public key is then $(n = 143, e = 7)$ and the private key is $(p = 11, q = 13, d = 103)$. Continuing with the example, the plain text message needs to be encrypted before transmission. To encrypt, Alice takes each byte m_i and computes $c_i = m_i^7 \mod 143$. By doing the encryption, the plaintext message is transformed into the ciphertext

```
97 39 40 8 24 73 56 104 101 24 8 32 101 8 24 101 115 101 73 115 99
104 105 80 7 16
```

which reads

```
a'(..I8he.. e..eseIschiP..
```

where dots represent some non-symbolic ASCII characters. The ciphertext is then sent to Bob and it is decrypted using the private key in the same manner as it was encrypted.

An important note is that this method of encryption is by no means secure. This example breaks several basic rules of information security. It is here just to illustrate the mathematics used to encrypt a message using an RSA cipher.

1.3 Optimization

The following section covers a field of mathematics called optimization. By modelling factoring as an optimization problem, an attempt can be made to intelligently step towards a solution of a factoring problem. This is fundamentally different from

the current factoring methods that use guessing. Optimization is provided here as background for experiments conducted in chapter 3. Several different types of optimization are attempted in the experiments section and discussed in the conclusion section. They are briefly covered here.

The purpose of optimization is to find an optimal solution to a given mathematical problem [19]. Problems are presented as a minimization or maximization of a function called a cost function. In this section, only minimization of cost functions will be considered because the same principles can be applied to maximization. This cost function f takes inputs x from some set A and computes a value of the input.

$$f : A \rightarrow \mathbb{R}$$

Typically, A is a subset of $\mathbb{R}^n$ but is also frequently a subset of $\mathbb{I}^n$. In addition to the cost function, the problem may also have limitations or conditions such as $g(x) \geq 0$. These conditions also restrict A to a subset of valid inputs. The subset of allowable inputs is known as the feasible solutions. A feasible solution in A that minimizes the cost function is known as an optimal solution. To illustrate these definitions an example is provided. For the minimization problem

$$\begin{aligned} min. \quad f(x) = \quad & (x^2 - x)^2, x \in \mathbb{R} \\ s.t. \qquad x < \quad & 1 \end{aligned}$$

has a cost function $f(x) = (x^2 - x)^2$ and a condition on x such that $x < 1$. Figure 1.2 shows the graph of $f(x) = (x^2 - x)^2$. In Figure 1.2, it is shown that the cost function $f(x)$ has two minimums at 0 and 1. With the condition of $x < 1$, the point $(1, 0)$ is not a feasible solution and it follows that the only optimal solutions is $(0, 0)$. In this example the solution can be found using simple analysis. However,

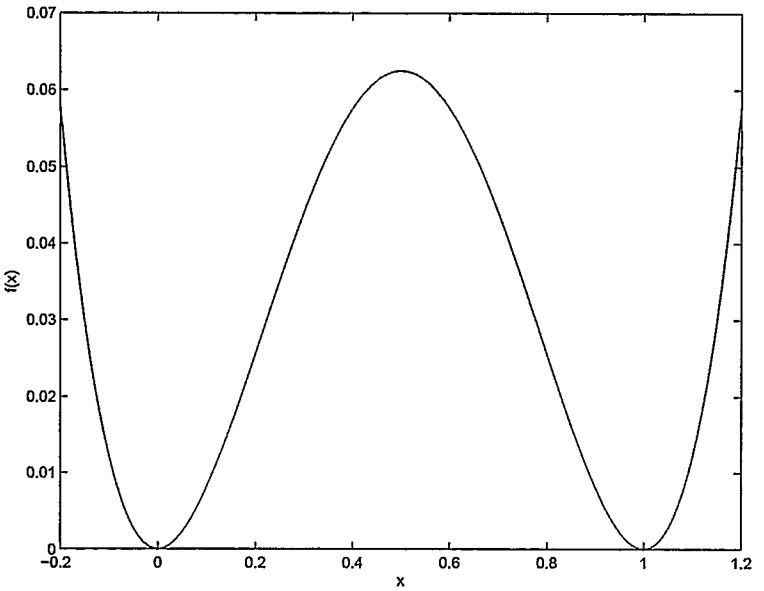


Figure 1.2: A simple minimization of $f(x) = (x^2 - x)^2$ such that $x < 1$.

most optimization problems deal with complicated or multi-solution systems, where analysis may not be possible. Also, in many real world problems, such as dynamical systems, finding the overall best solution may not be necessary. In these cases, where the system's complexity is beyond direct analysis, numerical analysis is used and can be an effective tool for solving most optimization problems.

For complex systems, optimization methods generally start by selecting an initial point x_0 at random, with the limited knowledge of the system. This could be as simple as choosing a random point within an interval the solution is excepted to fall in. Once an initial point is chosen, the initial point is then moved to a point that results in a better solution than the original guess. It is important to note that in some complex systems, a bad movement could be made in hopes of getting to a more advantageous path to an optimum solution. This seemingly bad movement can be illustrated in 1.2. If the system were to start at 0.9, the only movement that would

allow the system to find the globally optimum point is to move away from the local minimum near 1. It is easy to see that a seemingly bad movement to $x < 0.5$ would make it easy for the system to find the global minimum. The point exactly at $x = 0.5$ where both directions seem to look equally good, is called a local maximum.

Regardless of how the system moves, the general objective of an optimization method is to start at some initial point x_0 and calculate movements within the set of feasible solutions to eventually find a satisfactory solution.

In general, an optimization problem is approached by first classifying its type. This is done to exploit some property already discovered for that type of optimization problem. Below is a list of some classes of optimization types that are relevant to this thesis.

- Linear Programming. In linear programming, the optimization problem has the form

$$\begin{aligned} min. \quad & f(x) = c^T x \\ s.t. \quad & Ax \leq b, x \geq 0 \end{aligned}$$

  where x is the current solution, c is a vector of cost coefficients, and A is a matrix of constraint coefficients [3, 21]. Linear programming problems frequently come up in business and economic fields. Algorithms for solving linear programming problems include the simplex, interior point, and projective methods.

- Integer Programming. Integer programming is exactly what the name suggests; optimization in the set of integers. While most methods are concerned with linear problems such as branch-and-bound, there are few efficient methods for

nonlinear integer optimization problem and are usually made for a specific problem [21].

- Quadratic Programming. As one would expect, quadratic programming is a set of methods tailored towards optimization problems that have the quadratic form of

$$\begin{aligned} min. \quad & f(x) = \tfrac{1}{2}x^T Q x + c^T x \\ s.t. \quad & Ax \leq b \\ s.t. \quad & Ex = d. \end{aligned}$$

  Clearly, this is a generalization of a linear programming problem. If Q is zero, the system becomes linear and can be solved with linear programming. If Q is nonzero, different methods can be used depending on whether it is positive definite, positive semi definite, or negative definite. Methods for solving this class of optimization include interior point, active set, and conjugate gradient methods[3, 9].

- Nonlinear Programming. The nonlinear programming class covers nonlinear cost functions and constraints. One of the major challenges with nonlinear programming is local minima[9]. In nonlinear programming, often the algorithms are able to find a local minima but not the global minimum. Algorithms tend to get trapped in a local minima before finding a global minimum. This presents a challenge when applying these methods to cryptographic applications because they require global solutions that must be transformed into integers.

  These types of optimization are usually very difficult as they can have very little predictability or present the method with many undesirable local minima [9].

Methods for solving this class of optimization problem depend greatly on what type of cost function and constrains are involved. Very few algorithms consider finding the global optimum solution because of the difficulty in evaluating or formulating the cost function. Most algorithms attempt to find a local minima in an area of the function that can be approximated by a well-known function. However, this approach means that only a decent solution will be found and not a perfect solution. Unfortunately, cryptography requires that exact solutions be found. These leaves few nonlinear optimization algorithm available for use in this topic.

Other approaches include Newton, quasi-Newton and secant methods [9]. These also suffer from the problem of not identifying global minimums.

- Simulated Annealing Simulated annealing is an optimization method that was developed from the idea of forming crystals from heating and cooling metal. The general idea of annealing is that when the metal is heated, the molecules become excited, giving them a better chance to find a place in the lattice of the metal. Analogous to how the properties of a metal can be altered by continuously heating and cooling a metal, simulated annealing uses probability to help find a solution. The simulated annealing algorithm utilizes a random number to perturb a potentially stuck solution into new possible pathways of finding a solution. While this can be applied to finding global minimum, its random nature makes it difficult for the system to find the best solution [18]. This method can be used together with other algorithms to escape poor solutions.

One algorithm that is used in this thesis is the line search algorithm. In this algorithm the function is approximated as a linear decent to a solution. The algorithm operates as follows:

1. Compute the decent direction p_K.
2. Find an $\alpha_k \in \mathbb{R}$ such that $\phi(\alpha) = f(x_k + \alpha p_k)$ is minimized within some acceptable error.
3. Set $x_{k+1} = x_k + \alpha_k p_k$.
4. Do this until $||\nabla f(x_k)||$ is sufficiently small.

This algorithm requires a initial guess x_0 as a starting point for the optimization. Every iteration of the algorithm increments k by one. One major drawback for this algorithm is that it is inefficient compared to other more specific algorithms. Another drawback is that it has no way of escaping local minima on its own.

One interesting optimization algorithm already present in this thesis is the Euclidean algorithm. Although few seem to suggest it, the Euclidean algorithm exhibits many similarities with an integer programming algorithm using a recursive cost function. It terminates when the GCD cost function is 1.

1.4 Goals

The goal of this thesis is explore the use of optimization to break the RSA cipher by factoring n into p and q. The optimization method of factoring will also be compared to the state of the art. While many believe that optimization will not perform as well as current methods, their is little literature to support this claim. This thesis

will explore this claim by providing several experiments with optimization factoring. Several different approaches to the problem of factoring are given. Experiments in real and integer number sets are covered. Following the experiments, the thesis will conclude with possible reasons why optimization methods lack the means to factor quickly.

Chapter 2

Current State-of-the-Art Factoring

As stated previously, the security of the RSA cryptosystem is heavily reliant on the difficulty of factoring large numbers. While there are many other ways to attack the cryptosystem, factoring is considered the heart of the problem. Currently, there it is no publicly known factoring algorithm available that can factor in a reasonable amount of time [17]. Developing an efficient method for factoring would also lead to other consequences, such as a compromise of the Rabin's digital signature method [16] and its variants [25].

Interestingly, it is unknown whether factorization is intractable. The trust of security is held together by the system's resistance to a barge of failed factoring methods. The strength of security for the RSA cryptosystem is a belief [12, 14] and not necessarily a truth. Furthermore, if a proof of intractability for factorization is developed, this would show that there is no method that can factor any arbitrary number in a reasonable amount of time. However, this still does not rule out the quick factorization of all numbers. For some numbers of a particular form, heuristic methods can be used to factor, regardless of a proof for intractability. A simple example of such an algorithm is developed in section 3.2.

The current record for integer factoring is a 663 bit biprime on May 9th, 2005. This record was part of an RSA Factoring challenge and used an algorithm called the general number field sieve (GNFS) [7].

Properly covering the GNFS is well beyond the scope of this thesis. Therefore,

this section will cover the near current state-of-the-art, the quadratic sieve (QS). The quadratic sieve is the second fastest factoring algorithm and is similar to the GNFS [14].

2.1 Quadratic Sieve

To understand the quadratic sieve (QS), it is necessary to understand the algorithms it was built on. The start of the QS begins with Fermat. Fermat suggested that instead of factoring directly as

$$n = a \times b$$

n can be factored using a difference of squares [15, 2]

$$n = x^2 - y^2.$$

Clearly, n can be factored as

$$n = (x - y)(x + y).$$

Notice that $(x - y)$ and $(x + y)$ may not necessarily be prime. Furthermore, it turns out that x and y will always exist with the exception of a couple trivial cases. Consider x and y as

$$x = \frac{a + b}{2}$$

and

$$y = \frac{a - b}{2}.$$

As long as a or b are not divisible by 2, this holds true because

$$\begin{aligned} x^2 - y^2 &= \left(\frac{a+b}{2}\right)^2 - \left(\frac{a-b}{2}\right)^2 \\ &= \frac{(a+b)^2 - (a-b)^2}{4} \\ &= \frac{a^2 + 2ab + b^2 - a^2 + 2ab - b^2)^2}{4} \\ &= \frac{4ab}{4} \\ &= a \times b \\ &= n. \end{aligned}$$

If $2|a$ or $2|b$ then it would be possible for one to be even while the other is not, making x and y a fraction. The solutions for x and y will then yield non-integer results. Thus, we restrict a and b to odd integers. This is a reasonable restriction because n can be easily checked to be even. If n is found to be even then divide out two until it is not.

The problem is now finding x and y such that their squared difference is n. In Fermat's algorithm, we start x at $\lceil\sqrt{n}\rceil$ and find y until $n = x^2 - y^2$. An important note about this algorithm is that in general, it does not run faster than a trial by division [2]. However, we can gain some interesting results when implementing the algorithm. First, lets consider solving for y^2 given an x^2

$$y^2 = x^2 - n.$$

This reduces the problem to deciding when y^2 is a perfect square, given some x. If y^2 is a perfect square then a solution has been found. If y^2 is not, another x is chosen.

Suppose $y^2 = d$ and that d' is the next trial of x, increasing x means

$$\begin{aligned} d &= x^2 - n \\ d' &= (x+1)^2 - n \\ &= x^2 + 2x + 1 - n \\ &= d + 2x + 1 \end{aligned}$$

so subsequent trails can be found simply by adding 2 to a temporary variable and then adding that. To see how this works, the basic Fermat factoring method is given below.

```
int factor(int n)
{
  int sqrt_n, u;

  sqrt_n = ceil(sqrt(n));
  d = sqrt_n*sqrt_n - n;

  for (u=2*sqrt_n+1; d<n; u+=2)
  {
    if (is_square(d)) return sqrt(d+n) - sqrt(d);
    d+=u;
  }
  return n;
}
```

As stated above, the problem is checking d to be a perfect square. To improve performance of the function that checks d, the last digits can be used to quickly determine if it is not a perfect square. In particular, the last digit of a square can not be 2, 3, 7, or 8. Similarly, there are only 22 possible values for the last two digits.

Another speed increase can also be added. Consider varying both x and y such that some value $r = x^2 - y^2 - n$ is zero. The variables x and y are varied to find a solution $r = 0$. Using the same idea of increasing x by one, two alternate variables $u = 2x + 1$ and $v = 2y + 1$ are used to speed up the incrementing process. The following pseudo-C code can be developed which eliminates the square finding function.

```
int factor(int n)
{
  int sqrt_n, u, v, r;

  sqrt_n = ceil(sqrt(n));
  u = 2*sqrt_n + 1;
  v = 1;
  r = sqrt_n*sqrt_n - n;

  while (r != 0) {
    if (r > 0) {
      // use y to match x
      for (; r > 0; v += 2)
```

```
            r -= v; // r = r - (2y + 1)
        }
        if (r < 0) {
            r += u; // r = r + (2x + 1)
            u += 2;
        }
    }

    return (u - v)/2;        // = b
    //return (u + v - 2)/2; // = a
}
```

In this algorithm, x is incremented like in the previous version, and y is incremented to see if its square will match the square of x. The advantage of this algorithm is that both division and multiplications have been removed, resulting in faster loop cycles. Table 2.1 shows an example of the algorithm factoring 3131.

r	-4	-3	-23	-11	-15	-35	-16	-3	-59	-62	0
u	113	115	117	119	121	123	125	127	129	131	133
v	7	23	33	39	45	51	55	59	65	69	66
x	56	57	58	59	60	61	62	63	64	65	66
y	3	11	16	19	22	25	27	29	32	34	35

Table 2.1: Fermat's factoring algorithm factoring 3131 with square testing eliminated

In the example shown in Table 2.1, y is increased to match the increasing x in hopes of finding a solution to $x^2 - y^2 = n$. A solution is found when $r = x^2 - y^2 - n = 0$, seen in the last column when $x = 66$ and $y = 35$. This is because $66 - 35 = 31$ and $3131 = 31 * 101$.

While Fermat's factoring algorithm as a whole is slower than trial division, it lays the ground work for the next algorithm: Dixon's algorithm [5]. Before getting into Dixon's algorithm, it is important to take a look at an improvement on Fermat's algorithm realized by Maurice Kraitchik [23]. Kraitchik's idea was to look for x and y to satisfy

$$x^2 \equiv y^2 \pmod{n}$$

instead of a difference of squares [2]. If a solution to the congruence can be found and n is odd and contains at least two different primes, there is a 50 percent chance that it will yield a nontrivial factor of n. The congruence will give us a factor because

$$\begin{aligned} x^2 - y^2 &\equiv 0 \pmod{n} \\ n &\mid x^2 - y^2 \\ n &\mid (x-y)(y+x) \end{aligned}$$

. We expect the divisors of n to spread evenly among $(x-y)(y+x)$. If the factors are spread evenly, there is then a 50 percent chance that

$$\gcd(n, x-y) = a$$

will yield a nontrivial factor a of n. The trick then becomes finding the x and y values. To ensure at least one side is a squares, a number r is selected and squared modulo n such that

$$f(r) = r^2 \pmod{n}.$$

If $f(r)$ is a perfect square then we got lucky. Unfortunately, this is usually not the case, so picking these numbers at random is not feasible. However, if we look at the factorization of a several $f(r)$'s, we see that it is possible to construct a congruence

of squares from two or more non-square $f(r)$'s. To construct the relation $x^2 \equiv y^2 \pmod{n}$, Kraitchik made use of the following property concerning congruences. Given the congruences

$$a \equiv b \pmod{n}$$
$$c \equiv d \pmod{n}$$

we know that from the previous chapter we can multiply them together to get

$$ac \equiv bd \pmod{n}.$$

So with a bunch of smaller non-square congruences, a much larger solution to the congruence of squares can be found by combining the smaller ones together. This basic idea led to the first industrial strength factoring algorithms. Among them was one called continued fraction (CFRAC) algorithm [4], which was arguably one of the first truly large factoring algorithms. This improvement also led to Dixon's algorithm. It was yet another stepping stone to even more powerful factoring algorithms.

Dixon built on this idea by using a more systematic approach than Kraitchik originally proposed. Dixon used the properties of linear algebra to find a combination of the factored exponents to satisfy that a product of $f(r)$'s be a perfect square [5]. The exponents work linearly because multiplying two congruency together results in an addition of exponents. For example,

$$2^4 \times 3^2 = 144 \equiv 1 \pmod{143}$$

and

$$3^2 \times 5^2 = 225 \equiv 82 \pmod{143}$$

can be multiplied together to make

$$2^4 \times 3^{2+2} \times 5^2 = 32400 \equiv 82 \equiv 82 * 1 \pmod{143}.$$

Using this property, non-square congruences can be put together to make a large congruency of squares. With both sides being squares, their difference can be tested with the gcd for factors of n as proposed by Kraitchik. The steps performed in Dixon's algorithm are given below.

1. Generate a set of completely factored smaller congruences which can be used to construct a larger solution to $x^2 \equiv y^2 \pmod{n}$.

   (a) Choose a random integer r_i and compute

   $$f(r_i) = r_i^2 \pmod{n}.$$

   (b) Attempt to find trivial factors for $f(r_i)$ up to some divisor d. If it is prime or has factors above d, choose another r_i and try again.

   (c) If $f(r_i)$ factors easily, record the number of times each prime factors out modulo 2. For example, if $f(r_i) = 30870 = 2^1 \times 3^2 \times 5^1 \times 7^3$, record

   $$v_i = \begin{bmatrix} 1 \\ 2 \\ 1 \\ 3 \end{bmatrix} \equiv \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \pmod{2}.$$

   (d) Loop at least d times to gather enough factors for construction.

2. Solve the equation

$$\begin{bmatrix} v_{11} & v_{12} & \dots & v_{1n} \\ v_{21} & v_{22} & \dots & v_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \dots & v_{nn} \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \pmod 2$$

where v_i are the factorizations in 1.c and c is a solution to make the congruence a square. This can be solved using Gaussian elimination. Although typically, more sophisticated algorithms are used in practice.

3. The potential solution to $x^2 \equiv y^2 \pmod n$ is then given as

$$\prod_k f(r_k) \equiv \prod r_k^2 \pmod n$$

where the products are taken over all k for which $c_k = 1$.

4. Check gcd for non-trivial factor of n.

5. If the algorithm fails to produce a factor, start over with a different set of r_i's.

In step 1.c, the factored exponents are recorded modulo 2 because the only interesting part is if all the exponents have the same parity. For example, if we factor a number to be $22325625 = 3^6 \times 5^4 \times 7^2$ then we know it is a square because $22325625 = (3^3 \times 5^2 \times 7)^2 = 4725^2$. By doing this we can simplify the complexity of solving the system of equations in step 2 by taking advantage of binary values and the sparsity of the matrix. In step 2, a linear combination of the exponents from the factored numbers is used to construct a large product of squares.

In this algorithm, most of the work is done factoring all the $f(r_i)$'s, followed by the work required to solve the V matrix. Although, the matrix V will be very large,

it will be sparse and contain only binary entries. Recall that only binary values will be present in the matrix because the computation is done modulo 2. Specialization on current algorithms can be used to take advantage of these properties. Gaussian elimination is commonly suggested for solving the V matrix for simplicity; however, more efficient methods such as structured Gaussian elimination, the Wiedemann algorithm, and conjugate gradient methods exist [5, 2].

Not surprisingly, this algorithm has a large overhead cost from all the factoring needed to build up the V matrix and thus, is not suited for small n. In fact, for the case where n is large but made up of both small and large primes, it is worth trying simpler algorithms first. By using methods that are simpler, the smaller factors can be quickly removed without the overhead. At this point, if a large n still remains and is tested to be composite with a primality testing algorithm, then using an algorithm such as Dixon's algorithm can be considered.

The quadratic sieve (QS) [15] is known to be the second fastest factoring algorithm to date. It was invented by Carl Pomerance in 1981 by building on top of Dixon's algorithm. By incorporating a sieve into Dixon's algorithm, the V matrix can be constructed much faster than by selecting integers to factor at random. The name of the quadratic sieve comes from the sieving process used to remove factors that will not factor easily. The quadratic part of the name refers to the squaring done while generating number to be factored.

There are many different variations on the QS [2, 15, 14], ranging from changes in the sieving process, to changes in the function generator of the number to be factored. The variation chosen for this discussion is one that sieves factors using something similar to the sieve of Eratosthenes.

To begin with the QS, a list of primes needs to be found, known as the factor base. To construct our factor base, some brief discussion on Legendre symbols is required. During the late 18^{th} and early 19^{th} centuries, mathematicians were concerned with finding solutions to problem of quadratic residues. A quadratic residue is a integer n that can be expressed as

$$x^2 \equiv n \pmod{p}.$$

The Legendre symbol [8] is then defined to be

$$\left(\frac{n}{p}\right) = \begin{cases} 0 & \text{if } p|n \\ 1 & \text{if } n \text{ is a quadratic residue modulo p} \\ -1 & \text{if } n \text{ is a not quadratic residue modulo p} \end{cases}$$

By considering the Legendre symbol, a list of d primes are found for which quadratic residues are possible. With this list of primes, we solve the quadratic residue associated with the prime. The Legendre symbol is considered first because theorems associated with Legendre symbols can be used to determine if a prime is a quadratic residue quicker than solving the quadratic residue itself [2]. The quadratic residue

$$t^2 \equiv n \pmod{p_i}$$

will yield two values, t and $-t$. The factor base and the solution are stored for later use in the sieving process. With the factor base prepared, we begin to generate the list of numbers that could be potentially factorized. Similar to Dixon's algorithm, numbers of the form

$$f(r) = r^2 - n$$

are generated. Normally, r is started at $\lfloor\sqrt{n}\rfloor$ and increased by one until $d+1$ factors have been found [15, 2]. Starting r at $\lfloor\sqrt{n}\rfloor$ causes $f(r)$ to be smaller and have a better chance of being factored. A sieve is then used to find integers that can be factored completely over the factor base.

There are many ways to approach this sieving process. One such method is simply to factor each number and check its prime components. However, a better approach is to make use of the ideas from the sieve of Eratosthenes. Beginning with the first prime in the factor base, we locate the first number r in the factor set such that $r \equiv t \pmod{p_1}$, where t is one of the solutions to the quadratic residue of p_1. The corresponding value $f(r)$ will contain the prime factor p_1 because

$$r^2 \equiv t^2 \equiv n \pmod{p_1}$$

so

$$p_i | r^2 - n = f(r)$$

and every p_1^{th} number after that will also divisible by p_1. All of the number that are p_1 steps from the first, as well as all the matches for the alternative solution $-t$ are then divided by p_1 until no p_1 factors remain in that number. This same process is done for each p_i in the factor base. The resulting list will contain 1's only where the $f(r)$'s can be completely factored by the factor base. All of these numbers are then completely factored and exponents recorded modulo 2. Alternatively, the number of times the number was divided by p_i can be recorded during the sieving process to avoid doing the factoring work twice. Finally, the system is solved for a linear combination of the factored number in the same manner as Dixon's algorithm.

A more concise summary of the algorithm is given below.

1. Find a factor base of primes (p_i) which satisfy

$$t^2 \equiv n \pmod{p_i}$$

or in other words, primes whose Legendre symbol equals 1 or

$$\left(\frac{n}{p}\right) = +1.$$

2. Solve the congruences

$$t^2 \equiv n \pmod{p_i}$$

for each p_i in the factor base found in the previous step.

3. Using a sieve, find r and $f(r) = r^2 - n$ pairs such that $f(r)$ can be completely factored by the factor base. If the factor base uses d primes, then $d+1$ factored $f(r)$'s are needed for to solve the system.

4. Solve the system

$$Vc = 0$$

using the factored $f(r)$'s found from the sieving process.

5. Use the solution to V to construct a large product yielding a perfect square.

6. Check the congruence of squares with the gcd for a factor in n.

7. If the process fails to find a factor, try a different $f(r)$ generator or try a different range for r.

2.1.1 An Example of the Quadratic Sieve

An example is provided to get a better understanding of the process. Suppose the number $n = 1817$ is to be factored using an $f(r) = r^2 - n$, where r starts at $\lfloor\sqrt{1817}\rfloor = 42$.

1. To find a factor base of primes, the Legendre symbol of each prime starting from 2 is checked[1] and the primes

$$F = \{2, 7, 13\}$$

are found to have a Legendre symbol of 1.

2. To help with the sieving in the next step, the quadratic residues for the factor base are solved. The solutions for the factor base are shown in table 2.2.

p	x	$-x$
2	1	1
7	2	5
13	6	7

Table 2.2: Factor base and solutions for an example using the quadratic sieve

3. With 3 factors in the factor base, 4 solutions will be needed. The $(r, f(r))$ pairs $(43, 32), (45, 208), (51, 784), (123, 13312)$ are found from sieving. Each can be factored with the factor base and are shown in table 2.3.

4. The V matrix is then constructed from the system of quadratic congruences

[1]Usually this list of primes starts at -1 to realize $f(r)$'s that are negative.

r	$f(r)$		
43	2^5	7^0	13^0
45	2^4	7^0	13^1
51	2^4	7^2	13^0
123	2^{10}	7^0	13^1

Table 2.3: System of quadratic residues needed to construct a larger congruence of squares

and then reduced modulo 2.

$$V = \begin{bmatrix} 5 & 0 & 0 \\ 4 & 0 & 1 \\ 4 & 2 & 0 \\ 10 & 0 & 1 \end{bmatrix} \equiv \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \pmod 2$$

By inspection, row three gives a solution right away because

$$f(51) = (51)^2 - 1817 = 784 = 28^2 \equiv 51^2 \pmod{1817}$$

is already a congruence of squares.

5. Checking the gcd of this result gives

$$\gcd(51 + 28, 1817) = 79$$

and

$$\gcd(51 - 28, 1817) = 23$$

so there is no need to try something else.

Chapter 3

Optimization Factoring

In contrast to current state-of-the-art methods, this thesis approaches the factoring problem from the perspective of optimization. Instead of attempting to make an intelligent guess at a solution as seen in the quadratic sieve, optimization factoring uses directed movement towards a goal.

Two types of factoring optimization were considered. The first type of factoring was done in the reals. Releasing the restriction of keeping p and q only in the integers allows the use of standard optimization techniques. The second type of factoring is done within the integers but considers the binary representation in matrix form and attempts to factor using a bit moving game.

In all the cases covered, it is assumed that n is a biprime. If n was not a biprime, a modern primality test can be used to quickly determine the primality of the factors. If any of the factors fail the primality test, the algorithm can be recursed on the composite factor until the prime factors are found.

3.1 Optimization Factoring in the Reals

Finding the factors of a biprime using optimization in the reals presents two major problems. The first problem is the definition of the cost function that allows the minimization to get closer to a solution each iteration. For example, a cost function could be $E = |n - xy|$, where x and y are the potential solutions. As this thesis

will discuss, this cost function alone is insufficient to locate factors of n. The second problem is finding a solution that is in or near the integers. This process will be called *digitization* because it is responsible for bringing the solution back into an integer or digit. In this approach, the digitization of the solution is done by using either conditional optimization or building it into the cost function.

To test the functions quickly, MATLAB was used to compute the convergence of the optimization. The functions *fminunc* and *fmincon* were used to compute unconditional and conditional optimization problems, respectfully. The specific method of optimization was chosen by MATLAB as a line search unless otherwise stated.

3.1.1 Unconstrained Digitizer Addition

The first group of systems that were considered primarily use addition to obtain the contour. They all have the form

$$f(x,y) = s_1 e(x,y) + s_2 d(x,y) \tag{3.1}$$

where x and y are the attempted factors of the optimization, $e(x,y)$ is an error function, $d(x,y)$ is a digitization function, and s_1 and s_2 are scaling factors for the error and digitization functions, respectively. Several different types of $e(x,y)$ and $d(x,y)$ were chosen for testing.

The first function tested was

$$f(x,y) = s_1(n - xy)^2 + s_2(2 - \cos(2\pi x) - \cos(2\pi y)). \tag{3.2}$$

In this function, the error portion $e(x,y) = (n - xy)^2$ calculates the error directly and sinusoids as $d(x,y) = 2 - \cos(2\pi x) - \cos(2\pi y)$ persuade x and y into integer

values. To help visualize how the sinusoids are working, $f(x) = 1 - \cos(2\pi x)$ is graphed in Figure 3.1. The cost function in equation 3.2 was primarily chosen for

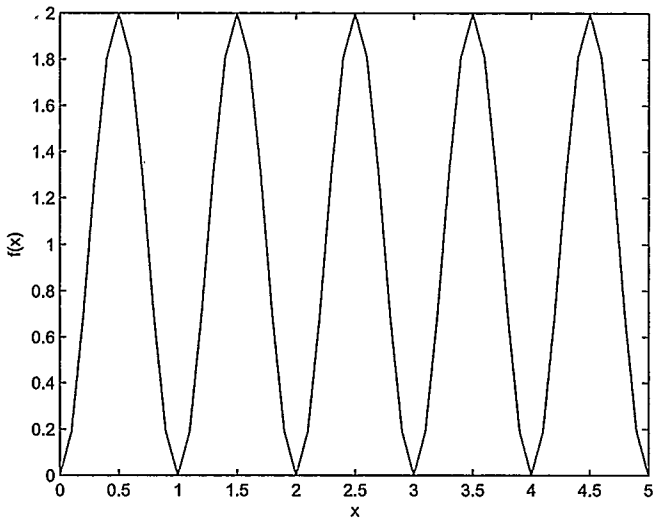


Figure 3.1: A sinusoid as a digitizer

its simplicity. The function only has two variables. Thus, it is easy to graph and visualize. Figure 3.2 shows the function graphed. It is easy to see that the slope is contributed by the error function portion $e(x, y) = (n - xy)^2$. The digitization function contributes by adding bumps to the contour. In Figure 3.2, the scaling values of s_2 is increased to show the contribution of the digitization function. The bottom of the trench or trough made by $e(x, y)$ on the graph represents solutions in the reals. By adding $d(x, y)$ the function is limited to only the (p, q) and (q, p). Also, notice in Figure 3.2 that there is some symmetry in the graph and that the solutions to the system are similar distances from the edge of the graph. This symmetry comes from the commutativity property of multiplication. Looking at the graph, this mirror means that everything on one side of the $x = y$ line is the same as on

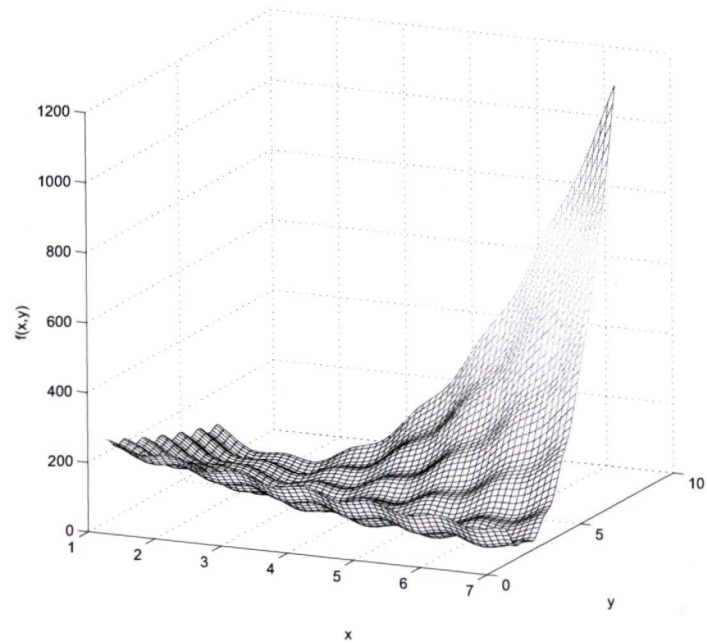


Figure 3.2: Equation 3.2 is graphed

the other side. The number of solutions is guaranteed by the uniqueness of prime factorization and the assumption that n is a biprime.

In the computations, the scaling variables s_1 and s_2 were set to 1 and 2, respectfully. The scaling variables were arbitrarily chosen as a starting point for designing the system. The starting point for the optimization is picked at random between zero and slightly higher than the largest expected prime. The initial points were selected slightly higher than the highest expected prime to evaluate the robustness of the system and to see how the optimization would behave around the local minima found near the highest prime. For the biprime 481, the range of initial vectors is $[0, 40]$ as the expected factors are 13 and 37. Figure 3.3 shows the results of the system as a histogram of 1000 trials attempting to factor the biprime 481. Figure 3.3

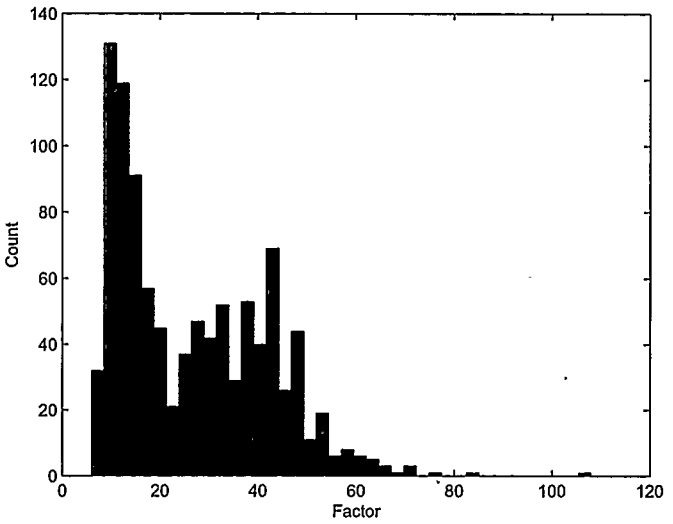


Figure 3.3: Results of direct optimization equation 3.2 factoring 481

shows that 13 is found 130 times and 37 is found 30 times, giving an approximate 16% success rate. This result is promising for a first attempt, but insufficient for practical use. While this system shows success 16% of the time, this is just good luck. However, it is clear that the system is trying to find a correct answer as there is a collection of solutions around 13 and 37. A second experiment was run on the larger biprime of 102313 which is the product of 101 and 1013. In contrast to the previous experiment, the factors were only found approximately 0.04% of the time.

Finally, to get an idea how this approach works for a range of biprimes, a list of biprimes was factored for success rate and timing. Table 3.1 shows the optimization attempting to factor increasingly larger biprimes. The timing was recorded by MATLAB's profiling functionality. It reported the amount of time the CPU spent in the main program. The success rate was computed from the number of times the optimization found a solution near a correct biprime. A correct solution was defined

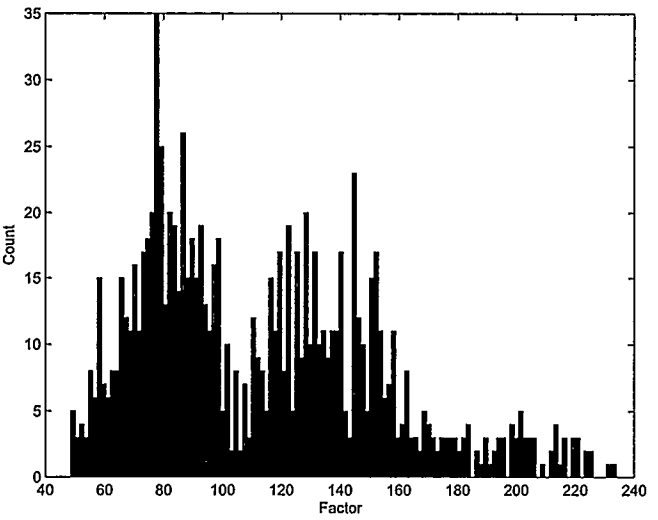


Figure 3.4: Results of the direct optimization equation 3.2 factoring 102313

Biprime	*Factors*	*Success(%)*	*Time(s)*
35	5×7	3	59.801
481	13×37	16	76.341
1243	11×113	0.06	81.580
11413	101×113	0.05	88.958
102313	101×1013	0.04	116.191
1571099	157×10007	0.01	254.776

Table 3.1: Table of direct optimization success and timing results

as the correct factor with the faction part truncated. Table 3.1 shows an increase in time of the computations as the size of the number grows. The success rate is also dropping off as the biprime increases in size.

To investigate why the optimization was failing, a more detailed look of the results were generated. Figure 3.5 shows a more detailed histogram of the results for the biprime 481. In Figure 3.5, it can be observed that some of the solutions are failing to be persuaded into integers. Solutions between 5 and 20 frequently do not have

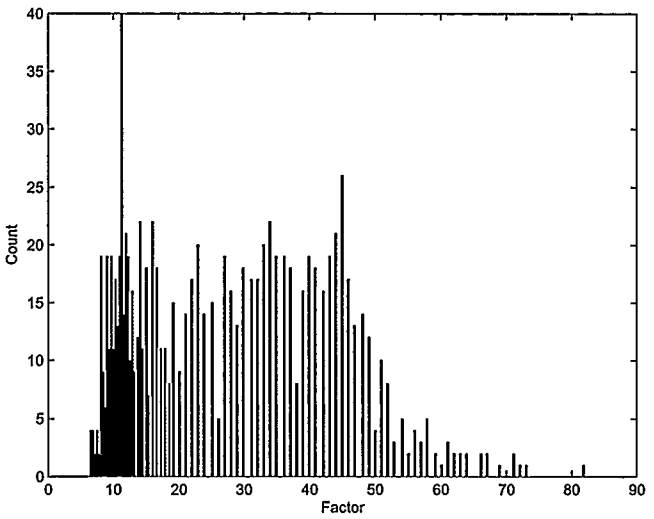


Figure 3.5: Detailed result of the direct optimization equation 3.2 factoring 481

integer solutions. To correct this, experiments with the scaling variables s_1 and s_2 were conducted.

Adjusting s_2 modifies the systems tendency to find integer values and produces the same overall results. In Figure 3.6, s_2 is increased from 2 to 10. These values for s_1 and s_2 are chosen arbitrarily to see how the system would react. When s_2 is modified, no clear factor is identified. It is clear that more sophisticated methods for scaling the optimization would be needed. These are covered in the next subsection with constrained optimization. Before that is done, a comparison between the results of 481 and 102313 show that these two numbers have similar distributions. These two biprimes are expected to have different solutions at different locations. More specifically, 481 is expected to have more solutions around 13 and 37, and 102313 is expect to have solutions around 101 and 1013. Looking at both graphs shows a decrease in the optimization's tendency to find larger numbers in general. Most of

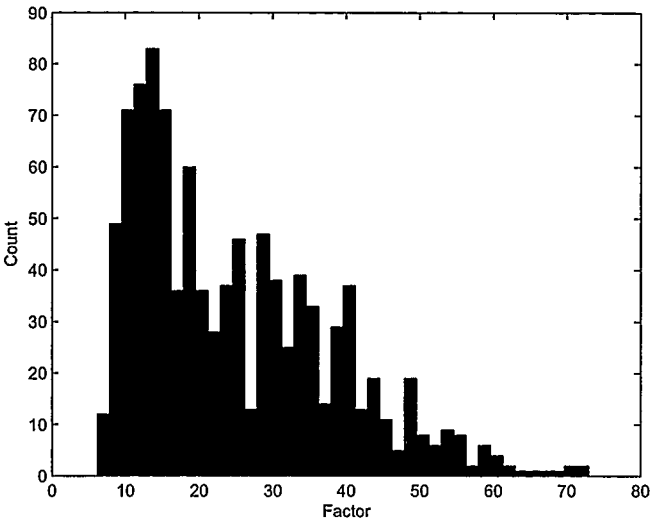


Figure 3.6: Results of the direct optimization equation 3.2 factoring 481 and adjusted scaling variables

the solutions tend to be within the lower valued sections of the figures. Recall that when constructing the cost function, the error function $e(x, y)$ was responsible for solving for the factors of n. This suggests a failure of the error function portion $e(x, y)$.

With the results of the direct approach failing to identify factors, a different error function was considered. The next cost function that was tried was

$$f(x, y) = s_1(n - x^2 + y^2)^2 + s_2(2 - \cos(2\pi x) - \cos(2\pi y)). \tag{3.3}$$

Instead of solving for p and q directly, like in equation 3.2, Fermat's factoring method is incorporated into the optimization. Recall that any composite n not divisible by 2 can be written as a difference as squares $n = x^2 - y^2$. The difference of squares can then be factored $n = (x + y)(x - y)$. If the number 481 is to be factored, then the answers return from the optimization would be $\frac{37+13}{2} = 25$ and $\frac{37-13}{2} = 12$. Using

this idea, equation 3.3 is constructed and run in the same experiment as equation 3.2. The results of the experiment are given in Figure 3.7. As Figure 3.7 shows, there is

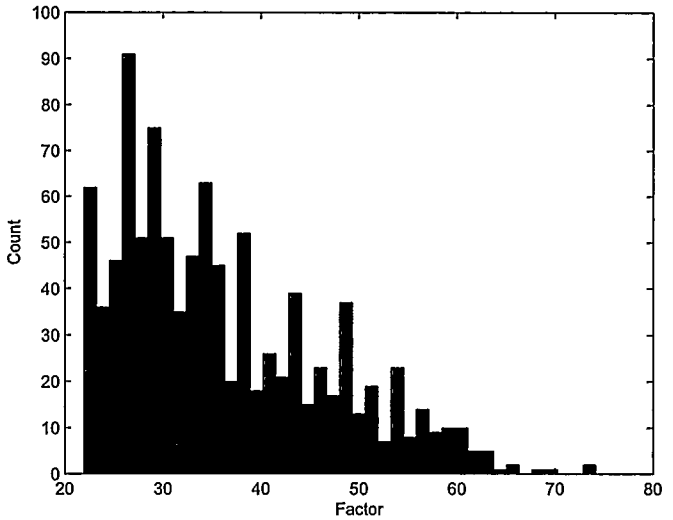


Figure 3.7: Results of the Fermat optimization equation 3.3 factoring 481

not much evidence that the number 481 is being factored properly. Figure 3.7 also bears some similarity with Figure 3.3. This suggests that the change in cost function has not given the system any advantage in finding factors of n.

Finally, in an effort to change both the error function and digitization function, the problem is transformed into a binary representation. The last equation discussed using the addition form is

$$f(\mathrm{x},\mathrm{y}) = [s_1(n - \sum_{i=1}^{L} x_i 2^{i-1} \sum_{j=1}^{L} y_j 2^{j-1}) + s_2(\sum_{i=1}^{L}(x_i^2 - x_i)^2 + \sum_{j=1}^{L}(y_j^2 - y_j)^2)]^2. \quad (3.4)$$

In equation 3.4, the factors are found by considering the binary representation of x as

$$\mathrm{x} = \sum_{i=1}^{L} x_i 2^{i-1} \quad (3.5)$$

where i is the i^{th} bit, L is the total number of expected bits, and x_i is an entry in vector $\mathbf{x}$ that is a potential solution. The vector $\mathbf{y}$ is considers in the same manner as $\mathbf{x}$. Also, in equation 3.4 the scalars s_1 and s_2 are used as scaling variables as in the first two experiments.

It is worth noting that because the factors are unknown, the length of the vector L must be large enough to hold the largest prime in the biprime. This means that $\lceil \frac{\log_2 n}{2} \rceil \leq L \leq \lceil \log_2 n \rceil$ because we expect one factor to be less than or equal to $\sqrt{n}$ and the other greater than or equal to $\sqrt{n}$. It is also unknown which factor will end up in which variable. Thus, both $\mathbf{x}$ and $\mathbf{y}$ are of size L. For simplicity reasons, some cheating is done with regards to choosing the size of L. Instead of calculating L from n alone, the maximum of the two chosen prime factors is used for the bit length L. The size of L is then simply the length of the largest binary represent prime factor.

By considering equation 3.5, an integer solution can be found if each bit is either zero or one. This can be achieved by considering a variable that is "pulled" into zero or one. In the case of equation 3.4, the function

$$f(x) = (x^2 - x)^2$$

is used to cause the bits to tend towards zero or one. The graph of $f(x)$ above is shown in Figure 3.8 As Figure 3.8 shows, the function has two global minimum at 0 and 1. The function makes for a good digitizer of binary numbers because the function is zero at both global minima. The function also tends towards either of these minima except at precisely $\frac{1}{2}$. Considering $f(x)$ for all bits causes the entire system to tend towards an integer solutions. Summing up all digitizers yields the

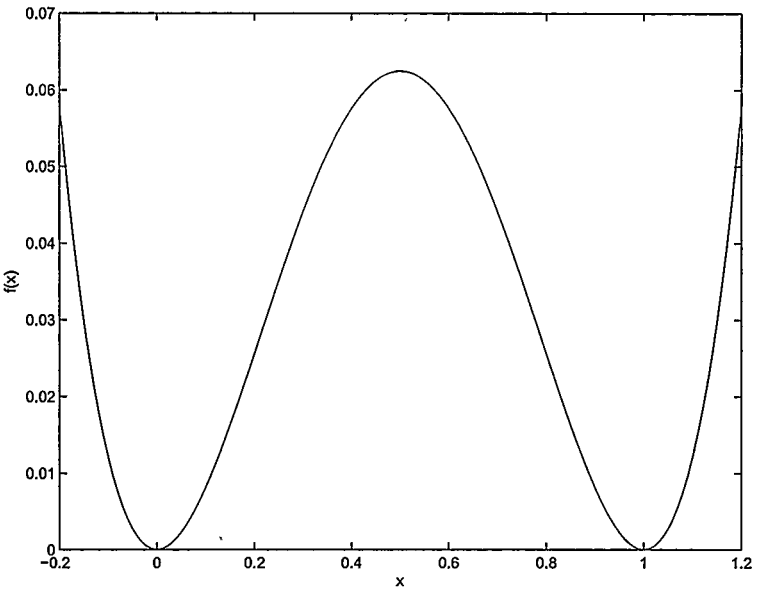


Figure 3.8: Graph of the function $f(x) = (x^2 - x)^2$

first half of equation 3.4

$$d(\mathbf{x}) = \sum_{i=1}^{L} (x_i^2 - x_i)^2$$

and is nonzero for all $x_i \neq 0$ or $x_i \neq 1$.

The error of the factorization is similar to equation 3.2, except that the binary represented vector $\mathbf{x}$ and $\mathbf{y}$ are converted to its corresponding value. Considering the summation of 3.5 makes it easy to see that

$$e(\mathbf{x}, \mathbf{y}) = n - xy = n - \sum_{i=1}^{L} x_i 2^{i-1} \sum_{j=1}^{L} y_j 2^{j-1}.$$

For the optimization experiments on equation 3.4, the scaling variables s_1 and s_2 were both set to 1. To compute the results, 1000 trials were done on the biprime 481 and are shown in Figure 3.9. The initial vector was chosen at random within the range $[0, 1]$ for each bit. This system also fails to factor 481. However, the distribution of the chosen solutions has changed.

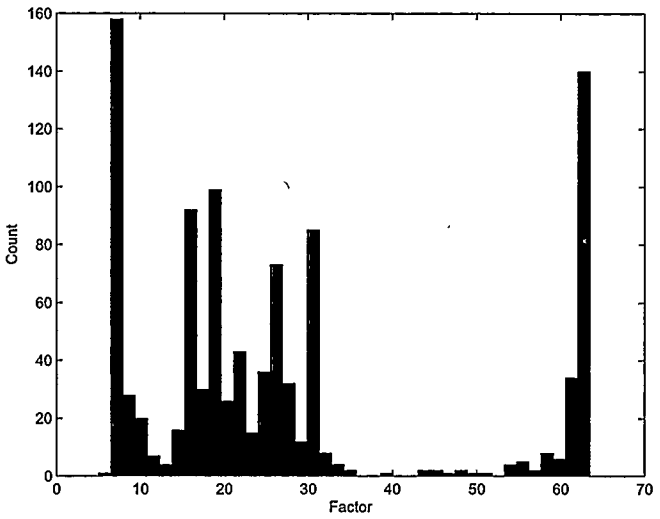


Figure 3.9: Results of the bit optimization equation 3.4 factoring 481

3.1.2 Constrained Digitizer Addition

By considering the integer persuasion function as a constraint and not as part of the cost function, the problem can be rewritten as a constrained optimization problem. In the previous section, scaling variables were used to amplify different potions of the unconstrained cost function. In constrained optimization, those scaling variables are built into the optimization itself. This is accomplished as a Lagrangian multiplier within KKT conditions[10].

In the context of the previous experiments, the digitizer portion $d(x, y)$ of equation 3.1 is treated as a set of constraints instead of part of the cost function. This section goes over all the optimizations done in section 3.1.1 as constrained optimizations. Each experiment is conducted in the same manner as the unconstrained case. Note that no scaling factors s_1 and s_2 were necessary because they are included as part of the KKT conditions used in constrained optimization.

The direct optimization given in equation 3.2 is transform into

$$\begin{aligned} min. \quad f(x,y) &= \quad (n-xy)^2 \\ s.t. \quad d(x) &= \quad 1-\cos(2\pi x) \\ d(y) &= \quad 1-\cos(2\pi y). \end{aligned} \tag{3.6}$$

Figure 3.10 shows the results of the constrained direct optimization. By using con-

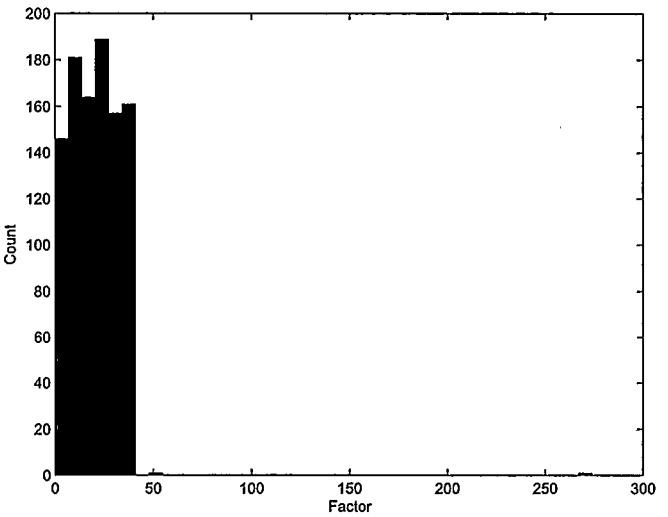


Figure 3.10: Results of the constrained direct optimization equations 3.6 factoring 481

straints, Figure 3.10 shows the results are still similar to the methods use in the unconstrained optimization. However, a couple differences are observed. First, the optimization chose a couple very large valued solutions. Second, the values were strictly in the integers. Looking at a more detailed view of the results show that the solutions are integers.

Figure 3.11 shows a detailed view of the experiment and gives a better view of the distribution as all the large solutions that were greater than 40 have been removed

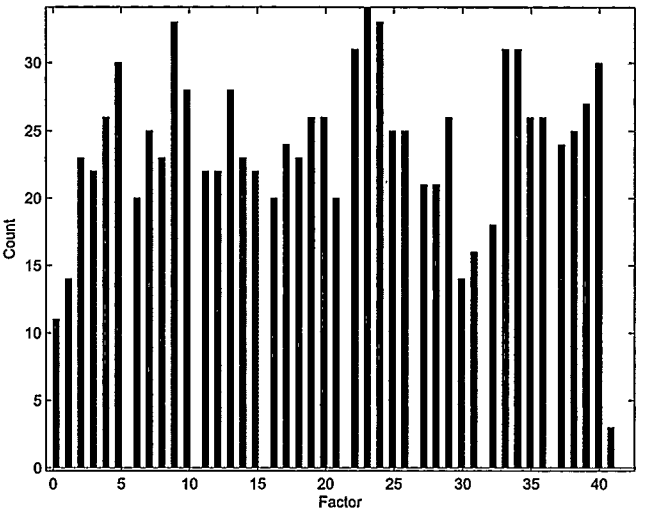


Figure 3.11: Detailed results of the constrained direct optimization equations 3.6 factoring 481

from view. Figure 3.11 shows that all the solutions were integers. Figure 3.11 also shows that the factors 13 and 37 and not identified by the optimization. While the solutions are successfully identifying numbers in the integers, the factors are not.

In an attempt to take advantage of the success in the constraints, the constraints and cost function are switched. The optimization would then be

$$\begin{aligned} min. \quad f(x,y) = \quad & 2 - \cos(2\pi x) - \cos(2\pi y) \\ s.t. \quad d(x,y) = \quad & n - xy. \end{aligned} \tag{3.7}$$

In this optimization the square is removed in the expression $(n - xy)^2$ because the constraint can be expressed as an equality constraint. Figure 3.12 shows the results of the constrained direct optimization with the cost function and constrains switched. The results shown in Figure 3.12 look similar to other experiments. Figure 3.12 shows 13 being found approximately 110 times and and 37 found approximately 25 times, giving an approximate 14% success rate. These results are similar to the

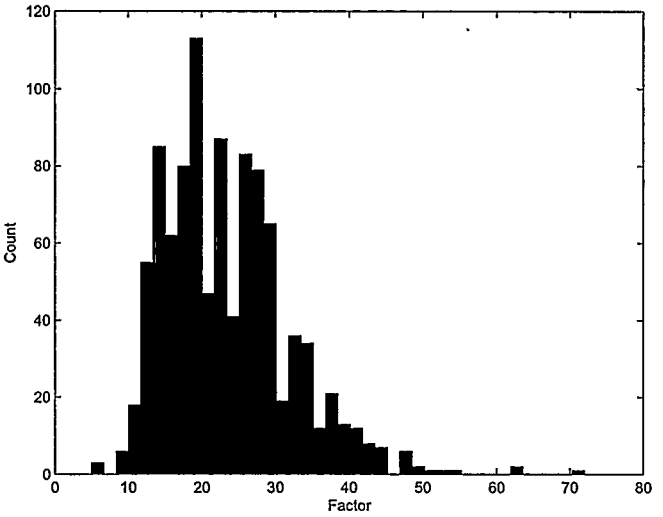


Figure 3.12: Results of the constrained direct optimization equations 3.7 factoring 481

results given in the first experiment shown in Figure 3.3.

The constrained Fermat optimization is done next, equation 3.3 is transformed into

$$\begin{aligned} min. \quad f(x,y) &= (n - x^2 + y^2)^2 \\ s.t. \quad d(x) &= 1 - \cos(2\pi x) \\ d(y) &= 1 - \cos(2\pi y). \end{aligned} \tag{3.8}$$

Running the experiment on this system yielded the results shown in Figure 3.13. By Figure 3.13, it is clear that no advantage is gained by using this approach.

Lastly, the bit represented constrained optimization is transformed from equation 3.4 as

$$\begin{aligned} min. \quad f(\mathbf{x},\mathbf{y}) &= n - \textstyle\sum_{i=1}^{L} x_i 2^{i-1} \sum_{j=1}^{L} y_j 2^{j-1} \\ s.t. \quad d(x_i) &= (x_i^2 - x_i)^2 = 0 \\ d(y_i) &= (y_i^2 - y_i)^2 = 0. \end{aligned} \tag{3.9}$$

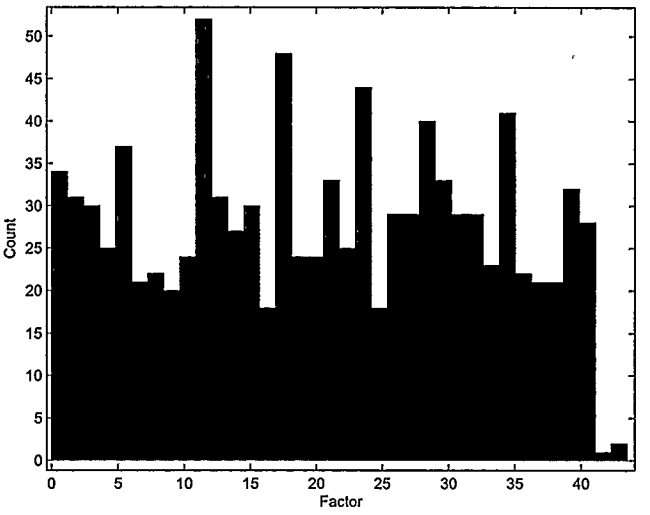


Figure 3.13: Results of the constrained Fermat optimization equations 3.8 factoring 481

Figure 3.14 shows a histogram of the factors chosen during the optimization. Again, Figure 3.14 shows factors are not being identified any better.

3.1.3 Integer Persuasion Scaling

A second form for factoring in the reals was considered. In this form, the digitizer from equation 3.1 is divided by the entire system to create amplification of the global minima. The idea is to create a more prominent global minima for the optimization to find. Consider the scaling variable s_2 from equation 3.1. By putting emphasis on the digitizer only at the solution point, the system can move freely towards a solution. Thus, can consider the form

$$f(x,y) = s_1 e(x,y) + s_2 \frac{d(x,y)}{e(x,y) + d(x,y) + \epsilon} \tag{3.10}$$

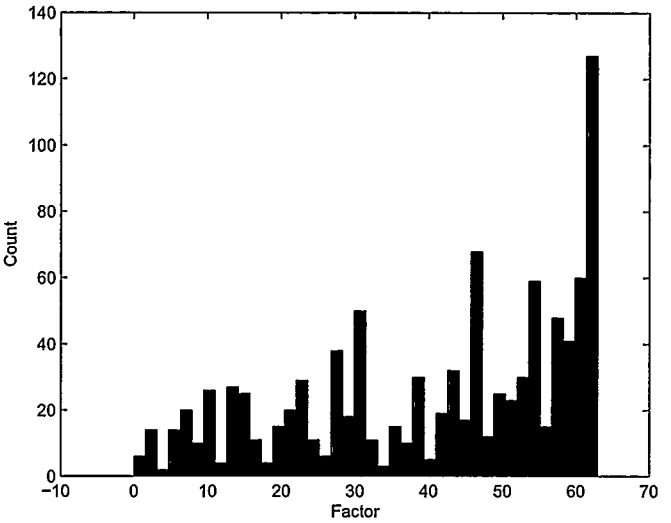


Figure 3.14: Results of the constrained bit optimization equations 3.9 factoring 481

and a specific cost function for the optimization using the same approach as equation 3.2 can be written as

$$f(x,y) = s_1(n-xy)^2 + s_2\frac{2-\cos(2\pi x)+\cos(2\pi y)}{(n-xy)^2+2-\cos(2\pi x)+\cos(2\pi y)+\epsilon}. \tag{3.11}$$

In Figure 3.15, equation 3.11 is graphed for $n = 143$ and the interesting part of the trench is shown. It should also be noted that s_1 and s_2 are set to 1 and 100 respectively. In this graph, s_2 is set to 100 to indicate the effects of the variable scaling. In Figure 3.15, one of the solutions is marked. The solution is clearly visible on the graph by a break in the hill within the trench. Figure 3.16 shows an experiment run on the biprime 481 with both scaling variables set to 1 using 1000 trials. In Figure 3.16, the results look very similar to the direct method used in equation 3.2. This is likely from the created local minima by the scaling contribution added to the equation. These local minima can be seen in Figure 3.15 on either sides of the hill created within the trench.

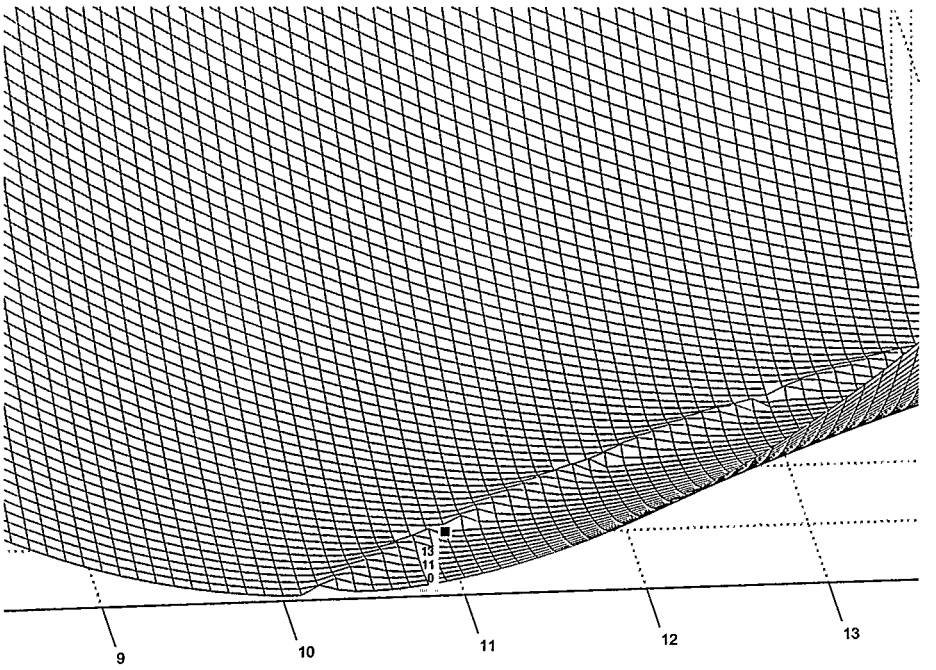


Figure 3.15: Effects of varying the scaling variable of the digitizer function

3.1.4 Multiplication Logic

Taking a closer look at the multiplication of p and q brings about the multiplication algorithm. It is a similar to the approach done in [3]. Multiplication in binary is done with shifts and adds. Table 3.2 shows binary multiplication for two 3-bit numbers. In Table 3.2, n_i is the i^{th} bit of n, x_i and y_i are the i^{th} bits of the primes p and q respectfully, and c_i are binary carries from previous additions. The carry bits, c_1,

	c_4					
c_5	c_3	c_2	c_1			
			x_3	x_2	x_1	y_1
		x_3	x_2	x_1		y_2
	x_3	x_2	x_1			y_3
n_6	n_5	n_4	n_3	n_2	n_1	

Table 3.2: Table of binary multiplication for 3-bit multiplier

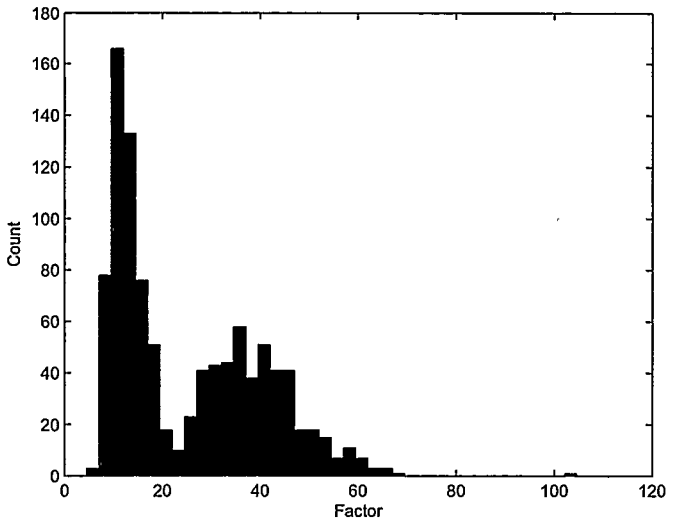


Figure 3.16: Results of equation 3.11 factoring 481

c_2, c_3 and c_5 are the carry bits from the previous row and c_4 is a carry from the two rows previous it. This configuration allows the equations for n_i to be written as a

sum of all bits in the i^{th} row modulo 2. The equations are

$$n_1 = x_1 y_1, \tag{3.12}$$

$$n_2 = x_2 y_1 \oplus x_1 y_2, \tag{3.13}$$

$$n_3 = x_3 y_1 \oplus x_2 y_2 \oplus x_1 y_3 \oplus c_1, \tag{3.14}$$

$$n_4 = x_3 y_2 \oplus x_2 y_3 \oplus c_2, \tag{3.15}$$

$$n_5 = x_3 y_3 \oplus c_3 \oplus c_4, \tag{3.16}$$

$$n_6 = c_5, \tag{3.17}$$

$$c_1 = \lfloor \frac{x_2 y_1 + x_1 y_2}{2} \rfloor \mod 2, \tag{3.18}$$

$$c_2 = \lfloor \frac{c_1 + x_3 y_1 + x_2 y_2 + x_1 y_3}{2} \rfloor \mod 2, \tag{3.19}$$

$$c_3 = \lfloor \frac{c_2 + x_3 y_2 + x_2 y_3}{2} \rfloor \mod 2, \tag{3.20}$$

$$c_4 = \lfloor \frac{c_1 + x_3 y_1 + x_2 y_2 + x_1 y_3}{4} \rfloor \mod 2, \tag{3.21}$$

$$c_5 = \lfloor \frac{c_3 + c_4 + x_3 y_3}{2} \rfloor \mod 2. \tag{3.22}$$

In these equation for n, the exclusive OR symbol $\oplus$ is used to denote addition modulo 2. Recall that two congruences $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ can be added together to make $a + c \equiv b + d \pmod{2}$.

To eliminate the carry variables and avoid tedious substitutions, some logic design is used to create a system of equations with 6 variables and 6 unknowns. The system of equations needs to be differentiable, if optimization is to be used, so the exclusive OR is transformed into an equivalent arithmetic equation. For example, $f = a \oplus b$ can be written $f = a+b-2ab$, if $a, b \in 0, 1$. It turns out that an arithmetic expression can be quickly found from the sum-of-products (SOP) truth table[22].

Consider a general arithmetic expression of the form

$$f = \sum_{i=1}^{2^m-1} a_i \cdot (x_1^{i_1} \cdots x_m^{i_m})$$

where m is the number of inputs, a_i is an arithmetic coefficient, $x_i$is the i^{th} input variable, and i_j is the j^{th} bit of i. For example, a function with two inputs, the arithmetic expression is

$$f = a_1 + a_2x_1 + a_3x_2 + a_4x_1x_2.$$

Using the general arithmetic expression above, the function can be expressed in matrix form as

$$f = \hat{X}P$$

where $\hat{X}$ is a vector of input variables and P is a vector of coefficients. Note that P can also be a matrix to represent multiple functions. To represent 6 equations, P will be a matrix of size $2^6 \times 6$. In the function with two inputs $\hat{X}$ is

$$\hat{X} = [1\ x_2\ x_1\ x_1x_2]\,.$$

In general, $\hat{X}$ is

$$\hat{X} = \bigotimes_{i=1}^{n}[1\ x_i].$$

To construct the P matrix, an arithmetic transform is used to transform a SOP truth table into coefficients. The transformation is

$$P = [\bigotimes_{i=1}^{m} P_2]F, P_2 = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \tag{3.23}$$

where P_2 is the transformation for a single variable, and F is the SOP truth table for the function.

The truth table for the bit multiplication was mapped out using equations constructed from table 3.2 and then transformed using equation 3.23. The tables as discussed above were of size $2^6 \times 6$ so they are shown in the appendix. The actual equations can also be found in the appendix. With the equations computed as

$$\begin{bmatrix} f_6 \\ f_5 \\ f_4 \\ f_3 \\ f_2 \\ f_1 \end{bmatrix} = \hat{X}[\bigotimes_{i=1}^{m} P_2]F$$

an optimization problem is constructed

$$\begin{aligned} min. \quad f(x,y) = & \sum_{i=1}^{6}(n_i - f_i)^2 \\ s.t. \quad d(x_i) = & \quad (x_i^2 - x_i)^2 \\ d(y_i) = & \quad (y_i^2 - y_i)^2. \end{aligned} \tag{3.24}$$

The optimization is tested against factoring 15. The experiment is ran 1000 times with initial conditions chosen randomly between $[0, 1.4]$. The results are shown in figure 3.17 as a histogram. Figure 3.17 shows no promise of factoring because neither 3 nor 5 is found with any usable probability.

3.2 Factoring Bit Game

A very different but related approach is to construct a bit factoring game. In this game, the game board consists of rows and columns representing a term of $2^i 2^j$. The

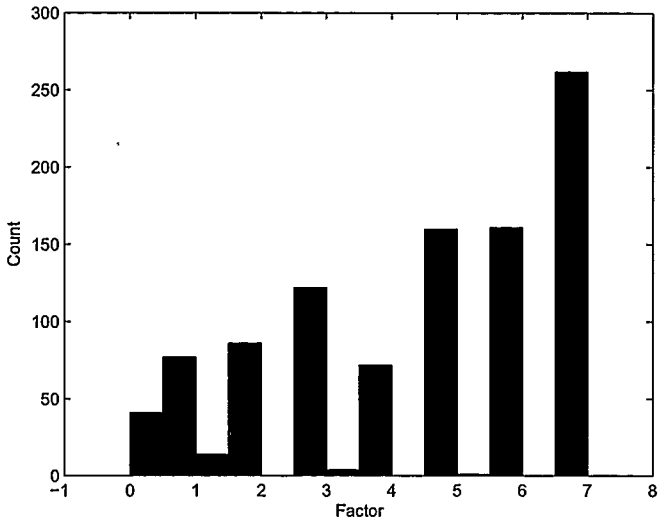


Figure 3.17: Results of using bit multiplication logic via equation 3.24 factoring 481

–	2^0	2^1	2^2	2^3	2^4	2^5
2^0	1	0	0	0	0	1
2^1	0	0	0	0	0	0
2^2	0	0	0	1	0	1
2^3	0	0	1	0	0	0
2^4	0	0	0	0	1	0
2^5	0	0	0	0	0	0

Table 3.3: A sample bit-board configuration for 481

entire board is summed up to represent the number in question. Table 3.2 shows an example of a bit-board for the number 481.

An entry of 1 in the matrix represents the existence of that term. The number represented can be written as

$$n = \sum_{i=1}^{L} \sum_{j=1}^{L} b_{ij} 2^{i-1} 2^{j-1}$$

where b_{ij} is the entry in the i^{th} row and j^{th} column, and L is sufficiently large to

cover all the bits. In the case of Table 3.2, the number 481 is represented because

$$2^0 2^0 + 2^5 2^0 + 2^3 2^2 + 2^2 2^3 + 2^5 2^2 + 2^4 2^4 = 481$$

Bits can be moved using certain rules. These rules ensure that the value of the number n is fixed. The game rules are:

1. Bits can be moved up or down along a diagonal, starting at the top right and ending at the bottom left. Mathematically, this means

   $$2^i 2^j = 2^{i+1} 2^{j-1} = 2^{i-1} 2^{j+1}$$

   and visually this means

   $$\begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 0 \\ \hline \end{array} \leftrightarrow \begin{array}{|c|c|} \hline 0 & 0 \\ \hline 1 & 0 \\ \hline \end{array}$$

2. Bits can be moved left or right by scaling by 2. Mathematically, this means

   $$2^i 2^j = 2 \times 2^{i-1} 2^j = \frac{1}{2} \times 2^{i+1} 2^j$$

   and visually, if values larger than 1 are allowed in the system, this means

   $$\begin{array}{|c|c|} \hline 2 & 0 \\ \hline 0 & 0 \\ \hline \end{array} \leftrightarrow \begin{array}{|c|c|} \hline 0 & 1 \\ \hline 0 & 0 \\ \hline \end{array}$$

   If the system is to be restricted to values of only 1, the first rule from above can be used to split the bit in either direction. This looks like

   $$\begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 0 & 1 \\ \hline 0 & 0 & 0 \\ \hline \end{array} \leftrightarrow \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 1 & 0 & 0 \\ \hline \end{array} \leftrightarrow \begin{array}{|c|c|c|} \hline 0 & 0 & 1 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

With the rules defined, the conditions for factoring can be examined. The factors of 481 are 13 and 37. In base 2, $481 = 13 \times 37 = (001101)_2 \times (100101)_2$. From binary multiplication, we know that

$$\begin{aligned}(001101)_2 \times (100101)_2 &= 2^3(100101)_2 + 2^2(100101)_2 + 2^0(100101)_2 \\ &= (2^3 2^5 + 2^3 2^2 + 2^3 2^0) + (2^2 2^5 + 2^2 2^2 + 2^2 2^0) + (2^0 2^5 + 2^0 2^2 + 2^0 2^0)\end{aligned}$$

Putting this result on the bit-board yields

1	0	1	0	0	1
0	0	0	0	0	0
1	0	1	0	0	1
1	0	1	0	0	1
0	0	0	0	0	0
0	0	0	0	0	0

Notice in the above bit-board that the configuration of the bits shows the factors of 481. Along the top row is the factor of 37 and along the leftmost column is the factor of 13. These factors can be seen by considering the binary number represented with the least significant bit in the top left corner of the board. This results brings an interesting game that can be used to factor biprimes. By using the rules discussed above, if bits can be moved into a similar form as above, the factors of the biprime can be found. In the factored bit-board, notice that all the rows are 37 or 0. This configuration exists for every biprime.

Approaching this problem is not an easy task. The complexity of the moves becomes very high because of rule 2. Even without rule 2, the complexity is very high. To help cope with the high complexity, a simplification is imposed.

3.2.1 Bit-Board Simplification

The high complexity of the bit-board game generates a need to simplify the problem. The rules of the game require that each row represent a factor or be zero because n is a biprime. However, the extra details of which bits are moved can be hidden by considering the bit-board as simply a vector of term values. For example, the bit-board for 481 can be rewritten as

1	0	1	0	0	1		37
0	0	0	0	0	0		0
1	0	1	0	0	1	=	37
1	0	1	0	0	1		37
0	0	0	0	0	0		0
0	0	0	0	0	0		0

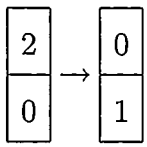
The game then becomes: balance the existing number such that each entry is either a factor of n or 0 and nothing else in between. This simplifies the game greatly as the individual bit movements are hidden by the distribution of the values. Writing a program to handle this new version of the game also becomes much simpler. Individual bit movements no longer need to be tracked or considered.

Rules for this new game are simplified to:

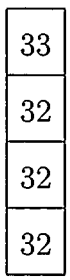
1. Any value can be move down a row by multiplying by a factor of 2. Below the movement is shown visually.

0	→	4	
2		0	

2. Any multiple of 2 can be moved up one row after being divided by 2. Again, a visualization of the movement is given below.



The game can start in any random configuration. Depending on the strategy of the solution, many configurations can be used as a starting point. For the first strategy, it is assumed that the smaller of the two factors of n will fall in the columns of the board. With this assumption, the number of rows can be limited to $L = \lfloor \frac{\log_2 n}{2} \rfloor$. After choosing the number of rows, the rows are filled such that each row only differs by 1. For example, the starting point of 481 would be



A balance for the vector can be found with the following formulae.

$$t = \left\lfloor \frac{n}{2^L - 1} \right\rfloor \tag{3.25}$$

$$r = n \mod 2^L - 1 \tag{3.26}$$

All rows have the value of t except those that have a 1 in the binary representation for n. For 481, we have

$$\begin{aligned} L &= \left\lfloor \frac{\log_2 481}{2} \right\rfloor = 4 \\ t &= \left\lfloor \frac{481}{2^4 - 1} \right\rfloor = 32 \\ r &= 481 \pmod{2^4 - 1} = 1 = (0001)_2 \end{aligned}$$

$$\begin{bmatrix} 32 \\ 32 \\ 32 \\ 32 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 33 \\ 32 \\ 32 \\ 32 \end{bmatrix}$$

Now that the vector is balanced, the next step is to choose a row to clear. When clearing a row, the value in the row is distributed among the other rows. From previous discussions it is easy to see the second row from the top gives the results we require to factor. In this case factoring is a single step away. Unfortunately, this is equivalent to guessing each bit of the smallest factor of n and therefore inefficient. This method was explored to observe how the other terms changed as bits were chosen to clear.

To see how viable an optimization using bit manipulations would be, an experiment was constructed. In this experiment, the cost function $f(x) = n \pmod{x}$ is used while each bit in x is changed. The optimization then follows the path of bit manipulations that decreases the cost function. The program iterates through all potential solutions of x and tests to see if there exists a decreasing cost path to the real solution. Each number in the solution space of $2 < x < 2^L$, where L is a sufficiently large bit length of a potential solution x, is tested. If there exists a path from a potential solution x to the real solution p, it is marked as nonisolated. The ratio of solution space to nonisolated solutions is calculated for all odd 20 bit biprimes and shown in figure 3.18. Figure 3.18 shows that most biprimes have initial points that can not reach the solution; however, there is a high probability that there does exist a path, depending on the biprime. It is clear that the cost function $f(x) = n$

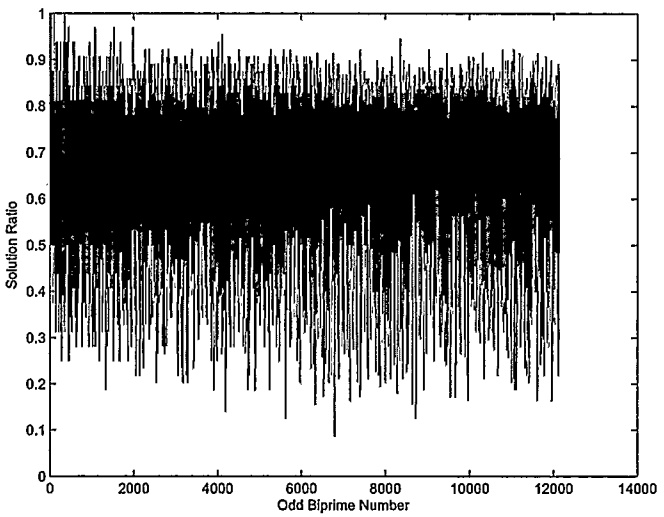


Figure 3.18: Graph of potential solution ratio for odd 20 bit biprimes

(mod x) is insufficient to find a path.

An interesting result of this approach is the quick factorization of special cases that have particular forms. Which form to take advantage of is dependent on the type of moves made. For example, biprimes constructed with a prime in the Mersenne-like form $p = 2^n - 1$ factor trivially because of the calculations done in the balancing stage. If the remainder r is zero, a factor has already been found.

Another interesting result is with the original bit-board. If one of the primes has the form $p = 2^n + 1$, the solution can be found by only moving bits (rule 1).

These methods are impractical because knowing which method to use depends on knowing the prime factors.

3.3 Branch and Bound Multiplication Logic

Revisiting the multiplication logic covered in section 3.1.4, a closer look at the equations created by Table 3.2 promoted a method inspired by branch and bound integer optimization. Looking at Table 3.2, it is clear that there is a lot of terms that disappear when a variable is zero. It is easy to see that a zero in a y_i variable would cause an entire row to disappear. In this section, a method is constructed that makes systematic assumptions based on the logic equations to find a solution. The appeal of this method is the simplification of the equations as more assumptions are introduced. If a contradiction in the equations is found, the last assumption is changed and a different assumption is tried.

To see in more detail how this works, some more logic design is used. It turns out that frequently, Reed-Muller (RM) expressions tend to have a smaller number of terms compared to other types of logical expression. Equations 3.22 also show that the addition logic uses the exclusive OR operation which is a basis for RM expressions. A Reed-Muller expression has the form

$$f = \bigoplus_{i=1}^{2^m-1} r_i \cdot (x_1^{i_1} \cdots x_m^{i_m}),$$

where r_i is a binary coefficient, x_i is the i^{th} input variable, and i_j is the j^{th} bit of i. For a logic function with two inputs, the Reed-Muller equation is then

$$f = r_1 \oplus r_2 x_1 \oplus r_3 x_2 \oplus r_4 x_1 x_2.$$

Similarly to multiplication logic equations solved in section 3.1.4, this form allows the function to be written in matrix form

$$f = \hat{X} R$$

where $\hat{X}$ is a vector of input variables same as in section 3.1.4 and R is a vector of binary coefficients. Again, R can be a matrix for multi-output systems. The R matrix is calculated in the same manner as the P matrix in section 3.1.4. The transformation is

$$R = [\bigotimes_{i=1}^{n} R_2]F, R_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

where R_2 is the transformation for a single variable, and F is the sum-of-products truth table for the function. The resulting matrix can also be computed modulo 2 because $x \oplus x = 0$.

Solving for the R matrix for a 2-bit multiplier gives

$$R = [\bigotimes_{i=1}^{n} R_2]F$$

$$\begin{bmatrix} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 1&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0 \\ 1&0&1&0&1&0&1&0&0&0&0&0&0&0&0&0 \\ 1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0&1&1&0&0&0&0&0&0 \\ 1&0&1&0&0&0&0&0&1&0&1&0&0&0&0&0 \\ 1&1&1&1&0&0&0&0&1&1&1&1&0&0&0&0 \\ 1&0&0&0&1&0&0&0&1&0&0&0&1&0&0&0 \\ 1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0 \\ 1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0 \\ 1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1 \end{bmatrix} \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 1&0&0&0 \\ 0&1&0&0 \\ 1&1&0&0 \\ 0&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&1&1&0 \\ 0&0&0&0 \\ 1&1&0&0 \\ 0&1&1&0 \\ 1&0&0&1 \end{bmatrix} \quad \text{mod } 2 = \begin{bmatrix} 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&1&0&0 \\ 0&0&1&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&0&0 \\ 0&0&1&1 \end{bmatrix}.$$

Reading off the R matrix, the equations are then

$$\begin{aligned} f_1 &= x_1y_1, \\ f_2 &= x_2y_1 \oplus x_1y_2, \\ f_3 &= x_1y_2 \oplus x_1y_1x_2y_2, \\ f_4 &= x_1y_1x_2y_2. \end{aligned}$$

Notice that the assumption of $y_1 = 0$ changes all ones to zeroes in 8 rows. Also, an assumption of $y_1 = 0$ moves a one up the matrix into a simpler term. The algorithm

$n/2$	1	2	3	4	5	6	7	8
n_1	1	1	1	1	1	1	1	1
n_2	0	2	2	2	2	2	2	2
n_3		2	4	4	4	4	4	4
n_4		1	8	10	10	10	10	10
n_5			9	24	26	26	26	26
n_6			3	37	76	78	78	78
n_7				41	155	286	288	288
n_8				19	187	659	1130	1132
n_9					139	841	2803	4658
n_{10}					71	745	3847	12095
n_{11}						583	3829	16959
n_{12}						303	3123	17645
n_{13}							2079	16007
n_{14}							991	12527
n_{15}								8517
n_{16}								4055
Total	1	6	27	138	671	3538	18211	94004
Eqns.	2	4	6	8	10	12	14	16
Terms	4	16	64	256	1024	4096	16384	65536
Ratio	0.13	0.09	0.07	0.07	0.07	0.07	0.08	0.09

Table 3.4: Table of term totals for RM matrix representation of multiplication logic

manipulates the matrix in this way until all variables x_i and y_i are solved for and no contradiction can be found. All assumptions are recorded on a stack so that when a contradiction is found, the system can pop the last assumption used to simplify the equations and try the negative of the assumption. This is essentially a branch and bound optimization approach.

The major drawback to this approach is the memory used to store the matrix to be solved. An experiment was conducted to compute the number of terms used in a RM representation of the multiplication logic. Table 3.4 shows a computation for the number of terms used in the RM matrix for multipliers up to 8 bits. Table 3.4

shows that the number of terms used for multiplication of numbers under 8 bits is roughly 8% of the matrix size of $n \times 2^n$. The slightly increasing trend of this ratio over increasing n rules out storing the matrix as a list of bits. Unless a more compact way for representing the matrix can be found, this method is unusable for large n. Fortunately, the matrix seems to have a pattern, making it a likely candidate for compression. Figure 3.19 shows a visualization of the RM matrix for multiplication up to 4 bits. In Figure 3.19, a black dot represents a 1 in the RM matrix. A pattern

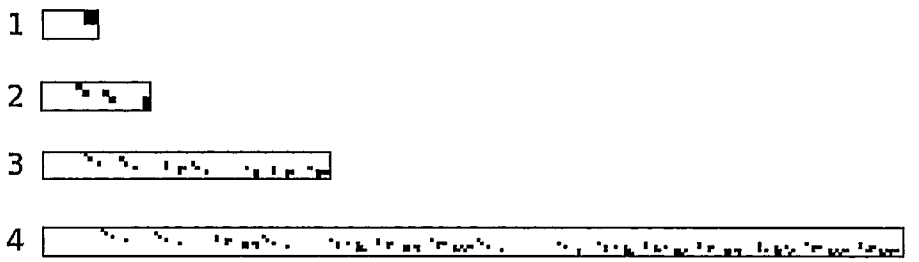


Figure 3.19: RM matrix visualization of 1, 2, 3, and 4 bit multiplications

can be seen in the distribution of the 1s. Within the matrices themselves, there is a repeated pattern that occurs at the beginning of matrix, starting from the left. Also, as n increases, the top of the matrix represented in Figure 3.19 shows there are no terms after a certain point for lower equations. This can also be seen in the numbers given in Table 3.4. As n increases, the terms for the lower significant bits stop producing terms. This is because of the dependence of carry bits on previous equations, as shown in the equations 3.22. Lastly, their are similarities between each of the matrices. Each next increasing n contains the same terms as before plus new terms introduced by the new equations. These similarities could be used to help compress the resulting R matrix used to determine the correct assumptions when running the branch and bound algorithm discussed above.

Chapter 4

Discussion and Conclusion

4.1 Discussion

The first experiments involved unconstrained optimization. In the unconstrained optimization experiments, the simplest cost function had the best chance of factoring. Most of the solutions found by the system, as shown in Figure 3.3, are around 13 and 37. The optimization of equation 3.2 correctly factored 481 approximately 16% of the time. Attempts to improve the success rate shown in Figure 3.7 and Figure 3.9 show decreased rate of factoring. In Figure 3.7, the results show unconstrained Fermat optimization correctly factoring approximately 5% of the time and Figure 3.9 shows the unconstrained bit optimization factoring $< 1\%$ of the time. In all the experiments, MATLAB chose to use line search for these experiments.

In an attempt to improve the component of the system that is responsible for persuading solutions into integer values, constrained optimization was used. Figure 3.11 showed that the constrained optimization successfully found integer solutions but did not find proper factors. This suggests that the method used to solve the constraint problem was putting too much focus on the constraint and inhibited the random nature of the initial point from finding a good solution. The inability to find a good solution can be visualize by imagining the digitization bumps seen in Figure 3.2 being too large to traverse the solution create by the trench, namely $x = \frac{n}{y}$.

To correct the emphasis on the digitizer of the optimization, the next system

varied the scale of the digitizer, similar to s_2 seen in equation 3.1. This varied scaling of the integer persuasion function $d(x, y)$ eliminated local minima outside the solution area of $x = \frac{n}{y}$. However, the similarity between the results shown in Figure 3.3 and Figure 3.16 suggest that local minima induced by the digitization is not the problem. After several different trials with the system outlined by the integer persuasion scaling equation 3.11, it was evident that the system had no where to go within the trench itself. All of the above systems fail because they lack a global attractor to a solution of $n = xy$ itself. For the simple cost function of $f(x, y) = n - xy$, no manner of digitizing or local minima escape will assist the system to find the solution efficiently. This problem is even more prominent when considering the solution space for a practical key of 1024 bits. Without a global attractor the system cannot be pulled towards the correct solution and the system breaks down into a brute force attack. Employing local minima escape methods would not fix the problem of pulling the system to the correct solution. To find a cost function that would attract the solution globally would require a deeper understand of number theory itself.

In an attempt to deconstruct multiplication and take a slightly different approach than in [3], equation 3.22 was created using logic design and solved for the carry bits. The results were unfavorable; however, the experiment did give an insight into the complexity of factoring. When constructing the equations for a 3-bit multiplier, the number of terms required to represent the logic was very high. This large number of terms makes the system infeasible at large n. However, because all the equation solving is precomputation, clever methods may still make the approach possible.

The bit-board optimization was developed simultaneously with the real optimiza-

tion experiments. It offers an integer optimization perspective to factoring. Interestingly, it suffers from the same problem as the real optimization experiments. It lacks an effective cost function. Without an effective cost function, the optimization fails to direct the movements towards the solution. However, it does offer some interesting results regarding the chance of factoring given a similar cost function to the real optimization. If an analogy can be drawn between the real and integer programming experiments, then the initial points that have no path towards the solution can be thought of as starting in a local minima of a real optimization. Interpolating on this analogy sheds some light on the probability of the real cost function. However, this result does not suggest anything about avoiding minima, which is a difficult task. It simply gives insight into the chances of starting in a local minima.

Results shown in Figure 3.3 suggest that many trials could loosely identify where the factors may be. This approach, as well as simulated annealing, was considered. However, both methods require many iterations of an already complex process. Moreover, without a cost function directing the optimization, this approach would break down to brute force by guessing. All the optimization experiments took over a minute to complete 1000 trials of the relatively small number 481. For practical key sizes the time to find a solution would be much too large. The purpose of applying optimization to factoring is to make complex but intelligent movements towards a potential solution. If there are many local minima as expected with a large key, the system will likely be slower than current methods that use many small unintelligent guesses, such as the quadratic sieve.

None of the algorithms in this thesis compare to the effectiveness of constructing a difference of squares from many well picked smaller guesses. The probability of

getting a correct factor is also much higher in the QS than in factoring optimization. Clearly, the choice is to use standard factoring methods over the non-standard methods presented in this thesis. While the methods presented in this thesis do not present methods to factor better, they do cover interesting approaches to the problem. Many new methods could be researched.

4.2 Future Work

There are many avenues for future development in the area of factoring optimization. One obvious and mentioned route is in the research of an effective cost function. However, finding one such cost function would require further analysis in number theory. This would most likely produce a more specific method than a cost function for general optimization. A similarity can be seen in the Euclidean algorithm, which is – in essences – a specialized optimization algorithm for finding the GCD. Further study into Diophantine equations would likely give a better understanding of constructing an effective cost function.

Another area that may be fruitful is neural networks. The advantage of using neural nets is that the cost function can be found by training the net. To be efficient the neural net would have to find a cost function that finds the global minimum fast enough that it could overcome the massive number of calculations required for large keys. Unfortuntely, like the many local minima created by a cost function, finding a cost function itself, would likely have many local minima and finding an effective cost function via neural net may also prove to be a difficult task.

4.3 Conclusion

The use of encryption is becoming more important as information becomes more available. Without it many businesses and new ideas would not be possible. These applications rely on encryption to help police valuable information. One of the most effective and widely used cryptosystem is the RSA cryptosystem. Consequently, much of todays infrastructure is built on this cryptosystem. With so much counting on the success of the cryptosystem, it is imparative that the cryptosystem stay secure. However, the security comes from the apparent intractibility of factoring. Many believe that it is secure through its intractibility because of its current resistance to attacks but that this is not enough for critical applications. For this reason, if a flaw in the cryptosystem is present, it is very important that the information be made available so that it can be substituted before too much damage is done.

While methods like the general number field sieve and the quadratic sieve show some promise at factoring efficiently, optimization factoring in this thesis does not. This failure is largely because of an inadequate cost function. If a cost function that directs movements at the global level can be found then an optimization approach may be viable. On the other hand, finding this cost function may not be an easy task. Until an efficient algorithm can be found, the RSA cryptosystem will remain secured by its mystery.

Even though the results of this thesis proved to be unsuccessful, many interesting ideas were touched on and most importantly, much was learned.

Bibliography

[1] Francois Morain Arthru O. L. Atkin. Elliptic curves and primality proving. *Math. Comput*, 61:29–68, 1993.

[2] David M. Bressoud. *Factorization and Primality Testing.* Springer-Verlag, 1989.

[3] Christopher J.C. Burges. Factoring as optimization. *Microsoft Technical Report MSR-TR-2002-83*, 2002.

[4] R. E. Powers Derrick H. Lehmer. On factoring large numbers. 37:770–776, 1931.

[5] John D. Dixon. Asymptotically fast factorization of integers. *Mathemicatical Computation*, 36, 1981.

[6] Titu Andreesscu et al. *104 Number Theory Problems.* Birkhauser, 2007.

[7] J. Franke F. Bahr, M. Boehm and T. Kleinjung. Factorization of rsa-200. Public announcement.

[8] Edward M. Wright Godfrey. H. Hardy. *An Introduction to the Theory of Numbers (fifth ed.).* Oxford University Press, 1979.

[9] Stephen J. Wright Jorge Nocedal. *Numerical Optimization.* Springer, 1999.

[10] W. Karush. Minima of functions of several variables with inequalities as side constraints. Master's thesis, M.Sc. Dissertation.Dept. of Mathematics, Univ. of Chicago, Chicago, Illinois, 1939.

[11] Donald Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Addison-Wesley, third edition, 1997.

[12] Kevin McCurley Leonard Adleman. Open problems in number theoretic complexity. *Proceedings of the Algorithmic Number Theory Symposium*, 1:3–13, 1997.

[13] Kevin S. McCurley. A key distribution system equivalent to factoring. 1.

[14] Peter Montgomery. A survey of modern integer factorization algorithms. *CWI Quarterly*, 1996.

[15] Carl Pomerance. A tale of two sieves. *Notices of the AMS*, 1996.

[16] Michael Rabin. Digital signature and public-key functions as intractable as factorization. *Technical Report 212, MIT*, 1979.

[17] Leonard Adleman Ronald Rivest, Adi Shamir. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2), 1987.

[18] M. P. Vecchi Scott Kirkpatrick, C. D. Gelatt. Optimization by simulated annealing. 220:671–680, 1983.

[19] Lieven Vandenberghe Stephen Boyd. *Convex Optimization.* Cambridge University Press, 2004.

[20] Raymond Sroul. *Programming for Mathematicians.* Berlin: Springer-Verlag, 2000.

[21] Ronald L. Rivest Clifford Stein Thomas H. Cormen, Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill, second edition, 2001.

[22] Masahiro Fujita Tsutomu Sasao. *Representations of Discrete Functions.* Springer, 1996.

[23] Eric W. Weisstein. *CRC Concise Encyclopedia of Mathematics.* CRC Press, 2003.

[24] Martin E. Hellman Whitfield Diffie. New directions in cryptographyy. *IEEE Trans. Info. Th. 22*, pages 644–654, 1976.

[25] Hugh C. Williams. Some public-key crypto-functions as intractable as factorization. *Proceedings of CRYPTO 84*, 196:66–70, 1985.

Appendix A

Experiment Code

A.1 Real Optimization in MATLAB

The following is the code used to conduct factorization experiments in MATLAB. Many of the experiments used similar code so not all functions will be given here.

A.1.1 Direct Factorization Code

The follow code was used to generate data for the equation 3.2 experiment. The code is split in two files. The first file is the main program, and the second file is the minimization function required by the *fminunc* function in MATLAB.

```
cap = 100; % use all primes less than cap
opt_iterations = 1000; % number of iterations to minimize

err_data=0; % store error data in here

% generate a range of biprime to test over
bip = [0 0];
i=1;
for p = primes(cap)
  for q = primes(cap)
    % exclude the case of p^2 and q^2 and when bip = q*p
    if p == q | intersect(err_data(:,1), p*q) == p*q
      continue
    end
    bip(i,:) = [ p q ];
    i = i+1;
  end
```

```
end

% uncomment to test only a single biprime
bip = [13 37];

for i = 1:size(bip,1)
  p = bip(i, 1);
  q = bip(i, 2);

  % first column of err_data is the biprime
  err_data(i,1) = p*q;

  % the following columns will be minimization trials on
  % random starting points
  for j=2:opt_iterations+1

    % form a vector of random numbers between 0.0 and some max
    initpq = rand(2,1)*1.1*max(p,q);

    % do the minimization
    [pq, min, f] = fminunc(@(x) factsin(x,p*q), initpq,
                           optimset('GradObj','on', 'Display', 'off'));

    % store the average error
    err_data(i,j) = sum(abs([p; q] - pq));

    % store the solution
    err_data(i,j+opt_iterations) = pq(1);
    j-1
  end

end

% make sure the data is in order
err_data = sortrows(err_data,1);
```

The minimization function is given below.

```
function [f, g] = factsin(x,bip)
  scale_err = 1;
  scale_sin = 10;

  f = scale_err*(bip - x(1)*x(2))^2 +
        scale_sin*(sin(pi*(2*x(1)+1/2)) + sin(pi*(2*x(2)+1/2)) + 2);
  g = [2*scale_err*(bip - x(1)*x(2))*-x(2) +
         scale_sin*cos(pi*(2*x(1)+1/2))*2*pi;
       2*scale_err*(bip - x(1)*x(2))*-x(1) +
         scale_sin*cos(pi*(2*x(2)+1/2))*2*pi];

%   f = (bip - x(1)*x(2))^2
%   % f = (x(1)*x(2))^2 - 2*bip*x(1)*x(2) + bip^2
%   % g = [2*x(1)*x(2)^2 - 2*bip*x(2); 2*x(1)^2*x(2) - 2*bip*x(1)];
%   % g = [2*x(2)*(x(1)*x(2) - bip); 2*x(1)*(x(1)*x(2) - bip)];
%   g = [2*(bip - x(1)*x(2))*-x(2);
%        2*(bip - x(1)*x(2))*-x(1)];
```

A.1.2 Bit Factorization Code

The follow code was used to generate data for the equation 3.4 experiment. The code is split in two files. The first file is the main program, and the second file is the minimization function required by the *fminunc* function in MATLAB.

```
cap = 100; % use all primes less than cap
opt_iterations = 1000; % number of iterations to minimize

err_data=0; % store error data in here

% generate a range of biprime to test over
bip = [0 0];
i=1;
for p = primes(cap)
```

```
  for q = primes(cap)
    % exclude the case of p^2 and q^2 and when bip = q*p
    if p == q | intersect(err_data(:,1), p*q) == p*q
      continue
    end
    bip(i,:) = [ p q ];
    i = i+1;
  end
end

% uncomment to test only a single biprime
bip = [13 37];
%bip = [101 1013];

for i = 1:size(bip,1)
  p = bip(i, 1);
  q = bip(i, 2);

  % binarize will turn p in an array of ones and zeros
  bp = binarize(p);
  bq = binarize(q);

  % use the largest binary representation for storing put correct
  % factorization of p and q in variable correct
  if length(bp) > length(bq)
    correct = [bp bq zeros(1,length(bp)-length(bq))]';
  elseif length(bp) < length(bq)
    correct = [bp zeros(1,length(bq)-length(bp)) bq]';
  else
    correct = [bp bq]';
  end

  % first column of err_data is the biprime, second is the
  % length of p (or q)
  err_data(i,1) = p*q;
  err_data(i,2) = length(correct)/2;
```

```
  % the following columns will be minimization trials on random
  % starting points
  for j=3:opt_iterations+2

    % form a vector of random numbers between 0.0 and 1.0
    initpq = rand(size(correct))*1.4;

    % do the minimization
    [pq, min, f] = fminunc(@(x) factquad(x,p*q), initpq,
                           optimset('GradObj','on', 'Display', 'off'));

    % store the average error per bit
    err_data(i,j) = sum(abs(correct - pq));

    % store the solution
    err_data(i,j+opt_iterations) =
      2.^(0:err_data(i,2)-1)*pq(1:err_data(i,2));
    j-2
  end

end

% make sure the data is in order
err_data = sortrows(err_data,1);
```

The minimization function is given below.

```
function [f, g] = factquad(x,bip)
  % x is bits of p and then q, and bip is the biprime

  scale_err = 1;
  scale_bit = 1;

  [sn, sm] = size(x);
  n = sn/2; % half the bits are p the other half are q
```

```
r = 2.^(0:n-1); % sequence [ 1 2 4 8 16 ... ] for bits

% result of f
%a = scale_err*(bip - (r*x(1:n))*(r*x(n+1:sn))) +
     scale_bit*sum((x.^2-x).^2);

% compute the gradient needed by fminunc()
%p = sum(r.*x(1:n)');
%q = sum(r.*x(n+1:sn)');
%b = -scale_err*[r'*q; p*r'] + scale_bit*2*(x.^2-x).*(2*x-1);
%-scale_err*[r'*q; p*r']
%scale_bit*2*(x.^2-x).*(2*x-1)
%f = a^2;
%g = 2*a.*b;

p = sum(r.*x(1:n)');
q = sum(r.*x(n+1:sn)');
f = scale_err*(bip - p*q)^2 + scale_bit*sum((x.^2-x).^2);
g = scale_err*(bip - p*q)*2*-[r'*q; p*r'] +
    scale_bit*2*(x.^2-x).*(2*x-1);
```

A.1.3 Constrained Bit Factorization Code

The follow code was used to generate data for the equation 3.9 experiment. The code is split in three files. The first file is the main program, and the second and third files are the minimization function and constrain vector required by the *fmincon* function in MATLAB.

```
cap = 100; % use all primes less than cap
opt_iterations = 1000; % number of iterations to minimize

err_data=0; % data store in here

% generate a range of biprime to test over
```

```
bip = [0 0];
i=1;
for p = primes(cap)
  for q = primes(cap)
    % exclude the case of p^2 and q^2 and when bip = q*p
    if p == q | intersect(err_data(:,1), p*q) == p*q
      continue
    end
    bip(i,:) = [ p q ];
    i = i+1;
  end
end

% uncomment to test only a single biprime
%bip = [13 37];
bip = [101 1013];

 for i = 1:size(bip,1)
  p = bip(i, 1);
  q = bip(i, 2);

  % binarize will turn p in an array of ones and zeros
  bp = binarize(p);
  bq = binarize(q);

  % use the largest binary representation for storing put correct
  % factorization of p and q in variable correct
  if length(bp) > length(bq)
      correct = [bp bq zeros(1,length(bp)-length(bq))]';
  elseif length(bp) < length(bq)
      correct = [bp zeros(1,length(bq)-length(bp)) bq]';
  else
      correct = [bp bq]';
  end

  % first column of err_data is the biprime, second is the
```

```
  % length of p (or q)
  err_data(i,1) = p*q;
  err_data(i,2) = length(correct)/2;

  % the following columns will be minimization trials on random
  % starting points
  for j=3:opt_iterations+2

      % form a vector of random numbers between 0.0 and 1.0
      initpq = rand(size(correct))*1.4;

      % do the minimization
      [pq, min, f] = fmincon(@(x) factquad(x,p*q), initpq,
                             [],[],[],[],[],[], @(x) factquadcon(x),
                             optimset('GradObj','on', 'Display', 'off'));

      % store the average error per bit
      err_data(i,j) = sum(abs(correct - pq));

      % store the average error
      err_data(i,j+opt_iterations) =
        2.^(0:err_data(i,2)-1)*pq(1:err_data(i,2));
      j-2
  end

 end


% make sure the data is in order
err_data = sortrows(err_data,1);
```

The following code is the minimization function.

```
function [f, g] = factquad(x,bip)
 % x is bits of p and then q, and bip is the biprime
```

```
  [sn, sm] = size(x);
  n = sn/2; % half the bits are p the other half are q
  r = 2.^(0:n-1); % sequence [ 1 2 4 8 16 ... ] for bits

  % result of f
  a = bip - (r*x(1:n))*(r*x(n+1:sn));

  % compute the gradient needed by fmincon()
  p = sum(r.*x(1:n)');
  q = sum(r.*x(n+1:sn)');
  b = -[r'*q; p*r'];

  f = a^2;
  g = 2*a.*b;
```

The following code is the constraint vectors for the *fmincon* function.

```
function [c, ceq] = factquadcon(x)
    % inequality constraints
    c = [];

    % equality constraints
    %ceq = sum((x.^2-x).^2);
    ceq = (x.^2-x).^2;
```

A.1.4 Scaled Factorization Code

The code for the experiment involving equation 3.11 is similar to that of the direct method used in the experiment with equation 3.2. The main program is identical to the direct method with the exception of the minimization function. The minimization function used is given below.

```
function [f, g] = factsin(x,bip)
  epsilon = 0.000001;
```

```
c = (bip-x(1)*x(2))^2;
d = sin(2*pi*(x(1)-1/4)) + sin(2*pi*(x(2)-1/4)) + 2;
f = c + d/(c + epsilon);

dc1 = 2*(bip - x(1)*x(2))*-x(2);
dd1 = cos(pi*(2*x(1)+1/2))*2*pi;

dc2 = 2*(bip - x(1)*x(2))*-x(1);
dd2 = cos(pi*(2*x(2)+1/2))*2*pi;
g = [dc1 + ((c+epsilon)*dd1 - d*dc1)/(c+epsilon)^2;
     dc2 + ((c+epsilon)*dd2 - d*dc2)/(c+epsilon)^2];
```

A.1.5 Multiplication Logic Equations and Code

The following are the matrices for the truth table and resulting arithmetic coefficients, respectfully.

$$F = \begin{bmatrix}
0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\\0&0&0&0&1&0\\0&0&0&0&1&1\\0&0&0&1&0&0\\0&0&0&1&0&1\\0&0&0&1&1&0\\0&0&0&1&1&1\\0&0&0&0&0&0\\0&0&0&0&1&0\\0&0&0&1&0&0\\0&0&0&1&1&0\\0&0&1&0&0&0\\0&0&1&0&1&0\\0&0&1&1&0&0\\0&0&1&1&1&0\\0&0&0&0&0&0\\0&0&0&0&1&1\\0&0&0&1&1&0\\0&0&1&0&0&1\\0&0&1&1&0&0\\0&0&1&1&1&1\\0&1&0&0&1&0\\0&1&0&1&0&1\\0&0&0&0&0&0\\0&0&0&1&0&0\\0&0&1&0&0&0\\0&0&1&1&0&0\\0&1&0&0&0&0\\0&1&0&1&0&0\\0&1&1&0&0&0\\0&1&1&1&0&0\\0&0&0&0&0&0\\0&0&0&1&0&1\\0&0&1&0&1&0\\0&0&1&1&1&1\\0&1&0&1&0&0\\0&1&1&0&0&1\\0&1&1&1&1&0\\1&0&0&0&1&1\\0&0&0&0&0&0\\0&0&0&1&1&0\\0&0&1&1&0&0\\0&1&0&0&1&0\\0&1&1&0&0&0\\0&1&1&1&1&0\\1&0&0&1&0&0\\1&0&1&0&1&0\\0&0&0&0&0&0\\0&0&0&1&1&1\\0&0&1&1&1&0\\0&1&0&1&0&1\\0&1&1&1&0&0\\1&0&0&0&1&1\\1&0&1&0&1&0\\1&1&0&0&0&1
\end{bmatrix} \quad P = \begin{bmatrix}
0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&1\\0&0&0&0&1&0\\0&0&0&0&0&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&1&0\\0&0&0&1&0&0\\0&0&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&1&1&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&1&1&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&0&0&1&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&1&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\1&1&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&1&1&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\1&1&0&0&0&0\\0&1&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\1&1&0&0&0&0\\0&1&0&0&0&0\\0&0&0&0&0&0\\0&0&0&0&0&0\\0&0&1&0&0&0\\0&0&0&0&0&0\\0&1&0&0&0&0
\end{bmatrix}$$

The resulting arithmetic matrix represents the following equations.

$$
\begin{aligned}
n_1 &= y_1x_1 \\
n_2 &= y_1x_2 + y_2x_1 \\
n_3 &= y_1x_3 + y_2x_2 + y_3x_1 + y_2y_1x_2x_1 \\
n_4 &= y_3y_2y_1x_2x_1 + y_2y_1x_3x_2 + y_2x_3 + y_3x_2 + y_2y_1x_2x_1 + y_3y_1x_3x_1 \\
&+ y_3y_2x_2x_1 + y_2y_1x_3x_2x_1 \\
n_5 &= y_3y_1x_3x_2x_1 + y_3y_2x_3x_2x_1 + y_3y_2y_1x_3x_1 + y_3y_2y_1x_3x_2x_1 \\
&+ y_2y_1x_3x_2 + y_3y_2x_3x_2 + y_3y_2y_1x_3x_2 + y_3x_3 + y_3y_2x_2x_1 \\
n_6 &= y_3y_1x_3x_2x_1 + y_3y_2y_1x_3x_1 + y_3y_2x_3x_2
\end{aligned}
$$

The code used to conduct the multiplication logic optimization experiment is shown below. The code is split in three files. The first file is the main program, and the second file is the minimization function required by the *fminunc* function in MATLAB. The third file is the constraints used for the function in the second file.

```
cap = 100; % use all primes less than cap
opt_iterations = 1000; % number of iterations to minimize

err_data=0; % data store in here

% generate a range of biprime to test over
bip = [0 0];
i=1;
for p = primes(cap)
  for q = primes(cap)
    % exclude the case of p^2 and q^2 and when bip = q*p
    if p == q | intersect(err_data(:,1), p*q) == p*q
      continue
    end
```

```
    bip(i,:) = [ p q ];
    i = i+1;
  end
end

% uncomment to test only a single biprime
bip = [3 5];

 for i = 1:size(bip,1)
  p = bip(i, 1);
  q = bip(i, 2);

  % binarize will turn p in an array of ones and zeros
  bp = binarize(p);
  bq = binarize(q);

  % use the largest binary representation for storing put correct
  % factorization of p and q in variable correct
  if length(bp) > length(bq)
      correct = [bp bq zeros(1,length(bp)-length(bq))]';
  elseif length(bp) < length(bq)
      correct = [bp zeros(1,length(bq)-length(bp)) bq]';
  else
      correct = [bp bq]';
  end

  % first column of err_data is the biprime, second is the length
  % of p (or q)
  err_data(i,1) = p*q;
  err_data(i,2) = length(correct)/2;

  % the following columns will be minimization trials on random
  % starting points
  for j=3:opt_iterations+2

      % form a vector of random numbers between 0.0 and 1.0
```

```
      initpq = rand(size(correct))*1.4;

      % do the minimization
      [pq, min, f] = fminunc(@(x) factmltlog(x,p*q), initpq,
                             optimset('GradObj','on', 'Display', 'off'));

      % store the average error per bit
      err_data(i,j) = sum(abs(correct - pq));

      % store the average error
      err_data(i,j+opt_iterations) =
        2.^(0:err_data(i,2)-1)*pq(1:err_data(i,2));
      j-2
  end

 end


% make sure the data is in order
err_data = sortrows(err_data,1);
```

The minimization function for the experiment is given below.

```
function [f, g] = factmltlog(x,bip)

    n = [ mod(floor(bip/2^0),2);
          mod(floor(bip/2^1),2);
          mod(floor(bip/2^2),2);
          mod(floor(bip/2^3),2);
          mod(floor(bip/2^4),2);
          mod(floor(bip/2^5),2)];

  a = (n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-
       x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))^2+(n(5)-x(6)*x(3)-
       x(5)*x(4)*x(3)*x(2)+x(6)*x(5)*x(3)*x(2)*x(1)+
       x(6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-
```

```
      x(6)*x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6)*x(4)*x(3)*x(2)*x(1)+
      x(6)*x(5)*x(4)*x(3)*x(2))^2+(n(4)+x(5)*x(4)*x(3)*x(2)-
      2*x(6)*x(5)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(2)*x(1)+
      2*x(6)*x(5)*x(4)*x(3)*x(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-
      x(6)*x(2)-x(5)*x(3)+x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(4)*x(3)*x(1)+
      x(6)*x(5)*x(2)*x(1)+2*x(6)*x(5)*x(3)*x(2)+
      2*x(6)*x(4)*x(3)*x(2)*x(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)-
      x(5)*x(4)*x(2)*x(1))^2+(n(3)+2*x(5)*x(4)*x(3)*x(2)-
      2*x(6)*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4)*x(3)*x(2)*x(1)+
      2*x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1)+x(5)*x(4)*x(2)*x(1)-
      x(6)*x(1)-x(5)*x(2))^2+(n(2)+2*x(5)*x(4)*x(2)*x(1)-x(5)*x(1)-
      x(4)*x(2))^2+(n(1)-x(4)*x(1))^2;

% these are much to long to format nicely
b = [
       2*(n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)* ...
       x(1)-x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))*(-x(5)*x( ...
       4)*x(3)*x(1)+2*x(5)*x(4)*x(3)*x(2)*x(1)-x(5)*x(3)*x(2)-x(4)* ...
       x(3)*x(2)*x(1))+2*(n(5)-x(6)*x(3)-x(5)*x(4)*x(3)*x(2)+x(6)*x ...
       (5)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4) ...
       *x(3)*x(2)*x(1)-x(6)*x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6) ...
       *x(4)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(3)*x(2))*(-x(3)+x(5)*x ...
       (3)*x(2)*x(1)+x(5)*x(4)*x(3)*x(1)-3*x(5)*x(4)*x(3)*x(2)*x(1) ...
       -x(5)*x(2)*x(1)+x(5)*x(3)*x(2)+x(4)*x(3)*x(2)*x(1)+x(5)*x(4) ...
       *x(3)*x(2))+2*(n(4)+x(5)*x(4)*x(3)*x(2)-2*x(6)*x(5)*x(3)*x(2 ...
       )*x(1)+x(6)*x(5)*x(4)*x(2)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(1)-2 ...
       *x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(2)-x(5)*x(3)+x(5)*x(4) ...
       *x(3)*x(2)*x(1)-x(6)*x(4)*x(3)*x(1)+x(6)*x(5)*x(2)*x(1)+2*x( ...
       6)*x(5)*x(3)*x(2)+2*x(6)*x(4)*x(3)*x(2)*x(1)-2*x(6)*x(5)*x(4 ...
       )*x(3)*x(2)-x(5)*x(4)*x(2)*x(1))*(-2*x(5)*x(3)*x(2)*x(1)+x(5 ...
       )*x(4)*x(2)*x(1)+2*x(5)*x(4)*x(3)*x(1)-2*x(5)*x(4)*x(3)*x(2) ...
       *x(1)-x(2)-x(4)*x(3)*x(1)+x(5)*x(2)*x(1)+2*x(5)*x(3)*x(2)+2* ...
       x(4)*x(3)*x(2)*x(1)-2*x(5)*x(4)*x(3)*x(2))+2*(n(3)+2*x(5)*x( ...
       4)*x(3)*x(2)-2*x(6)*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4 ...
       )*x(3)*x(2)*x(1)+2*x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1) ...
       +x(5)*x(4)*x(2)*x(1)-x(6)*x(1)-x(5)*x(2))*(-2*x(5)*x(4)*x(2) ...
```

```
*x(1)+2*x(4)*x(3)*x(1)+2*x(5)*x(2)*x(1)-x(1));

2*(n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)* ...
x(1)-x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))*(-x(6)*x( ...
4)*x(3)*x(1)+2*x(6)*x(4)*x(3)*x(2)*x(1)-x(6)*x(3)*x(2))+2*(n ...
(5)-x(6)*x(3)-x(5)*x(4)*x(3)*x(2)+x(6)*x(5)*x(3)*x(2)*x(1)+x ...
(6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-x(6) ...
*x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6)*x(4)*x(3)*x(2)*x(1) ...
+x(6)*x(5)*x(4)*x(3)*x(2))*(-x(4)*x(3)*x(2)+x(6)*x(3)*x(2)*x ...
(1)+x(6)*x(4)*x(3)*x(1)-3*x(6)*x(4)*x(3)*x(2)*x(1)-x(6)*x(2) ...
*x(1)+x(6)*x(3)*x(2)+x(6)*x(4)*x(3)*x(2))+2*(n(4)+x(5)*x(4)* ...
x(3)*x(2)-2*x(6)*x(5)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(2)*x(1 ...
)+2*x(6)*x(5)*x(4)*x(3)*x(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)*x(1) ...
-x(6)*x(2)-x(5)*x(3)+x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(4)*x(3) ...
*x(1)+x(6)*x(5)*x(2)*x(1)+2*x(6)*x(5)*x(3)*x(2)+2*x(6)*x(4)* ...
x(3)*x(2)*x(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)-x(5)*x(4)*x(2)*x(1 ...
))*(x(4)*x(3)*x(2)-2*x(6)*x(3)*x(2)*x(1)+x(6)*x(4)*x(2)*x(1) ...
+2*x(6)*x(4)*x(3)*x(1)-2*x(6)*x(4)*x(3)*x(2)*x(1)-x(3)+x(4)* ...
x(3)*x(2)*x(1)+x(6)*x(2)*x(1)+2*x(6)*x(3)*x(2)-2*x(6)*x(4)*x ...
(3)*x(2)-x(4)*x(2)*x(1))+2*(n(3)+2*x(5)*x(4)*x(3)*x(2)-2*x(6 ...
)*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4)*x(3)*x(2)*x(1)+2 ...
*x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1)+x(5)*x(4)*x(2)*x( ...
1)-x(6)*x(1)-x(5)*x(2))*(2*x(4)*x(3)*x(2)-2*x(6)*x(4)*x(2)*x ...
(1)-2*x(4)*x(3)*x(2)*x(1)+2*x(6)*x(2)*x(1)+x(4)*x(2)*x(1)-x( ...
2))+2*(n(2)+2*x(5)*x(4)*x(2)*x(1)-x(5)*x(1)-x(4)*x(2))*(2*x( ...
4)*x(2)*x(1)-x(1));

2*(n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)* ...
x(1)-x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))*(-x(6)*x( ...
5)*x(3)*x(1)+2*x(6)*x(5)*x(3)*x(2)*x(1)-x(6)*x(3)*x(2)*x(1)) ...
+2*(n(5)-x(6)*x(3)-x(5)*x(4)*x(3)*x(2)+x(6)*x(5)*x(3)*x(2)*x ...
(1)+x(6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4)*x(3)*x(2)*x(1) ...
-x(6)*x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6)*x(4)*x(3)*x(2) ...
*x(1)+x(6)*x(5)*x(4)*x(3)*x(2))*(-x(5)*x(3)*x(2)+x(6)*x(5)*x ...
(3)*x(1)-3*x(6)*x(5)*x(3)*x(2)*x(1)+x(6)*x(3)*x(2)*x(1)+x(6) ...
*x(5)*x(3)*x(2))+2*(n(4)+x(5)*x(4)*x(3)*x(2)-2*x(6)*x(5)*x(3 ...
```

```
)*x(2)*x(1)+x(6)*x(5)*x(4)*x(2)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x ...
(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(2)-x(5)*x(3)+x(5) ...
*x(4)*x(3)*x(2)*x(1)-x(6)*x(4)*x(3)*x(1)+x(6)*x(5)*x(2)*x(1) ...
+2*x(6)*x(5)*x(3)*x(2)+2*x(6)*x(4)*x(3)*x(2)*x(1)-2*x(6)*x(5 ...
)*x(4)*x(3)*x(2)-x(5)*x(4)*x(2)*x(1))*(x(5)*x(3)*x(2)+x(6)*x ...
(5)*x(2)*x(1)+2*x(6)*x(5)*x(3)*x(1)-2*x(6)*x(5)*x(3)*x(2)*x( ...
1)+x(5)*x(3)*x(2)*x(1)-x(6)*x(3)*x(1)+2*x(6)*x(3)*x(2)*x(1)- ...
2*x(6)*x(5)*x(3)*x(2)-x(5)*x(2)*x(1))+2*(n(3)+2*x(5)*x(4)*x( ...
3)*x(2)-2*x(6)*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4)*x(3 ...
)*x(2)*x(1)+2*x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1)+x(5) ...
*x(4)*x(2)*x(1)-x(6)*x(1)-x(5)*x(2))*(2*x(5)*x(3)*x(2)-2*x(6 ...
)*x(5)*x(2)*x(1)-x(3)-2*x(5)*x(3)*x(2)*x(1)+2*x(6)*x(3)*x(1) ...
+x(5)*x(2)*x(1))+2*(n(2)+2*x(5)*x(4)*x(2)*x(1)-x(5)*x(1)-x(4 ...
)*x(2))*(2*x(5)*x(2)*x(1)-x(2))-2*(n(1)-x(4)*x(1))*x(1);

2*(n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)* ...
x(1)-x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))*(-x(6)*x( ...
5)*x(4)*x(1)+2*x(6)*x(5)*x(4)*x(2)*x(1)-x(6)*x(5)*x(2)-x(6)* ...
x(4)*x(2)*x(1))+2*(n(5)-x(6)*x(3)-x(5)*x(4)*x(3)*x(2)+x(6)*x ...
(5)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4) ...
*x(3)*x(2)*x(1)-x(6)*x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6) ...
*x(4)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(3)*x(2))*(-x(6)-x(5)*x ...
(4)*x(2)+x(6)*x(5)*x(2)*x(1)+x(6)*x(5)*x(4)*x(1)-3*x(6)*x(5) ...
*x(4)*x(2)*x(1)+x(6)*x(5)*x(2)+x(6)*x(4)*x(2)*x(1)+x(6)*x(5) ...
*x(4)*x(2))+2*(n(4)+x(5)*x(4)*x(3)*x(2)-2*x(6)*x(5)*x(3)*x(2 ...
)*x(1)+x(6)*x(5)*x(4)*x(2)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(1)-2 ...
*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(2)-x(5)*x(3)+x(5)*x(4) ...
*x(3)*x(2)*x(1)-x(6)*x(4)*x(3)*x(1)+x(6)*x(5)*x(2)*x(1)+2*x( ...
6)*x(5)*x(3)*x(2)+2*x(6)*x(4)*x(3)*x(2)*x(1)-2*x(6)*x(5)*x(4 ...
)*x(3)*x(2)-x(5)*x(4)*x(2)*x(1))*(x(5)*x(4)*x(2)-2*x(6)*x(5) ...
*x(2)*x(1)+2*x(6)*x(5)*x(4)*x(1)-2*x(6)*x(5)*x(4)*x(2)*x(1)- ...
x(5)+x(5)*x(4)*x(2)*x(1)-x(6)*x(4)*x(1)+2*x(6)*x(5)*x(2)+2*x ...
(6)*x(4)*x(2)*x(1)-2*x(6)*x(5)*x(4)*x(2))+2*(n(3)+2*x(5)*x(4 ...
)*x(3)*x(2)-2*x(6)*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4) ...
*x(3)*x(2)*x(1)+2*x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1)+ ...
x(5)*x(4)*x(2)*x(1)-x(6)*x(1)-x(5)*x(2))*(2*x(5)*x(4)*x(2)-x ...
```

```
(4)-2*x(5)*x(4)*x(2)*x(1)+2*x(6)*x(4)*x(1));

2*(n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)* ...
x(1)-x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))*(2*x(6)*x ...
(5)*x(4)*x(3)*x(1)-x(6)*x(5)*x(3)-x(6)*x(4)*x(3)*x(1))+2*(n( ...
5)-x(6)*x(3)-x(5)*x(4)*x(3)*x(2)+x(6)*x(5)*x(3)*x(2)*x(1)+x( ...
6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-x(6)* ...
x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6)*x(4)*x(3)*x(2)*x(1)+ ...
x(6)*x(5)*x(4)*x(3)*x(2))*(-x(5)*x(4)*x(3)+x(6)*x(5)*x(3)*x( ...
1)-3*x(6)*x(5)*x(4)*x(3)*x(1)-x(6)*x(5)*x(1)+x(6)*x(5)*x(3)+ ...
x(6)*x(4)*x(3)*x(1)+x(6)*x(5)*x(4)*x(3))+2*(n(4)+x(5)*x(4)*x ...
(3)*x(2)-2*x(6)*x(5)*x(3)*x(2)*x(1)+x(6)*x(5)*x(4)*x(2)*x(1) ...
+2*x(6)*x(5)*x(4)*x(3)*x(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)- ...
x(6)*x(2)-x(5)*x(3)+x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(4)*x(3)* ...
x(1)+x(6)*x(5)*x(2)*x(1)+2*x(6)*x(5)*x(3)*x(2)+2*x(6)*x(4)*x ...
(3)*x(2)*x(1)-2*x(6)*x(5)*x(4)*x(3)*x(2)-x(5)*x(4)*x(2)*x(1) ...
)*(x(5)*x(4)*x(3)-2*x(6)*x(5)*x(3)*x(1)+x(6)*x(5)*x(4)*x(1)- ...
2*x(6)*x(5)*x(4)*x(3)*x(1)-x(6)+x(5)*x(4)*x(3)*x(1)+x(6)*x(5 ...
)*x(1)+2*x(6)*x(5)*x(3)+2*x(6)*x(4)*x(3)*x(1)-2*x(6)*x(5)*x( ...
4)*x(3)-x(5)*x(4)*x(1))+2*(n(3)+2*x(5)*x(4)*x(3)*x(2)-2*x(6) ...
*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4)*x(3)*x(2)*x(1)+2* ...
x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1)+x(5)*x(4)*x(2)*x(1 ...
)-x(6)*x(1)-x(5)*x(2))*(2*x(5)*x(4)*x(3)-2*x(6)*x(5)*x(4)*x( ...
1)-2*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(1)+x(5)*x(4)*x(1)-x(5 ...
))+2*(n(2)+2*x(5)*x(4)*x(2)*x(1)-x(5)*x(1)-x(4)*x(2))*(2*x(5 ...
)*x(4)*x(1)-x(4));

2*(n(6)-x(6)*x(5)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x(2)* ...
x(1)-x(6)*x(5)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)*x(1))*(-x(6)*x( ...
5)*x(4)*x(3)+2*x(6)*x(5)*x(4)*x(3)*x(2)-x(6)*x(4)*x(3)*x(2)) ...
+2*(n(5)-x(6)*x(3)-x(5)*x(4)*x(3)*x(2)+x(6)*x(5)*x(3)*x(2)*x ...
(1)+x(6)*x(5)*x(4)*x(3)*x(1)-3*x(6)*x(5)*x(4)*x(3)*x(2)*x(1) ...
-x(6)*x(5)*x(2)*x(1)+x(6)*x(5)*x(3)*x(2)+x(6)*x(4)*x(3)*x(2) ...
*x(1)+x(6)*x(5)*x(4)*x(3)*x(2))*(x(6)*x(5)*x(3)*x(2)+x(6)*x( ...
5)*x(4)*x(3)-3*x(6)*x(5)*x(4)*x(3)*x(2)-x(6)*x(5)*x(2)+x(6)* ...
x(4)*x(3)*x(2))+2*(n(4)+x(5)*x(4)*x(3)*x(2)-2*x(6)*x(5)*x(3) ...
```

```
        *x(2)*x(1)+x(6)*x(5)*x(4)*x(2)*x(1)+2*x(6)*x(5)*x(4)*x(3)*x( ...
        1)-2*x(6)*x(5)*x(4)*x(3)*x(2)*x(1)-x(6)*x(2)-x(5)*x(3)+x(5)* ...
        x(4)*x(3)*x(2)*x(1)-x(6)*x(4)*x(3)*x(1)+x(6)*x(5)*x(2)*x(1)+ ...
        2*x(6)*x(5)*x(3)*x(2)+2*x(6)*x(4)*x(3)*x(2)*x(1)-2*x(6)*x(5) ...
        *x(4)*x(3)*x(2)-x(5)*x(4)*x(2)*x(1))*(-2*x(6)*x(5)*x(3)*x(2) ...
        +x(6)*x(5)*x(4)*x(2)+2*x(6)*x(5)*x(4)*x(3)-2*x(6)*x(5)*x(4)* ...
        x(3)*x(2)+x(5)*x(4)*x(3)*x(2)-x(6)*x(4)*x(3)+x(6)*x(5)*x(2)+ ...
        2*x(6)*x(4)*x(3)*x(2)-x(5)*x(4)*x(2))+2*(n(3)+2*x(5)*x(4)*x( ...
        3)*x(2)-2*x(6)*x(5)*x(4)*x(2)*x(1)-x(4)*x(3)-2*x(5)*x(4)*x(3 ...
        )*x(2)*x(1)+2*x(6)*x(4)*x(3)*x(1)+2*x(6)*x(5)*x(2)*x(1)+x(5) ...
        *x(4)*x(2)*x(1)-x(6)*x(1)-x(5)*x(2))*(-2*x(6)*x(5)*x(4)*x(2) ...
        -2*x(5)*x(4)*x(3)*x(2)+2*x(6)*x(4)*x(3)+2*x(6)*x(5)*x(2)+x(5 ...
        )*x(4)*x(2)-x(6))+2*(n(2)+2*x(5)*x(4)*x(2)*x(1)-x(5)*x(1)-x( ...
        4)*x(2))*(2*x(5)*x(4)*x(2)-x(5))-2*(n(1)-x(4)*x(1))*x(4)];
    ];

    scale_bit = 2;
    f = a + scale_bit*(sum((x.^2-x).^2));
    g = b + scale_bit*2*(x.^2-x).*(2*x-1);
```

Finally, the constraints for the minimization function is as follows.

```
function [c, ceq] = factmltlogcon(x)
    % inequality constraints
    c = [];

    % equality constraints
    %ceq = sum((x.^2-x).^2);
    ceq = (x.^2-x).^2;
```

A.2 Integer Optimization

A.2.1 Factoring Bit Game

The code for the factoring bit game experiment is not listed here because it was over 5,000 lines of code. Please contact the author for a compressed tarball of the code.

A.2.2 Branch and Bound Multiplication Code

The following C code was used to count the number of terms used in the branch and bound multiplication experiment.

```
#include <stdio.h>
#include <stdlib.h>

#define pow2(a) ((unsigned long)1<<a)

/****
 * returns the bit b of the multiplication of x and y
 */
int bitmult(unsigned int x, unsigned int y, unsigned char b)
{
  return x*y>>b&1;
}

/*****
 * returns the entry at (i,j) in knokecker product matrix with a
 * 2x2 base matrix to the exponent of pow.
 */
int kronpow(int *base, char pow, unsigned long i, unsigned long j)
{
  unsigned long k, size = 1<<pow;
  int entry=1;
  for (k=0; k<size && entry != 0; k++) {
    entry *= base[2*(j&1)+(i&1)];
```

```
    i>>=1;
    j>>=1;
  }

  return entry;
}

int main(int argc, char *argv[])
{
  int R2[4] = { 1, 0, 1, 1 };
  unsigned long i, j, k;
  unsigned long entry;
  unsigned int size, *terms;
  unsigned int bitstream;
  FILE *imm = NULL;

  /* get the size argument or assume size = 2. size is the number of bit
   in n. open a file to store actual matrix */
  if (argc > 1) {
    size = 2*atoi(argv[1]);
    if (size < 1) size = 2;
    if (argc > 2)
      imm = fopen(argv[2], "w+");
  } else
    size = 2;

  /* create some space to store the number of total terms and initialize */
  terms = (unsigned int *) malloc(sizeof(unsigned long)*size);
  for (k=0; k<size; k++)
    terms[k] = 0;

  bitstream = 1;
  for (j=0; j<pow2(size); j++) {

    /* display progress every 100th j */
    if (j%100==0) {
```

```
    printf("[%2.f\%]: ", ((float) j / pow2(size)*100));
    for (k=0; k<size; k++)
      printf("%4.1d ", terms[k]);
    printf("\n");
  }

  for (k=0; k<size; k++) {
    entry=0;
    for (i=0; i<pow2(size); i++)
      /* main matrix calculation of R = K(R2,2*n)*P */
      entry += kronpow(R2, size, i, j)*
                bitmult(i>>size/2, i&pow2(size/2)-1, k);
    //printf ("%d ", entry%2);
    terms[k] += entry%2;

    /* if we're using a matrix file, spit out the results */
    if (imm) {
      /* spit out as 0 or -1 */
      fputc(0xff*(entry%2), imm);

      /* spit out as a byte bundle of bits */
      /*
      bitstream |= entry%2;
      bitstream <<= 1;
      if (bitstream > 0xff) {
        fputc(bitstream, imm);
        bitstream = 1;
      }
      */
    }
  }
}

/* done, print total */
printf("total terms: ");
for (k=0; k<size; k++)
```

```
    printf("%4.1d ", terms[k]);
  printf("\n");

  free(terms);

  if (imm) {
    /*
    if (bitstream > 1)
      fputc(bitstream, imm);
    */
    fclose(imm);
  }
}
```