YAŞAR UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

MASTER THESIS

AUTOMATED LINEAR ALGEBRA FOR MODERN INTEGER FACTORIZATION ALGORITHMS

FİLİNTA BEDİRHAN YILDIZ

THESIS ADVISOR: ASST.PROF. HÜSEYİN HIŞIL

COMPUTER ENGINEERING

PRESENTATION DATE: 14.08.2018

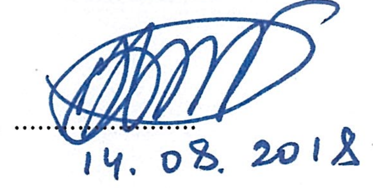
BORNOVA / İZMİR
SEPTEMBER 2018

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

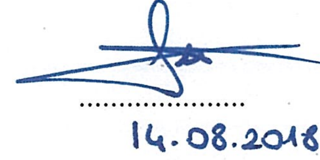
Jury Members: **Signature:**

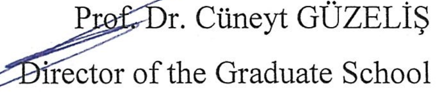
Dr. Öğr. Üy. Hüseyin HIŞIL
YAŞAR University

Prof. Dr. Urfat NURİYEV
EGE University



Dr. Öğr. Üy. Serap ŞAHİN
İZMİR INSTITUTE OF TECHNOLOGY University





Prof. Dr. Cüneyt GÜZELİŞ
Director of the Graduate School

ABSTRACT

AUTOMATED LINEAR ALGEBRA FOR MODERN INTEGER FACTORIZATION ALGORITHMS

Yıldız, Filinta Bedirhan
Msc, Computer Engineering
Advisor: Asst.Prof. Hüseyin HIŞIL, Ph.D.
September 2018

This thesis is on the development of a tool that can convert linear algebra scripts written in Magma language to optimized C codes. Integer factorization algorithms such as Quadratic Sieve (QS) and Number Field Sieve (NFS) produces a large sparse matrix. The nullspace of this matrix is computed as one of the main steps of both QS and NFS. The most convenient algorithms that compute the nullspace are: (distributed) Block Wiedemann and (parallel) Block Lanczos algorithms. Both algorithms are fairly easily implemented on a high level computer algebra tool e.g. Magma. On the other hand, implementing these algorithms in C language in an optimized fashion allowing parallelism between processors/computers is tedious. This thesis work provides a tool that inputs a Magma script written in the context of Block Lanczos and Block Wiedemann algorihtms and outputs an optimized C code which is capable of providing parallelism over a TCP/IP network.

Key Words: Block Wiedemann, block Lanczos, quadratic sieve, linear algebra, number field sieve, integer factorization

ÖZ

MODERN TAMSAYI FAKTORİZASYON ALGORİTHMALARI İÇİN
OTOMATİK LİNEER CEBİR

Yıldız, Filinta Bedirhan
Yüksek Lisans, Bilgisayar Mühendisliği
Danışman: Yrd.Doç.Dr. Hüseyin HIŞIL
Eylül 2018

Bu tez, Magma dili ile yazılmış lineer cebir betiklerini optimize C kodlarına dönüştüren bir aracın geliştirilmesi üzerindedir. Kuadratik Elek (QS) ve Sayı Cismi Eleği (NFS) gibi tamsayıları çarpanlarına ayırma algoritmaları büyük bir seyrek (sparse) matris üretir. Bu matrisin çekirdeğinin (nullspace), hem QS'nin hem de NFS'nin ana adımlarından biri olarak hesaplanması gerekir. Çekirdeği (nullspace) hesaplayan en uygun algoritmalar (dağıtılmış) Wiedemann ve (paralel) Blok Lanczos algoritmalarıdır. Her iki algoritma, yüksek seviyeli bir bilgisayar cebiri aracı üzerinde örneğin Magma, oldukça kolay bir şekilde uygulanabilmektedir. Öte yandan, bu algoritmaların C dilinde işlemciler/bilgisayarlar arasında paralellik sağlacak optimize bir şekilde uygulanması oldukça zahmetlidir. Bu tez çalışması, Block Lanczos ve Block Wiedemann algoritmaları bağlamında yazılmış bir Magma betiğini, TCP/IP ağı üzerinden paralellik sağlayabilecek şekilde optimize edilmiş bir C kodu üreten bir araç sağlamaktadır.

Anahtar Kelimeler: block Wiedemann, block Lanczos, kuadratik elek, lineer cebir, sayı cismi eleği, çarpanlara ayırma

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Dr. Hüseyin Hışıl for his guidance, patience and encouragement during the study.

I would also like to thank my dear friend Ozan Murat for his support and his friendship.

I would also like to thank my colleagues in Yaşar University Department of Computer Engineering for their supports.

Finally, I would like to express my special thanks to my dear family for their endless love, encouragement and patience.

Filinta Bedirhan Yıldız

İzmir, 2018

TEXT OF OATH

I declare and honestly confirm that my study, titled “AUTOMATED LINEAR ALGEBRA FOR MODERN INTEGER FACTORIZATION ALGORITHMS” and presented as a Master’s Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Filinta Bedirhan Yıldız

Signature

....

September 10, 2018

TABLE OF CONTENTS

FRONT MATTER i

ABSTRACT v

ÖZ vii

ACKNOWLEDGEMENTS ix

TEXT OF OATH xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF CODES xvii

1 ABOUT THESIS 1

1.1 MOTIVATION 2

1.2 GOALS OF THESIS 2

1.3 CONTRIBUTIONS OF THESIS 3

1.4 OUTLINE OF THESIS 3

2 INTRODUCTION 5

2.1 INTEGER FACTORIZATION 5

2.2 DISCRETE LOGARITHMS 10

2.3 LINEAR ALGEBRA TECHNIQUES 13

3 BLOCK LANCZOS 17

3.1 SEQUENTIAL LANCZOS ALGORITHM 17

3.2 BLOCK LANCZOS METHOD 19

3.3 LITERATURE REVIEW ON BLOCK LANCZOS 23

3.4 BLOCK LANCZOS EXAMPLE 23

4 BLOCK WIEDEMANN **29**

4.1 WIEDEMANN METHOD 29

4.2 BLOCK WIEDEMANN METHOD 30

4.3 LITERATURE REVIEW ON BLOCK WIEDEMANN 34

5 AUTOMATED LINEAR ALGEBRA **37**

5.1 SOFTWARE LIBRARIES 37

5.2 PROBLEM DEFINITION 37

5.3 C CODE GENERATOR 38

5.4 DISTRIBUTION 42

6 CONCLUSION AND FUTURE WORK **49**

REFERENCES **49**

LIST OF TABLES

3.1 Block Lanczos input 23

3.2 Block Lanczos iteration 24

3.3 Block Lanczos result 25

3.4 Finding nullspace 25

5.1 Our implementation of Block Lanczos test results 44

5.2 Our implementation of Block Wiedemann test results 46

5.3 C implementation of Block Wiedemann test results 46

LIST OF FIGURES

5.1 Our Distribution Model 42

5.2 Matrix Distribution Model 43

LIST OF CODES

5.1 project/Generator/src/com/generator/source/Function.java . . . 39
5.2 project/magma/bl.txt 39
5.3 project/c/bl.c 39
5.4 project/magma/bw.txt 40
5.5 project/c/bw.c 40
5.6 project/c/bw.c 40
5.7 project/magma/bl.txt 40
5.8 project/c/bl.c 40
5.9 project/magma/bl.txt 40
5.10 project/c/bl.c 40
5.11 project/magma/bl.txt 41
5.12 project/c/bl.c 41
5.13 project/magma/bl.txt 43
5.14 project/magma/bw.txt 45
6.1 project/c/bl.c 55
6.2 project/c/bw.c 58
6.3 project/Generator/src/com/generator/source/Function.java . . . 63
6.4 project/Generator/src/com/generator/source/CodeGenerator.java 64
6.5 project/Generator/src/com/generator/io/FileOperations.java . . . 69

CHAPTER 1

ABOUT THESIS

A research area has potential to effect developments in one other. One example to this is the interaction between cryptanalysis and integer factorization. A generic attack on the RSA cryptosystem can be made through factorization of RSA modulus. This connection has been the deriving force on the developments of several integer factorization ideas and algorithms. Two of the distinguished algorithms with sub-exponential time complexity are Quadratic Sieve (QS) and Number Field Sieve (NFS) algorithms. Both algorithms are bottlenecked by an overly involved linear algebra step on sparse matrices. As a consequence, these algorithms have been a deriving force on the development of fast linear algebra techniques on sparse matrices for several decades.

In the ages of Quadratic Sieve (QS) algorithm the linear algebra phase was not a burdon. The integers attacked in 1980s where around 60 decimal digits (200 bits)(Pomerance, 1996). The resulting sparse matrix was solvable in a single multi-processor computer with a large memory.

With the advent of NFS the attack range increased gradually to 230 decimal digits (768 bits)(Kleinjung et al., 2010). The size of the matrix increased not only due to the size of attack range but also due to special structure of NFS algorithm. Such an increase in the matrix size made Block Lanczos impractical. Nevertheless, with the recent advances, block Wiedemann method became the most convenient choice in performing the linear algebra step of NFS.

1.1 MOTIVATION

The optimized implementations of both block Lanczos and block Wiedemann requires extensive programmer effort. This thesis work grew from a simplification attempt of this programming challenge. This thesis aims to answer the following research questions:

Research questions 1 *Can we simplify the code development phase of block Lanczos and block Wiedemann algorithms?*

Research questions 2 *Can we decouple network parallelization efforts from the main body of these algorithms?*

Research questions 3 *Can we decouple the underlying linear algebra engine from the main body of these algorithms without sacrificing the efficiency?*

We did not find any attempt of this sort in the literature to date and it was not clear at the start of the project whether these question could be positively answered.

1.2 GOALS OF THESIS

This thesis has the following goals:

- Understand the principles of Lanczos and Wiedemann algorithms.
- Understand the principles of block Lanczos and block Wiedemann algorithms.
- Implement block Lanczos and block Wiedemann algorithms in Magma language.
- Develop an automated code generator that can input the Magma script(s) and then produce implementations in C language using a suitable linear algebra engine at low-level.
- Improve the automated code generator as to distribute the workload over a TCP/IP network of processors or computers.

1.3 CONTRIBUTIONS OF THESIS

This thesis has the following contributions:

- We developed proof of concept Magma scripts for Block Lanczos and Block Wiedemann algorithms.
- We developed an automated code generator written in Java language that can input the above mentioned Magma scripts and output C implementations of block Lanczos and block Wiedemann algorithms with currently limited parallelization options.

1.4 OUTLINE OF THESIS

The thesis is organized as follows. Chapter 2 provides preliminary information on integer factorization and discrete logarithm problems. The chapter also makes an introduction to linear algebra techniques that are central to this thesis. Chapter 3 and Chapter 4 provides mathematical background on block Lanczos and block Wiedemann algorithms, respectively. These chapters also set the notation and provides literature reviews. Chapter 5 is central to this thesis. The chapter shows how to automate block Lanczos and block Wiedemann algorithms so that their high level implementations in Magma language can turn into an optimized and parallel C code at the touch of a button. Chapter 6 provides test results, conclusions, and possible future research directions.

CHAPTER 2

INTRODUCTION

This chapter provides preliminary information on integer factorization and discrete logarithm problems. The chapter also makes an introduction to linear algebra techniques that are central to this thesis.

This chapter is organized as follows. Section 2.1 states the integer factorization problem and provides an overview of selected methods of integer factorization. Section 2.2 states the discrete logarithm problem with pointers to literature. Section 2.3 summarizes chronological development of linear algebra techniques to compute the nullspace (kernel) of extremely sparse matrices.

2.1 INTEGER FACTORIZATION

The fundamental theorem of number theory states that every integer can be written as the product of finitely many prime factors up to reordering of the primes in a unique way. Integer factorization is the process of finding these primes.

Definition 2.1.1 (Integer Factorization Problem) *Let p and q be two large distinct primes. Assume that $N = pq$. The integer factorization problem is finding p and q when only N is provided.*

Remark 1 *In fact, there are extra conditions that can be added to the definition to circumvent known polynomial time solutions to integer factorization problem. E.g. the problem is very esily solved if p and q are close integers.*

Integer factorization became a very active algorithmic research area after the advent of the RSA cryptosystem since the conjectured security of the RSA

cryptosystem is very closely related to the hardness of integer factorization problem. The RSA algorithm is developed by Ron Rivest, Adi Shamir and Leonard Adleman and was published in 1977. It is one of the most popular public key cryptosystems. RSA cryptosystem separates the encryption and the decryption keys with respect to the public key encryption notion developed by Diffie-Hellman and Merkle independently in 1976. RSA cryptosystem uses a public key to encrypt a message and produce the ciphertext, and the private key associated with the public key to decrypt the ciphertext and obtain the initial message.

Remark 2 *There is no polynomial time algorithm in the literature to solve the integer factorization problem. The best algorithms have sub-exponential complexity.*

A naive approach is the trial division algorithm that trivially checks whether each prime up to $\lfloor\sqrt{N}\rfloor$ divides N. Trial division has exponential time complexity, but trial division is useful for extracting small factors. Integer factorization algorithms are divided into two categories with respect to purpose of usage (Montgomery, 1994) :

- Trial division, Pohlig-Hellman, Pollard's ρ, $p\pm1$ and Elliptic Curve Method (ECM) algorithms are a few examples in the first category that, find small prime factors of N quickly,

- Second category algorithms include continued fractions, quadratic sieve and number field sieve. These algorithms are developed to find large prime factors. Also, these algorithms have sub-exponential time complexity which mainly depend on size of the N. These algorithms usually call first category algorithms as subroutine.

RSA factoring challenge was put forward by RSA laboratories to encourage researchers to study the integer factorization problem in 1990. Since then,

there had been a great research motivation towards the factorization problem. Challenge was ended in 2007 and the factorization of RSA-768 is the record that was broken with using general number field sieve(Kleinjung et al., 2010).

Fermat's Difference of Squares. Fermat's method is the starting point for both quadratic sieve and number field sieve. Fermat tried to express N as a difference of two squares $X^2 - Y^2$ with the pair X and Y different from $(N+1)/2$ or $(N-1)/2$. Any other representation of N as $X^2 - Y^2$ gives a nontrivial factorization $N = (X - Y)(X + Y)$. Clearly, $x \geq \sqrt{N}$(Wagstaff, 2002).

Kraitchik's Method. The advent of continued fractions algorithm (`CFRAC`) is a turning point in the history of integer factorization. `CFRAC` is the first sub-exponential integer factorization algorithm and advent of the later algorithms build on the basics of `CFRAC`.

Theorem 1 (Continued Fractions) *If n is a composite positive integer, X and Y are integers and $X^2 \equiv Y^2 \pmod{N}$, but $x \neq \pm y$ then $gcd(X - Y, N)$ and $gcd(X + Y, N)$ are proper factors of N.*

These concepts are used for all known subquadratic integer factorization algorithms. The algorithms try different ways to satisfy the equation $X^2 \equiv Y^2 \pmod{N}$ but all algorithms use some same definitions. It is infeasible to find X and Y for the big number N by trying random integers, therefore, the algorithms collect many relations to construct the equation. Historically, usage of the relations starts with Kraitchik(Wagstaff, 2002). Kraitchik represents relations as

$$x_i^2 \equiv q_i = p_1^{e_1} p_2^{e_2} \dots p_k^{e_k}.$$

Also, he suggests a method to find a subset of a given set of relations $x_i^2 = q_i \pmod{N}$ that product of these q_i's is a square. If q_i is factored to small primes, then try to match these primes with the primes in the other relations that primes have occurred even number of times. This method finds X and Y when both

sides are the perfect square. Y is perfect square when all exponents of the primes are even. Exponents of the relations can be viewed as a vector

$$v_i = (e_1, e_2, \ldots, e_k).$$

The method has set of these v_i's and find the subset of v_i's whose addition of the vectors is an even. These vectors constructs homogeneous system of linear equations $\sum_{i=0} z_i e_{i,j}$ for $1 \le j \le k$. Linear algebra is used for solving this system.

Quadratic Sieve. Quadratic sieve algorithm (`QS`) is developed by Carl Pomerance. The advent of `QS` is a major milestone in the history of integer factorization (Pomerance, 1985). `QS` provided solid speed-ups and later was improved by many other researchers. `QS` is a similar with `CFRAC`. Sieve is the major difference between the two algorithms. Sieve mainly speeds up the finding relation and increases the ratio of smooth relations. Along time, the major improvements include Montgomery's Multiple polynomial variants and Contini's Self-Initializing variant(Contini, 1997). `QS` is the second fastest algorithm after general number field sieve and `QS` is still faster than number field sieve for numbers under 100 digits.

First step of `QS` starts with a polynomial $f(x) = x^2 - N$ and integer starting point $s = \lceil \sqrt{N} \rceil$. The original version of `QS` uses single polynomial f. `QS` precomputes factor base. Factor base is the collection of primes $p < B$ which the Legendre symbol $(\frac{N}{p} = \pm 1)$. Basically, If a prime divides the polynomial $f(x)$, then $x^2 \equiv N \pmod{p}$. N has a square root in modulo p. Also, two roots of x satisfy the congruence. If x_i is the solution of congruence, then $x_i + p, x_i + p, \ldots$ are solutions. These roots x_i can be precomputed for each p and will be used for sieving. The sieving begins with an empty array that called as sieve array. `QS` marks xth index of the sieving array and continue for $x+p, x+2p, \ldots$ the indexes. The marking process is repeated for each prime in the factor base. Another array F is initialized and is used to store marking values. The logarithm of primes is

used for marking process that logarithm of the prime is added to $F[i]$. The major advantage of sieving is detecting indexes that is divisible by prime p without using trial division. After the sieve, sieving array is scanned and more marked indexes $F[i]$ are determined. Indexes are selected with the threshold T. If the value of $F[i]$ is bigger than T, then $f(x+i)$ is most likely smooth relation. Factorize $f(x+i)$ with using trial division. If all factors of $f(x+i)$ are smaller than B, then these factors represent as a relation. Linear algebra starts after enough relation is collected. Linear algebra constructs the congruences $x^2 \equiv y^2 \pmod{N}$ that gives factorization of N.

Number Field Sieve. Number Field Sieve (`NFS`) is developed by John M. Pollard(Lenstra et al., 1990), (Buhler et al., 1993). `NFS` is the fastest integer factorization algorithm over 100 digits. The aim of the `NFS` is same as quadratic sieve (Finding relation in form $x^2 \equiv y^2 \pmod{N}$). `NFS` brings integer factorization problem to a new domain. The algorithm uses other rings along with $\mathbb{Z}$. This improvement significantly speeds up finding relation process. In the quadratic sieve, x is an integer and y is a product of primes, but in `NFS`, x is a product of prime ideals from the algebraic field and y is same as the quadratic sieve.

`NFS` is consist of three steps: polynomial selection, sieving and linear algebra. Polynomial selection process is more complex than `QS`'s polynomial selection. `NFS` selects an irreducible monic polynomial $f(x)$ and m is an integer that satisfies $f(m) \equiv 0 \pmod{N}$. α is a complex root of f and ring $\mathbb{Z}[\alpha]$ is a sub-ring of the number field $Q[\alpha]$. The root α can be mapped to $m \pmod{N}$ and this is a ring homomorphism ϕ_a

$$\phi_a : \mathbb{Z}[\alpha] \rightarrow \mathbb{Z}/n\mathbb{Z}.$$

`NFS` requires rational and algebraic factor bases. Rational factor base is collection of the primes p that satisfies $p < B$.

Algebraic factor base includes elements $a + b\alpha \in Z[\alpha]$. It is difficult to

represent $Z[\alpha]$ on a computer language, prime p and integer r are used for represent $a + b\alpha$ (p and r satisfies the polynomial $f(r) \equiv 0 \pmod{p}$). Pairs (p, r) are stored in algebraic factor base. There are two sieving arrays for each factor base.

The sieving want to find a set S of (a, b) pairs such that

$$\prod_{(a,b)\in S} (a - b\alpha) = \gamma^2 \quad \text{and} \quad \prod_{(a,b)\in S} (a + bm) = x^2.$$

If $a \equiv -br \pmod{p}$, then the pair (p, r) divides ideal $a + b\alpha$. a is represent as $a = -br + kp$ for integer k. Sieving is done with iterating k with respect to $[-a, a]$. Sieving for rational side can be evaluated with the same logic. After the sieve, algebraic sieve array is scanned and more marked indexes are determined. Factorize $f(x + i)$ with using trial division. If all factors of $f(x + i)$ are smaller than B, then this index is checked in the rational sieve array. These factors are represented as a relation when $f(x + i)$ is smooth for both sieve arrays. Linear algebra starts after enough relation is collected. Linear algebra constructs the congruences

$$y^2 \equiv \prod_{(a,b)\in S} \phi_a(a + b\alpha) \equiv \prod_{(a,b)\in S} (a + bm) \equiv x^2$$

that gives factorization of N.

2.2 DISCRETE LOGARITHMS

Definition 2.2.1 (Discrete Logarithm Problem) *Let GF_q denotes the finite field of order Q. The discrete logarithm problem in GF_q is that of determining, given a generator g of GF_q^* and an element $h \in GF_q^*$ the integer $x = \log_g h$ satisfying $h = g^x \in [0, q-2]$ (Adj et al., 2015).*

Many cryptographic algorithms based on the hardness of the discrete logarithm problem. It is difficult to find x such that $a^x = b$ for given integers a, b. The discrete logarithm algorithms are applicable in any group. In elliptic

curves, the algorithm finds x that $Q = Px$ for given points P, Q on an elliptic curve E. There is no polynomial time algorithm in the literature to solve the discrete logarithm problem in the multiplicative group of a finite field or in an elliptic curve group. Diffie-Helman key-exchange and ElGamal encryption are cryptographic systems that are very related to the hardness of the discrete logarithm problem. Coppersmith has published the first subquadratic discrete logarithm algorithm. Also, integer factorization algorithms can be used for discrete logarithm problem with some modifications (Wagstaff, 2002).

Index calculus. Integer factorization algorithms are used to solve $a^x \equiv b \pmod{p}$ with some modifications. Index calculus method is the analogue for quadratic sieve. The algorithm chooses a factor base of primes $p_1, p_2, \ldots, p_k$ that all primes $p_i \leq B$. The algorithm try to factorize $a^x \pmod{p}$ for random integer x. Trial division or other factorization algorithms (Integer factorization algorithms in the first category) can be used for this factorization. If all factors of $a^x \pmod{p}$ are smaller than B, then these factors represent as a relation (Wagstaff, 2002). The algorithm needs to find at least $k + 20$ factored relations:

$$a^{x_i} \equiv \prod_{i=1}^{k} p_i^{e_{i,j}} \pmod{p} \quad \text{for} \quad 1 \leq j \leq k + 20.$$

Relations can be viewed as:

$$x_i \equiv \sum_{i=1}^{k} e_{i,j} \log_a p_i \pmod{p-1} \quad \text{for} \quad 1 \leq j \leq k + 20.$$

These relations are stored in the matrix and x_i's are stored in vector. The sparse matrix is defined over modulo $p - 1$ which is different than integer factorization problem. Linear algebra solves the matrix with the vector. Using Gauss elimination is infeasible because row operations causes the same problem as the integer factorization. Block Lanczos or block Wiedemann can be used for solving this sparse matrix. After linear algebra, the algorithm do computation to

find $\log_a b$ and tries the random values s $(s < p-2)$ until $ba^s \pmod p$ is smooth. If all prime factors of $ba^s \pmod p$ is in the factor base, then $ba^s \pmod p$ is smooth.

$$ba^s \equiv \prod_{i=1}^{k} p_i^{c_i} \pmod p.$$

This congruence is equal to

$$\log_a b + s \equiv \prod_{i=1}^{k} c_i \log_a p_i \pmod{p-1}.$$

The value of $\log_a p_i$'s can be precomputed and c_i's are gathered from solution vector.

Joux's improvement. The algorithm has different complexities for the different characteristic. Antonie Joux improves index calculus algorithm that solves the discrete logarithm problem with heuristic complexity for finite fields of small characteristic(Joux, 2014). His variant improves finding relation step of index calculus algorithm. He constructs new variant of index calculus algorithm with a few basic ideas. The polynomial f can be converted into different polynomial with changing the variable $f(X) \to f(aX)$ for constant a. He changes this X with homographies

$$X \to \frac{aX+b}{cX+d}.$$

This idea increase number of copies of the polynomial. The second idea is dividing polynomial systematically by

$$f(X) = X^q - X.$$

The polynomial can be factorized because every elements in GF_q i a root of the polynomial. The last idea is creating a field with using polynomial $f(X) = X^q - X$ that only contains the terms X^{q+1}, X^q, X and 1. This terms have multiplicative relation that can generates a finite field. These ideas creates

principles of new algorithm. He replaces sieving step with new algorithm that directly creates relations.

2.3 LINEAR ALGEBRA TECHNIQUES

Linear algebra is the last part of QS and NFS. It is necessary for solving $Bx = 0$ in modulo 2 where B is a sparse matrix. There are many methods for finding x, but we focus block Lanczos, block Wiedemann and Gaussian elimination. Gauss elimination has the worst complexity which is $O(n^3)$.

Three subquadratic algorithms need to solve a large system of linear equations over $\mathbf{F}_2$. The most of the elements in this large system are zero. For example, the matrix has nearly 15-20 non zero elements in 50000 size column. This sparseness brings significant advantage for storing the matrix and reduce the cost of matrix-vector products. Storage requirement is reduced because only non-zero elements are stored in a special format. There are several data structure types for storing sparse matrices. The most primitive way is the coordinate list (COO). This method stores the coordinates and values of non zero elements. There are three one dimensional arrays: row array, columns array and value array. Value array is not needed because integer factorization algorithms do calculation over $\mathbf{F}_2$. Compressed Sparse Row (CSR) format is an improved version of COO. CSR has the same columns array and row array as COO but, uses different row array format. CSR fills row array with a number of non zero elements in the column. This customization has reduced the size of row array. Also, row array contains information about the number of non zero elements in a row. This is a CSR example for example matrix. The matrix with 30 elements can be stored by 9 elements.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$column_indexes = [0, 3, 5, 0, 4]$$

$$row_offsets = [0, 3, 4, 5]$$

Gauss elimination is the most primitive way to solve this matrix. The algorithm has $O(n^3)$ time complexity and causes storage problems. Elementary row operations disturb the sparsity of the matrix because non-zero elements are added to zero elements with high probability. When the number of non zero elements increase, storage of the matrix is increased. The Lanczos method (Lanczos, 1950) and Wiedemann algorithm (Wiedemann, 1986) are algorithms that protect the sparsity of matrix. These algorithms are not specified in the field $\mathbf{F}_2$. These algorithms are processing a single bit but current computers can compute 64 bits or powers of 64 bits. Block Lanczos (Montgomery, 1995) and block Wiedemann (Coppersmith, 1994) are block versions of these algorithms.

Montgomery's Block Lanczos. Montgomery brings Lanczos algorithm to the new domain (Montgomery, 1995). Lanczos algorithm works over the rational field and the algorithm has the fundamental problem over $\mathbf{F}_2$. Montgomery develops the extended version of Lanczos iteration that uses block matrices and works over $\mathbf{F}_2$. Block Lanczos algorithm solves $Ax = b$ for a symmetric matrix A. A matrix B is not a symmetric matrix for integer factorization problem. The matrix A can be computed with BB^T, but this multiplication disturbs the sparseness. The matrix A is never computed. When multiplication with A is needed, multiplication with B^T and B is used $(Ax = (B^T Bx))$. The algorithm uses computer word size or powers of computer word size to represent

block matrices because. After block Lanczos, few more computation is necessary for finding the nullspace of B. Montgomery's block Lanczos algorithm will be reviewed in 3.2.

Coppersmith's Block Wiedemann. Coppersmith develops Block Wiedemann algorithm that generalizes Wiedemann algorithm (Coppersmith, 1994). Block Wiedemann algorithm consist of three stages: `BW1`, `BW2` and `BW3`. `BW1` generates matrix sequence with using B and vectors. `BW1` has quadratic time complexity. `BW2` finds linear generator polynomial f for the matrix sequence. This stage had quadratic time complexity, but Thomé develops a subquadratic variant. `BW3` finds the vectors in the nullspace of B with using f. Block Lanczos had been used primarily for integer factorization problem before 2005. After, block Wiedemann is preferred because block Wiedemann provides better distribution opportunities than block Lanczos. The algorithm will be explained in detail at 4.2.

Thomé's Subquadratic Approach. Thomé improves the time complexity of the block Wiedemann algorithm. He changes the second step (`BW2`) of the algorithm. Coppersmith's version of `BW2` has quadratic time complexity. The reason of quadratic complexity that the algorithm evaluates the coefficient matrix at each iteration t. This coefficient matrix is created with the multiplication of the matrix with a polynomial. His divide-and-conquer approach replaces this multiplication with big polynomial multiplications (Thomé, 2001), (Thomé, 2002). Also, Coppersmith's `BW2` generates one transformation matrix P for each iteration. This replacement generates multiple transformation matrices $P^t, P^{t+1}, \ldots, P^{t+k-1}$ for known k. The algorithm does calculations with more transformation matrices and this change reduces the time complexity of `BW2` to subquadratic time complexity.

CHAPTER 3

BLOCK LANCZOS

This chapter summarizes Lanczos's method of generating nullspace of the matrix. Although the original method solves matrix equations, our treatment is specialized to nullspace computation. Section 3.1 provides classic algorithm of Lanczos. Section 3.2 explains Montgomery's block version of the Block Lanczos algorithm.

3.1 SEQUENTIAL LANCZOS ALGORITHM

Lanczos iteration solves homogeneous or non-homogeneous equations. The purpose of the algorithm is solving $Ax = b$. The algorithm requires a symmetric matrix A that n has rows and columns. A matrix B is never symmetric form for some cases such as integer factorization. The matrix A can be created with $A = BB^T$, but this multiplication disturbs the sparseness of the matrix. The matrix A is never computed. Multiplication with A is the same computation as multiplying with B^T and B ($Ax = BB^Tx$). The algorithm applies the Gram-Schmidt process to Krylov subspace. Krylov subspace is generated by the matrix A and the vector b. The subspace is viewed as $w_i = b, Ab, A^2b, \ldots, A^ib$.

The algorithm iterates the vector space w until each vector is linearly dependent. The algorithm applies A-orthogonal process to each vector w.

$$w_0 = b,$$

$$w_i = Aw_{i-1} - \sum_{j=0}^{i-1} c_{ij} w_j \quad \text{where} \quad c_{ij} = \frac{w_j^T A^2 w_{i-1}}{w_j^T A w_j}.$$

The iteration stops at step m, when $w_m = 0$.

Solution x of algorithm can be found by

$$x = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_j \,.$$

Solution can be verified by

$$Ax = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} A w_j$$

$$l \in 0, 1 \ldots, m-1 \quad w_l^T A x = \sum_{j=0}^{m-1} \frac{w_j^T b}{w_j^T A w_j} w_l^T A w_j = w_l^T b.$$

Since all vectors in the subspace are A-orthogonal. All $w_l^T A w_j$'s are zero apart from $w_l^T A w_l (l = j)$. When $l = j$, $w_l^T A w_l (l = j)$ is simplified with denominator. This shows that $w_l^T A x = w_l^T b$ and $w_l^T (Ax - b) = 0$.

$$Ax - b \in \mathbf{span}(Aw_0, Aw_1, \ldots, Aw_{m-1}, b) = \mathbf{span}(w_0, w_1, w_{m-1})$$

This construction shows that $Ax - b$ is in the vector subspace. $w_j^T A x = w_j^T b$, so $(Ax - b)^T (Ax - b) = 0$.

Some c_{ij}'s are vanished when $j < i - 2$:

$$\begin{aligned} w_j^T A^2 w_{i-1} &= (Aw_j)^T (Aw_{i-1}) \\ &= (w_{j+1} + \sum_{j=0}^{k} c_{j+1,k} w_k)^T (Aw_{i-1}) = 0. \end{aligned} \tag{3.1}$$

This operation simplifies the iteration to $w_i = Aw_{i-1} - c_{i,i-1} w_{i-1} - c_{i,i-2} w_{i-2}$. All $c_{i,j}$'s are zero except $c_{i,i-1}$ and $c_{i,i-2}$.

$$c_{i,i-1} = \frac{(Aw_{i-1})^T(Aw_{i-1})}{w_{i-1}^T Aw_{i-1}} \quad c_{i,i-2} = \frac{(Aw_{i-2})^T(Aw_{i-2})}{w_{i-2}^T Aw_{i-2}}$$

The algorithm satisfies the condition $w_m = 0$ at most N step. The variables Aw_{i-2},Aw_{i-1},$w_{i-2}^T Aw_{i-2}$ are known from previous steps. This shifting of vectors reduces the number of multiplications. The algorithm stores the matrix A and some vectors. Running time of the algorithm is $O(dN^2)+O(N^2)$. d is the average number of non-zero elements in the column.

3.2 BLOCK LANCZOS METHOD

Coppersmith develops a block version of Lanczos algorithm (Coppersmith, 1993) and Coppersmith suggests to use block matrices over instead of vectors. Later, Montgomery develops his version of block Lanczos algorithm. Montgomery's block Lanczos has better complexity than Coppersmith's version. Block Lanczos algorithm moves standard Lanczos iteration to the field $\mathbf{F}_2$. The original Lanczos iteration works over rational field. The iteration requires $w_i^T Aw_i \neq 0$ to compute w_i. This requirement causes problems in the field $\mathbf{F}_2$ because half of all vectors are A-orthogonal to themselves ($w_i^T Aw_i = 0$). Montgomery uses inverse instead of division, when $c_{i,j}$s are created. He generalizes the iteration and replaces the vectors w_i with sequence of vectors W_i (Montgomery, 1995). These modifications change the rules of Lanczos iteration. Also, working with the block of vectors creates an advantage for multiplications with the matrix over $\mathbf{F}_2$. N is equal to computer word size or powers. Elements of the sparse matrix are represented by bits and bitwise operations are significantly faster.

Block Lanczos algorithm requires a symmetric matrix A that has $n \times n$ size over $\mathbf{F}_2$. The algorithm generates A-orthogonal vector subspaces that satisfies following definitions(Montgomery, 1995):

Definition 3.2.1 *A subspace $W \subset K^n$ is said to be A-invertible if it has a basis W of column vectors such that $W^T A W$ is invertible.*

Definition 3.2.2 *Let V and W be a subspace of K^n. Then, V and W are called A-orthogonal subspaces, if $v^T A w = 0$ for all $v \in V$, $w \in W$. This is written as* $V^T A W = 0$.

Solution x of algorithm can be found by

$$x = \sum_{j=0}^{m-1} W_j (W_j^T A W_j)^{-1} W_j^T b$$

The verification of the solution is similar to Lanczos iteration.

Montgomery replaces the Lanczos iteration by

$$\begin{aligned} W_i &= V_i S_i \\ V_{i+1} &= A W_i S_i^T + V_i - \sum_{j=0}^{i} W_j C_{i+1,j} \qquad (i \geq 0) \\ \mathcal{W}_i &= \langle W_i \rangle. \end{aligned}$$

The iteration stops at step m, when $V_m A V_m = 0$.

The algorithm generates the matrix W_i that must be A-invertible. W_i selects every possible columns of V_i that columns satisfy the condition W_i is A-invertible. The iteration ensures that V_i is A-orthogonal to all previous W_j $(j < i)$.

S_i is an $N \times N_i$ projection matrix and N_i is the number of selected columns of V_i. S_i ensures that W_i is A-invertible. S_i has only one 1 per column. For example, if first, second and fifth columns of V_i are selected, then

$$S_i = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

$$V_{i+1} = AW_iS_i^T + V_i - \sum_{j=0}^{i} W_jC_{i+1,j} \qquad (i \geq 0)$$

$$C_{i+1,j} = (W_j^TAW_j)^{-1}W_j^TA(AW_iS_i^T + V_i).$$

The creation of matrix V_{i+1} ensures that V_{i+1} is A-orthogonal to all previous W_j's for $j \leq i$ ($W_j^TAV_i = 0$). The algorithm selects columns of V_i that is not used for generation of W_i. These selected columns are already A-orthogonal to $W_0, W_1, \ldots, W_{i-1}$, but not to W_i. These columns become A-orthogonal to W_i with the operation $V_i - W_iC_{i+1,i}$. Also, V_i prevents the rank problem.

This iteration requires a lot of multiplications, so Montgomery try to simplify $C_{i,j}$s for $j \leq i - 3$

$$V_{i+1} = AW_iS_i^T - W_iC_{i+1,i} - W_{i-1}C_{i+1,i-1} - W_{i-2}C_{i+1,i-2} \qquad (i \geq 2).$$

The algorithm gets replaces $C_{i,j}$s with D_{i+1}, E_{i+1} and F_{i+1}. The findWinv function 3 selects S_i and creates W_{inv}. D_{i+1}, E_{i+1}, and F_{i+1} are block matrices with of size $N \times N$.

$$W_i^{inv} = S_i(W_i^TAWi)^{-1}S_i^T = S_i(S_i^TV_i^TAV_iS_i)^{-1}S_i^T,$$

$$V_{i+1} = AV_iS_iS_i^T + V_{i-1}D_{i+1} + V_iE_{i+1} + V_{i-2}F_{i-2},$$

$$D_{i+1} = I_N - W_i^{inv}(V_iA^2V_iS_iS_i^T + V_iAV_i),$$

$$E_{i+1} = -W_{i-1}^{inv}V_i^TAV_iS_iS_i^T,$$

$$F_{i+1} = -W_{i-2}^{inv}(I_N - V_{i-1}^TAV_{i-1}W_{i-1}^{inv})(V_{i-1}A^2V_{i-1}S_{i-1}S_{i-1}^T + V_{i-1}AV_{i-1})S_iS_i^T.$$

This simplification reduces numbers of multiplications. The algorithm stores some temporary vectors during the iteration and returns the solution X and V_m.

Runtime complexity is $O(dn) + O(Nn)$ for each iteration and $O(dn^2/N) + O(n^2)$ is for the algorithm. The pseudo code example of the algorithm can be found in Algorithm 1.

Selecting S_i and W^{inv} W^{inv} and SS^T are matrices of size $N \times N$. W^{inv} is generated by calculating the inverse of $V_i^T AV_i$. Gauss elimination is the primitive way to compute the inverse of a matrix. The identity matrix is added the right side of the matrix $[V_i^T AV_i|I_N]$ and elementary row operations are applied. The left side of the result $[I_N|V_i^T AV_i^{-1}]$ is the inverse of a matrix. Montgomery calculates the inverse of the matrix with some modifications. Rows and columns of the matrix are sorted by S_{i-1} that selected columns are put last indexes (Algorithm primarily wants to use new columns). Selecting columns of S_i and inverse operation are performed together. The pseudocode of this operation can be found in Algorithm 3.

Finding Nullspace of A Sparse Matrix Block Lanczos algorithm solves the equation $Ax = b$, but integer factorization problem needs to find $Bx = 0$. The matrix B is a large sparse matrix. The algorithm requires a symmetric matrix, so $A = B^T B$ (The matrix A is never computed). If x is the solution of $Bx = 0$, then satisfies the equation $Ax = 0$. Block size N is selected as computer word size or powers of computer word size. The matrix Y is selected randomly that has n rows and N columns. The matrix V_0 is $N \times n$ and is initialized $V_0 = AY$. Block lanczos iterates the V_i until $V_m^T AV_m = 0$. The algorithm returns the matrix X and V_m. There are two possibilities. If $V_m = 0$ then $AX = AY$. $A(X - Y) = 0$ and columns of $X - Y$ belongs to the null space of the matrix A. If $V_m \neq 0$, then V_m and $X - Y$ are belongs to nullspace of A. Linear combinations of $X - Y$ and V_m are computed to find the null space of A. Gaussian elimination can be used for this calculation. Let Z is of size $n \times 2N$ that combine $X - Y$ and V_m ($[X - Y|V_m]$). U is a basis of nullspace of BZ. The U ensures that $B(ZU) = (BZ)U = 0$, so ZU is the nullspace of B. The pseudocode of algorithm can be found in Algorithm 2.

3.3 LITERATURE REVIEW ON BLOCK LANCZOS

Flesch suggest parallel approach for block Lanczos (Flesch, 2002). He parallelizes the sparse matrix vector multiplication. The pseudocodes 1, 3, 2 are taken from (Flesch, 2002). Thomé tries to reduce the number of stored vectors (Thomé, 2016).

3.4 BLOCK LANCZOS EXAMPLE

For Example, B is a example matrix of size 20×20 with %50 sparsity over the field $\mathbf{F}_2$. Y is a random matrix of size 20×4 over the field $\mathbf{F}_2$. Block size is defines as N = 4. The initial values of $W_i^{inv} = 0$, $V_i = 0$ and $SS_i^T = I_N$ for $i < 0$.

$$
\mathbf{B} = \begin{bmatrix}
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}, \quad
\mathbf{Y} = \begin{bmatrix}
1 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1
\end{bmatrix}
$$

Table 3.1: Block Lanczos input

Matrix	1	2	3	4	5	6
V_i	0 0 1 0 1 0 1 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0 1 1 1 1 1 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 0 1 1 1 0 1 0 1 1 1 0 1 1 0 1 0 0 0 0 0 1 0 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1	0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 1 1 0 0 1 0 0 1 1 1 1 0 1 0 1 0 0 1 1 0 0 0 0 0 0 1 0 0 1 0 0 1 0 1 1 1 0 0 0 0 1 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 0 0 0 1 0 0 1 1	0 0 0 1 0 0 1 0 0 0 1 1 0 1 0 0 1 0 1 1 0 0 0 0 1 1 0 0 0 0 0 1 1 0 1 0 0 1 1 1 1 0 1 0 0 0 1 0 1 1 1 0 0 1 0 0 0 1 1 1 1 0 0 0 1 0 0 1 1 1 0 0 0 0 0 1 0 0 1 0	0 0 0 0 1 1 1 0 0 1 1 0 1 1 1 0 0 0 1 1 1 0 0 0 1 0 1 1 1 0 0 0 0 1 1 0 1 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 1 0 0 0 0 0 1 0 1 0 1 1 0 1 1 0 1 1 0 1 1	0 1 1 0 0 1 1 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0 1 0 0 0 1 0 0 0 1 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0 1 1 1 0 1 0 0 0 0 1 1 0 0 1 1 0 1 0 0 0 1 1 1 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 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
SS_i^T	1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1	1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1	0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1	1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1	0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0	1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
D_i	1 1 0 1 0 1 1 1 0 0 0 1 1 1 0 1	1 0 0 0 0 1 0 0 1 1 1 0 1 1 0 1	1 0 0 0 0 1 1 1 0 0 1 0 0 1 0 0	1 0 0 0 0 1 1 1 0 0 1 0 1 0 0 1	1 0 0 0 0 1 0 1 1 1 1 1 0 0 0 1	1 1 0 0 1 0 1 0 0 0 1 0 0 0 0 1
E_i	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 1 0 0 1 0 0	0 0 0 1 0 1 0 1 0 1 1 1 0 1 0 1	0 0 0 0 1 0 0 1 0 0 0 1 0 0 0 0	0 0 1 0 0 1 0 0 0 0 0 0 0 1 1 0	0 0 0 0 1 1 0 0 1 1 0 0 0 0 0 0
F_i	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 1 0 0 0 1 0 0 0 0 0 0	0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0	0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0

Table 3.2: Block Lanczos iteration

The example takes 6 steps to find the matrix X and block Lanczos returns X and V_m. Also, if $V_m = 0$, then the nullspace of the matrix B can be found by calculated with $X - Y$.

$$
\mathbf{X} = \begin{bmatrix}
0 & 0 & 1 & 1 \\
1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
1 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 \\
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1
\end{bmatrix}, \quad
\mathbf{V_m} = \begin{bmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0
\end{bmatrix}
$$

Table 3.3: Block Lanczos result

$\mathbf{Z}$	$\mathbf{U}$	$\mathbf{ZU}$	$\mathbf{B * ZU}$
$\begin{bmatrix} 1&1&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$	$\begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix}$	$\begin{bmatrix} 1&1&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 1&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$	$\begin{bmatrix} 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0 \end{bmatrix}$

Table 3.4: Finding nullspace

Block Lanczos satisfies $B^TBX = B^TBY$ and $V_mTAV_m = 0$. After the block Lanczos, nullspace of the matrix B is calculated with the Algorithm 2. $X - Y$ is appended to V_m to create the matrix Z of size 8×20. This matrix is multiplied with B to create BZ. U is the null space of the BZ and can be calculated with Gauss elimination. The algorithm wants to find null space of B and that is $Z \cdot U$ because $BZ \cdot (U) = B \cdot (ZU) = 0$.

This matrix sizes are just an example. The sparse matrix B is of size $192,796,550 \times 192,796,550$ for number field sieve challenge. Each row has average 144 non zero entries. Storage of the sparse matrix requires about 105 gigabytes (Kleinjung et al., 2010).

Algorithm 1 Block Lanczos

1: *Input*:2
2: B size of $n_1 \cdot n_2$
3: Y of size $n_2 \cdot N$
4: *Output*:
5: The matrices X and V_m
6: *Initialize*:
7: $W_{-2}^{inv} \leftarrow 0_{N\text{x}N}$, $W_{-1}^{inv} \leftarrow 0_{N\text{x}N}$
8: $V_{-2} \leftarrow 0_{n_2\text{x}N}$, $V_{-1} \leftarrow 0_{n_2\text{x}N}$
9: $BV_{-1} \leftarrow 0_{n_1\text{x}N}$
10: $V^T B_{-1}^T \leftarrow 0_{N\text{x}n_1}$
11: $SS_{-1}^T \leftarrow I_N$
12: $X \leftarrow 0_{n_2\text{x}N}$
13: $V_0 \leftarrow B^T \cdot (B \cdot Y)$
14: $BV_0 \leftarrow B \cdot V_0$
15: $V^T B_0^T \leftarrow V_0^T \cdot B^T$
16: $Cond_0 \leftarrow V^T B_0^T \cdot BV_0$
17: $i \leftarrow 0$
18: **while** $Cond_i \neq 0$ **do**
19: $\quad W_i^{inv}, SS_i^T \leftarrow \text{findWinv}(Cond_i, SS_{i-1}^T)$
20: $\quad X \leftarrow X + V_i \cdot (W_i^{inv} \cdot (V_i^T \cdot V_0))$
21: $\quad B^T BV_i \leftarrow B^T \cdot BV_i$
22: $\quad K_i \leftarrow (V^T B_i^T \cdot (B \cdot B^T BV_i)) \cdot SS_i^T + Cond_i$
23: $\quad D_{i+1} \leftarrow I_N - W_i^{inv} \cdot K_i$
24: $\quad E_{i+1} \leftarrow -W_{i-1}^{inv} \cdot (Cond_i \cdot SS_i^T)$
25: $\quad F_{i+1} \leftarrow -W_{i-2}^{inv} \cdot (I_N - Cond_{i-1} \cdot W_{i-1}^{inv}) \cdot K_{i-1} \cdot SS_i^T$
26: $\quad V_{i+1} \leftarrow B^T BV_i \cdot SS_i^T + V_i \cdot D_{i+1} + V_{i-1} \cdot E_{i+1} + V_{i-2} \cdot F_{i+1}$
27: $\quad V^T B_{i+1}^T \leftarrow V_{i+1}^T \cdot B^T$
28: $\quad BV_{i+1} \leftarrow B \cdot V_{i+1}$
29: $\quad Cond_{i+1} \leftarrow V^T B_{i+1}^T \cdot BV_{i+1}$
30: $\quad i \leftarrow i + 1$.
31: $V_m \leftarrow V_i$. **return** X and V_m

Algorithm 2 Block Lanczos Nullspace

1: *Input*: The matrix B
2: *Output*: The nullspace X
3: *Algorithm*:
4: $(X, V_m) = blockLanczos(B, Y)$
5: $Z = [X - Y, V_m]$
6: $BZ = [B \cdot (X - Y), B \cdot V_m]$
7: $U = Nullspace(BZ)$
8: $ZU = Z \cdot U$ **return** ZU

Algorithm 3 Finding W^{inv}

1: *Input*: $Cond = V_i^T B_i^T B_{V_i}$ and S_{i-1}
2: *Output*: W_i^{inv} and $S_i S_i^T$
3: *Algorithm*:
4: $M = [Cond, I_N]$ //$N \times 2N$ `matrix with` $Cond$ `on the left and` I_N `on the right`
5: $c = []$
6: $S = []$
7: **for** $j = 1$ to N **do**
8: $\quad k = j$
9: $\quad$ **while** $M[c_j, c_j] = 0$ and $k \leq N$ **do**
10: $\quad\quad$ **if** $M[c_k, c_j] \neq 0$ **then**
11: $\quad\quad\quad$ Swap row j and k of M
12: $\quad$ **if** $M[c_j, c_j] \neq 0$ **then**
13: $\quad\quad$ Add multiples of row c_j to other rows of M, to zero rest of column c_j
14: $\quad$ **else**
15: $\quad\quad k = j$
16: $\quad\quad$ **while** $M[c_j, c_{j+N}] = 0$ and $k \leq N$ **do**
17: $\quad\quad\quad$ **if** $M[c_k, c_{j+N}] \neq 0$ **then**
18: $\quad\quad\quad\quad$ Swap row j and k of M
19: $\quad\quad$ Add multiples of row c_j to other rows of M, to zero rest of column c_{j+N}
20: $\quad\quad$ Zero row c_j of M
21: Copy the right half of M into W_i^{inv} **return** W_i^{inv} and SS_i^T

CHAPTER 4

BLOCK WIEDEMANN

This chapter summarizes Wiedemann's method of generating nullspace of matrices. Although the original method solves matrix equations, our treatment is specialized to nullspace computation. Section 4.1 provides the classic algorithm of Wiedemann, which makes a connection to Berlekamp/Massey algorithm. Section 4.2 explains the Coppersmith's block version of Wiedemann's algorithm.

4.1 WIEDEMANN METHOD

Wiedemann algorithm solves linear equations over $\mathbf{F}_2$. The algorithm requires singular square matrix B that has N rows and N columns. The algorithm finds non zero vector w which satisfies the equation $Bw = 0$. Algorithm consists of three main steps. In the first step, algorithm initialize two random vectors x and z that are of size N. Set $y = Bz$ and compute

$$A_i = x^T B^i y$$

for $1 \leq i \leq 2N$. Each A_i is an integer and whose coefficient of the polynomial.

$$A(X) = \sum_{i=0}^{2N} A_i X^i$$

The second part finds a linear generator for this sequence $A(X)$. Various methods can be applied for this computation. Wiedemann suggests using Berlekamp-Massey algorithm (Wiedemann, 1986). Berlekamp-Massey and Extended Euclidean algorithm are two examples and both algorithms have time complexity $O(N^2)$. Berlekamp-Massey algorithm use the equation $A(X)F(X) =$

$G(X) \bmod X^{2N+1}$ to find $F(X)$ (Massey, 1969). The connection between Berlekamp-Massey and Extend Euclidean was explained in (Dornstetter, 1987). The equation

$$F(X)A(X) = G(X) \pmod{X}^{2N+1}$$

is equal to

$$F(X)A(X) = G(X) + E(X)X^{2N+1}$$

which can be written as invariant equation

$$F^{rev}(X)A^{rev}(X) + G^{rev}(X)X^{2N+1} = E^{rev}(X)$$

where $A^{rev}(X)$ and X^{2N+1} are inputs of the extended Euclidean algorithm. Degrees of F and G are initially 0 and degree of $E(X)$ starts with $2N$. Degrees of F and G tends to increase. On the other hand, the degree of E decreases. At each turn, quotient q is calculated and the polynomials F,G,E are updated accordingly. The algorithm works until the degree of F has not reached N. F is the linear generator polynomial of $A(X)$.

In the last stage, the algorithm uses the generating polynomial F to find the nullspace vector w. The algorithm evaluates the polynomial f with the matrix B. The algorithm already knows that $B^i f(B)z = 0$ for some $i < N$. If $B^i f(B)z = 0$, then $w = B^{i-1}f(B)z = 0$. The vector w is in the null space of the matrix B that satisfies

$$Bw = B(B^{i-1}f(B)z) = 0.$$

4.2 BLOCK WIEDEMANN METHOD

Coppersmith developed the multi-dimensional variant of the Wiedemann algorithm. He suggests replacing vectors with block matrices. Block matrix is a set of vectors and his algorithm processes computer word size vectors. Therefore, the number of matrix-vector products are decreased from $3N$ to $3N/CW$. This

replacement mainly effects the second step of the algorithm and his study focus on the development of matrix Berlekamp-Massey (Coppersmith, 1994).

The aim of the algorithm is the same as sequential Wiedemann. m and n are defined as block sizes. The algorithm requires the case $m \leq n$. x and z are random block matrices instead of vectors. x is of size $n \times N$ and z is of size $N \times m$. Linear sequence is computed with

$$A_n i = x^T B^i y$$

for $1 \leq i \leq (N/m) + (N/n)$. The length of the sequence is reduced $2N$ to $L = (N/m) + (N/n)$. Each element of the sequence A is $n \times m$ matrix.

$$A(X) = \sum_{i=0}^{L} A_i X^i \quad \in K[[X^{m \times n}]].$$

Each A_i has more information than the original algorithm because it is generated with extra vectors. These matrices are coefficients of the polynomial. Also, $A(X)$ can represent as $n \times m$ matrix with polynomial coefficients over $\mathbf{F}_2$. This sequence requires $O(\gamma N^2)$ multiplications that γ is average non zero elements per row. This computation can be done in parallel. Each computer in the distributed network gets the sparse matrix B and block matrix x. Columns of the block matrix y is distributed to computers and each computer do the calculation for specific columns of A_i. The computers do not need to communicate with each other. The distribution of block Wiedemann will be explained in 5.4.

Second stage finds generating polynomial for $A(X)$ and it is the same computation as original Wiedemann when block sizes are one. This step is have to be modified for sequence whose coefficients are matrix. The algorithm develops the polynomial F by each iteration t. The iteration starts with $t_0 = \lceil \frac{m}{n} \rceil$ and increases by 1. The algorithm computes $F^{t+1}(X)$ with using $F^t(X)$. The algorithm uses the same equation as the original Wiedemann. Coppersmith

defines nominal degrees instead of true degrees (Coppersmith, 1994). Nominal degree is the upper bound of true degree of the column. This degree has some properties

$$\deg_{\textbf{nom}}(F+G) = \max(\deg_{\textbf{nom}} F + \deg_{\textbf{nom}} G)$$

$$\deg_{\textbf{nom}}(xF) = \deg_{\textbf{nom}} F + 1.$$

The nominal degree δ is associated with generator polynomial F and is an array of size $n \times m$. The algorithm defines coefficient matrix $E(X)$ with multiplication of F^t and $A(X)$. The aim of the algorithm is canceling this coefficient matrix $E(X)$ for each iteration t with maintaining these conditions:

Theorem 2 *(Thomé, 2001) Let $A(X) \in K[[X]]^{m \cdot n}$. Suppose that $A(X)$ has a left rational form $D(X)^{-1}N(X)$. if we have matrices $F(X) \in K[X]^{n \cdot r}$, $G(X) \in K[X]^{m \cdot r}$, $E(X) \in K[X]^{m \cdot r}$, and an integer t such that:*

$$A(X)F(X) = G(X) + X^t E(x),$$

$$t - \delta(F, G) \geq \delta(D, N) \Rightarrow E(X) = 0.$$

Theorem 3 *(Thomé, 2001) Rank* $([X^t]e) = m$.

The algorithm maintains two conditions (C1) and (C2) for each iteration t. Condition (C1) is extended version of the original equation. The algorithm uses auxiliary Gauss elimination to cancel the coefficient matrix. Auxiliary Gauss generates transformation matrix P with of size $(m+n) \times (m+n)$ and increases last m nominal degrees by 1. This transformation matrix P updates polynomials F, G and E with

$$F^{t+1} = F^t P^t, \quad G^{t+1} = G^t P^t, \quad E^{t+1} = E^t P \frac{1}{X}.$$

Also, auxiliary Gauss maintains (C2) that the last m rows of $F(X)P$ is linearly independent. This iteration stops two cases. The first case is $F^{t+1} = F^t$ when

the transformation matrix is identity matrix.

Linear generator is used to find a solution of the matrix B in the last stage of the block Wiedemann. δ satisfies $A(X)F(X) = g(X), \delta(F,G) \leq d$, After generating polynomial $F(X)$ is obtained(Thomé, 2001).

$$\forall t, t \leq \delta, \quad \sum_{k=0}^{\delta}([X^{t-k}]A)([X^k]F) = 0,$$

$$\sum_{k=0}^{\delta}(x^T B^{t-k} y)([X^k]f) = x^T B^{t-\delta} \sum_{k=0}^{\delta}(B^{\delta-k} y)([X^k]f) = 0.$$

$$\forall t, x^T B^t \sum_{k=0}^{\delta}(B^{\delta-k} z)([X^k]f) = 0.$$

Then, the $w = \sum_{k=0}^{\delta}(B^{\delta-k}z)([X^k]f)$ and w is orthogonal to the span of $x^T B^t$. This satisfies $Bw = 0$ and w is the solution.

Auxiliary Gauss Elimination. This method ensures the two conditions of the algorithm. The method gets coefficient matrix $[X^t]e$ and nominal degrees δ. The method creates transformation matrix P that is of size $(m+n) \times (m+n)$ and first m columns of P matrix is the coefficient matrix and other n columns is $n \times n$ identity matrix. The aim of this method is processing these m columns that become linearly independent.

The method starts with ordering row of the transformation matrix by nominal degrees and applies elementary row operations on the transformation matrix with extra rules. Rows that have lower nominal degrees is never added to one of higher nominal degree. The algorithms start with sorting nominal degrees by increasing order. The method applies Gauss elimination on the coefficient matrix for n columns. After elimination, first, n rows of the coefficient matrix becomes zero. This ensures the condition (C1) and The method develops that last m rows of the coefficient matrix become linearly independent.

Theorem 4 *Assuming conditions $(C1)$ and $(C2)$ hold at step t, there is an algorithm Auxiliary Gauss that, knowing $[X^0]E(X)$ and $(\delta_1^{(t)}, \ldots, \delta_{m+n}^{(t)})$,*

computes a $(m+n) \times (m+n)$ *matrix* $P_{(t)}$ *along with integers* $(\delta_1^{(t+1)}, \ldots, \delta_{m+n}^{(t+1)})$ *such that:*

$$F^{t+1} = F^t P^t, \quad G^{t+1} = G^t P^t, \quad E^{t+1} = E^t P \frac{1}{X}$$

and the $\delta_1^{(t+1)}$*'s satisfy conditions (C1) and (C2) at step* $t+1$*. Furthermore, we have* $\sum_j \delta_1^{(t+1)} - \delta_1^{(t)} = m$*.*

4.3 LITERATURE REVIEW ON BLOCK WIEDEMANN

Kaltofen suggest different generating algorithm for `BW2` (Kaltofen, 1995). Kaltofen and Yuhasz work on matrix Berlekamp-Massey (Kaltofen and Yuhasz, 2013), (Yuhasz, 2009) that solves block Toeplitz matrices. Also, They review algorithms that are used for `BW2`. Penninga implements a version block Wiedemann(Penninga, 1998) that is suggested to him by Villard(Villard, 1997). Villard and Kaltofen develop the extended Euclidean algorithm that works over matrix polynomial (Kaltofen and Villard, 2005). (Yang et al., 2017) They develop parallel general number field sieve algorithm with block Wiedemann.

Algorithm 4 BW 1

1: *Input*:
2: The matrix B of size $N \cdot N$
3: The matrix x of size $N \cdot n$, the matrix z of size $N \cdot m$
4: *Output*:
5: The matrix polynomial A
6: *Initialize*:
7: $v \leftarrow B \cdot z$
8: **while** $i \leq L$ **do**
9: $\quad A[i] \leftarrow x^T \cdot v$
10: $\quad v \leftarrow B \cdot v$
return a

Algorithm 5 BW 2

1: *Input:*
2: The matrix polynomial $A(X)$
3: *Output:*
4: The minimal generating polynomial $F(X) \in K^{n \cdot n}$
5: *Initialize:*
6: $F[1] \leftarrow [I_N, 0]$
7: $t \leftarrow 0$
8: d is array with of size $n + m$ and first n elements are 1 others 0.
9: **while** $t < L$ **do**
10: $E \leftarrow Coeff(t, A(X) \cdot F(X))$
11: $P, d \leftarrow AuxiliaryGauss(E, d)$
12: $d_{N+i} \leftarrow d_{N+i} + 1$
13: $F(X) \leftarrow F(X) \cdot P \cdot \frac{1}{X}$
14: $t \leftarrow t + 1$
15: Sequence $F(X)$ is converted to $(n \cdot n + m)$ matrix.
16: $F \leftarrow Determinant(F(X))$ **return** F

Algorithm 6 BW 3

1: *Input:*
2: B size of $N \cdot N$, v size of $N \cdot n$
3: f size of $n \cdot m$
4: *Output:*
5: The matrix w
6: *Initialize:*
7: $t \leftarrow f(B) \cdot z$
8: **while** $i \leq \delta$ **do**
9: $s \leftarrow B^i \cdot t$
10: **if** $s = 0$ **then**
11: $w \leftarrow B^{i-1} \cdot t$
12: $B^i \leftarrow B^i \cdot B$
13: $i \leftarrow i + 1$
return w

This `BW2` is the Coppersmith's version. $A(X)$ and $F(X)$ are stored in arrays that each element of the array is a matrix. After the while loop end, the algorithm converts reverse of the polynomial sequence $F(X)$ into a single matrix that every

element of the matrix is a polynomial. A determinant of this matrix gives a linear generator.

CHAPTER 5

AUTOMATED LINEAR ALGEBRA

This chapter explains automated linear algebra. Automated linear algebra generates C codes from a given magma script. Section 5.2 explains to the reason of a creating code generator. Section 5.3 introduces code generation process. Section 5.4 introduces distribution process of block Wiedemann.

5.1 SOFTWARE LIBRARIES

These are used libraries and technologies for the automated code generator. FLINT is a library for matrix operations.

FLINT. Fast linear algebra for number theory (FLINT) is a number theory library that is written in C language (Hart, 2010). The library supports arithmetic with numbers, polynomials, power series and matrices over many rings. Matrix polynomial arithmetic is used for block Wiedemann algorithm.

Magma. Magma is a large framework that solves mathematical problems. it has software packages for computations in algebra, number theory, and linear algebra. Magma is distributed by the Computational Algebra Group at Sydney University (Bosma et al., 1997). Block Lanczos and block Wiedemann are implemented over Magma.

5.2 PROBLEM DEFINITION

Many researchers use software tools for solving mathematical problems. They prefer the software package that does calculations in algebra, number theory,

and linear algebra. This software has a user-friendly language and high-level functions. The functions need to solve hard level problems for experimental data. Researchers develop experimental scripts at the initial step of their works. The scripts are very useful for understanding the problem but, this software is not practical for real-world data. Researchers need the low-level software to experiment on the problem with real data. Our aim is developing a tool that generates C code from Magma script. Our software is only useful for linear algebra because we want to solve linear equations over $\mathbf{F}_2$. The software is used to automate block Lanczos and block Wiedemann algorithms with using their high-level Magma implementations. The software is developed in Java.

5.3 C CODE GENERATOR

The generator requires magma script in a structural format. The input magma script needs to have some extra comments due to the syntax of the magma. For example, variables are not identified with specific types in magma. The generator requires extra comments for these syntactic features. The generator uses FLINT library for linear algebra operations. Also, the user can use other linear algebra libraries than FLINT. If the user wants to use own library, then the user needs to add the names of linear algebra functions. This functions should agree with the required function types. The generator creates the C code that requires the FLINT library by default. Also, magma has many special functions for a mathematical operation that functions are not defined in C. We add some extra C functions that are matched with some of these magma functions. These functions are frequently used operations in linear algebra and can be replaced with functions that have better performance in the future.

The code generator should handles:

- Variable initialization (Matrix, Array, Integer, Polynomial),
- Magma function calls (`Determinant()`, `InsertBlock ()`),

- User defined functions,
- Loops and conditions,
- Extra specialities for linear algebra.

```
1 public class Function {
2
3   private String name;
4   private ArrayList<Variable> inputs = new ArrayList<Variable>();
5   private ArrayList<Variable> definedVariables = new ArrayList<Variable>();
```

Code 5.1: project/Generator/src/com/generator/source/Function.java

The generator has a variable class that stores the name and type of the variable. The function class handles functions that defined by the user. The function class has two variable lists. Input list stores parameters with name and type. The other list is filled with variables that are defined in function scope.

```
1 blockLanczos:=function(B,trB,Y,N,NT,F) //
      Matrix X2,Matrix V_m,Matrix B,Matrix
      trB,int N,field F
```

Code 5.2: project/magma/bl.txt

```
1 void blockLanczos(nmod_mat_t X2, nmod_mat_t
      V_m, nmod_mat_t B, nmod_mat_t Y,
2     nmod_mat_t trB, int N, mp_limb_t F) {
```

Code 5.3: project/c/bl.c

The code generator primarily determines the Magma function declarations and stores these function in the function list. `//Matrix X2,Matrix V_m,Matrix B,Matrix trB,int N,field F` comment helps the generator to understand types of the parameters. These parameters are converted to corresponding data types of FLINT. Since FLINT does not have sparse matrix library, all matrices are defined as in`nmod_mat_t` type. The field $GF(2)$ is converted to `mp_limb_t`. These parameters are stored in the input list of the function. Also, Magma functions can return multiple variables directly. In C, the function does not allow to return multiple variables so the generator defines the returned variable as a parameter. The function comments should include returned variables before function parameters. These code snippets are example of array declarations in block Wiedemann.

```
1  f:=[]; // type:Matrix size:N r:N c:M
2  d:=[]; // type:int size:N
```
Code 5.4: project/magma/bw.txt

```
1  initMatrixSequence(f, M, NM, L, F);
2  int *d = (int *) malloc(sizeof(int) * nm);
```
Code 5.5: project/c/bw.c

```
1 void initMatrixSequence(nmod_mat_struct **A,
      int r, int c, int length,
2   mp_limb_t F) {
3   int i;
4   for (i = 0; i < length; i++) {
5     A[i] = (nmod_mat_struct *) malloc(sizeof
        (nmod_mat_t[1]));
6     nmod_mat_init(A[i], r, c, F);
7   }
8 }
```
Code 5.6: project/c/bw.c

In magma implementation, we do not identify the types and length of the arrays, but it is mandatory for C application. `f` is array that have matrix elements. The comment `// type:Matrix size:L r:M c:MN` explains length of the array and size of the matrix elements. `initMatrixSequence()` allocates the array and initializes each matrix element in the array.

```
1      while M[c[j]][c[j]+N] eq 0 and k le N
          do
2        if M[c[k]][c[j]+N] ne 0 then
3          SwapRows(~M,c[j],c[k]);
4        end if;
5        k+:=1;
6      end while;
7      for i2:=1 to N do
8        if i2 ne c[j] then
9          if M[i2][c[j]+N] eq 1 then
10           M[i2]:=M[i2]+M[c[j]+N]; //Row Add
11         end if;
12       end if;
13     end for;
```
Code 5.7: project/magma/bl.txt

```
1      while (M->rows[c[j]][c[j] + N] == 0 &&
          k < N) {
2        if (M->rows[c[k]][c[j] + N] != 0) {
3          swap(M, c[j], c[k]);
4        }
5        k += 1;
6      }
7      for (i2 = 0; i2 < N; i2++) {
8        if (i2 != c[j]) {
9          if (M->rows[i2][c[j] + N] == 1) {
10           addRow(M, i2, c[j] + N);
11         }
12       }
13     }
```
Code 5.8: project/c/bl.c

The character ˜ 5.3 symbolizes pass by reference in Magma. `SwapRows()` is a default magma function. The comment `//Row Add` notifies for row addition. In Magma, we can access row of matrix with `M[i]` and can do calculation with this format. This row addition is not suitable with FLINT so, we create `swap()` and `addRow()` as a default functions that works same as corresponding magma functions. Frequently used functions are automatically added to the generated C code (transpose, swap rows, add row). User can customizes these functions.

```
1  BY     := B*Y;
2  V_i    := trB * BY;
3  BV_i   := B * V_i;
4  trBV_i := Transpose (BV_i);
5  trV_i  := Transpose (V_i);
6  Cont_i := trBV_i * BV_i;
7  V_0    := V_i;
```
Code 5.9: project/magma/bl.txt

```
1  nmod_mat_mul(BY, B, Y);
2  nmod_mat_mul(V_i, trB, BY);
3  nmod_mat_mul(BV_i, B, V_i);
4  nmod_mat_transpose(trBV_i, BV_i);
5  nmod_mat_transpose(trV_i, V_i);
6  nmod_mat_mul(Cont_i, trBV_i, BV_i);
7  nmod_mat_set(V_0, V_i);
```
Code 5.10: project/c/bl.c

In Magma, the symbol of addition and multiplication operations are same for matrices and integers. The generator should identify the multiplication terms between matrix multiplication and polynomial multiplication. The generator decides the multiplication with types of variables. If two terms are matrices, then `nmod_mat_mul()` is called.

```
1  main:=function()
2    sparsity:=0.03; // Means x% sparse matrix.
3    n1:=1000;
4    n2:=1200;
5    N:=16;
6    NT:= 2*N;
7    F:=GF(2);
8
9    //Initialize Matrix
10   B := Parent(ZeroMatrix(F,n1,n2))!
          RandomSparseMatrix(F,n1,n2,sparsity)
          ;
11   Y := Parent(ZeroMatrix(F,n2,N))!
          RandomSparseMatrix(F,n2,N,sparsity);
12   trB := ZeroMatrix (F,n2,n1);
13   X := ZeroMatrix (F,n2,N);
14   V_m := ZeroMatrix (F,n2,N);
15   //End Initialize Matrix
16   trB:= Transpose(B);
17
18   X,V_m:=blockLanczos(B,trB,Y,N,NT,F);
19   ZU :=findNullspace(B,X,Y,V_m,NT,F);
```

Code 5.11: project/magma/bl.txt

```
1  int main() {
2    double sparsity:=0.03;
3    int n1 = 1000;
4    int n2 = 1200;
5    int N = 16;
6    int NT = 2*N;
7    mp_limb_t limb = 2;
8    flint_rand_t rand;
9    flint_randinit(rand);
10
11   /*Initialize Matrix*/
12   nmod_mat_t B;
13   nmod_mat_t Y;
14   nmod_mat_t trB;
15   nmod_mat_t X;
16   nmod_mat_t V_m;
17   nmod_mat_init(B, n1, n2, limb);
18   nmod_mat_init(Y, n2, N, limb);
19   nmod_mat_init(trB, n2, n1, limb);
20   nmod_mat_init(X, n2, N, limb);
21   nmod_mat_init(V_m, n2, N, limb);
22   randomMatrix(B);
23   randomMatrix(Y);
24   nmod_mat_transpose(trB, B);
25
26   blockLanczos(X, V_m, B, Y, trB, limb, N);
27   findNullspace(B,X,Y,V_m,NT,limb);
28
29   return 0;
30 }
```

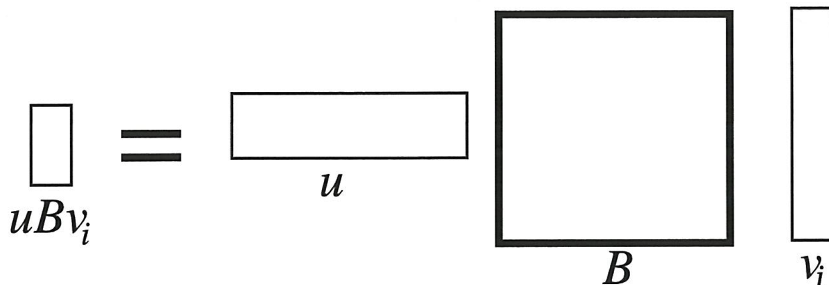
Code 5.12: project/c/bl.c

Matrices are defined in the functions because they can only be allocated inside the functions. `//Initialize` comment notifies the generator. The user should initialize matrices, variables, and fields until the `//End Initialize`. `definedVariables` list stores all variables between these two comments. The matrix are defined `//nmod_mat_t` and initialized `//nmod_mat_init()` with given sizes. Random matrix is a customizable function that creates the random matrix with given size. The user should create the main function that matches with the main function of C. Basically, Magma implementation should follow the order of structural format.

5.4 DISTRIBUTION

We developed the distributed version of block Wiedemann. We already mention that the first step of the algorithm can be done in the distributed network over TCP/IP. The application uses Berkley sockets and is developed in C language. The application has a simple protocol. The protocol requires one server at most and supports multiple clients. The server creates a thread for communication and sends a certain part of the job for each client. The clients that do the calculation with their inputs. Each client sends the result to the server when the calculation is done. The protocol requires some synchronizations. The server collects results from the clients. Clients do not communicate with each other. This approach is used by (Coppersmith, 1994), (Kaltofen and Lobo, 1999).

Figure 5.1: Our Distribution Model

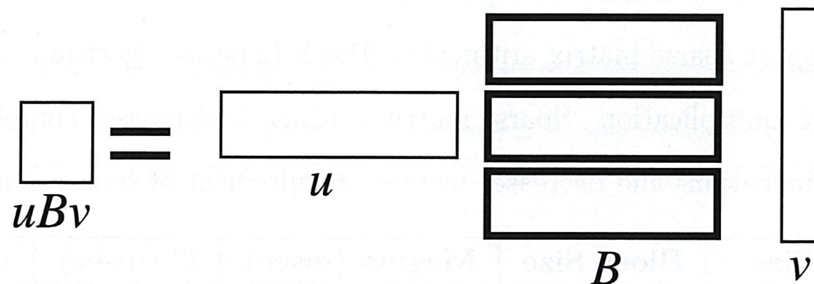


This protocol is used for application of block Wiedemann. We need to create a matrix sequence with some matrix multiplications 4.2. The server has the information of the parameters and all matrices are defined in the server. The parameters are N, n, m, L and matrices are B, u, v. N is the row and the column size of the sparse matrix B. m and n are the sizes of the block matrix. L is the length of the sequence A. The server sends these parameters to connected clients. After, the server sends the sparse matrix B and block matrix u to the connected clients. These quantities do not change during the application, so the parameters, B and u are sent only one time. The server sends columns i of the

matrix v. Each clients calculates v_i^1 with Bv_i^0. The product uv_i^k computes k the column of the A^j. Each client generates specific column of A_j for $j < L$. The server collects all subsequences and merges to create the main sequence A.

There are other distribution options. Distribution of the sparse matrix B is an alternative approach. All parameters, block matrices u and v are send to clients. The server sends rows of the sparse matrix B that represented by B_i. Each clients computes rows of the matrix v_j^{i+1} with Bv_j^i. The server collects all rows of v_{j+1}. After, all clients wait v^{i+1} from server to compute uv_i. This approach brings more data transfers and each client have to wait for v^{i+1} (Kaltofen and Lobo, 1999).

Figure 5.2: Matrix Distribution Model



Block Lanczos Implementation. Montgomery's version of block Lanczos is implemented in Magma with using 2. The generator creates C code from Magma implementation. This code is the main procedure of the algorithm.

```
1  main:=function()
2    sparsity:=0.03; // Means x% sparse matrix.
3    n1:=1000;
4    n2:=1200;
5    N:=16;
6    NT:= 2*N;
7    F:=GF(2);
8
9    //Initialize Matrix
10   B := Parent(ZeroMatrix(F,n1,n2))!RandomSparseMatrix(F,n1,n2,sparsity);
11   Y := Parent(ZeroMatrix(F,n2,N))!RandomSparseMatrix(F,n2,N,sparsity);
12   trB := ZeroMatrix (F,n2,n1);
13   X := ZeroMatrix (F,n2,N);
14   V_m := ZeroMatrix (F,n2,N);
15   //End Initialize Matrix
16   trB:= Transpose(B);
17
18   X,V_m:=blockLanczos(B,trB,Y,N,NT,F);
19   ZU :=findNullspace(B,X,Y,V_m,NT,F);
```

Code 5.13: project/magma/bl.txt

`sparsity` represents percentage of zero elements in the matrix `B`. The sparse matrix `B` is of size `n1`$\times$ `n2`. `N` is the block size and `F` is the field. `trB` is transpose

of the matrix. Transpose is necessary for calculating `V_i`. Magma supports sparse matrix arithmetic and `RandomSparseMatrix` function generates sparse matrix with given parameters.

The table 5.1 displays block Lanczos test results. The results are based on Magma and C. We test the algorithm with different matrix sizes and block sizes. 32, 64 and 128 are chosen block sizes that are equal to CW or powers of CW because nowadays, computers support 256 and 512 bit arithmetic (Advanced vector extension). The matrix has %0.0001 dense elements. Magma implementation works with sparse matrices that bellow the size 100000×100000. If the matrix size passes 100000×100000, memory problem occurs in Magma. Magma implementation has better performance than C version because FLINT does not support sparse matrix arithmetic. Block Lanczos algorithm consist of many matrix multiplication. Sparse matrix arithmetic decreases complexity of matrix multiplications and decreases memory requirement of sparse matrix.

Matrix Size	Block Size	Magma (msec)	C (msec)	number(bit)
5000×5000	32	0.060	3.740	100
5000×5000	64	0.060	5.140	100
10000×10000	32	0.250	17.520	150
10000×10000	64	0.290	48.270	150
20000×20000	32	14.740	-	150
20000×20000	64	11.680	-	150
40000×40000	64	69.440	-	150
40000×40000	128	48.470	-	150
50000×50000	128	86.740	-	150
70000×70000	128	216.740	-	150
100000×100000	256	385.560	-	200

Table 5.1: Our implementation of Block Lanczos test results

Block sizes do not make a difference for small matrices because iteration takes less time for small matrices with this sparsity. Block size increases memory requirement but decreases the number of iteration. The factorized number determines the size of the sparse matrix.

Block Wiedemann Implementation. We implement Coppersmith's version of block Wiedemann 4.1. We develop the algorithm in Magma and C version is generated from Magma implementation. This is the main function of block Wiedemann. `F` and `sparsity` are same as block Lanczos. `last` is the length of the sequence `a`.

```
1  main:= function ()
2    //Initialize Variables
3    sparsity:=0.0001; // Means x% sparse matrix.
4    N:=1400;
5    n:=8;
6    m:=8;
7    nm:= n+m;
8    F:=GF(2);
9    last := (N div n) + (N div m) +10;
10   B:=Parent(ZeroMatrix(F,N,N))!RandomSparseMatrix(F,N,N,sparsity);
11   u:=RandomMatrix(F,m,N);
12   v:=RandomMatrix(F,N,n);
13   //End Initialize Variables
14
15   a:=bw1(B,v,u,last);
16   xfg,f,fg:=BerlekampMasseyMatrix(a,F,n,m,nm,last);
17   w:=bw3 (f,B,v,N);
18
19   return 1;
20 end function;
```

Code 5.14: project/magma/bw.txt

This table 5.2 represents test results of Magma implementation of block Wiedemann. The sparse matrix has %0.0001 dense elements. We test Magma implementation with different block sizes and matrix sizes. Time of the `BW1` primarily depends the block size because length of the sequence is determined by $L = (N/n) + (N/m) + O(1)$. If L increases, then elapsed time in `BW1` decreases. In `BW2`, a determinant of the matrix polynomial is calculated and the calculation is a big part of the running time.

Matrix Size	Block Size	BW1 (msec)	BW2 (msec)	BW3 (msec)
5000 × 5000	4	0.140	0.000	0.050
5000 × 5000	8	0.150	0.000	0.050
10000 × 10000	4	5.400	0.040	0.081
10000 × 10000	8	4.980	0.160	1.010
12000 × 12000	4	19.360	0.820	16.040
12000 × 12000	8	8.890	2.440	18.580
14000 × 14000	4	29.010	8.250	53.260
14000 × 14000	4	12.310	27.470	50.150

Table 5.2: Our implementation of Block Wiedemann test results

This table 5.3 represents test results of C version of block Wiedemann. The sparse matrix has %0.0001 dense elements. We test C version with different block sizes and matrix sizes. Magma implementation of block Wiedemann is faster than C version because FLINT does not support sparse matrix arithmetic.

Matrix Size	Block Size	BW1 (msec)	BW2 (msec)	BW3 (msec)
1000 × 1000	4	0.150	0.020	0.180
1000 × 1000	8	2.780	0.010	0.070
2000 × 2000	4	20.400	1.040	6.140
2000 × 2000	8	38.280	0.090	2.010
3000 × 3000	4	106.360	2.620	38.920
3000 × 3000	8	118.890	1.840	34.520

Table 5.3: C implementation of Block Wiedemann test results

These results are based on our implementation. We developed Coppersmith's version of block Wiedemann (Coppersmith, 1994). `BW1` and `BW3` have distribution opportunities. Thomé develops subquadratic version of `BW2` (Thomé, 2002). Nowadays, block Wiedemann is more efficient than block Lanczos for large matrices.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, an automated code generator for block Lanczos and block Wiedemann algorithms are developed. The generator which was developed with Java language is capable of inputting a handcrafted Magma script that implements the two algorithms, and then, generates a ready to run C code.

The C code has a limited capacity for working in distributed environments at the moment. We aim to improve parallelization options in a future work. The current generator uses Flint library to carry out low-level matrix operations. We aim to extend Flint's code by adding special functions tailored for sparse matrices as a future work. We want to add a block matrix concept and block matrix specialized functions as a future work. We aim to add Thomé's (Thomé, 2001) subquadratic improvement for `BW2` to our implementation. We want to solve larger matrices with this improvement.

Advanced vector extension (AVX-512) is a set of instructions that do single instruction multiple data operations (SIMD). AVX-512 supports 512-bit registers and presents three operand SIMD instruction format. Also, this instruction has 512 bit and operation and 512-bit xor operation. We can use these instructions for matrix multiplication. Each row of first matrix stores 512 elements and this elements multiplied with a single instruction. We can use 512 as a block size for block Lanczos algorithm and performance of block matrix multiplications are increases. This is also left as a future work.

REFERENCES

Adj, G., Menezes, A., Oliveira, T., and Rodríguez-Henríquez, F. (2015). Computing discrete logarithms using joux's algorithm. ACM Comm. Computer Algebra, 49(2):60. 10

Bosma, W., Cannon, J., and Playoust, C. (1997). The Magma algebra system. I. The user language. J. Symbolic Comput., 24(3-4):235–265. Computational algebra and number theory (London, 1993). 37

Buhler, J. P., Jr., H. W. L., and Pomerance, C. (1993). Factoring integers with the number field sieve. In The development of the number field sieve, pages 50–94. Springer. 9

Contini, S. P. (1997). Factoring Integers with the Self-initializing Quadratic Sieve. University of Georgia. 8

Coppersmith, D. (1993). Solving linear equations over GF(2): block Lanczos algorithm. Linear Algebra and its Applications, 192:33 – 60. 19

Coppersmith, D. (1994). Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. Math. Comput., 62(205):333–350. 14, 15, 31, 32, 42, 47

Dornstetter, J. L. (1987). On the equivalence between Berlekamp's and Euclid's algorithms (corresp.). IEEE Transactions on Information Theory, 33(3):428–431. 30

Flesch, I. (2002). A new parallel approach to the block Lanczos algorithm for finding nullspaces over GF(2). Master's thesis, Utrecht University. 23

Hart, W. B. (2010). Fast library for number theory: An introduction. In Proceedings of the Third International Congress on Mathematical Software, ICMS'10, pages 88–91, Berlin, Heidelberg. Springer-Verlag. 37

Joux, A. (2014). A new index calculus algorithm with complexity L(1/4+o(1)) in small characteristic. In Lange, T., Lauter, K., and Lisoněk, P., editors, Selected Areas in Cryptography – SAC 2013, pages 355–379, Berlin, Heidelberg. Springer Berlin Heidelberg. 12

Kaltofen, E. (1995). Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. Mathematics of Computation, 64(210):777–806. 34

Kaltofen, E. and Lobo, A. A. (1999). Distributed matrix-free solution of large sparse linear systems over finite fields. Algorithmica, 24(3):331–348. 42, 43

Kaltofen, E. and Villard, G. (2005). On the complexity of computing determinants. Computational complexity, 13(3):91–130. 34

Kaltofen, E. and Yuhasz, G. (2013). On the matrix Berlekamp-Massey algorithm. ACM Transactions on Algorithms (TALG), 9(4):33. 34

Kleinjung, T., Aoki, K., Franke, J., Lenstra, A. K., Thomé, E., Bos, J. W., Gaudry, P., Kruppa, A., Montgomery, P. L., Osvik, D. A., et al. (2010). Factorization of a 768-bit RSA modulus. In Annual Cryptology Conference, pages 333–350. Springer. 1, 7, 26

Lanczos, C. (1950). An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. United States Governm. Press Office Los Angeles, CA. 14

Lenstra, A. K., Lenstra, Jr., H. W., Manasse, M. S., and Pollard, J. M. (1990). The number field sieve. In Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing, STOC '90, pages 564–572, New York, NY, USA. ACM. 9

Massey, J. (1969). Shift-register synthesis and BCH decoding. IEEE Transactions on Information Theory, 15(1):122–127. 30

Montgomery, P. L. (1994). A survey of modern integer factorization algorithms. CWI Quarterly, 7:337–366. 6

Montgomery, P. L. (1995). A Block Lanczos Algorithm for Finding Dependencies over GF(2), pages 106–120. Springer Berlin Heidelberg, Berlin, Heidelberg. 14, 19

Penninga, O. (1998). Finding column dependencies in sparse systems over f_2 by block Wiedemann. Master's thesis, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands. 34

Pomerance, C. (1985). The quadratic sieve factoring algorithm. In Beth, T., Cot, N., and Ingemarsson, I., editors, Advances in Cryptology, pages 169–182, Berlin, Heidelberg. Springer Berlin Heidelberg. 8

Pomerance, C. (1996). A tale of two sieves. Notices Amer. Math. Soc, 43:1473–1485. 1

Thomé, E. (2001). Fast computation of linear generators for matrix sequences and application to the block Wiedemann algorithm. In Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation, ISSAC '01, pages 323–331, New York, NY, USA. ACM. 15, 32, 33, 49

Thomé, E. (2002). Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. Journal of symbolic computation, 33(5):757–775. 15, 47

Thomé, E. (2016). A modified block Lanczos algorithm with fewer vectors. IACR Cryptology ePrint Archive, 2016:329. 23

Villard, G. (1997). A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. IMAG, Institut d'informatique et de mathématiques appliquées de Grenoble. 34

Wagstaff, S. S. (2002). Cryptanalysis of Number Theoretic Ciphers. CRC Press, Inc. 7, 11

Wiedemann, D. H. (1986). Solving sparse linear equations over finite fields. IEEE Transactions on Information Theory, 32(1):54–62. 14, 29

Yang, L. T., Huang, G., Feng, J., and Xu, L. (2017). Parallel gnfs algorithm integrated with parallel block wiedemann algorithm for rsa security in cloud computing. Information Sciences, 387:254 – 265. 34

Yuhasz, G. (2009). Berlekamp/Massey Algorithms for Linearly Generated Matrix Sequences. PhD thesis, North Carolina State University. 34

APPENDIX

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <gmp.h>
4  #include "flint.h"
5  #include "fmpz.h"
6  #include "fmpz_mat.h"
7  #include "nmod_mat.h"
8
9  void print_sparse_matrix(nmod_mat_t A) {
10   int i, j;
11   printf("\n");
12   printf("B:=Matrix(GF(2),n1,n2,[ ");
13   for (i = 0; i < A->r; i++) {
14     for (j = 0; j < A->c; j++) {
15       if((i==(A->r-1)) && (j==(A->c-1)) ){
16         printf("%d",(int) A->rows[i][j]);
17       }
18       else{
19         printf("%d,",(int) A->rows[i][j]);
20       }
21     }
22     printf("\n");
23   }
24   printf(" ]);");
25   printf("\n");
26 }
27
28 void print_y_matrix(nmod_mat_t A){
29   int i, j;
30   printf("\n");
31   printf("Y:=Matrix(GF(2),n2,N,[ ");
32   for (i = 0; i < A->r; i++) {
33     for (j = 0; j < A->c; j++) {
34       if((i==(A->r-1)) && (j==(A->c-1)) ){
35         printf("%d",(int) A->rows[i][j]);
36       }
37       else{
38         printf("%d,",(int) A->rows[i][j]);
39       }
40     }
41     printf("\n");
42   }
43   printf(" ]);");
44   printf("\n");
45 }
46
47 void swap(nmod_mat_t A, int i, int j) {
48   mp_limb_t* swap;
49   swap = A->rows[i];
50   A->rows[i] = A->rows[j];
51   A->rows[j] = swap;
52 }
53
54 void randomMatrix(nmod_mat_t A){
55   int i,j;
56   for (i = 0; i < A->r; i++) {
57     for (j = 0; j < A->c; j++) {
58       int num = rand()%100;
59       if(num>95){
60         A->rows[i][j]=1;
61       }
62     }
63   }
64 }
65
66 void addRow(nmod_mat_t A, int i, int j) {
67   int k;
68   int max = (int) A->c;
69   for (k = 0; k < (int) A->c; k++) {
70     A->rows[i][k] = (A->rows[i][k] + A->rows[j][k])%2;
71   }
72 }
73
74 void findWinv(nmod_mat_t SST_i, nmod_mat_t W_i_inv, nmod_mat_t TT, int N,
75     nmod_mat_t SSi_1, nmod_mat_t I, mp_limb_t F) {
76   /* Initialize variables */
77   nmod_mat_t S;
78   nmod_mat_t M;
```

```
79    /* Allocate variables */
80    int *c = (int *) malloc(sizeof(int) * N);
81    int *Q = (int *) malloc(sizeof(int) * N);
82    nmod_mat_init(S, N, N, F);
83    nmod_mat_init(M, N, 2 * N, F);
84    /* Assing matrices */
85    nmod_mat_zero(S);
86    nmod_mat_zero(M);
87    nmod_mat_concat_horizontal(M, TT, I);
88
89    int i = 0, j = 0, k = 0, k2 = 0, i2 = 0;
90
91    for (i = 0; i < N; i++) {
92      if (SSi_1->rows[i][i] == 0) {
93        c[j] = i;
94        j++;
95      }
96    }
97    for (i = 0; i < N; i++) {
98      if (SSi_1->rows[i][i] == 1) {
99        c[j] = i;
100       j++;
101     }
102   }
103   for (j = 0; j < N; j++) {
104     k = j;
105     while (M->rows[c[j]][c[j]] == 0 && k < N) {
106       if (M->rows[c[k]][c[j]] != 0) {
107         swap(M, c[j], c[k]);
108       }
109       k += 1;
110     }
111     if (M->rows[c[j]][c[j]] != 0) {
112       for (i2 = 0; i2 < N; i2++) {
113         if (i2 != c[j]) {
114           if (M->rows[i2][c[j]] == 1) {
115             addRow(M, i2, c[j]);
116           }
117         }
118       }
119       Q[k2] = c[j];
120       k2++;
121     } else {
122       k = j;
123       while (M->rows[c[j]][c[j] + N] == 0 && k < N) {
124         if (M->rows[c[k]][c[j] + N] != 0) {
125           swap(M, c[j], c[k]);
126         }
127         k += 1;
128       }
129       for (i2 = 0; i2 < N; i2++) {
130         if (i2 != c[j]) {
131           if (M->rows[i2][c[j] + N] == 1) {
132             addRow(M, i2, c[j] + N);
133           }
134         }
135       }
136       for (i2 = 0; i2 < 2 * N; i2++) {
137         M->rows[c[j]][i2] = 0;
138       }
139     }
140   }
141
142   for (i = 0; i < k2; i++) {
143     S->rows[Q[i]][Q[i]] = 1;
144   }
145   for (i = 0; i < N; i++) {
146     for (j = N; j < N + N; j++) {
147       W_i_inv->rows[i][j - N] = M->rows[i][j];
148     }
149   }
150   nmod_mat_set(SST_i,S);
151   /*nmod_mat_print_pretty(M);*/
152   /*nmod_mat_print_pretty(W_i_inv);*/
153 }
154 void blockLanczos(nmod_mat_t X2, nmod_mat_t V_m, nmod_mat_t B, nmod_mat_t Y,
155     nmod_mat_t trB, int N, mp_limb_t F) {
156   /* Initialize variables */
157   nmod_mat_t W_i_minus_1_inv;
158   nmod_mat_t W_i_minus_2_inv;
159   nmod_mat_t V_i_plus_1;
160   nmod_mat_t V_i;
161   nmod_mat_t trV_i;
162   nmod_mat_t V_i_minus_1;
163   nmod_mat_t V_i_minus_2;
164   nmod_mat_t V_0;
165   nmod_mat_t K_i_minus_1;
166   nmod_mat_t K_i;
167   nmod_mat_t Cont_i_minus_1;
168   nmod_mat_t Cont_i;
169   nmod_mat_t D_i_plus_1;
170   nmod_mat_t E_i_plus_1;
171   nmod_mat_t F_i_plus_1;
172   nmod_mat_t X;
173   nmod_mat_t SST_i_minus_1;
174   nmod_mat_t SST_i;
175   nmod_mat_t W_i_inv;
176   nmod_mat_t I;
```

```
177   nmod_mat_t BV_i;
178   nmod_mat_t trBV_i;
179   nmod_mat_t BY;
180   nmod_mat_t BTBV_i;
181   nmod_mat_t TMP;
182   nmod_mat_t TMP2;
183   nmod_mat_t TMP3;
184
185   int i = 0;
186   int n2 = B->c;
187   int n1 = B->r;
188   /* Allocate variables */
189   printf("Init started.\n");
190   nmod_mat_init(W_i_minus_1_inv, N, N, F);
191   nmod_mat_init(W_i_minus_2_inv, N, N, F);
192   nmod_mat_init(V_i_plus_1, n2, N, F);
193   nmod_mat_init(V_i, n2, N, F);
194   nmod_mat_init(trV_i, N, n2, F);
195   nmod_mat_init(V_i_minus_1, n2, N, F);
196   nmod_mat_init(V_i_minus_2, n2, N, F);
197   nmod_mat_init(V_0, n2, N, F);
198   nmod_mat_init(K_i_minus_1, N, N, F);
199   nmod_mat_init(K_i, N, N, F);
200   nmod_mat_init(Cont_i_minus_1, N, N, F);
201   nmod_mat_init(Cont_i, N, N, F);
202   nmod_mat_init(X, n2, N, F);
203   nmod_mat_init(SST_i_minus_1, N, N, F);
204   nmod_mat_init(SST_i, N, N, F);
205   nmod_mat_init(W_i_inv, N, N, F);
206   nmod_mat_init(D_i_plus_1, N, N, F);
207   nmod_mat_init(E_i_plus_1, N, N, F);
208   nmod_mat_init(F_i_plus_1, N, N, F);
209   nmod_mat_init(I, N, N, F);
210   nmod_mat_init(BV_i, n1, N, F);
211   nmod_mat_init(trBV_i, N, n1, F);
212   nmod_mat_init(BY, n1, N, F);
213   nmod_mat_init(BTBV_i, n2, N, F);
214   nmod_mat_init(TMP, N, N, F);
215   nmod_mat_init(TMP2, n1, N, F);
216   nmod_mat_init(TMP3, n2, N, F);
217   /* Assing matrices */
218
219   nmod_mat_zero(W_i_minus_1_inv);
220   nmod_mat_zero(W_i_minus_2_inv);
221   nmod_mat_zero(V_i_minus_1);
222   nmod_mat_zero(V_i_minus_2);
223   nmod_mat_zero(K_i_minus_1);
224   nmod_mat_zero(Cont_i_minus_1);
225   nmod_mat_zero(X);
226   nmod_mat_one(SST_i_minus_1);
227   nmod_mat_one(I);
228   nmod_mat_zero(BV_i);
229   nmod_mat_zero(BY);
230   nmod_mat_zero(TMP);
231   nmod_mat_zero(TMP2);
232   nmod_mat_zero(TMP3);
233
234   nmod_mat_mul(BY, B, Y);
235   nmod_mat_mul(V_i, trB, BY);
236   nmod_mat_mul(BV_i, B, V_i);
237   nmod_mat_transpose(trBV_i, BV_i);
238   nmod_mat_transpose(trV_i, V_i);
239   nmod_mat_mul(Cont_i, trBV_i, BV_i);
240   nmod_mat_set(V_0, V_i);
241
242   printf("Assign finished.\n");
243   printf("Iteration started.\n");
244   nmod_mat_print_pretty(Cont_i);
245   while (nmod_mat_is_zero(Cont_i) == 0) {
246   /* while (i<10) { */
247
248     findWinv(SST_i, W_i_inv, Cont_i, N, SST_i_minus_1, I, F);
249
250     /*printf("W_i_inv.\n");*/
251     /*nmod_mat_print_pretty(W_i_inv);*/
252
253     nmod_mat_mul(TMP, trV_i, V_0);
254     nmod_mat_mul(TMP, W_i_inv, TMP);
255     nmod_mat_mul(TMP3, V_i, TMP);
256     nmod_mat_add(X, X, TMP3);
257     printf("X.\n");
258     nmod_mat_print_pretty(X);
259
260     nmod_mat_mul(BTBV_i, trB, BV_i);
261     nmod_mat_mul(TMP2, B, BTBV_i);
262     nmod_mat_mul(K_i, trBV_i, TMP2);
263     nmod_mat_mul(K_i, K_i, SST_i);
264     nmod_mat_add(K_i, K_i, Cont_i);
265     nmod_mat_mul(D_i_plus_1, W_i_inv, K_i);
266     nmod_mat_add(D_i_plus_1, I,D_i_plus_1);
267     nmod_mat_mul(E_i_plus_1, W_i_minus_1_inv, Cont_i);
268     nmod_mat_mul(E_i_plus_1, E_i_plus_1, SST_i);
269     nmod_mat_mul(F_i_plus_1, Cont_i_minus_1, W_i_minus_1_inv);
270     nmod_mat_add(F_i_plus_1, F_i_plus_1, I);
271     nmod_mat_mul(TMP, F_i_plus_1, K_i_minus_1);
272     nmod_mat_mul(F_i_plus_1, TMP, SST_i);
273     nmod_mat_mul(F_i_plus_1, W_i_minus_2_inv, F_i_plus_1);
274     nmod_mat_mul(V_i_plus_1, BTBV_i, SST_i);
```

```
275     nmod_mat_mul(TMP3, V_i, D_i_plus_1);
276     nmod_mat_add(V_i_plus_1, V_i_plus_1, TMP3);
277     nmod_mat_mul(TMP3, V_i_minus_1, E_i_plus_1);
278     nmod_mat_add(V_i_plus_1, V_i_plus_1, TMP3);
279     nmod_mat_mul(TMP3, V_i_minus_2, F_i_plus_1);
280     nmod_mat_add(V_i_plus_1, V_i_plus_1, TMP3);
281
282     nmod_mat_set(V_i_minus_2, V_i_minus_1);
283     nmod_mat_set(V_i_minus_1, V_i);
284     nmod_mat_set(V_i, V_i_plus_1);
285     nmod_mat_set(SST_i_minus_1, SST_i);
286     nmod_mat_set(Cont_i_minus_1, Cont_i);
287     nmod_mat_set(W_i_minus_2_inv, W_i_minus_1_inv);
288     nmod_mat_set(W_i_minus_1_inv, W_i_inv);
289     nmod_mat_set(K_i_minus_1, K_i);
290     nmod_mat_mul(BV_i, B, V_i);
291     nmod_mat_transpose(trBV_i, BV_i);
292     nmod_mat_transpose(trV_i, V_i);
293     nmod_mat_mul(Cont_i, trBV_i, BV_i);
294     /*nmod_mat_transpose(trV_i, V_i);*/
295     /*nmod_mat_print_pretty(D_i_plus_1);*/
296     /*nmod_mat_print_pretty(E_i_plus_1);*/
297
298     i += 1;
299
300     printf("i = %d\n", i);
301   }
302   nmod_mat_set(X2, X);
303   nmod_mat_set(V_m,V_i);
304   printf("Iteration finished.\n");
305 }
306 void findNullspace(nmod_mat_t X, nmod_mat_t V_m, nmod_mat_t B, nmod_mat_t Y, int N ,mp_limb_t F) {
307   nmod_mat_t Z;
308   nmod_mat_t BZ;
309   nmod_mat_t trBZ;
310   nmod_mat_t XY;
311   nmod_mat_init (Z,n2,2*N,F);
312   nmod_mat_init (XY,X->r,X->c,F);
313   nmod_mat_init (BZ,B->r,Z->c,F);
314   int n2 = B->c;
315   nmod_mat_sub(XY, X, Y);
316   nmod_mat_concat_horizontal(Z,XY,V_m);
317   nmod_mat_mul(BZ, B, Z);
318   nmod_mat_transpose(trBZ, BZ);
319   nmod_mat_nullspace(X,BZ);
320 }
321
322
323 int main() {
324   double sparsity:=0.03;
325   int n1 = 1000;
326   int n2 = 1200;
327   int N = 16;
328   int NT = 2*N;
329   mp_limb_t limb = 2;
330   flint_rand_t rand;
331   flint_randinit(rand);
332
333   /*Initialize Matrix*/
334   nmod_mat_t B;
335   nmod_mat_t Y;
336   nmod_mat_t trB;
337   nmod_mat_t X;
338   nmod_mat_t V_m;
339   nmod_mat_init(B, n1, n2, limb);
340   nmod_mat_init(Y, n2, N, limb);
341   nmod_mat_init(trB, n2, n1, limb);
342   nmod_mat_init(X, n2, N, limb);
343   nmod_mat_init(V_m, n2, N, limb);
344   randomMatrix(B);
345   randomMatrix(Y);
346   nmod_mat_transpose(trB, B);
347
348   blockLanczos(X, V_m, B, Y, trB, limb, N);
349   findNullspace(B,X,Y,V_m,NT,limb);
350
351   return 0;
352 }
```

Code 6.1: project/c/bl.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <gmp.h>
4  #include <sys/time.h>
5  #include "flint.h"
6  #include "fmpz.h"
7  #include "fmpz_mat.h"
8  #include "nmod_mat.h"
9  #include "nmod_poly_mat.h"
10 #include "nmod_poly.h"
11
12 void print_sparse_matrix(nmod_mat_t A) {
13   int i, j;
14   printf("\n");
15   printf("B:=Matrix(GF(2),n1,n2,[ ");
```

```
16    for (i = 0; i < A->r; i++) {
17      for (j = 0; j < A->c; j++) {
18        if ((i == (A->r - 1)) && (j == (A->c - 1))) {
19          printf("%d", (int) A->rows[i][j]);
20        } else {
21          printf("%d,", (int) A->rows[i][j]);
22        }
23      }
24      printf("\n");
25    }
26    printf(" ]);");
27    printf("\n");
28  }
29
30  void print_y_matrix(nmod_mat_t A) {
31    int i, j;
32    printf("\n");
33    printf("Y:=Matrix(GF(2),n2,N,[ ");
34    for (i = 0; i < A->r; i++) {
35      for (j = 0; j < A->c; j++) {
36        if ((i == (A->r - 1)) && (j == (A->c - 1))) {
37          printf("%d", (int) A->rows[i][j]);
38        } else {
39          printf("%d,", (int) A->rows[i][j]);
40        }
41      }
42      printf("\n");
43    }
44    printf(" ]);");
45    printf("\n");
46  }
47
48  void print_matrix_sequence(nmod_mat_struct **A, int L) {
49    int i;
50    for (i = 0; i < L; i++) {
51      nmod_mat_print_pretty(A[i]);
52    }
53  }
54
55  void swap(nmod_mat_t A, int i, int j) {
56    mp_limb_t* swap;
57    swap = A->rows[i];
58    A->rows[i] = A->rows[j];
59    A->rows[j] = swap;
60  }
61
62  void addRow(nmod_mat_t A, int i, int j) {
63    int k;
64    int max = (int) A->c;
65    /*printf("Add row start i = %d, j = %d, max = %d. \n", i, j, max);*/
66    for (k = 0; k < max; k++) {
67      A->rows[i][k] = (A->rows[i][k] + A->rows[j][k]) % 2;
68    }
69    /*printf("Add row end.\n");*/
70  }
71
72  void swapColumn(nmod_mat_t A, int c1, int c2) {
73    int i;
74    int j;
75    for (i = 0; i < A->r; i++) {
76
77      mp_limb_t tmp = A->rows[i][c1];
78      A->rows[i][c1] = A->rows[i][c2];
79      A->rows[i][c2] = tmp;
80    }
81  }
82
83  void swapElement(int *arr, int i, int j) {
84    int tmp;
85    tmp = arr[i];
86    arr[i] = arr[j];
87    arr[j] = tmp;
88  }
89
90  void removeArray(int *arr, int e, int size) {
91    int i;
92    int j;
93    for (j = 0; j < size; j++) {
94      if (arr[j] == e) {
95        break;
96      }
97    }
98    for (i = j; i < size - 1; i++) {
99      arr[i] = arr[i + 1];
100   }
101 }
102
103 void initMatrixSequence(nmod_mat_struct **A, int r, int c, int length,
104     mp_limb_t F) {
105   int i;
106   for (i = 0; i < length; i++) {
107     A[i] = (nmod_mat_struct *) malloc(sizeof(nmod_mat_t[1]));
108     nmod_mat_init(A[i], r, c, F);
109   }
110 }
111
112 void clearMatrixSequence(nmod_mat_struct **A, int r, int c, int length,
113     mp_limb_t F) {
```

```
114   int i;
115   for (i = 0; i < length; i++) {
116     nmod_mat_clear(A[i]);
117   }
118 }
119
120 void addColumn(nmod_mat_t A, int i, int j) {
121   int k;
122   int max = (int) A->r;
123   /*printf("Add row start i = %d, j = %d, max = %d. \n", i, j, max);*/
124   for (k = 0; k < max; k++) {
125     A->rows[k][i] = (A->rows[k][i] + A->rows[k][j]) % 2;
126   }
127   /*printf("Add row end.\n");*/
128 }
129
130 void polyToMatrix(nmod_poly_t frev, nmod_mat_struct **f, int len, mp_limb_t F) {
131
132   int i = 0;
133   int j = 0;
134   int k = 0;
135   int n = f[0]->c;
136   int m = f[0]->r;
137   nmod_poly_mat_t pol;
138
139   nmod_poly_t det;
140   nmod_poly_t ndet;
141   nmod_poly_mat_init(pol, m, n, F);
142   nmod_poly_init(det, F);
143   nmod_poly_init(ndet, F);
144   for (i = 0; i < m; i++) {
145     for (j = 0; j < n; j++) {
146       nmod_poly_t tmp;
147       nmod_poly_init(tmp, F);
148       for (k = 0; k < len; k++) {
149         nmod_poly_set_coeff_ui(tmp, k, f[k]->rows[i][j]);
150       }
151       pol->rows[i][j].coeffs = tmp->coeffs;
152       pol->rows[i][j].length = tmp->length;
153       /*nmod_poly_clear(tmp);*/
154     }
155   }
156   nmod_poly_set_coeff_ui(ndet, 1, 1);
157   nmod_poly_mat_det(det, pol);
158   i = 0;
159   /*
160   while (i < 10) {
161     nmod_poly_div(det, det, ndet);
162     i++;
163   }*/
164   /*printf("%u",det->coeffs[0]);*/
165 #ifdef DEBUG
166   nmod_poly_mat_print(pol, "x");
167   nmod_poly_print_pretty(det, "x");
168 #endif
169   nmod_poly_print_pretty(det, "x");
170   nmod_poly_set(frev, det);
171
172 }
173
174 void AuxGauss(nmod_mat_t P, nmod_mat_t DELTA, int nm, int *d) {
175   int N = DELTA->r;
176   nmod_mat_one(P);
177   int *g = (int *) malloc(sizeof(int) * N);
178   int i;
179   int j;
180   int k = 0;
181   int l;
182   int gSize = N;
183   for (i = 0; i < N; i++) {
184     g[i] = i;
185   }
186
187   for (i = 0; i < N; i++) {
188     int *pi = (int *) malloc(sizeof(int) * N);
189     k = 0;
190     for (j = 0; j < gSize; j++) {
191       if (DELTA->rows[i][g[j]] != 0) {
192         pi[k] = g[j];
193         k++;
194       }
195     }
196     pi[k] = N + i;
197     k++;
198     l = N + i;
199
200     for (j = 0; j < k; j++) {
201       if (d[pi[j]] == d[l]) {
202         if (pi[j] < l) {
203           l = pi[j];
204         }
205       } else if (d[pi[j]] < d[l]) {
206         l = pi[j];
207       }
208     }
209     /*printf("%d \n", l);*/
210
211     removeArray(pi, l, k);
```

```
212     k--;
213 #ifdef DEBUG
214     for (j = 0; j < k; j++) {
215       printf(" %d ,", pi[j]);
216     }
217     printf("\n");
218 #endif
219     for (j = 0; j < k; j++) {
220       if (l == N + i) {
221 #ifdef DEBUG
222         printf("if 1\n");
223 #endif
224         addColumn(DELTA, pi[j], N + i);
225         addColumn(P, pi[j], N + i);
226       }
227       if (l < N + i) {
228         if (pi[j] < N + i) {
229 #ifdef DEBUG
230           printf("if 2\n");
231 #endif
232           addColumn(DELTA, pi[j], l);
233           addColumn(P, pi[j], l);
234         }
235         if (pi[j] == N + i) {
236           if (DELTA->rows[i][N + i] != 0) {
237 #ifdef DEBUG
238             printf("if 3\n");
239 #endif
240             addColumn(DELTA, N + i, l);
241             addColumn(P, N + i, l);
242             swapColumn(DELTA, l, N + i);
243             swapColumn(P, l, N + i);
244             swapElement(d, l, N + i);
245           } else {
246 #ifdef DEBUG
247             printf("if 4\n");
248 #endif
249             addColumn(DELTA, N + i, l);
250             addColumn(P, N + i, l);
251             swapElement(d, l, N + i);
252             removeArray(g, l, gSize);
253             gSize--;
254           }
255         }
256
257       }
258
259     }
260
261   }
262
263 }
264
265 void bw1(nmod_mat_struct **A, nmod_mat_t B, nmod_mat_t v, nmod_mat_t u, int L,
266     mp_limb_t F) {
267   int m = u->r;
268   int n = v->c;
269   int N = B->r;
270   int i;
271   nmod_mat_t t;
272   nmod_mat_t b;
273   nmod_mat_init(t, N, n, F);
274   nmod_mat_init(b, N, n, F);
275   nmod_mat_mul(b, B, v);
276   nmod_mat_mul(t, B, b);
277   for (i = 0; i < L; i++) {
278     nmod_mat_mul(A[i], u, t);
279     nmod_mat_set(b, t);
280     nmod_mat_mul(t, B, b);
281
282   }
283 }
284
285 void bw2(nmod_poly_t frev, nmod_mat_struct **A, nmod_mat_t B, int n, int m,
286     int L, mp_limb_t F) {
287   int NM = n + m;
288   /*int beta = 1;*/
289   int mu = 0;
290   int sigma = 0;
291   int i;
292   int t = n / m;
293   int k = 0;
294   int j = 0;
295   int f_index = 0;
296   nmod_mat_t DELTA;
297   nmod_mat_t Z;
298   nmod_mat_t I;
299   nmod_mat_t fi;
300   nmod_mat_t TMP;
301   nmod_mat_t TMP2;
302   nmod_mat_t P;
303   nmod_mat_t PI;
304   nmod_mat_t ZERO;
305   int N = B->r;
306   nmod_mat_init(DELTA, n, NM, F);
307   nmod_mat_init(TMP, n, NM, F);
308   nmod_mat_init(Z, m, n, F);
309   nmod_mat_init(I, m, m, F);
```

```
310   nmod_mat_init(fi, m, NM, F);
311   nmod_mat_init(TMP2, m, NM, F);
312   nmod_mat_init(P, NM, NM, F);
313   nmod_mat_init(PI, NM, NM, F);
314   nmod_mat_init(ZERO, NM, NM, F);
315   nmod_mat_struct **f = (nmod_mat_struct**) malloc(sizeof(nmod_mat_t[1]) * L);
316   initMatrixSequence(f, M, NM, L, F);
317   int *d = (int *) malloc(sizeof(int) * nm);
318   nmod_mat_one(I);
319   nmod_mat_one(PI);
320
321   nmod_mat_concat_horizontal(f[f_index], I, Z);
322   /*nmod_mat_print_pretty(f[f_index]);*/
323   f_index = 1;
324   for (i = 0; i < NM; ++i) {
325     if (i < n) {
326       d[i] = 1;
327     } else {
328       d[i] = 0;
329     }
330   }
331
332   while (t < L) {
333     nmod_mat_zero(DELTA);
334
335     for (i = 0; i < t; i++) {
336       nmod_mat_mul(TMP, A[i], f[t - i - 1]);
337       nmod_mat_add(DELTA, DELTA, TMP);
338     }
339 #ifdef DEBUG
340     nmod_mat_print_pretty(DELTA);
341 #endif
342     AuxGauss(P, DELTA, NM, d);
343 #ifdef DEBUG
344     nmod_mat_print_pretty(P);
345     for (i = 0; i < NM; ++i) {
346       printf("%d", d[i]);
347     }
348     printf("\n");
349 #endif
350
351     for (i = 0; i < m; i++) {
352       d[n + i] = d[n + i] + 1;
353     }
354
355     sigma = 0;
356 #ifdef DEBUG
357     printf("Apply P \n");
358 #endif;
359     for (i = 0; i < f_index; i++) {
360       nmod_mat_mul(TMP2, f[i], P);
361       nmod_mat_set(f[i], TMP2);
362     }
363
364     /*f[f_index] = ZERO;*/
365     f_index++;
366 #ifdef DEBUG
367     printf("Shifting f \n");
368 #endif;
369     for (k = f_index; k > 0; k--) {
370       for (i = 0; i < n; i++) {
371         for (j = n; j < NM; j++) {
372           f[k]->rows[i][j] = f[k - 1]->rows[i][j];
373         }
374       }
375     }
376     for (i = 0; i < n; i++) {
377       for (j = n; j < NM; j++) {
378         f[0]->rows[i][j] = 0;
379       }
380     }
381 #ifdef DEBUG
382     printf("%d f\n", t);
383     print_matrix_sequence(f, f_index);
384 #endif;
385
386     if (nmod_mat_equal(P, PI)) {
387       printf("Iteration end %d\n", t);
388       break;
389     }
390     t++;
391   }
392
393   int max = 0;
394   for (i = 0; i < n; i++) {
395     if (d[i] > max) {
396       max = d[i];
397     }
398   }
399   nmod_mat_struct **res = (nmod_mat_struct**) malloc(
400       sizeof(nmod_mat_t[1]) * (max + 1));
401   initMatrixSequence(res, n, n, max + 1, F);
402 #ifdef DEBUG
403   printf("max %d\n", max);
404   print_matrix_sequence(f, f_index);
405   printf("\n");
406 #endif;
407
```

```
408   int sizer = 0;
409   for (i = 0; i < max; i++) {
410     for (j = 0; j < n; j++) {
411       for (k = 0; k < n; k++) {
412         if (d[j] > i) {
413           res[d[j] - i]->rows[k][j] = f[i]->rows[k][j];
414           sizer++;
415         }
416       }
417     }
418   }
419 #ifdef DEBUG
420   print_matrix_sequence(res, max + 1);
421 #endif
422   polyToMatrix(frev, res, max + 1, F);
423 
424 }
425 void bw3(nmod_poly_t f, nmod_mat_t B, nmod_mat_t v, mp_limb_t F) {
426   int N = B->r;
427   int n = v->c;
428   int i = 0;
429   nmod_mat_t eval;
430   nmod_mat_t t;
431   nmod_mat_t Bi;
432   nmod_mat_t s;
433   nmod_mat_init(eval, N, N, F);
434   nmod_mat_init(t, N, n, F);
435   nmod_mat_init(s, N, n, F);
436   nmod_mat_init(Bi, N, N, F);
437   nmod_poly_evaluate_mat(eval, f, B);
438   nmod_mat_mul(t, eval, v);
439   nmod_mat_one(Bi);
440   /*nmod_mat_print_pretty(eval);*/
441   while (i < N) {
442     printf("%d\n",i);
443     nmod_mat_mul(s, Bi, t);
444     if (nmod_mat_is_zero(s) == 1) {
445       printf("sadsadsadasdas %d\n", i);
446       break;
447     }
448     nmod_mat_mul(eval, Bi, B);
449     nmod_mat_set(Bi, eval);
450     i++;
451   }
452 }
453 
454 int main() {
455   nmod_mat_t B;
456   nmod_mat_t u;
457   nmod_mat_t v;
458   nmod_poly_t f;
459   int N = 5000;
460   int n = 8;
461   int m = 8;
462   int L = (N / n) + (N / m) + 10;
463   mp_limb_t limb = 2;
464   flint_rand_t rand;
465   flint_randinit(rand);
466   nmod_mat_init(B, N, N, limb);
467   nmod_mat_init(u, m, N, limb);
468   nmod_mat_init(v, N, n, limb);
469   nmod_poly_init(f, limb);
470   nmod_mat_randtest(B, rand);
471   nmod_mat_randfull(u, rand);
472   nmod_mat_randfull(v, rand);
473 
474   nmod_mat_struct **A = (nmod_mat_struct**) malloc(sizeof(nmod_mat_t[1]) * L);
475 
476   initMatrixSequence(A, m, n, L, limb);
477   /*nmod_mat_print_pretty(B);*/
478 
479 #ifdef DEBUG
480   print_sparse_matrix(B);
481   print_y_matrix(u);
482   print_y_matrix(v);
483 #endif
484   printf("BW 1 starts\n");
485   bw1(A, B, v, u, L, limb);
486   /*timersub(t_end, t_start, t_diff);*/
487   print_matrix_sequence (A,L);
488   /*nmod_mat_print_pretty(A[0]);*/
489   printf("BW 2 starts\n");
490   bw2(f, A, B, n, m, L, limb);
491   printf("BW 3 starts\n");
492   bw3(f, B, v, limb);
493   /*int res = nmod_mat_equal(X2, Y);*/
494   /*printf("res %d",res);*/
495   /*nmod_mat_print_pretty(X2);*/
496   return 0;
497 }
```

Code 6.2: project/c/bw.c

```
1 package com.generator.source;
2 
3 import java.util.ArrayList;
```

```
4
5  public class Function {
6
7    private String name;
8    private ArrayList<Variable> inputs = new ArrayList<Variable>();
9    private ArrayList<Variable> definedVariables = new ArrayList<Variable>();
10
11   public String getName() {
12     return name;
13   }
14   public void setName(String name) {
15     this.name = name;
16   }
17   public ArrayList<Variable> getInputs() {
18     return inputs;
19   }
20   public void setInputs(ArrayList<Variable> inputs) {
21     this.inputs = inputs;
22   }
23   public ArrayList<Variable> getDefinedVariables() {
24     return definedVariables;
25   }
26   public void setDefinedVariables(ArrayList<Variable> definedVariables) {
27     this.definedVariables = definedVariables;
28   }
29 }
```

Code 6.3: project/Generator/src/com/generator/source/Function.java

```
1  package com.generator.source;
2
3  import java.util.ArrayList;
4
5  import com.generator.io.FileOperations;
6
7  public class CodeGenerator {
8
9    private String sourceFile;
10   private String outFile;
11
12   private FileOperations op = new FileOperations();
13
14   private ArrayList<Function> functionList = new ArrayList<Function>();
15
16   public CodeGenerator(String source, String out) {
17     this.sourceFile = source;
18     this.outFile = out;
19   }
20
21   public boolean generateCode() {
22     op.readFile(sourceFile);
23     String tab = "";
24     int functionIndex = 0;
25     Function gfunc = new Function();
26     gfunc.setName("Global");
27     functionList.add(gfunc);
28     addHeaders();
29     addAxuliaryFunctions();
30     for (int i = 0; i < op.readerList.size(); i++) {
31       i = handleInitializeVariables(op.readerList, op.writerList, i, functionIndex, tab);
32       i = handleMagmaFunctions(op.readerList, op.writerList, i, tab);
33       String line = op.readerList.get(i);
34       if (line.contains("][")) {
35         int index = line.indexOf("[");
36         String out = line.substring(0, index) + "->rows" + line.substring(index);
37         op.readerList.set(i, out);
38         line = out;
39       }
40       if (line.contains(":=function")) {
41         String functionName = op.readerList.get(i).substring(0, op.readerList.get(i).indexOf(":"));
42         op.writerList.add("void " + functionName + "(" + getFunctionParameters(op.readerList.get(i)) + ")
           {");
43         tab += "\t";
44         Function func = new Function();
45         func.setName(functionName);
46         functionList.add(func);
47       }
48
49       else if (line.contains("Int Assign")) {
50         String out = line.replace(":", "").trim();
51         op.writerList.add(tab + out);
52       } else if ((line.contains("while")) && (line.contains("do"))) {
53         String out = multiWhileLoop(line);
54         op.writerList.add(tab + out);
55         tab += "\t";
56       } else if ((line.contains("for")) && (line.contains("do"))) {
57         String out = defineForLoop(line);
58         op.writerList.add(tab + out);
59         tab += "\t";
60       } else if ((line.contains("if")) && (line.contains("then"))) {
61         String out = multiIfCondition(line);
62         op.writerList.add(tab + out);
63         tab += "\t";
64       } else if (line.contains("else")) {
65         tab = tab.substring(0, tab.length() - 1);
66         op.writerList.add(tab + "}");
```

```
67          op.writerList.add(tab + "else {");
68          tab += "\t";
69        } else if (line.contains("+:=")) {
70          String out = line.replace(":=", "=").trim();
71          op.writerList.add(tab + out);
72        }
73
74        else if (line.contains("return")) {
75
76        } else if (isFunctionCall(line)) {
77          System.out.println(line);
78        } else if (line.contains("end function")) {
79          tab = tab.substring(0, tab.length() - 1);
80          op.writerList.add(tab + "}");
81          functionIndex++;
82        } else if (line.contains("end while")) {
83          tab = tab.substring(0, tab.length() - 1);
84          op.writerList.add(tab + "}");
85        } else if (line.contains("end for")) {
86          tab = tab.substring(0, tab.length() - 1);
87          op.writerList.add(tab + "}");
88        } else if (line.contains("end if")) {
89          tab = tab.substring(0, tab.length() - 1);
90          op.writerList.add(tab + "}");
91        } else {
92          // line = line.replace(":", "");
93          System.out.println("****" + line);
94          assignVariables(i, functionIndex);
95        }
96      }
97      op.writeFile(outFile);
98
99      for (int i = 0; i < functionList.size(); i++) {
100       System.out.println(functionList.get(i).getDefinedVariables().size());
101     }
102     return true;
103   }
104
105   private int handleInitializeVariables(ArrayList<String> inList, ArrayList<String> outList, int index,
106       int functionIndex, String tab) {
107     int i = index;
108
109     if (inList.get(index).contains("//Initialize Variables")) {
110       ArrayList<String> init = new ArrayList<String>();
111       ArrayList<String> alloc = new ArrayList<String>();
112       ArrayList<String> assign = new ArrayList<String>();
113       ArrayList<Variable> varList = new ArrayList<Variable>();
114       while (!inList.get(i).contains("//End Initialize Variables")) {
115         String line = inList.get(i);
116         if (line.trim().length() != 0) {
117           if (line.contains("Matrix")) {
118             String variable = line.substring(0, line.indexOf(":")).trim();
119             init.add(tab + "nmod_mat_t " + variable + ";");
120             String parameters = getParameters(line);
121             if (!parameters.contains(",")) {
122               parameters = parameters + "," + parameters;
123             }
124             alloc.add(tab + "nmod_mat_init(" + variable + "," + parameters + ",F);");
125             assign.add(tab + defineMatrix(line, variable));
126             Variable var = new Variable("Matrix", variable);
127             varList.add(var);
128           } else if (line.contains("[]") && line.contains("type") && line.contains("size")) {
129             String name = line.substring(0, line.indexOf(":")).trim();
130             String type = line.substring(line.indexOf("type:") + 5, line.indexOf("size")).trim();
131             String size = line.substring(line.indexOf("size:") + 5, line.length()).trim();
132             outList.add(tab + type + "*" + name + "= (" + type + "*" + ") malloc (sizeof(" + type + ") * "
133                 + size + ");");
134             Variable var = new Variable("Array", name);
135             varList.add(var);
136           } else if (line.contains("Nrows")) {
137             line = line.replace(":=", "= ");
138             String var = line.substring(0, line.indexOf("Nrows")).trim();
139             String name = line.substring(line.indexOf("(") + 1, line.indexOf(")")).trim();
140             outList.add(tab + "int " + var + name + "-> r");
141           } else if (line.contains("Ncols")) {
142             line = line.replace(":=", "= ");
143             String var = line.substring(0, line.indexOf("Ncols")).trim();
144             String name = line.substring(line.indexOf("(") + 1, line.indexOf(")")).trim();
145             outList.add(tab + "int " + var + name + "-> c");
146           } else if (line.contains("GF")) {
147             String var = line.substring(0, line.indexOf(":")).trim();
148             String val = line.substring(line.indexOf("(") + 1, line.indexOf(")"));
149             outList.add(tab + "mp_limb_t " + var + " = " + val + ";");
150
151           } else {
152             line = line.replace(":", "");
153             line = "int " + line.trim();
154             outList.add(tab + line);
155             Variable var = new Variable("Int", line);
156             varList.add(var);
157           }
158           i++;
159         }
160
161       }
162
163       functionList.get(functionIndex).setDefinedVariables(varList);
164       for (int j = 0; j < init.size(); j++) {
```

```
165         outList.add(init.get(j));
166       }
167       op.writerList.add("/* Allocate variables */");
168       for (int j = 0; j < alloc.size(); j++) {
169         outList.add(alloc.get(j));
170       }
171       op.writerList.add("/* Assing matrices */");
172       for (int j = 0; j < assign.size(); j++) {
173         outList.add(assign.get(j));
174       }
175     }
176
177     return i;
178   }
179
180   private int handleMagmaFunctions(ArrayList<String> inList, ArrayList<String> outList, int index, String
          tab) {
181     int i = index;
182     String line = inList.get(i);
183     if (line.contains("SwapRows") && line.contains(";")) {
184       op.writerList.add(tab + "swap" + "(" + getFunctionCallParameters(line) + ");");
185       i = i + 1;
186     } else if (line.contains("Concat")) {
187       String out = concatanete(line);
188       outList.add(tab + out);
189       i = i + 3;
190     } else if (line.contains("Row Add")) {
191       outList.add(tab + "addRow" + "(" + addRow(line) + ");");
192       i = i + 1;
193     } else if (line.contains("Row Assign")) {
194       outList.add(tab + "assingRow" + "(" + assignRow(line) + ");");
195       i = i + 1;
196     } else if (line.contains("Append")) {
197       append(line, tab);
198       i = i + 1;
199     } else if (line.contains("ColumnSubmatrix")) {
200       outList.add(tab + columnSubmatrix(line,tab));
201       i = i + 1;
202     } else if (line.contains("//Remove")) {
203       i = i+1;
204     } else if (line.contains("SwapColumns") && line.contains(";")) {
205       i = i+1;
206     }
207
208     return i;
209   }
210
211   private String defineMatrix(String str, String name) {
212     String returnStr = "";
213     if (str.contains("IdentityMatrix")) {
214       returnStr = "nmod_mat_one (" + name + ");";
215     } else if (str.contains("ZeroMatrix")) {
216       returnStr = "nmod_mat_zero (" + name + ");";
217     } else if (str.contains("RandomMatrix")) {
218
219     }
220     return returnStr;
221   }
222
223   private String defineForLoop(String str) {
224     String returnStr = "for (";
225     String iterator = str.substring(str.indexOf("for") + 3, str.indexOf(":")).trim();
226     String start = str.substring(str.indexOf("=") + 1, str.indexOf("to")).trim();
227     returnStr += iterator + "=" + start + ";";
228     if (str.contains("by")) {
229
230     } else {
231       String end = str.substring(str.indexOf("to") + 2, str.indexOf("do")).trim();
232       returnStr += iterator + "<" + end + "; i++ ){";
233     }
234     return returnStr;
235   }
236
237   private String multiWhileLoop(String str) {
238     String out = "while ( ";
239     str = str.replace("while", "");
240     str = str.replace("do", "");
241     if (str.contains("and")) {
242       String[] arr = str.split("and");
243       for (int i = 0; i < arr.length; i++) {
244         if (i != 0) {
245           out += " &&" + defineWhileLoop(arr[i]);
246         } else {
247           out += defineWhileLoop(arr[i]);
248         }
249
250       }
251     } else {
252       out += defineWhileLoop(str);
253     }
254     out = out + "){";
255     return out;
256   }
257
258   private String multiIfCondition(String str) {
259     String out = "if( ";
260     str = str.replace("if", "");
261     str = str.replace("then", "");
```

```
262     String[] arr = str.split(" ");
263     String compare = loopCompare(arr[2]);
264     out += arr[1] + compare + arr[3];
265     out += "){";
266     return out;
267   }
268
269   private String addRow(String str) {
270     String out = "";
271     String matrix = str.substring(0, str.indexOf("[")).trim();
272     String row1 = str.substring(str.indexOf("[") + 1, str.indexOf("]")).trim();
273     String row2 = str.substring(str.lastIndexOf(matrix + "[") + 2, str.lastIndexOf("]")).trim();
274     out += matrix + "," + row1 + "," + row2;
275     // System.out.println(out);
276     return out;
277   }
278
279   private String assignRow(String str) {
280     String matrix = str.substring(0, str.indexOf("[")).trim();
281     String row1 = str.substring(str.indexOf("[") + 1, str.lastIndexOf("]")).trim();
282     String val = str.substring(str.indexOf("=") + 1, str.indexOf(";"));
283     String out = matrix + "," + row1 + "," + val;
284     return out;
285   }
286
287   private String concatanete(String str) {
288     String out = "";
289     String res = str.substring(str.indexOf("Concat") + 6, str.indexOf("=")).trim();
290     String matrix1 = str.substring(str.indexOf("[") + 1, str.indexOf("|")).trim();
291     String matrix2 = str.substring(str.indexOf("|") + 1, str.indexOf("]")).trim();
292     out = "nmod_mat_concat_horizontal(" + res + "," + matrix1 + "," + matrix2 + ");";
293     return out;
294   }
295
296   private void append(String str, String tab) {
297     str = str.replace("~", "");
298     String arr = str.substring(str.indexOf("(") + 1, str.indexOf(","));
299     String value = str.substring(str.indexOf(",") + 1, str.indexOf(")"));
300     op.writerList.add(tab + arr + "[j] = " + value + ";");
301     op.writerList.add(tab + "j++;");
302
303   }
304
305   private String columnSubmatrix(String str, String tab) {
306     String destMatrix = str.substring(0, str.indexOf(":")).trim();
307     String sourceMatrix = str.substring(str.indexOf("(") + 1, str.indexOf(",")).trim();
308     String row = str.substring(str.indexOf(",") + 1, str.lastIndexOf(",")).trim();
309     String col = str.substring(str.lastIndexOf(",") + 1, str.indexOf(")")).trim();
310     String out = "columnSubMatrix("+ destMatrix +","+ sourceMatrix +","+row+","+col+");";
311     return out;
312   }
313
314   private String defineWhileLoop(String str) {
315     String out = " ";
316     if (str.contains("IsZero")) {
317       String name = str.substring(str.indexOf("(") + 1, str.indexOf(")"));
318       out += "nmod_mat_is_zero(" + name + ")";
319       if (str.contains("eq")) {
320         out += "==";
321       } else if (str.contains("ne")) {
322         out += "!=";
323       }
324       if (str.contains("true")) {
325         out += " 1";
326       } else if (str.contains("false")) {
327         out += "0";
328       }
329     } else {
330       String[] arr = str.split(" ");
331       String compare = loopCompare(arr[2]);
332       out += arr[1] + compare + arr[3];
333     }
334
335     return out;
336   }
337
338   private String loopCompare(String str) {
339     String returnStr = "";
340     if (str.equalsIgnoreCase("ne")) {
341       returnStr = "!=";
342     } else if (str.equalsIgnoreCase("eq")) {
343       returnStr = "==";
344     } else if (str.equalsIgnoreCase("le")) {
345       returnStr = "<=";
346     } else if (str.equalsIgnoreCase("lt")) {
347       returnStr = "<";
348     } else if (str.equalsIgnoreCase("ge")) {
349       returnStr = ">=";
350     } else if (str.equalsIgnoreCase("gt")) {
351       returnStr = ">";
352     }
353
354     return returnStr;
355   }
356
357   private void addHeaders() {
358     String[] headers = { "#include <stdlib.h>", "#include <stdio.h>", "#include <gmp.h>", "#include \"flint.
          h\"",
```

```
359         "#include \"fmpz.h\"", "#include \"fmpz_mat.h\"", "#include \"nmod_mat.h\"\n" };
360     for (int i = 0; i < headers.length; i++) {
361       op.writerList.add(headers[i]);
362     }
363   }
364
365   private void addAxuliaryFunctions() {
366     op.writerList.add("void swap(nmod_mat_t A,int i, int j){");
367     op.writerList.add("\t mp_limb_t* swap;");
368     op.writerList.add("\t swap = A->rows[i];");
369     op.writerList.add("\t A->rows[i]=A->rows[j];");
370     op.writerList.add("\t A->rows[j] = swap;");
371     op.writerList.add("}\n");
372
373     op.writerList.add("void addRow(nmod_mat_t A,int i, int j){");
374     op.writerList.add("\t int k;");
375     op.writerList.add("\t int max = (int) A->c;");
376     op.writerList.add("\t for (k = 0; k < max; k++) {");
377     op.writerList.add("\t\t A->rows[i][k] = A->rows[i][k] + A->rows[j][k];");
378     op.writerList.add("\t}");
379     op.writerList.add("}\n");
380
381     op.writerList.add("\t void assignRow(nmod_mat_t A, int i, int val){");
382     op.writerList.add("\t int k;");
383     op.writerList.add("\t int max = (int) A->c;");
384     op.writerList.add("\t for (k = 0; k < max; k++) {");
385     op.writerList.add("\t\t A->rows[i][k] = val;");
386     op.writerList.add("\t}");
387     op.writerList.add("}\n");
388
389     op.writerList.add("void columnSubMatrix (nmod_mat_t dest, nmod_mat_t sour,int r, int c) {");
390     op.writerList.add("\t int max = sour->c;");
391     op.writerList.add("\t int i;");
392     op.writerList.add("\t int j;");
393     op.writerList.add("\t for (i = 0; i < r; i++) {");
394     op.writerList.add("\t\t for (j = c; j < max; j++) {");
395     op.writerList.add("\t\t\t dest->rows[i][j - c] = sour->rows[i][j];");
396     op.writerList.add("\t\t}");
397     op.writerList.add("\t}");
398     op.writerList.add("}\n");
399
400   }
401
402   private String getParameters(String str) {
403     int start = str.indexOf("(F", 0);
404     int end = str.indexOf(")", 0);
405     String parameters = str.substring(start + 3, end);
406     return parameters;
407   }
408
409   private String getFunctionParameters(String str) {
410     if (str.contains("//")) {
411       String returnStr = str.substring(str.indexOf("//") + 2);
412       String[] arr = returnStr.split(",");
413       returnStr = returnStr.replace("Matrix", "nmod_mat_t");
414       returnStr = returnStr.replace("field", "mp_limb_t");
415       returnStr = returnStr.replace("int", "int");
416       return returnStr;
417     } else {
418       return str;
419     }
420   }
421
422   private String getFunctionCallParameters(String str) {
423     str = str.replace("~", "");
424     String returnStr = "";
425     if (str.contains(":")) {
426       returnStr = str.substring(0, str.indexOf(":")).trim();
427       returnStr += "," + str.substring(str.indexOf("(") + 1, str.indexOf(")")).trim();
428     } else {
429       returnStr += str.substring(str.indexOf("(") + 1, str.indexOf(")")).trim();
430     }
431
432     // System.out.println(returnStr);
433     return returnStr;
434   }
435
436   private int initializeArrays(ArrayList<String> list, int i, String tab) {
437
438     while (!list.get(i).contains("//End Initialize Array")) {
439       if (list.get(i).contains("[]")) {
440         String name = list.get(i).substring(0, list.get(i).indexOf(":")).trim();
441         String type = list.get(i).substring(list.get(i).indexOf("type:") + 5, list.get(i).indexOf("size"))
442             .trim();
443         String size = list.get(i).substring(list.get(i).indexOf("size:") + 5, list.get(i).length()).trim();
444         op.writerList.add(tab + type + "*" + name + "= (" + type + "*" + ") malloc (sizeof(" + type + ") * "
445             + size + ");");
446       }
447       i++;
448     }
449     return i;
450   }
451
452   private void assignVariables(int i, int funcIndex) {
453     String str = op.readerList.get(i);
454     if (!str.contains("//") && str.trim().length() > 0) {
455       if (str.contains("*")) {
456         String res = str.substring(0, str.indexOf(":")).trim();
```

```
457            String num1 = str.substring(str.indexOf("=") + 1, str.indexOf("*")).trim();
458            String num2 = str.substring(str.indexOf("*") + 1, str.indexOf(";")).trim();
459            if (isMatrix(res, funcIndex) && isMatrix(num1, funcIndex) && isMatrix(num2, funcIndex)) {
460              op.writerList.add("\t nmod_mat_mul (" + res + "," + num1 + "," + num2 + ");");
461            }
462
463          } else if (str.contains("Transpose")) {
464            String res = str.substring(0, str.indexOf(":")).trim();
465            String num = str.substring(str.indexOf("(") + 1, str.indexOf(")")).trim();
466            op.writerList.add("\t nmod_mat_transpose(" + res + "," + num + ");");
467
468          } else if (str.contains("+")) {
469            String res = str.substring(0, str.indexOf(":")).trim();
470            String num1 = str.substring(str.indexOf("=") + 1, str.indexOf("+")).trim();
471            String num2 = str.substring(str.indexOf("+") + 1, str.indexOf(";")).trim();
472            if (isMatrix(res, funcIndex) && isMatrix(num1, funcIndex) && isMatrix(num2, funcIndex)) {
473              op.writerList.add("\t nmod_mat_add (" + res + "," + num1 + "," + num2 + ");");
474            }
475          }
476
477          else {
478            String res = str.substring(0, str.indexOf(":")).trim();
479            String num = str.substring(str.indexOf("=") + 1, str.indexOf(";")).trim();
480            if (isMatrix(res, funcIndex) && isMatrix(num, funcIndex)) {
481              op.writerList.add("\t nmod_mat_set(" + res + "," + num + ");");
482            } else if (str.contains("][")) {
483              str = str.replace(":", "");
484              op.writerList.add(str);
485            } else if (str.contains("[")) {
486              String name = str.substring(0, str.indexOf("[")).trim();
487              if (isArray(name, funcIndex)) {
488                str = str.replace(":", "");
489                op.writerList.add(str);
490              }
491
492            }
493
494          }
495        } else {
496          // comment
497        }
498        i++;
499        // }
500
501    }
502
503    private boolean isMatrix(String text, int index) {
504
505      if (functionList.size() == 0) {
506        return false;
507      }
508      for (int i = 0; i < functionList.get(index).getDefinedVariables().size(); i++) {
509        if (functionList.get(index).getDefinedVariables().get(i).getName().equals(text)) {
510          return true;
511        }
512      }
513      return false;
514    }
515
516    private boolean isFunctionCall(String text) {
517      for (int i = 0; i < functionList.size(); i++) {
518        if (text.contains(functionList.get(i).getName())) {
519          // System.out.println(text);
520          return true;
521        }
522      }
523      return false;
524    }
525
526    private boolean isArray(String text, int index) {
527      if (functionList.size() == 0) {
528        return false;
529      }
530      for (int i = 0; i < functionList.get(index).getDefinedVariables().size(); i++) {
531        if (functionList.get(index).getDefinedVariables().get(i).getName().equals(text)) {
532          return true;
533        }
534      }
535      return false;
536    }
537
538  }
```

Code 6.4: project/Generator/src/com/generator/source/CodeGenerator.java

```
 1  package com.generator.io;
 2
 3  import java.io.BufferedReader;
 4  import java.io.BufferedWriter;
 5  import java.io.File;
 6  import java.io.FileOutputStream;
 7  import java.io.FileReader;
 8  import java.io.IOException;
 9  import java.io.OutputStreamWriter;
10  import java.util.ArrayList;
11
```

```
12 public class FileOperations {
13
14   public ArrayList<String> readerList = new ArrayList<String>();
15   public ArrayList<String> writerList = new ArrayList<String>();
16
17   public void readFile(String fileName) {
18     File file;
19     BufferedReader buffer = null;
20     try {
21       file = new File(fileName);
22       buffer = new BufferedReader(new FileReader(file));
23       String readLine = "";
24       while ((readLine = buffer.readLine()) != null) {
25         readerList.add(readLine);
26       }
27       buffer.close();
28     } catch (IOException e) {
29       e.printStackTrace();
30     }
31
32   }
33
34   public void writeFile(String fileName) {
35     File file;
36     BufferedWriter buffer = null;
37     try {
38       file = new File(fileName);
39       FileOutputStream fos = new FileOutputStream(file);
40       buffer = new BufferedWriter(new OutputStreamWriter(fos));
41       String readLine = "";
42       for (int i = 0; i < writerList.size(); i++) {
43         buffer.write(writerList.get(i));
44         buffer.newLine();
45       }
46       buffer.close();
47     } catch (IOException e) {
48       e.printStackTrace();
49     }
50   }
51
52 }
```

Code 6.5: project/Generator/src/com/generator/io/FileOperations.java