Computing Characteristic Polynomials of Matrices of Structured Polynomials

by

Marshall Law

B.Sc., Simon Fraser University, 2014

Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
Department of Mathematics
Faculty of Science

© Marshall Law 2017
SIMON FRASER UNIVERSITY
Spring 2017

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, education, satire, parody, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name: **Marshall Law**

Degree: **Master of Science (Mathematics)**

Title: ***Computing Characteristic Polynomials of Matrices of Structured Polynomials***

Examining Committee: **Chair:** Luis Goddyn
Professor

Michael Monagan
Senior Supervisor
Professor

Marni Mishna
Supervisor
Associate Professor

Nils Bruin
Examiner
Professor

Date Defended: April 13, 2017

Abstract

We present a parallel modular algorithm for finding characteristic polynomials of matrices with integer coefficient bivariate monomials. For each prime, evaluation and interpolation gives us the bridge between polynomial matrices and matrices over a finite field so that the Hessenberg algorithm can be used. After optimizations, we are able to save a significant amount of work by incremental Chinese remaindering and early termination.

Keywords: Exact Linear Algebra, Characteristic Polynomial, High Performance Parallel Algorithms

Dedication

To my Lord and Saviour, the one who died for me though I am still unworthy.

Acknowledgements

I don't believe this list could ever be complete, but I have so much to be grateful for:

- Dr. Michael Monagan: For your kindness, patience, guidance and generosity. My time in graduate school has certainly been extremely valuable, I have learned so much.
- Dr. JF Williams: The Math461 course you taught spiked my interest in mathematics.
- Dr. David Muraki: Your invaluable advice has helped me a lot during my undergraduate studies.
- Drs. Marni Mishna and Nils Bruin: For excellent comments/suggestions for this thesis.
- Peter Cho Ho Lam: Your immense biblical knowledge is always an encouragement, and thanks for inspiring Theorem 4.
- John Kluesner: You are a great and helpful friend, it's always fun to work/chat with you.
- My dad, mom, and brother: Your unconditional love and support.
- My pastor and his wife: I am extremely honoured and privileged to be in your flock.
- My church: You are my home away from home.
- Romans 5:18, Ephesians 4:30.

Table of Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

Table of Contents vi

List of Algorithms viii

List of Figures viii

List of Tables viii

1 Introduction 1

1.1 Motivation . . . 3
1.2 Thesis Outline . . . 3
1.3 Original Contribution . . . 4
1.4 Related Presentations . . . 4

2 Background 5

2.1 Algebra . . . 5
2.1.1 Brief History . . . 5
2.1.2 Linear Algebra . . . 5
2.1.3 Abstract Algebra . . . 7
2.2 Algorithms for Computing Characteristic Polynomials . . . 8
2.2.1 The Bareiss Fraction-Free Algorithm . . . 8
2.2.2 The Berkowitz Method . . . 11
2.2.3 The Hessenberg Algorithm . . . 14
2.3 Other Algorithms and Tools . . . 17
2.3.1 Polynomial Evaluation . . . 17
2.3.2 Polynomial Interpolation . . . 17

2.3.3 Fast Evaluation and Interpolation . . . 18
2.3.4 Chinese Remainder Theorem and Algorithm . . . 18
2.4 Size of Characteristic Polynomial . . . 19
2.4.1 Degree Bounds . . . 19
2.4.2 P-Norms . . . 20
2.4.3 Coefficient Bound . . . 20
2.5 Prime Numbers . . . 21
2.5.1 Primality Testing . . . 21

3 The Bivariate Routine **23**
3.1 The Modular Algorithm . . . 23
3.1.1 Bounds on Characteristic Polynomial . . . 24
3.1.2 Cost of Modular Algorithm . . . 27
3.2 Overview of Routine . . . 28
3.2.1 Structures of $C(\lambda, x, y)$ . . . 28
3.2.2 High Level Description . . . 28
3.3 Phase 1 – Query . . . 29
3.3.1 Lowest Degree Factors . . . 30
3.3.2 Non-Zero Factors . . . 30
3.3.3 Required Points . . . 31
3.3.4 Unlucky Evaluations . . . 32
3.4 Phase 2 – Optimizations . . . 33
3.4.1 Lowest Degree . . . 33
3.4.2 Even Degree . . . 34
3.4.3 Non-zero Factors . . . 34
3.4.4 Savings with Combined Optimizations . . . 34
3.4.5 Chinese Remainder Algorithm (CRA) . . . 35

4 Implementation **42**
4.1 Data Structures . . . 42
4.2 Parallelization . . . 44
4.3 Space . . . 44
4.4 Prime Numbers . . . 48
4.5 Partial Code . . . 48

5 Output **52**
5.1 Validation . . . 52
5.2 Benchmarks . . . 55
5.3 General Routine . . . 57
5.4 Conclusion and Further Work . . . 58

List of Algorithms

1 Bareiss Fraction Free Determinant 9
2 Berkowitz Method 12
3 Hessenberg Method 15
4 Characteristic Polynomial on Bivariate Matrices 24

List of Figures

Figure 2.1 Berkowitz algorithm in Maple code. 13

Figure 3.1 Modular algorithm homomorphism diagram. 23

Figure 4.1 Main Functions in C Code (part 1/3). 49
Figure 4.2 Main Functions in C Code (part 2/3). 50
Figure 4.3 Main Functions in C Code (part 3/3). 51

List of Tables

Table 2.1 Summary of algorithms. 16

Table 3.1 Bounds on the specific characteristic polynomials $C = C(\lambda, x, y)$. . . 25
Table 3.2 Data for the coefficients of $C(\lambda, x, y)$ for $n = 16$. 29

Table 5.1 Bivariate routine timings in seconds (s), minutes (m) or hours (h). . 56
Table 5.2 Maple and Magma timings in seconds (s), minutes (m) or hours (h). 56
Table 5.3 Time spent in parallel algorithm for $n = 256$. 56

Chapter 1

Introduction

Computing the characteristic polynomial of a matrix is a classical and fundamental problem in mathematics. For numerical matrices, there are many optimized algorithms to compute eigenvalues. Finding the characteristic polynomial of symbolic matrices, however, is usually more difficult and requires more computation power and memory.

The central goal of this thesis is to compute characteristic polynomials of matrices of polynomials. We start by considering matrices whose entries are monomials in two variables with integer coefficients. Since these matrices can be highly structured, we also exploit possible extra structures for optimizations. A typical computer today contains multiple cores, which when utilized properly, provides a significant speed up. By combining the above ideas to compute characteristic polynomials, we have developed an optimized and parallel routine that may be used for general matrices of bivariate monomials. Our routine is probabilistic, as it computes the characteristic polynomial correctly with high probability.

Let $A(x, y)$ be an n by n matrix with entries of the form

$$c\ x^a y^b$$

where c is an integer and the exponents a, b are non-negative integers. Let $C(\lambda, x, y) \in \mathbb{Z}[\lambda, x, y]$ be the characteristic polynomial of $A(x, y)$, which is

$$C(\lambda, x, y) = \det\left(\lambda I_n - A(x, y)\right)$$

by definition, where I_n is the $n \times n$ identity matrix.

On the next page, we give an example of a matrix with bivariate monomial entries.

Matrix Example

$$
\begin{bmatrix}
x^8 & x^5y & x^5y & x^4y^2 & x^5y & x^2y^2 & x^4y^2 & x^3y^3 & x^5y & x^4y^2 & x^2y^2 & x^3y^3 & x^4y^2 & x^3y^3 & x^3y^3 & x^4y^4 \\
x^7 & x^6y & x^4y & x^5y^2 & x^4y & x^3y^2 & x^3y^2 & x^4y^3 & x^4y & x^5y^2 & xy^2 & x^4y^3 & x^3y^2 & x^4y^3 & x^2y^3 & x^5y^4 \\
x^7 & x^4y & x^6y & x^5y^2 & x^4y & xy^2 & x^5y^2 & x^4y^3 & x^4y & x^3y^2 & x^3y^2 & x^4y^3 & x^3y^2 & x^2y^3 & x^4y^3 & x^5y^4 \\
x^6 & x^5y & x^5y & x^6y^2 & x^3y & x^2y^2 & x^4y^2 & x^5y^3 & x^3y & x^4y^2 & x^2y^2 & x^5y^3 & x^2y^2 & x^3y^3 & x^3y^3 & x^6y^4 \\
x^7 & x^4y & x^4y & x^3y^2 & x^6y & x^3y^2 & x^5y^2 & x^4y^3 & x^4y & x^3y^2 & xy^2 & x^2y^3 & x^5y^2 & x^4y^3 & x^4y^3 & x^5y^4 \\
x^6 & x^5y & x^3y & x^4y^2 & x^5y & x^4y^2 & x^4y^2 & x^5y^3 & x^3y & x^4y^2 & y^2 & x^3y^3 & x^4y^2 & x^5y^3 & x^3y^3 & x^6y^4 \\
x^6 & x^3y & x^5y & x^4y^2 & x^5y & x^2y^2 & x^6y^2 & x^5y^3 & x^3y & x^2y^2 & x^2y^2 & x^3y^3 & x^4y^2 & x^3y^3 & x^5y^3 & x^6y^4 \\
x^5 & x^4y & x^4y & x^5y^2 & x^4y & x^3y^2 & x^5y^2 & x^6y^3 & x^2y & x^3y^2 & xy^2 & x^4y^3 & x^3y^2 & x^4y^3 & x^4y^3 & x^7y^4 \\
x^7 & x^4y & x^4y & x^3y^2 & x^4y & xy^2 & x^3y^2 & x^2y^3 & x^6y & x^5y^2 & x^3y^2 & x^4y^3 & x^5y^2 & x^4y^3 & x^4y^3 & x^5y^4 \\
x^6 & x^5y & x^3y & x^4y^2 & x^3y & x^2y^2 & x^2y^2 & x^3y^3 & x^5y & x^6y^2 & x^2y^2 & x^5y^3 & x^4y^2 & x^5y^3 & x^3y^3 & x^6y^4 \\
x^6 & x^3y & x^5y & x^4y^2 & x^3y & y^2 & x^4y^2 & x^3y^3 & x^5y & x^4y^2 & x^4y^2 & x^5y^3 & x^4y^2 & x^3y^3 & x^5y^3 & x^6y^4 \\
x^5 & x^4y & x^4y & x^5y^2 & x^2y & xy^2 & x^3y^2 & x^4y^3 & x^4y & x^5y^2 & x^3y^2 & x^6y^3 & x^3y^2 & x^4y^3 & x^4y^3 & x^7y^4 \\
x^6 & x^3y & x^3y & x^2y^2 & x^5y & x^2y^2 & x^4y^2 & x^3y^3 & x^5y & x^4y^2 & x^2y^2 & x^3y^3 & x^6y^2 & x^5y^3 & x^5y^3 & x^6y^4 \\
x^5 & x^4y & x^2y & x^3y^2 & x^4y & x^3y^2 & x^3y^2 & x^4y^3 & x^4y & x^5y^2 & xy^2 & x^4y^3 & x^5y^2 & x^6y^3 & x^4y^3 & x^7y^4 \\
x^5 & x^2y & x^4y & x^3y^2 & x^4y & xy^2 & x^5y^2 & x^4y^3 & x^4y & x^3y^2 & x^3y^2 & x^4y^3 & x^5y^2 & x^4y^3 & x^6y^3 & x^7y^4 \\
x^4 & x^3y & x^3y & x^4y^2 & x^3y & x^2y^2 & x^4y^2 & x^5y^3 & x^3y & x^4y^2 & x^2y^2 & x^5y^3 & x^4y^2 & x^5y^3 & x^5y^3 & x^8y^4
\end{bmatrix}
$$

As seen above, the matrix is dense as it does not contain any zeroes. The entries also have low degrees in the two variables x and y. Note that our routine is not limited to matrices where the elements have identical unit coefficients.

1.1 Motivation

The choice of matrix elements (bivariate monomials with integer coefficients) arises from specific matrices that occur in the Ising model [21] of statistical physics. This model seeks to explain phase transitions – when substance changes from one state to another. These substance molecules may be in one of two possible states. Their transition depends on their current state and neighbouring molecules, which are represented by the matrix elements. We have specific transfer matrices for the square Ising model wrapped around a torus obtained from M. Kauers, which were generated automatically by a combinatorial procedure. The variables x and y represent the temperature and magnetic field, respectively. As n (the dimension) approaches infinity, the largest eigenvalue of the matrices (as a function of x and y) approaches a limiting function. This limiting function has cusps which correspond to temperatures where phase transitions occur. M. Kauers and D. Zeilberger proposed to find the first few characteristic polynomials, then detect some interesting pattern in the eigenvalues, and finally predict for the limiting function.

We have five square matrices with the dimensions $16, 32, 64, 128$ and 256 obtained from M. Kauers [10]. The smallest matrix with dimension 16 is shown on the previous page. Computer algebra systems (CAS) such as Maple and Magma have general purpose routines for computing characteristic polynomials. To compute the characteristic polynomial of the size 64 matrix, Magma takes about 15 hours, Maple takes three hours, and our routine takes less than a minute. Based on the complexity of the algorithm used by Maple, the characteristic polynomial of the size 128 and 256 matrices should be solved in two days and one month, respectively. But for the size 128 matrix, Maple is still computing for the answer even after running for one week. On the other hand, our optimized and parallel algorithm can compute the size 256 matrix in four hours on a 6-core 3.2 GHz machine.

1.2 Thesis Outline

In Chapter 2, we discuss background material regarding characteristic polynomials, algorithms for computing them and other algorithms that will be needed. Chapter 3 is our new routine in two phases, query and optimization. Chapter 4 involves implementation details, including our parallel algorithm in Cilk C. Chapter 5 contains output validation, comparisons and benchmarks/timings. After Chapter 5 are the Appendix and Bibliography.

1.3 Original Contribution

Within the process of optimizing a modular algorithm to solve for characteristic polynomials, we have discovered the following which we believe to be original.

- For a general matrix, the number of terms in the characteristic polynomial follows a well defined pattern. Theorem 4 on page 26 gives more details.
- For any system of congruences, the solution in mixed radix form has a constant tail under some conditions. Theorem 8 on page 37 has more details.
- A modular algorithm for computing the characteristic polynomial of a matrix with bivariate monomials. The algorithm may be generalized to a matrix of bivariate polynomials.
- A parallel implementation (in C code) of our modular algorithm which incorporates various optimizations. The optimizations (if applicable) are automatically detected and applied for any user-input matrices. After computing the characteristic polynomial, it also validates the solution independently and probabilistically.

1.4 Related Presentations

This work has also been published and presented on other occasions as written below.

- Published and presented as a paper [11] at the 19th International Workshop on Computer Algebra in Scientific Computing (CASC), a conference held in Bucharest, Romania in September 2016.
- Presented as a poster at the 41st International Symposium on Symbolic and Algebraic Computation (ISSAC), a conference held in Waterloo, Canada in July 2016.
- Presented at the poster competition on the SFU Symposium on Mathematics and Computation day in August 2016. This poster also won the runner up prize in the graduate student category.

Chapter 2

Background

In this chapter, we present and recall background material on algebra, algorithms in computer algebra and relevant mathematical tools.

2.1 Algebra

2.1.1 Brief History

Matrices and determinants were studied as early as 3 B.C. as seen in the Chinese mathematics textbook `The Nine Chapters on the Mathematical Art` [24]. The same idea behind matrices existed in array methods for solving simultaneous equations. The determinant was considered to be the property of a linear system of equations, as it determines whether the system has a unique solution. A system has a unique solution if and only if the determinant is non-zero. It was not until around 1600 A.D. when matrices and determinants received much more attention from many famous mathematicians. The 2 by 2 determinant was given by Cardano, and larger dimensions by Leibniz. Matrices (arrays) were further used to represent linear transformations/mappings.

2.1.2 Linear Algebra

An m by n matrix is a rectangular array consisting of elements (numbers or variables) with m rows and n columns. A vector is a matrix with only one row or one column. We are primarily working with square matrices, meaning $m = n$. Let A be an n by n matrix and $k < n$. The k by k matrix composed of the first k rows and k columns of A is called the principal k by k submatrix of A .

Definition 1 (Determinant)**.** *Let $A = (a_{ij})$ for $1 \leq i, j \leq n$ be a square matrix with dimension n. The determinant for $n = 2$ is*

$$\det A = |A| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{12}a_{21}.$$

When $n > 2$, the <u>determinant</u> by expanding along the top row ($i = 1$) is

$$\det A = |A| = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} M[i,j]$$

where $M[i,j]$ is the determinant of a square matrix of dimension $n-1$ obtained by removing the ith row and jth column from the matrix A.

As an example, consider the 3 by 3 matrix $A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$. The determinant by expanding along the first row is given by

$$\begin{aligned} \det A &= \sum_{j=1}^{3} (-1)^{1+j} a_{1j} M[1,j] \\ &= (-1)^{1+1} a \begin{vmatrix} e & f \\ h & i \end{vmatrix} + (-1)^{1+2} b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + (-1)^{1+3} c \begin{vmatrix} d & e \\ g & h \end{vmatrix} \\ &= a(ei - fh) - b(di - fg) + c(dh - eg). \end{aligned}$$

If the determinant of a square matrix is zero, the matrix is said to be singular (or degenerate) and the matrix inverse does not exist. Two other key objects of matrices are are the eigenvalues and eigenvectors.

Definition 2 (Eigenvalue and Eigenvector). *A non-zero vector v is an <u>eigenvector</u> of a matrix A if and only if there exists a scalar λ such that*

$$Av = \lambda v.$$

The scalar of λ is the corresponding <u>eigenvalue</u> of the eigenvector v.

Every matrix may be viewed as a linear transformation. Applying a matrix to an eigenvector v gives a scalar multiple (λ) of v. As taught in a first course in linear algebra, one method to compute the eigenvalues is to first solve $\det(\lambda I_n - A) = 0$ for λ. Then substitute each λ into $(\lambda I_n - A)v = 0$ and solve for the corresponding (non-zero) eigenvector v.

Definition 3 (Characteristic Polynomial/Matrix). *The <u>characteristic polynomial</u> $C(\lambda)$ of a matrix A is obtained by taking the determinant of the <u>characteristic matrix</u> $\lambda I_n - A$, that is*

$$C(\lambda) = \det(\lambda I_n - A).$$

The characteristic polynomial of our running example matrix A is

$$\begin{aligned}C(\lambda) &= \det(\lambda I_3 - A)\\ &= (\lambda - a)\left((\lambda - e)(\lambda - i) - fh\right) - b\left(d(\lambda - i) - fg\right) + c\left(dh - (\lambda - e)g\right)\\ &= \lambda^3 - (a + e + i)\lambda^2 + (ae + ai - bd - cg + ei - fg)\lambda - \det A.\end{aligned}$$

Note that some definitions may write $\det(A - \lambda I_n)$. It differs from the above by a scalar of $(-1)^n$, and we will use them interchangeably. It can also be verified easily for a square matrix $A = (a_{ij})$ with dimension n that

$$C(\lambda) = \lambda^n + (-1)^{n-1}\text{trace}(A)\lambda^{n-1} + \cdots + (-1)^n \det(A)$$

where $\text{trace}(A) = \sum_{i=1}^{n} a_{ii}$ by definition. The roots of the polynomial $C(\lambda)$ are the eigenvalues of A. The Caley–Hamilton theorem states that any square matrix satisfies its own characteristic polynomial. By substituting the matrix into the characteristic polynomial, the result is $C(A) = 0$, the zero matrix. Hence the minimal polynomial of A is a factor of the characteristic polynomial, another useful property.

2.1.3 Abstract Algebra

Abstract algebra (or modern algebra) is a general term for the study of algebraic structures, such as rings, fields and vector spaces. Their terminology will be used throughout this thesis, thus we define them here for clarity.

Definition 4 (Commutative Ring with Identity)**.** *A commutative ring R with identity is a set with two binary operations $+$ (addition) and $\times$ (multiplication) satisfying the following axioms.*

1. *There is an element $0_R \in R$ such that $a + 0_R = a$ for all $a \in R$ (additive identity).*
2. *For all $a \in R$ there exists $-a \in R$ such that $a + (-a) = 0_R$ (additive inverse).*
3. *$(a + b) + c = a + (b + c)$ for all $a, b, c \in R$ ($+$ is associative).*
4. *$a + b = b + a$ for all $a, b \in R$ ($+$ is commutative).*
5. *There is an element $1_R \in R$ such that $1_R \neq 0_R$ and $a \times 1_R = 1_R \times a = a$ for all $a \in R$ (multiplicative identity).*
6. *$(a \times b) \times c = a \times (b \times c)$ for all $a, b, c \in R$ ($\times$ is associative).*
7. *$a \times b = b \times a$ for all $a, b \in R$ ($\times$ is commutative).*
8. *$(a + b) \times c = a \times c + b \times c$ and $c \times (a + b) = c \times a + c \times b$ for all $a, b, c \in R$ ($\times$ is distributive with respect to $+$).*

An example of a commutative ring with identity is the integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$. Note that a general non-commutative ring without identity would satisfy all the axioms above except for axioms 5 and 7. An example of a non-commutative ring would be matrices, as matrix multiplication is generally not commutative. Rings are one of the most fundamental structures, and the following structures are obtained by introducing more requirements.

Definition 5 (Integral Domain)**.** *An integral domain is a commutative ring with identity where the product of any two non-zero elements is non-zero.*

Definition 6 (Field)**.** *A field F is a commutative ring with identity where every non-zero element has a multiplicative inverse, that is: For all $a \in F$ there exists $a^{-1} \in F$ such that $a \times a^{-1} = 1_F$.*

The set of all integers $\mathbb{Z}$ is an integral domain. Common examples of fields include the rational numbers $\mathbb{Q}$, real numbers $\mathbb{R}$ and complex numbers $\mathbb{C}$. Suppose there are two non-zero elements a, b in a field F and that $ab = 0_F$. Multiplying by $a^{-1}b^{-1}$ on both sides gives $b^{-1}a^{-1}ab = 1_F = 0_F$, which is a contradiction. Thus it follows that all fields are also integral domains.

Our algorithm will be using finite fields where there are only a finite number of elements. The integers modulo a prime p form a finite field, denoted by $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$. We use the common notation $K[x]$ to represent polynomials over K in variable x: that is, polynomials with coefficients in K. So $\mathbb{Z}[x]$ represents polynomials over the integers: i.e., with integer coefficients. Similarly, $\mathbb{Z}[x]^{m \times n}$ represents m by n matrices whose entries are from $\mathbb{Z}[x]$.

2.2 Algorithms for Computing Characteristic Polynomials

In the following subsections we describe several methods for computing $C(\lambda)$, which are currently implemented in common CAS (computer algebra systems). We give a summary of these algorithms in Table 2.1 on page 16.

Chapter 2 in [4] also gives two more methods for computing characteristic polynomials. One of them uses adjoint matrices and the other uses Lagrange interpolation.

2.2.1 The Bareiss Fraction-Free Algorithm

The Bareiss fraction-free algorithm [2, 12] is currently implemented and used to compute the determinant and characteristic polynomial in Magma [20]. It does exact divisions (hence the adjective fraction-free) for matrices over an integral domain D, such as polynomials

Algorithm 1 Bareiss Fraction Free Determinant

```
    Input: n × n matrix A = (a_ij), entries from an integral domain D.
    Output: The determinant of the matrix A.
 1: function BareissDet(A)
 2:     Initialize a_00 ← 1
 3:     for k = 1, 2, ..., n − 1 do          ▷ Assume diagonals/pivots are non-zero.
 4:         for i = k + 1, ..., n do
 5:             for j = k + 1, ..., n do
 6:                 a_ij ← (a_kk a_ij − a_kj a_ik)/a_{k−1,k−1}    ▷ Divisions are exact in D.
 7:             end for
 8:         end for
 9:     end for
10:     return a_nn
11: end function
```

with integer coefficients. This method computes the determinant of a matrix by modifying Gaussian elimination based on Sylvester's determinant identity. The identity computes the determinant for a matrix partitioned into blocks, namely

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & 0 \\ A_{21} & I_{n-k} \end{vmatrix} \cdot \begin{vmatrix} I_k & A_{11}^{-1}A_{12} \\ 0 & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{vmatrix} = |A_{11}| \cdot |A_{22} - A_{21}A_{11}^{-1}A_{12}|$$

where A_{11} is a square matrix with dimension $k < n$. Since the core routine is similar to that of Gaussian elimination, it does $O(n^3)$ arithmetic operations in the integral domain D. Each iteration builds up the determinant of leading principal submatrices and they are stored as entries on the main diagonal.

This method may be applied directly on the characteristic matrix $(\lambda I_n - A(x, y))$, as it has polynomial entries in $\mathbb{Z}[\lambda, x, y]$. Alternatively, one may evaluate λ at $\{0, 1, \ldots, n\}$, for $n+1$ matrices, apply the algorithm to obtain $n+1$ determinants and interpolate λ for $C(\lambda, x, y)$.

Ignoring the details of pivoting for simplicity, the algorithm works as follows. Let $A = (a_{ij})$ be an n by n matrix. Let $A^{(k)}$ denote the matrix after k iteration(s), so $A^{(0)} = A$. Initialize $a_{00}^{(-1)} = 1$, and for $k = 1, 2, \ldots n-1$, compute $A^{(k)}$ using

$$a_{ij}^{(k)} = \frac{a_{kk}^{(k-1)} a_{ij}^{(k-1)} - a_{kj}^{(k-1)} a_{ik}^{(k-1)}}{a_{k-1,k-1}^{(k-2)}}$$

for $k+1 \leq i \leq n$, and $k+1 \leq j \leq n$ assuming pivots (diagonals) $a_{k-1,k-1}^{(k-2)}$ are non-zero. Pseudo-code is given in Algorithm 1. Assuming no pivoting, one can show for $0 \leq k < n$ that $A_{k+1,k+1}^{(k)}$ is the determinant of the principal $(k+1)$ by $(k+1)$ submatrix of A. After

$k > 1$ steps we have this intermediate expansion

$$a_{kk}^{(k-1)}a_{ij}^{(k-1)} - a_{kj}^{(k-1)}a_{ik}^{(k-1)} = a_{k-1,k-1}^{(k-2)}a_{ij}^{(k)} \tag{2.1}$$

where the right hand side is larger than the result $a_{ij}^{(k)}$. So each step of the algorithm computes a determinant by dividing Equation (2.1) by a previously computed determinant.

Here is this algorithm on the running example matrix $A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$. We obtain

$$A^{(1)} = \begin{bmatrix} a & b & c \\ d & ae-bd & af-cd \\ g & ah-bg & ai-cg \end{bmatrix} \text{ and } A^{(2)} = \begin{bmatrix} a & b & c \\ d & ae-bd & af-cd \\ g & ah-bg & (\det A)a/a \end{bmatrix}.$$

Notice in $A^{(1)}$ the second diagonal entry $ae - bd$ is the determinant of the principal 2 by 2 submatrix. Also in $A^{(2)}$ the bottom right entry is the determinant of A, and the exact division of a previous diagonal term is shown for illustration. For larger matrices, all iterations starting from the second one will require quotients. At the last step, the intermediate expansion is given by

$$a_{n-1,n-1}^{(n-2)}a_{n,n}^{(n-2)} - a_{n-1,n}^{(n-2)}a_{n,n-1}^{(n-2)} = a_{n-2,n-2}^{(n-3)} \det A.$$

Because our matrix entries contain polynomials, the product $a_{n-2,n-2}^{(n-3)} \det A$ will be much larger (in degree and terms) than $\det A$. Effectively, the algorithm computes a large multiple of the answer and involves expensive divisions that become the bottleneck.

One method to reduce work in the divisions is to use lazy and forgetful polynomial arithmetic [22]. The two products in the numerator of

$$\frac{a_{n-1,n-1}^{(n-2)}a_{n,n}^{(n-2)} - a_{n-1,n}^{(n-2)}a_{n,n-1}^{(n-2)}}{a_{n-2,n-2}^{(n-3)}} = \det A$$

may be computed separately. Dividing the two products by $a_{n-2,n-2}^{(n-3)}$ may also be computed simultaneously.

2.2.2 The Berkowitz Method

The Berkowitz method [3, 1] is currently implemented in Maple, and works for all matrices over a commutative ring R with identity. It is used when the `CharacteristicPolynomial` routine in the `LinearAlgebra` package is called. This algorithm does $O(n^4)$ arithmetic operations in the ring R, but has no divisions. Here we show the mathematical idea behind this method, an algorithm, an example and our implementation in Maple.

Consider the matrix $A = (a_{ij})$, where $a_{ij} \in R$ for $1 \le i, j \le n$. Let $A_r = (a_{ij})$, $1 \le i, j \le r$ be the principal r by r submatrix of A. Suppose the characteristic polynomial of A_r is

$$C_r(\lambda) = \det(A_r - \lambda I_r) = \sum_{k=0}^{r} c_{r,r-k}\lambda^k = c_{r,r} + c_{r,r-1}\lambda + \cdots + c_{r,0}\lambda^r.$$

Let $F = (f_{ij})$ be the cofactor matrix of A, meaning $f_{ij} = (-1)^{i+j}M[i,j]$, where $M[i,j]$ is the determinant of the matrix A after removing the ith row and jth column (as previously stated in Definition 1). The adjoint matrix is the matrix transpose of the cofactor matrix F. The adjoint of the characteristic matrix satisfies

$$\begin{aligned}\mathrm{Adj}(A_r - \lambda I_r) &= -\sum_{i=0}^{r-1}\sum_{j=0}^{r-i-1} c_{r,r-i-j-1}A_r^j\lambda^i = -\sum_{k=1}^{r}\sum_{j=0}^{r-k} c_{r,r-k-j}A_r^j\lambda^{k-1} \\ &= -\sum_{k=1}^{r}\left(c_{r,r-k}I_r + c_{r,r-k-1}A_r + c_{r,r-k-2}A_r^2 + \cdots + c_{r,0}A_r^{r-k}\right)\lambda^{k-1}.\end{aligned} \tag{2.2}$$

Consider the matrix A_{r+1} written in terms of A_r, the diagonal term $a_{r+1,r+1}$, row vector R_r and column vector S_r, as seen in Equation (2.3). The determinant is given by

$$\det A_{r+1} = \det\begin{pmatrix} A_r & S_r \\ R_r & a_{r+1,r+1}\end{pmatrix} = \det(A_r)a_{r+1,r+1} - R_r\mathrm{Adj}(A_r)S_r. \tag{2.3}$$

Combining Equations (2.2) and (2.3) on the characteristic matrix gives a recurrence formula for $C_{r+1}(\lambda)$ in terms of $C_r(\lambda)$ and matrix entries, which is

$$C_{r+1}(\lambda) = C_r(\lambda)(a_{r+1,r+1} - \lambda) + \sum_{k=1}^{r}\sum_{j=0}^{r-k} c_{r,r-k-j}\left(R_rA_r^jS_r\right)\lambda^{k-1}.$$

The Berkowitz algorithm may be implemented using matrix and vector multiplications. Given a polynomial $C_r(\lambda) = \sum_{k=0}^{r} c_{r,k}\lambda^{r-k}$, let the coefficients be encoded in the vector

$$\vec{C}_r = (c_{r,0}, c_{r,1}, \ldots, c_{r,r})^T.$$

Algorithm 2 Berkowitz Method

Input: $n \times n$ matrix $A = (a_{ij})$, entries from a commutative ring with identity.
Output: The characteristic polynomial of the matrix A (in vector form).

```
1: function Berkowitz(A)
2:     Initialize: C ← (−1, a_11)^T
3:     for i = 1, 2, ... n − 1 do
4:         Obtain row vector R_i, column vector S_i and principal matrix A_i
5:         Q ← −λ^{i+1} + a_{i+1,i+1} λ^i + Σ_{j=1}^{i} R_i A_i^{j−1} S_i λ^{i−j}
6:         C ← Toep(Q) × C
7:     end for
8:     return C
9: end function
```

Also with $C_r(\lambda) = \sum_{k=0}^{r} c_{r,k}\lambda^{r-k}$, we define the special lower triangular Toeplitz matrix as

$$\text{Toep}(C_r) = \begin{bmatrix} c_{r,0} & 0 & 0 & \dots & 0 \\ c_{r,1} & c_{r,0} & 0 & \dots & 0 \\ c_{r,2} & c_{r,1} & c_{r,0} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ c_{r,r-1} & c_{r,r-2} & c_{r,r-3} & \dots & c_{r,0} \\ c_{r,r} & c_{r,r-1} & c_{r,r-2} & \dots & c_{r,1} \end{bmatrix}$$

which has dimension $(r+1) \times r$. The equivalent recurrence formula for $\vec{C}_{r+1}$ is now

$$\vec{C}_{r+1} = \text{Toep}(Q_{r+1}) \times \vec{C}_r, \quad \text{where}$$

$$Q_{r+1} = -\lambda^{r+1} + a_{r+1,r+1}\lambda^r + (R_rS_r)\lambda^{r-1} + \cdots + (R_rA_r^iS_r)\lambda^{r-1-i} + \cdots + (R_rA_r^{r-1}S_r). \tag{F1}$$

Hence the algorithm computes the characteristic polynomial by

$$\vec{C}_n = \text{Toep}(Q_n) \times \cdots \times \text{Toep}(Q_3) \times \text{Toep}(Q_2) \times \vec{C}_1. \tag{F2}$$

The pseudo-code is given in Algorithm 2. Here is the algorithm applied on the running example matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

The first step is to initialize $\vec{C}_1 = (-1, a)^T$ from $C_1 = a - \lambda$. The next step is to compute $Q_2 = -\lambda^2 + e\lambda + bd$, and obtain $\text{Toep}(Q_2) = \begin{bmatrix} -1 & 0 \\ e & -1 \\ bd & e \end{bmatrix}$. For $\vec{C}_2$ we compute a matrix-

vector product as follows:

$$\vec{C_2} = \text{Toep}(Q_2) \times \vec{C_1} = (1, -a-e, ae-bd)^T .$$

One may easily check the correctness as the second and third term in $\vec{C_2}$ are the negative trace and determinant of the principal 2 by 2 matrix, respectively. For $\vec{C_3}$ we compute

$$Q_3 = -\lambda^3 + i\lambda^2 + (g,h)\,(c,f)^T\,\lambda + (g,h)\,A_2\,(c,f)^T$$

to obtain

$$\text{Toep}(Q_3) = \begin{bmatrix} -1 & 0 & 0 \\ i & -1 & 0 \\ cg+fh & i & -1 \\ acg+cdh+bgf+efh & cg+fh & i \end{bmatrix}.$$

Then finally we have

$$\begin{aligned}\vec{C_3} &= \text{Toep}(Q_3) \times \vec{C_2} \\ &= (-1, a+e+i, -ae-ai+bd+cg-ei+fh, \det A)^T\end{aligned}$$

which is correct. We have implemented this algorithm with Maple code in Figure 2.1.

```
Toep := proc(F, x) local n, C; ## Input: Polynomial F, variable x
    n := degree(F, x);
    C := [ 0, seq( coeff(F,x,n-i), i=0..n ) ]; # C = [0,c_n,...,c_1,c_0]
    return Matrix( n+1, n, (i,j) -> C[ max(i-j+2 , 1) ] );
end proc: ## Returns the special Toeplitz matrix

charpoly := proc(M, n, x) local i, j, A, C, Q, R, S, T;
## Input: Matrix M, dimension n, variable x
    C := Vector( [-1 , M[1,1]] );        # Initialize
    for i from 1 to n-1 do
        R := M[i+1,1..i];                # Row vector
        A := M[1..i,1..i];               # Principal matrix
        S := M[1..i,i+1];                # Column vector
        Q := M[i+1,i+1]*x^i - x^(i+1);
        for j from 1 to i do             # F1 loop (multiplications):
            Q := Q + R . S * x^(i-j);    # Vector-vector product (i)
            S := A . S;                  # Matrix-vector product (i^2)
        od;
        C := Toep(Q, x) . C;             # F2: Matrix-vector product
    od;                                  # ((i+1)*(i+2) multiplications)
    return C; ## Returns the characteristic polynomial in vector form
end proc:
```

Figure 2.1: Berkowitz algorithm in Maple code.

The following theorem on the complexity is found in [1]. Here we present our proof with complete details.

Theorem 1 (Berkowitz Algorithm Complexity). *The number of multiplications in the commutative ring R with identity for the Berkowitz algorithm is*

$$\frac{1}{4}n^4 + \frac{1}{6}n^3 + \frac{3}{4}n^2 + \frac{5}{6}n - 2 \in O(n^4). \tag{2.4}$$

Proof. The most expensive step in the algorithm is computing Q_{r+1} in Equation (F1, p.12) because, for $r > 2$, it involves computing the matrix power A_r^{r-1}. We may avoid matrix-matrix multiplications when computing matrix powers in $R_r A_r^j S_r$ for $1 \le j \le r-1$ by computing instead matrix-vector products. As seen in Figure 2.1, the `F1` double loop involves matrix-vector multiplications only. Equivalently, it computes for $R_r A_r^j S_r = R_r(A_r \ldots (A_r(A_r S_r)))$. For dimension r, matrix-vector products require r^2 multiplications. The number of multiplications in the ring R for the `F1` double loop (excluding line `F2` in Figure 2.1) is

$$\sum_{i=1}^{n-1}\left(\sum_{j=1}^{i} i + i^2\right) = \frac{1}{4}n^4 - \frac{1}{6}n^3 - \frac{1}{4}n^2 + \frac{1}{6}n \in O(n^4). \tag{2.5}$$

The next step is Equation (F2, p.12), that is, to multiply $n-1$ special Toeplitz matrices obtained from Q_r with one vector. Similarly, we only need matrix-vector products. In this step, we multiply dimension $(i+2)$ by $(i+1)$ matrix with dimension $(i+1)$ vector, for $1 \le i \le n-1$. Thus the number of multiplications in the ring R for (F2) is

$$\sum_{i=1}^{n-1}(i+1)(i+2) = \frac{1}{3}n^3 + n^2 + \frac{2}{3}n - 2 \in O(n^3). \tag{2.6}$$

Adding Equations (2.5) and (2.6) gives the result in Equation (2.4). □

Preliminary timings show that it is faster than the $O(n^3)$ fraction-free method for our matrices. This is because the Berkowitz method is division-free whereas the Bareiss fraction free method involves large divisions.

2.2.3 The Hessenberg Algorithm

For matrices over a field F, the Hessenberg algorithm [4] computes for the characteristic polynomial from leading principal submatrices. This algorithm requires $O(n^3)$ arithmetic operations in the field F. We use this method as the base algorithm of our modular routine. CAS such as Maple and `LinBox` [13] also use this Hessenberg method to compute the characteristic polynomial over a finite field $\mathbb{Z}_p$. For a matrix of integers, Maple uses a modular Hessenberg algorithm [14] to compute $C(\lambda)$, otherwise it uses the Berkowitz algorithm.

Algorithm 3 Hessenberg Method

```
    Input: n × n matrix H = (h_ij), entries from a field F.
    Output: The characteristic polynomial of the matrix H.
 1: function HESSENBERG(H, λ)
    Stage 1 - Decomposition
 2:     for m = 2, 3, ..., n − 1 do
 3:         if h_{i,m−1} = 0 for i > m then
 4:             Continue loop for next value of m
 5:         else
 6:             t ← h_{i,m−1} for smallest index i ≥ m
 7:             if i > m then
 8:                 Swap row H_i with row H_m, and swap column H_i with column H_m
 9:             end if
10:         end if
11:         for i = m + 1, ..., n do
12:             u ← h_{i,m−1}/t, row H_i ← H_i − uH_m and column H_m ← H_m + uH_i
13:         end for
14:     end for
    Stage 2 - Recurrence
15:     Initialize: C_0(λ) ← 1
16:     for m = 1, 2, ..., n do
17:         t ← 1 and C_m(λ) ← (λ − h_{m,m})C_{m−1}(λ)
18:         for i = 1, ..., m − 1 do
19:             t ← th_{m−i+1,m−i} and C_m(λ) ← C_m(λ) − th_{m−i,m}C_{m−i−1}(λ)
20:         end for
21:     end for
22:     return C_n(λ)
23: end function
```

There are two stages to the Hessenberg algorithm. The first stage decomposes the input matrix A into a matrix H in Hessenberg form, while preserving the characteristic polynomial. In the second stage, a recurrence is used to compute the characteristic polynomial of H. The first part involves divisions but the second part does not. The pseudo-code is given in Algorithm 3. The following matrix H below is in upper Hessenberg form as all entries under the sub-diagonal are zeroes:

$$H = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \dots & h_{1,n} \\ k_2 & h_{2,2} & h_{2,3} & \dots & h_{2,n} \\ 0 & k_3 & h_{3,3} & \dots & h_{3,n} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & k_n & h_{n,n} \end{bmatrix}$$

The recurrence begins with $C_0(\lambda) = 1$ and iterates using the formula

$$C_m(\lambda) = (\lambda - h_{m,m})C_{m-1}(\lambda) - \sum_{i=1}^{m-1}\left(h_{i,m}C_{i-1}(\lambda)\prod_{j=i+1}^{m} k_j\right)$$

for $1 \leq m \leq n$. Here is the Hessenberg algorithm applied on the running example matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

The first part of the algorithm reduces the matrix to the upper Hessenberg form:

$$H = \begin{bmatrix} a & \frac{bd+cg}{d} & c \\ d & \frac{de+fg}{d} & f \\ 0 & \frac{d^2h-deg+dgi-fg^2}{d^2} & \frac{di-fg}{d} \end{bmatrix}.$$

The eigenvalues (and hence characteristic polynomial) are preserved. Next, in the second stage, the recurrence is iterated:

$$\begin{aligned} C_1(\lambda) &= \lambda - a, \\ C_2(\lambda) &= \lambda^2 - \frac{ad + de + fg}{d}\lambda + \frac{ade + afg - bd^2 - cgd}{d}, \\ C_3(\lambda) &= \lambda^3 - (a + e + i)\lambda^2 + (ae + ai - bd - cg + ei - fg)\lambda - \det A. \end{aligned}$$

Notice the intermediate polynomial $C_2(\lambda)$ does not correspond to the characteristic polynomial of the original principal 2 by 2 submatrix.

Summary of Algorithms

Table 2.1 below gives a summary of these algorithms.

Algorithm	CAS	Complexity	Entries	Remarks
Bareiss fraction-free	Magma	$O(n^3)$	Integral domain	Gaussian elimination Expensive & exact divisions
Berkowitz	Maple	$O(n^4)$	Commutative ring	Matrix-vector products Division free
Hessenberg	(Maple) LinBox	$O(n^3)$	Field	Decomposition & recurrence Some divisions

Table 2.1: Summary of algorithms.

2.3 Other Algorithms and Tools

This section contains other building blocks required in our routine.

2.3.1 Polynomial Evaluation

Our code evaluates polynomials. Consider a degree d polynomial $f(x) = \sum_{i=0}^{d} a_i x^i$. A common way to evaluate polynomials is Horner's method, which rewrites the polynomial into the more computationally efficient form

$$f(x) = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{d-1} + x \cdot a_d) \ldots)).$$

To evaluate this polynomial at some value α, we obtain

$$f(\alpha) = a_0 + \alpha(a_1 + \alpha(a_2 + \cdots + \alpha(a_{d-1} + \alpha \cdot a_d) \ldots)).$$

By multiplying from right to left, evaluation for degree d polynomial requires d multiplications and d additions.

Square and Multiply

Our specific matrices only contain monomial entries with one term. A more efficient way to evaluate a monomial $g(x) = x^d$ than Horner's method is the square and multiply method. Suppose $d = 10$, Horner's method would require $d = 10$ multiplications to obtain $g(\alpha)$. While the square and multiply method computes for $g(\alpha) = \alpha^{10} = \left(\alpha(\alpha^2)^2\right)^2$, which is only four multiplications. In general to evaluate the polynomial $g(x)$, the square and multiply algorithm requires no more than $2\log_2 d$ multiplications. So for $d > 4$, the square and multiply method is more efficient than Horner's method. Thus for our matrices of monomials, we evaluate by the square and multiply method.

2.3.2 Polynomial Interpolation

Polynomial interpolation returns a polynomial when given pairs of points. Our presentation follows page 185 in [8].

Theorem 2 (Interpolation [8])**.** *Let F be a field. Given a set of $d+1$ data points $(x_i, y_i) \in F^2$ with distinct x_i, there exists a unique polynomial $f(x)$ over F of at most degree d such that $f(x_i) = y_i$ for all $0 \leq i \leq d$.*

The Newton interpolation algorithm is a common method for polynomial interpolation. It computes the polynomial by using Newton form, that is,

$$f(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)(x - x_1) + \cdots + c_d \prod_{i=0}^{d-1}(x - x_i)$$

where $c_0 = y_0$, and $c_i = \left(y_i - \sum_{j=0}^{i-1} c_j \prod_{k=0}^{j-1}(x_i - x_k)\right)\left(\prod_{j=0}^{i-1}(x_i - x_j)\right)^{-1}$ for $1 \leq i \leq d$. We have implemented this method in C code. For degree d, Newton interpolation requires $O(d^2)$ arithmetic operations in the field F.

We use the calling sequence `Interpolate`$([x_0, \ldots, x_d], [y_0, \ldots, y_d], z)$ to obtain a polynomial in the variable of the third argument.

2.3.3 Fast Evaluation and Interpolation

The fast Fourier transform (FFT) is a method for doing fast evaluations and interpolations. More details can be found in Section 4.5 of [8]. The FFT evaluates and interpolates a polynomial of degree $< n$ with $O(n \log n)$ field operations. If the problem involves large degrees, the FFT would be faster than regular evaluation and interpolation. We have experimented by using the FFT for evaluation and interpolation, but we were not able to adapt all optimizations in Chapter 3 to the FFT. We will proceed without them.

2.3.4 Chinese Remainder Theorem and Algorithm

Part of our algorithm uses the Chinese remainder algorithm (CRA) which arises from the Chinese remainder theorem (CRT). Our presentation follows page 175 in [8], where additional details can be found.

Theorem 3 (CRT [8])**.** *Consider a set of congruences $x \equiv c_i \pmod{p_i}$ where the p_i are pairwise relatively prime, for $1 \leq i \leq m$. For any $a \in \mathbb{Z}$, there exists a unique integer u that satisfies*

$$u \equiv c_i \pmod{p_i} \text{ and } a \leq u < a + (p_1 p_2 \ldots p_m).$$

We will use `CRA`$([c_1, \ldots, c_m], [p_1, \ldots, p_m])$ as the calling sequence to obtain the solution in the range $0 \leq u < \Pi_{i=1}^{m} p_i$. One method to compute the solution is based on the mixed radix representation, that is

$$u = v_1 + v_2 M_1 + v_3 M_2 + \cdots + v_m M_{m-1}$$

where $M_k = \prod_{i=1}^{k} p_i$ and $0 \leq v_i < p_i$. Garner's algorithm solves for u in this form by starting with $v_1 \equiv c_1 \pmod{p_1}$, and then using the recurrence

$$v_i \equiv \left(c_i - \sum_{j=1}^{i-1} v_j M_{j-1}\right) M_{i-1}^{-1} \pmod{p_i} \text{ for } 2 \leq i \leq m. \tag{2.7}$$

Note that there is an alternative approach using the Lagrange representation:

$$u = v_1 \frac{M_m}{p_1} + v_2 \frac{M_m}{p_2} + \cdots + v_m \frac{M_m}{p_m}.$$

This method is not compatible with our optimizations as we will explain in Chapter 3. We proceed with the mixed radix representation.

2.4 Size of Characteristic Polynomial

In general, the characteristic polynomial coefficients and degrees are larger than that of the original matrix entries. Our matrices consist of polynomials, so we must ensure sufficient primes and evaluation/interpolation points to recover the result correctly. In this section, we provide details on how to obtain degree and coefficient bounds for the characteristic polynomial.

2.4.1 Degree Bounds

For any polynomial f, let $\deg_x f$ represent the degree of f in x. The total degree of f is represented by just $\deg f$ without any subscripts. Let $A = (a_{ij})$ be a dimension n square matrix, where the entries are bivariate polynomials in x and y. The degrees of the determinant are bounded by the following:

$$\deg\ \det A \leq D_{tot} = \min\left(\sum_{i=1}^{n}\max_{j=1}^{n}\deg a_{ij}, \sum_{j=1}^{n}\max_{i=1}^{n}\deg a_{ij}\right)$$

$$\deg_x \det A \leq D_x = \min\left(\sum_{i=1}^{n}\max_{j=1}^{n}\deg_x a_{ij}, \sum_{j=1}^{n}\max_{i=1}^{n}\deg_x a_{ij}\right)$$

$$\deg_y \det A \leq D_y = \min\left(\sum_{i=1}^{n}\max_{j=1}^{n}\deg_y a_{ij}, \sum_{j=1}^{n}\max_{i=1}^{n}\deg_y a_{ij}\right)$$

In each equation, the first and second sums add the largest degree in each row and columns, respectively. Then we take the minimum of the two to obtain the lower degree bound in that variable.

The above bounds on the determinant also work for the characteristic polynomial $C(\lambda) = \sum_{i=0}^{n} c_i\lambda^i = \det(\lambda I_n - A)$. In general, many of the coefficients such as $c_{n-1} = (-1)^{n-1}\text{trace}(A)$ have degrees much lower than the bound.

Kronecker Substitution

A Kronecker substitution is a common tool when working with multiple variables. The problem will be simpler if the matrices of interest only contain univariate polynomials. This can be achieved by a Kronecker substitution.

Definition 7 (Kronecker Substitution). *A Kronecker substitution on $A(x, y)$, denoted by $K_b(A(x, y))$, replaces the variable y by x^b, giving*

$$K_b(A(x, y)) = A(x, y = x^b).$$

To ensure an invertible substitution, take $b > \deg_x C(\lambda, x, y)$. Then $C(\lambda, x, y)$ can be recovered from the characteristic polynomial of $A(x, y = x^b)$.

The smallest possible value for a reversible substitution would be $b = D_x + 1$. After applying, the degrees of the polynomial entries in $A(x, y = x^b)$ become quite large. Now the problem has effectively become solving for

$$\det(\lambda I_n - A(x, x^b)) = C(\lambda, x, x^b).$$

Note that we only use this for determining a coefficient bound on $C(\lambda, x, y = x^b)$.

2.4.2 P-Norms

We will use a p-norm to bound the size of the largest integer coefficient in $C(\lambda, x, y)$.

Definition 8 (p-norm). *Let $a_1, a_2, \ldots, a_t$ denote the coefficients of any given multivariate polynomial f (with t terms). The p-norm of f, denoted by $||f||_p$ for $p \geq 1$, is defined by*

$$||f||_p = \left(\sum_{i=1}^{t} |a_i|^p \right)^{1/p}.$$

When $p = 1$ we add up the absolute value of each coefficient, so $||f||_1 = \sum_{i=1}^{t} |a_i|$. When $p = 2$ we have the Euclidean norm, that is $||f||_2 = \sqrt{\sum_{i=1}^{t} |a_i|^2}$. When $p = \infty$ we have $||f||_\infty = \max_i |a_i|$, called the maximum norm or height. It is also clear that

$$||f||_\infty \leq \cdots \leq ||f||_2 \leq ||f||_1.$$

2.4.3 Coefficient Bound

The Hadamard inequality for a n by n integer matrix $M = (m_{ij})$ asserts the bound

$$|\det M| \leq H(M) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} |m_{ij}|^2 \right)^{1/2}.$$

A similar bound [15] exists for matrices with polynomial entries. Let $M(x) = (m_{ij}(x))$, where $m_{ij}(x)$ are polynomials in $\mathbb{Z}[x]$. Let $s_0, s_1, s_2, \cdots$ represent the coefficients of the determinant, so $\det M(x) = s_0 + s_1 x + s_2 x^2 + \cdots$. Let $T = (t_{ij})$ be the n by n matrix where

each entry is defined by $t_{ij} = ||m_{ij}(x)||_1$. Then the similar bound states that

$$|| \det M(x)||_2 = \left(\sum |s_i|^2\right)^{1/2} \leq H(T) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} |t_{ij}|^2\right)^{1/2}.$$

So the equivalent bound on the height is given by

$$|| \det M(x)||_\infty \leq H(T) = \prod_{i=1}^{n} \left(\sum_{j=1}^{n} ||m_{ij}(x)||_1^2\right)^{1/2}. \tag{2.8}$$

Here is an example with a 2 by 2 matrix with bivariate monomial entries.

$$M(x,y) = \begin{bmatrix} 2x^2y & xy^2 \\ 3xy & y \end{bmatrix}, \ \det M(x,y) = 2x^2y^2 - 3x^2y^3.$$

If we proceed with $K_b(M(x,y))$ where $b = 3$ we obtain

$$M(x, y = x^3) = \begin{bmatrix} 2x^5 & x^7 \\ 3x^4 & x^3 \end{bmatrix} \text{ and } T = \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix}.$$

Thus the bound on height of the determinant clearly satisfies

$$|| \det M(x,y)||_\infty = 3 \leq H(T) = \sqrt{(2^2+1)(3^2+1)} = \sqrt{50} \approx 7.071.$$

2.5 Prime Numbers

A prime number p is a positive integer with only two divisors, one and itself. Prime numbers are vital to our routine because we compute for characteristic polynomial over multiple finite fields.

Definition 9 (Prime Counter)**.** *Let* $\underline{\pi(x)}$ *denote the prime number counting function, that is, the number of primes less than or equal to* x*.*

A useful bound from [6] showed that

$$\pi(x) > \frac{x}{\ln x}(1 + \frac{1}{\ln x} + \frac{2}{\ln^2 x}) \text{ for } x \geq 88,783. \tag{2.9}$$

2.5.1 Primality Testing

The Miller–Rabin test [23] tells us whether an integer is a composite (non-prime). Given an odd integer $n = 2^q s + 1 > 2$, this test picks another integer $1 < a < n - 1$ to be the base. If

$$a^s \not\equiv 1 \pmod{n} \text{ and } a^{2^r s} \not\equiv -1 \pmod{n}$$

for some $1 \leq r < q$, then n is not prime. A prime n will pass this test for all a. It is possible for the the above condition to hold even when n is composite, then the integer n is a strong pseudo–prime with respect to base a. Thus for large n, it is common to choose several integers as the base.

Our routine generates primes p in the range $2^{30} < p < 2^{31}$ (assuming machine word size is 32 bits). In that range, if an integer n passes the Miller-Rabin test for bases $a = 2, 3, 5, 7$, then n is proven to be prime (see [17]).

Chapter 3

The Bivariate Routine

This chapter contains the mathematical ideas required for computing characteristic polynomials with bivariate entries. We start with the core modular algorithm in our routine. As mentioned in the Introduction, there is much structure to our characteristic polynomials. Hence, we identify how to take advantage of this structure. By combining all the optimizations, our routine saves a very significant amount of work.

3.1 The Modular Algorithm

As the modular name suggests, we compute the image of the characteristic polynomial modulo a sequence of primes $p_1, p_2, \ldots, p_m$. Finding m, the required number of primes will be addressed in the next subsection regarding bounds. The modular routine takes in a matrix and returns an image of the characteristic polynomial. With sufficient number of images, the CRA recovers the solution over the integers. We give the homomorphism diagram in Figure 3.1 and the pseudo-code in Algorithm 4 for $C(\lambda, x, y)$ (without optimizations).

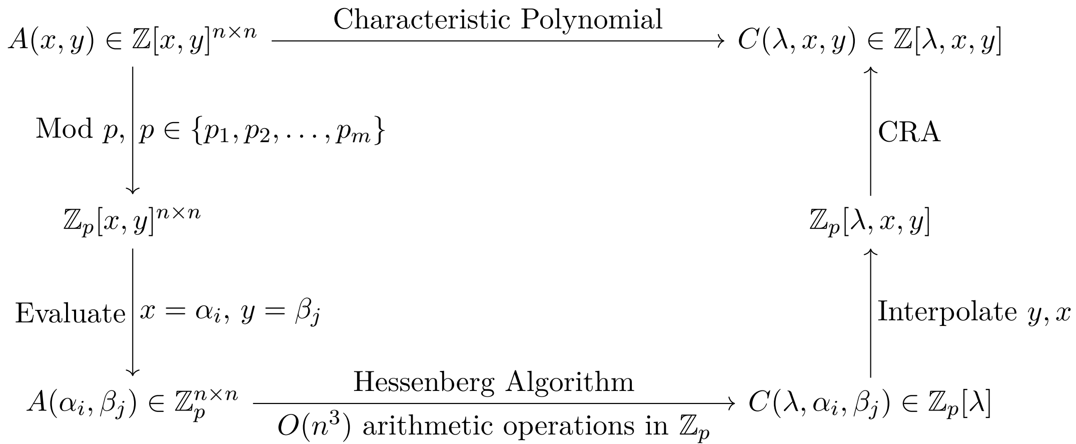

Figure 3.1: Modular algorithm homomorphism diagram.

Algorithm 4 Characteristic Polynomial on Bivariate Matrices

Input: $n \times n$ matrix $M(x,y) = (m_{ij}(x,y))$, positive integers $\alpha_1, \ldots, \alpha_{e_x}, \beta_1, \ldots, \beta_{e_y}$ and primes $p_1, \ldots, p_m$.
Output: Characteristic polynomial $C(\lambda, x, y)$ of the input matrix.

```
 1: for k = 1, 2, ..., m do                                  ▷ Assume m primes are sufficient
 2:     for i = 1, 2, ..., e_x do                            ▷ Assume e_x > deg_x C(λ, x, y)
 3:         for j = 1, 2, ..., e_y do                        ▷ Assume e_y > deg_y C(λ, x, y)
 4:             T_j ← Hessenberg(M(α_i, β_j), λ) mod p_k
 5:         end for
 6:         S_i ← Interpolate([β_1, ..., β_{e_y}], [T_1, ..., T_{e_y}], y) mod p_k
 7:     end for
 8:     C_k ← Interpolate([α_1, ..., α_{e_x}], [S_1, ..., S_{e_x}], x) mod p_k
 9: end for
10: return CRA([C_1, ..., C_m], [p_1, ..., p_m])
```

The following paraphrases the core algorithm above with more details.

1. For each prime $p \in \{p_1, p_2, \ldots p_m\}$ do the following.
   (a) For each $\alpha_i \in \{\alpha_1, \ldots, \alpha_{e_x}\}$, evaluate the matrix at $x = \alpha_i$ to obtain $A(\alpha_i, y)$ mod p, and do:
      i. Evaluate the matrix at all $\beta_j \in \{\beta_1, \ldots, \beta_{e_y}\}$ to obtain $A(\alpha_i, \beta_j)$ mod p;
      ii. Apply the Hessenberg algorithm on $A(\alpha_i, \beta_j)$ mod p for $C(\lambda, \alpha_i, \beta_j)$ mod p;
      iii. Interpolate the coefficients of λ in y for $C(\lambda, \alpha_i, y)$ mod p from the points β_j and values $C(\lambda, \alpha_i, \beta_j)$;
   (b) Interpolate the coefficients of λ in x for $C(\lambda, x, y)$ mod p from the points α_i and values $C(\lambda, \alpha_i, y)$ mod p;
2. Recover the integer coefficients of $C(\lambda, x, y)$ using the Chinese remainder algorithm.

3.1.1 Bounds on Characteristic Polynomial

In the previous chapter, we gave degree and coefficient bounds for determinant computations. The degree bound remains the same as given on 19. Table 3.1 below shows the various degree bounds based on our specific matrices.

Now we derive the coefficient bound based on Equation (2.8) for the characteristic polynomial. The bound is applied on the determinant of $M(\lambda, x, y) = \lambda I_n - A(x, y)$, where the entries of the matrix $A(x, y)$ are bivariate monomials. The T matrix is obtained by taking the 1-norm for each element in $M(\lambda, x, y)$, so we have $t_{ii} = ||\lambda - x^a y^b||_1 = 2$ and $t_{ij} = ||x^a y^b||_1 = 1$ for $1 \leq i \neq j \leq n$. Thus the height of $C(\lambda, x, y) = \det M(\lambda, x, y)$ is

bounded by

$$||C(\lambda, x, y)||_\infty \leq \prod_{i=1}^{n} \left(\sum_{j=1}^{n} t_{ij}^2 \right)^{1/2} = \prod_{i=1}^{n} (4 + n - 1)^{1/2} = (n+3)^{n/2}. \quad (3.1)$$

This bound tells us how many primes are needed at most to recover the integer coefficients of $C(\lambda, x, y)$. Allowing for positive and negative coefficients, we need $\prod_{i=1}^{m} p_i > 2(n+3)^{n/2}$. Since the standard integer word size is 32 bits, we use signed integers for primes between 30 and 31 bits. Thus the number of primes needed is given by

$$m \leq \lceil \log_{2^{30}} 2(n+3)^{n/2} \rceil. \quad (3.2)$$

Note that with optimizations to be presented, the integer coefficients can be recovered with fewer primes.

n	$\deg C$	$\deg_x C$	$\deg_y C$	$\|\|C\|\|_\infty$(bits)
16	128	96	32	34
32	288	208	80	83
64	768	576	192	195
128	1664	1216	448	451
256	4096	3072	1024	1027

Table 3.1: Bounds on the specific characteristic polynomials $C = C(\lambda, x, y)$.

We suspected a smaller/tighter bound on the characteristic polynomial $C(\lambda) = \sum_{i=0}^{n} c_i \lambda^i$ of an integer matrix A. After experimenting on matrices with random integers, we spotted a trend of

$$|c_n| \leq |c_{n-1}| \leq \cdots \leq |c_1| \leq |c_0| \leq H(A).$$

Because $c_0 = C(0) = \det A$ we suspected that $||C(\lambda)||_\infty \leq H(A)$. Thus for our matrices, we thought we could replace the above bound $(n+3)^{n/2}$ with the Hadamard bound $n^{n/2}$ on $C(\lambda, x, y)$. But in [5], there is a matrix

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 & -1 \\ 1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

with characteristic polynomial

$$\lambda^5 - 5\lambda^4 + 40\lambda^2 - 80\lambda + 48.$$

This is a counter example to our conjecture as $|c_1| = 80 > \lceil H(M) \rceil = 56 > |c_0| = 48$. However, in an attempt to prove the conjecture, we found Theorem 4 on the number of terms within $C(\lambda)$.

Number of Terms in the Characteristic Polynomial

Consider the square matrix $A = (a_{ij})$ for $1 \leq i, j \leq n$. Let the characteristic polynomial be $c_n\lambda^n + c_{n-1}\lambda^{n-1} + \cdots + c_1\lambda + c_0$. We know that $c_0 = \det A$ a sum of $n!$ terms since each term in the determinant is a product n elements from distinct rows and columns. The trace is a sum of the n elements on the main diagonal. The middle coefficients, c_k for $0 < k < n-1$ is a sum of $\frac{n!}{k!}$ terms, as shown here with an example.

Take $n = 5$ and $k = 2$. In order to contribute to the coefficient of λ^2, pick the first two elements on the diagonal as λ:

$$\det \begin{pmatrix} \lambda & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{21} & \lambda & a_{23} & a_{24} & a_{25} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{pmatrix} = \lambda^2 \times \det \begin{pmatrix} a_{33} & a_{34} & a_{35} \\ a_{43} & a_{44} & a_{45} \\ a_{53} & a_{54} & a_{55} \end{pmatrix}.$$

After removing the corresponding rows and columns, the remaining matrix has dimension 3 by 3. The determinant of the 3 by 3 matrix is a sum of 3! terms. There are $\binom{5}{2}$ ways to pick two λs, and 3! terms in the remaining submatrix determinant. So the number of terms for λ^2 is $\binom{5}{2}(5-2)! = \frac{5!}{2!}$. Now we prove in general for the other coefficients.

Theorem 4 (Terms in $C(\lambda)$)**.** *Let $c_n\lambda^n + \cdots + c_1\lambda + c_0$ be the characteristic polynomial of a square matrix $A = (a_{ij})$ of dimension n. The coefficient of λ^k, that is c_k, is a sum of $\frac{n!}{k!}$ terms from the original matrix entries a_{ij}, for $0 \leq k \leq n$.*

Proof. Each term in the determinant is a product of n elements from distinct rows and columns. Choose $k < n$ diagonal elements from the characteristic matrix and consider the terms which have λ^k. There are $\binom{n}{k}$ ways to choose $0 \leq k \leq n$ diagonal elements to get λ^k. After removing the corresponding rows and columns, the remaining matrix has dimension $n - k$. The determinant of the remaining matrix has $(n-k)!$ terms without λ. So the number of terms with λ^k is

$$\binom{n}{k}(n-k)! = \frac{n!}{k!}.$$

□

3.1.2 Cost of Modular Algorithm

Recall $D_x \geq \deg_x C(\lambda, x, y)$ and $D_y \geq \deg_y C(\lambda, x, y)$. Let d be the largest exponent in the input matrix, so $D_x, D_y \in O(nd)$. Let $T = (D_x + 1)(D_y + 1) \in O(n^2 d^2)$ denote maximum number of terms (for each λ^i). Here we give the cost of the core algorithm based on n (matrix dimension), m (number of 31 bit primes) and d (largest degree of polynomials in matrix).

Theorem 5 (Core Algorithm Cost)**.** *The cost of the core modular algorithm is*

$$O(mn^5 d^2 + mn^4 d^3 + m^2 n^3 d^2) \tag{3.3}$$

arithmetic operations of at most size 31 bits.

Proof. We give the arithmetic operations by the stages of the core algorithm: evaluation, Hessenberg algorithm, interpolation and CRA.

1. Reduce modulo p_i, for $1 \leq i \leq m$. This step is trivial as the matrices have unit coefficients as entries. But the following (sub) steps are to be computed m times.

   (a) Evaluation: Since the input matrix entries are of the form $x^a y^b$, we compute them by the square and multiply algorithm. The maximum degree in either variable is d for all matrix entries. Computing two exponents costs $2O(\log d)$ multiplications, and their product requires one more. Evaluation is needed on n^2 entries and for total of T integer matrix images. Thus this evaluation stage with m primes has a cost of

   $$mn^2 T(1 + 2O(\log d)) \in O(mn^4 d^2 \log d). \tag{3.4}$$

   (b) Hessenberg: The Hessenberg algorithm is called mT times, for m primes and T matrix images. Thus the cost is

   $$mTO(n^3) \in O(mn^5 d^2). \tag{3.5}$$

   (c) Interpolation: With T matrix images, we obtain T characteristic polynomial images, namely $C(\lambda, \alpha_i, \beta_j)$. We interpolate for $C(\lambda, \alpha_i, y)$ first. There are n coefficients to interpolate from $\mathbb{Z}[\lambda]$ to $\mathbb{Z}[\lambda, y]$. Each coefficient becomes a polynomial in y with degree up to D_y, so interpolation on y costs $nO(D_y^2)$. Then from $\mathbb{Z}[\lambda, y]$ to $\mathbb{Z}[\lambda, x, y]$, there are at most $n(D_y + 1)$ coefficients. Each coefficient has degree at most D_x, so interpolation on x costs $n(D_y + 1)O(D_x^2)$. The total cost to interpolate both variables for m primes is

   $$mnO(D_y^2) + mn(D_y + 1)O(D_x^2) \in O(mn^4 d^3). \tag{3.6}$$

2. Chinese remaindering for m congruences does $O(m^2)$ arithmetic operations. The total number of coefficients is at most nT, so the cost is

$$nTO(m^2) \in O(m^2 n^3 d^2). \tag{3.7}$$

Combining the above Equations (3.5), (3.6) and (3.7) gives the cost in Equation (3.3) as stated in the Theorem. Note that Equation (3.4) is not included in the Theorem because it is dominated by (3.6). □

3.2 Overview of Routine

There is much structure in our matrices, as they are generated by a combinatorial process. This section will discuss the structure in the characteristic polynomials, and establish our two–phase routine.

3.2.1 Structures of $C(\lambda, x, y)$

To see the structure, let $C(\lambda, x, y) = \sum_{i=0}^{n} c_i(x, y)\lambda^i$. Now the coefficients of λ^i of the characteristic polynomial factors:

$$c_i(x, y) = x^{f_i} y^{g_i} (x^2 - 1)^{h_i} s_i(x, y)$$

where the exponent values $f_i, g_i, h_i \in \mathbb{N} \cup \{0\}$, for $0 \le i < n$, and the $s_i(x, y)$ are bivariate polynomials with even degrees in x. These exponents are large enough such that we can speed up the computation by a significant factor. See Appendix A1 and A2 for some coefficients in the characteristic polynomial of the size 16 matrix. Table 3.2 contains the values for these parameters f_i, g_i, h_i for the size 16 matrix and also other information about $c_i(x, y)$. Notice that the largest coefficients in $c_i(x, y)$ in magnitude (see column $||c_i||_\infty$) decrease as i increases but those in s_i (see column $||s_i||_\infty$) increase and decrease.

As the structure suggests, the most complicated factors to recover are the bivariate polynomials $s_i(x, y)$. In the next two sections, we split into two phases to compute $C(\lambda, x, y)$. The first phase is to find f_i, g_i, h_i for $0 \le i < n$ and the number of evaluation points e_x and e_y. The second phase is to recover the “cofactor” polynomials $s_i(x, y)$.

3.2.2 High Level Description

The following list indicates how our entire routine works.

- Phase 1: Find the key values of f_i, h_i, g_i, e_x, e_y by making queries in the two variables.

i	f_i	g_i	h_i	$\deg_x c_i$	$\deg_y c_i$	$\|\|c_i\|\|_\infty$	$\deg_x s_i$	$\deg_y s_i$	$\|\|s_i\|\|_\infty$
0	32	32	32	96	32	601080390	0	0	1
1	32	28	28	92	32	160466400	4	4	4
2	24	25	25	88	31	28428920	14	6	31
3	26	22	22	82	30	16535016	12	8	128
4	20	19	19	76	29	3868248	18	10	382
5	22	16	16	70	28	946816	16	12	684
6	16	14	14	64	26	183648	20	12	1948
7	18	12	12	58	24	82492	16	12	3738
8	12	10	10	52	22	35264	20	12	4730
9	14	8	8	46	20	10876	16	12	3740
10	8	6	6	40	18	3242	20	12	2116
11	10	4	4	34	16	1556	16	12	806
12	4	3	3	28	13	322	18	10	454
13	6	2	2	22	10	108	12	8	142
14	0	1	1	16	7	22	14	6	31
15	4	0	0	8	4	4	4	4	4

Table 3.2: Data for the coefficients of $C(\lambda, x, y)$ for $n = 16$.

- Phase 2: Compute the polynomial images $s_i(x, y)$ modulo prime(s) with optimizations based on the previously computed exponents.
  - Apply the CRA incrementally on the image coefficients.
- Phase 3: Validate solution independently (details provided in Chapter 5).

3.3 Phase 1 – Query

This query phase scouts out the size of the problem, namely it determines various degrees within the characteristic polynomial. The factor degrees are applicable to each coefficient of λ, which are the exponents in

$$x^{f_i} y^{g_i} (x^2 - 1)^{h_i}.$$

Then we also require the other degrees within the characteristic polynomial, $\deg_x s_i(x, y)$ and $\deg_y s_i(x, y)$ so that we have sufficient number of evaluation/interpolation points.

Since the matrix entries are bivariate, we make two queries. In each query, we evaluate one variable of the matrix $A(x, y)$ to obtain an image of the characteristic polynomial in the other variable. Let p be a prime. We use $p = 2^{31} - 1$ in our C implementation because it is the largest prime for a 32-bit signed integer. Let γ be chosen at random from $\mathbb{Z}_p$ for evaluation. We evaluate the matrix $A(x, y)$ to obtain the two matrices where the entries

are univariate monomials with integer coefficients in $\mathbb{Z}_p$, namely

$$A(x, \gamma) \bmod p \text{ and } A(\gamma, y) \bmod p.$$

Their respective characteristic polynomials are

$$C(\lambda, x, y = \gamma) \bmod p \text{ and } C(\lambda, x = \gamma, y) \bmod p.$$

This is a much simpler problem, as there is one less variable to work with and it is in mod p. Even for our large matrices, the characteristic polynomial of univariate matrices can solved within minutes. The query phase concludes by finding the necessary parameters for phase 2, which include the exponents f_i, g_i, h_i for $0 \leq i < n$, and the minimal number of evaluation/interpolation points e_x and e_y.

Due to two random evaluations, there is a possibility of failure in this phase. The probability of failure is small and will be addressed in Section 3.3.4. Thus assume correctness for now.

3.3.1 Lowest Degree Factors

After making the two queries, we have images $C(\lambda, x, y = \gamma) \bmod p$ and $C(\lambda, x = \gamma, y) \bmod p$. For each coefficient of λ, we require the degrees of the first and last non-zero coefficient in x and y. Let the lowest degrees be f_i, g_i in x, y respectively for $0 \leq i < n$. Let the largest degrees be $\bar{f}_i, \bar{g}_i$ in x, y respectively for $0 \leq i < n$. Thus we have $f_i \leq \bar{f}_i$ and $g_i \leq \bar{g}_i$ for all $0 \leq i < n$. To see it in perspective, the coefficient of λ^i in $C(\lambda, x, y = \gamma)$ has the form

$$c_i(x, \gamma) \bmod p = \bullet x^{\bar{f}_i} + \bullet x^{\bar{f}_i - 1} + \cdots + \bullet x^{f_i + 1} + \bullet x^{f_i}$$

where the symbol $\bullet$ represents integers modulo p. Similarly for $C(\lambda, x = \gamma, y)$,

$$c_i(\gamma, y) \bmod p = \bullet y^{\bar{g}_i} + \bullet y^{\bar{g}_i - 1} + \cdots + \bullet y^{g_i + 1} + \bullet y^{g_i}$$

where the symbol $\bullet$ represents integers modulo p. The key values $\bar{f}_i, f_i, \bar{g}_i, g_i$ can be found easily by searching for first and last non-zero coefficients.

3.3.2 Non-Zero Factors

For our characteristic polynomials, most coefficients of λ have a factor of $(x^2 - 1)^{h_i}$ with large h_i. Removing this reduces the number of evaluation points needed and the integer coefficient size of $c_i(x, y)/(x^2 - 1)^{h_i}$, hence also the number of primes needed. For our matrices it happens that $h_i = g_i$. Otherwise, to find h_i, we divide $c_i(x, \gamma)$ by $(x^2 - 1)$ modulo p repeatedly.

In general, to determine if $(ax \pm b)$ is a factor of a coefficient of $c_i(x, y)$, for small integers $a > 0, b$, we could compute the roots of $c_i(x, \gamma) \bmod p$, a polynomial in $\mathbb{Z}_p[x]$, using Rabin's algorithm [18]. From each root, we try to reconstruct a small fraction $\mp\frac{b}{a}$ using rational number reconstruction ([7] Section 5.10). We have not implemented this.

Definition 10 (Small Unlucky Root)**.** *The value α is a small unlucky root if it can be written as $\alpha \equiv \frac{b}{a} \pmod p$ where $1 \leq a \leq q \ll p$, $1 \leq |b| \leq q$ and for some i*

$$c_i(\alpha, \gamma) \text{ mod } p = 0 \text{ but } c_i(\alpha, y) \neq 0.$$

Note if $b = 0$ then the root is zero, which corresponds to the lowest degree factor.

Theorem 6 (Unlucky Roots)**.** *With bound $q \leq \lceil \sqrt[r]{p} \rceil \ll p$, for $r \geq 3$, the probability of small unlucky roots occurring is $< 1\%$*

Proof. There are q and $2q$ choices for a and b respectively. The total number of possible unlucky roots with these restrictions is $2q^2$. We are working in $\mathbb{Z}_p$, so without restrictions there are at most p roots. So the probability of unlucky roots is given by

$$\Pr[f(\frac{\pm b}{a}, \gamma) \bmod p = 0] \leq \frac{2q^2}{p} = 2p^{\frac{2}{r}-1}$$

If we take $r = 3$, then the probability of unlucky roots is $2p^{-1/3}$. Since we use $p = 2^{31} - 1$, the probability of unlucky roots is under 1%. One could restrict unlucky roots to be smaller by taking $r > 3$. The larger the value r, the lower the probability. □

3.3.3 Required Points

After finding the factor degrees, we determine the minimal number of points required to recover all $s_i(x, y)$, for $0 \leq i < n$. Since we have already computed the largest, smallest and factor degrees, we can know the maximal degrees of $s_i(x, y)$ in both variables. In particular, the degree in x to be interpolated is at most $\max_{0 \leq i < n} \deg_x s_i(x, y)$. So the number of points needed to interpolate in x is given by

$$e_x := \max \frac{1}{2}\{\bar{f}_i - f_i - 2h_i\} + 1, \text{ for } 0 \leq i < n. \tag{3.8}$$

The subtraction of $2h_i$ corresponds to the factor $(x^2 - 1)^{h_i}$. The scalar of half is due to the even powers in x, which will be addressed later in Section 3.4.2 with more details. Similarly for y, the degree is at most $\max_{0 \leq i < n} \deg_y s_i(x, y)$. So the number of points for y is

$$e_y := \max\{\bar{g}_i - g_i\} + 1, \text{ for } 0 \leq i < n. \tag{3.9}$$

3.3.4 Unlucky Evaluations

As mentioned earlier, random evaluations may cause the algorithm to fail by returning an incorrect answer. Without loss of generality, consider one query of randomly evaluating at $y = \gamma$ for random $\gamma \in [0, p-1]$. Let $d_i = \deg_x c_i(x, y)$. We have

$$c_i(x, y) = \sum_{j=0}^{d_i} c_{ij}(y) x^j, \text{ where } c_{ij}(y) \in \mathbb{Z}[y].$$

If $c_{i0}(\gamma) \bmod p = 0$, then the value returned is greater than f_i (the correct lowest degree), which is incorrect. If the algorithm continues, the target for interpolation is compromised, and the final answer will be incorrect.

If $c_{id_i}(\gamma) \bmod p = 0$, then the largest degree becomes less than $\bar{f}_i$. This may affect e_x, the required number of evaluation points, as defined by taking the maximum of a set in Equation (3.8). But the final answer will still be correct as long as e_x is correct.

Definition 11 (Unlucky Evaluation)**.** *Let p be a prime, $0 \le \gamma < p$, and $d_y = \max_{ij} \deg_y a_{ij}(x, y)$ be the largest degree in y from the matrix of interest. γ is an unlucky evaluation if for any $0 \le i < n$*

$$c_{i0}(\gamma) \equiv 0 \pmod p \text{ or } c_{id_i}(\gamma) \equiv 0 \pmod p.$$

Theorem 7 (Unlucky Evaluations)**.** *If γ is chosen from $[0, p-1]$ at random, then*

$$\Pr[\gamma \text{ is unlucky}] \le \frac{n(n+1)d_y}{p}.$$

Proof. Since c_{i0} and c_{id_i} are part of the coefficient of λ^i, their degrees in y are at most $d_y(n-i)$. So for each $0 \le i < n$ there are at most $2d_y(n-i)$ choices for γ such that

$$c_{i0}(\gamma) \equiv 0 \pmod p \text{ or } c_{id_i}(\gamma) \equiv 0 \pmod p.$$

There are n of these cases, so adding them up gives a total of at most $\sum_{i=0}^{n-1} 2d_y(n-i) = d_y n(n+1)$ unlucky evaluations. Therefore the probability of an unlucky evaluation (for $0 \le i < n$) is given by

$$\Pr\left[c_{i0}(\gamma) \equiv 0 \pmod p \text{ or } c_{id_i}(\gamma) \equiv 0 \pmod p\right] \le \frac{d_y n(n+1)}{p}.$$

□

For our largest matrix, the parameters are $n = 256$, $d_x = 16$ and $d_y = 8$. Our algorithm makes two queries with prime $p = 2^{31} - 1$, and the probability of an unlucky evaluation is less than 0.001.

3.4 Phase 2 – Optimizations

The structure for each λ coefficient is already known, namely the factor degrees (f_i, g_i, h_i) and degree of interpolation target (e_x, e_y). In this section we show how to apply the optimizations on the characteristic polynomials. We have implemented each of the following optimizations with Newton interpolation.

We illustrate the optimizations on the size 16 matrix, so Table 3.2 (page 29) will be referenced frequently. Without optimization, the target is $c_0(x, y)$ because it is the largest coefficient in terms of degree and height. But with optimization, the target becomes $s_8(x, y)$ as its degrees and height are larger than all other $s_i(x, y)$. $s_8(x, y)$ contains 93 terms (see Appendix A1) and is irreducible over $\mathbb{Z}$. The savings are shown in the next paragraph.

Without optimizations, we require 97 and 33 evaluation/interpolation points for x and y, respectively (since $\max \deg_x c_i(x, y) + 1 = 97$ and $\max \deg_y c_i(x, y) + 1 = 33$). So for each prime, there would be would be $97 \times 33 = 3201$ calls to the Hessenberg algorithm. Two 31-bit primes are needed because the coefficient bound is $\log_2(n+3)^{n/2} = \log_2(16+3)^8 = 34$ bits. With optimization through two phases, we only need to interpolate for $s_8(x, y)$. Thus we have $e_x = 11 = \max \deg_x s_i(x, y)/2 + 1$ and $e_y = 13 = \max \deg_y s_i(x, y) + 1$. The number of calls to the Hessenberg algorithm is now $11 \times 13 = 143$ (previously 3201). Only one prime is required since the target height is now $\max ||s_i(x, y)||_\infty = 4730$, which is only 13 bits (previously 34).

In the next few subsections, consider the step of interpolating x after y is interpolated within the second phase. Let $E = \{\alpha_1, \alpha_2, \dots\}$ be the evaluation points, and $V_i = \{c_i(\alpha_1, y), c_i(\alpha_2, y), \dots\}$ be the values. We will illustrate savings by referencing the coefficient of $c_8(x, y) = x^{12}y^{10}(x^2 - 1)^{10}s_8(x, y)$. By the end of this section, we will only require $(10 + 1)(12 + 1) = 143$ points, which gives a gain of more than a factor of 12. For the larger matrices, the gain is even greater.

3.4.1 Lowest Degree

Since the lowest degree is known, that is $f_8 = 12$, we need to interpolate $c_8(x, y)/x^{12}$. For each $\alpha_j \in E$, divide V_i by $\alpha_j^{f_8}$ point wise. Proceeding with regular interpolation will give

$$c_8(x, y)/x^{12} = s_8(x, y)y^{10}(x^2 - 1)^{10}.$$

In this example there is a saving of $f_8 = 12$ points. This optimization also applies to the other variable y, as $g_8 = 10$. When applied on both variables, this optimization alone

reduces the total number of evaluation points from 3201 to

$$\max(\deg_x c_i - f_i + 1) \times \max(\deg_y c_i - g_i + 1) = (64 + 1) \times (12 + 1) = 845.$$

3.4.2 Even Degree

All the terms in $c_i(x, y)$ have even degrees in x. So if we interpolate for $c_i(x^{1/2}, y)$ instead of $c_i(x, y)$, the degree of the target is halved, and the number of evaluation points is also (approximately) halved. To do so, simply square each value in E, and proceed with interpolation as usual. The polynomial recovered will have half of the true degree, and $c_i(x, y)$ is recovered by doubling each exponent.

The even degrees structure for our matrices only applies to variable x. With this optimization, the number of evaluation points decreases from $(96 + 1)$ to $(96/2 + 1)$.

3.4.3 Non-zero Factors

This optimization is similar to that of the lowest degree, where the factor is $(x^2 - 1)^{h_i}$. For each $\alpha_j \in E$, we divide each V_i value by $(\alpha_j^2 - 1)^{h_i}$. Proceeding with regular interpolation will return

$$c_i(x, y)/(x^2 - 1)^{h_i} = s_i(x, y)x^{f_i}y^{g_i}.$$

E cannot contain ± 1, because there will be divisions by zero. This optimization is only applicable to variable x, and it alone decreases the number of evaluation points from $(96+1)$ to $\max(\deg_x c_i - 2h_i) + 1 = (38 + 1)$.

Since h_i is large, $||s_i(x, y)||_\infty$ will be much smaller than $||c_i(x, y)||_\infty$. Therefore the algorithm needs fewer primes to recover $C(\lambda, x, y)$. For the 16 by 16 case, the coefficient bound for $C(\lambda, x, y)$ is $(16+3)^8$, about 34 bits. The actual height $||C(\lambda, x, y)||_\infty$ is a 30 bit integer and $\max ||s_i(x, y)||_\infty = 4730 = ||s_8(x, y)||_\infty$ is a 13 bit integer. For the 64 by 64 case, $||C(\lambda, x, y)||_\infty$ is bounded by 195 bits and $\max ||s_i(x, y)||_\infty$ is 72 bits.

Due to this loose bound, the problem has effectively become much smaller in terms of integer coefficient size. The target is $\max_{0 \le i < n} ||s_i(x, y)||_\infty$, instead of the much larger $\max_{0 \le i < n} ||c_i(x, y)||_\infty$. So $C(\lambda, x, y)$ can be recovered with fewer primes, which allows terminating our routine earlier than expected. More details will follow in the Section on Early Termination on page 38.

3.4.4 Savings with Combined Optimizations

Combining all the optimizations together gives $e_x = (52 - 12 - 20)/2 + 1 = 11$ and $e_y = (22-10)+1 = 13$. So the number of Hessenberg calls reduces from 3201 to $11\times 13 = 143$

for the size 16 matrix (as stated previously).

Table 5.1 on page 56 contains details for the characteristic polynomial of the size 256 matrix. The table tells us $\deg_x C(\lambda, x, y) = 3072$ and $\deg_y C(\lambda, x, y) = 1024$. We also need 35 primes since the bound on $C(\lambda, x, y)$ is $\log_2(256 + 3)^{128} = 1027$ bits. So without optimizations, the routine will make $35 \times 3073 \times 1025 \approx 110$ million calls to the Hessenberg algorithm within each prime. Our routine ended up using only 14 primes instead of 35, which is $14 \times 261 \times 281 = 1,026,774$ Hessenberg calls. But we do not know in advance the number of necessary primes to recover the solution. The next CRA section will address this by introducing check primes.

3.4.5 Chinese Remainder Algorithm (CRA)

The last step of our routine involves the CRA, which was recalled in Chapter 2. So in this section, we will show how this algorithm is optimized to produce the characteristic polynomial. As seen previously, the optimizations of non-zero factors has lead to a loose coefficient bound, and that there are more primes than necessary. We show in Theorem 9 a pattern in the solution of the CRA exclusively due to this loose bound. This allows us to avoid computing additional image(s) of the characteristic polynomial, which saves time and computation power. In other words, our routine may terminate early and still return the correct answer.

Recall the solution for a system of congruences

$$u \equiv c_i \pmod{p_i} \quad \text{for } 1 \leq i \leq m$$

in the mixed radix form is

$$u = v_1 + v_2 M_1 + v_3 M_2 + \cdots + v_m M_{m-1} \tag{3.10}$$

where $M_k = \prod_{i=1}^{k} p_i$ and $0 \leq v_i < p_i$. Let $u^{(k)}$ be the truncated solution formed by $v_1, v_2, \ldots, v_k$, that is

$$u^{(k)} = v_1 + v_2 M_1 + \cdots + v_k M_{k-1}.$$

With this notation, the recurrence from Equation (2.7) becomes

$$v_i \equiv (c_i - u^{(i-1)}) M_{i-1}^{-1} \pmod{p_i}. \tag{3.11}$$

From an implementation stand point, we avoid multi-precision arithmetic for computing the products M_i by reducing them modulo a (31 bit) prime after each multiplication. Thus we only require double precision arithmetic and single precision storage to compute v_i. Fur-

thermore, no extra space is needed in this CRA stage as $0 \le c_i, v_i \le p_i$, so we may simply overwrite c_i with v_i. So after computing each image of $C(\lambda, x, y) \bmod p_1, p_2, \dots$, we build the solution in mixed radix form until it stabilizes, that is when $u = u^{(k)}$ for some integer $k \le m$. This way the algorithm may terminate much earlier when the truncated solution is correct.

Also mentioned back in Chapter 2 is the alternative Lagrange representation, where

$$u = v_1 \frac{M_m}{p_1} + v_2 \frac{M_m}{p_2} + \cdots + v_m \frac{M_m}{p_m}.$$

We are unable to find a truncated solution in this representation which can return the correct solution. But rather all values of v_i must be computed first to find the solution, thus it does not reduce any work. There is no apparent advantage with this method, thus we proceed with the mixed radix representation.

Negative Coefficients

One way to recover negative coefficients is solve for the coefficients of $C(\lambda, x, y)$ in the symmetric range, that is $-\frac{p_i}{2} < v_i < \frac{p_i}{2}$ and $-\frac{M_m}{2} < u < \frac{M_m}{2}$ where $M_m = \prod_{i=1}^{m} p_i$. Suppose k primes are sufficient, meaning $u = u^{(k)}$. Then we would have a tail of zeroes, that is $v_{k+1} = v_{k+2} = \cdots = v_m = 0$. This is because $u = u^{(k)} = u^{(k+1)}$, and so

$$u^{(k+1)} - u^{(k)} = 0 = v_{k+1} M_k \iff v_{k+1} = 0.$$

Similarly $v_{k+2} = 0$ for $u^{(k+2)}$ and so on.

Instead, we detect negative coefficients using the positive range as follows. The positive range restricts v_i such that $0 \le v_i < p_i$. The solution $u \equiv u^{(m)} \pmod{M_m}$ is also restricted such that $u^{(m)} \in [0, M_m)$. If k primes are sufficient and $u > 0$, there will be a tail of zeroes like in the symmetric case. But for $u < 0$, we discovered that there will be a tail of -1 or $p_i - 1$.

Here is an example of a system of congruences, but only two primes are sufficient to recover the negative integer $u = -7$.

$$\begin{aligned}
u &\equiv 2 \pmod{p_1 = 3} \\
u &\equiv 3 \pmod{p_2 = 5} \\
u &\equiv 0 \pmod{p_3 = 7} \\
u &\equiv 4 \pmod{p_4 = 11}
\end{aligned}$$

The symmetric range solution in mixed radix form is

$$u^{(4)} = (-1) + (-2)p_1 + 0p_1p_2 + 0p_1p_2p_3 = -7$$

with tail consisting of two zeroes. The truncated solutions in the positive range are shown below, along with how u is obtained from them.

$$\begin{aligned}
u^{(1)} &= 2 && =2 \equiv -1 \pmod{M_1 = 3} \neq u \\
u^{(2)} &= 2 + 2p_1 && =8 \equiv -7 \pmod{M_2 = 15} = u \\
u^{(3)} &= 2 + 2p_1 + 6p_1p_2 && =98 \equiv -7 \pmod{M_3 = 105} = u \\
u^{(4)} &= 2 + 2p_1 + 6p_1p_2 + 10p_1p_2p_3 && =1148 \equiv -7 \pmod{M_4 = 1155} = u
\end{aligned}$$

Notice the tail of -1 with $v_3 = 6 = p_3 - 1$ and $v_4 = 10 = p_4 - 1$. Now we prove the tail of -1 in the following theorem.

Theorem 8 (Coefficient Tail)**.** *Let $u \equiv c_i \pmod{p_i}$ for $1 \le i \le m$ and $u < 0$. Consider the solution in positive range mixed radix form, $u = v_1 + v_2p_1 + v_3p_1p_2 + \cdots + v_kp_1p_2 \ldots p_{k-1}$. If $0 < k < m$ primes are sufficient, that is $M_k = \prod_{i=1}^{k} p_i > |u|$, then $v_i = p_i - 1$ for $k < i \le m$.*

Proof. Note that $u \equiv u^{(i)} \pmod{M_i}$ for all $1 \le i \le m$. Given $|u| < M_k$ and $u < 0$, we have the first bound

$$-M_k < u < 0. \tag{3.12}$$

The truncated solution is defined by $u^{(k)} = v_1 + v_2M_1 + \cdots + v_kM_{k-1}$. The largest value it can attain is when $v_i = p_i - 1$ for $1 \le i \le k$, that is

$$\max u^{(k)} = (p_1 - 1) + (p_2 - 1)M_1 + \cdots + (p_k - 1)M_{k-1} = M_k - 1 < M_k.$$

$u^{(k)} \ge 0$ because $v_i \ge 0$ and $M_i > 0$ for all $i \ge 1$. If $u^{(k)} = 0$, then

$$0 = u^{(k)} \equiv u \pmod{M_k} \Rightarrow M_k \mid u.$$

But from Equation (3.12) we know that $-M_k < u < 0$, which contradicts the above $M_k \mid u$ and thus $u^{(k)} \neq 0$. That gives us the second bound

$$0 < u^{(k)} < M_k. \tag{3.13}$$

Combining Equations (3.12) and (3.13), the bound on their difference is

$$-2M_k < u - u^{(k)} < 0. \tag{3.14}$$

Consider the truncated solution with k primes in

$$0 > u \equiv u^{(k)} \pmod{M_k} \Longleftrightarrow u - u^{(k)} = aM_k$$

where $a \in \mathbb{Z}$. The bound on the difference from Equation (3.14) tells us

$$-2M_k < u - u^{(k)} = aM_k < 0.$$

It must be the case that $a = -1$. Now for $k+1$ primes, we have

$$0 > u \equiv u^{(k+1)} \pmod{M_{k+1}} \Longleftrightarrow u - u^{(k+1)} = bM_{k+1}$$

for some $b \in \mathbb{Z}$. The new bound on the difference is now given by

$$-2M_{k+1} < -M_k - M_{k+1} < u - u^{(k+1)} = bM_{k+1} < 0.$$

Similarly, $b = -1$. Continuing with the equality in the above bound we have:

$$\begin{aligned}
& & u - u^{(k+1)} &= bM_{k+1} \\
b = -1 &\Rightarrow & u - (u^{(k)} + v_{k+1}M_k) &= -M_{k+1} \\
u - u^{(k)} = -M_k &\Rightarrow & -M_k - v_{k+1}M_k &= -M_k p_{k+1} \\
\mid M_k &\Rightarrow & -1 - v_{k+1} &= -p_{k+1} \\
\text{solve} &\Rightarrow & v_{k+1} &= p_{k+1} - 1
\end{aligned}$$

The same would be true for the rest of the tails, thus $v_i = p_i - 1$ for $k < i \leq m$. □

Early Termination

If $k < m$ primes are sufficient to recover $C(\lambda, x, y)$, the tail in the mixed radix representation becomes a consistent value, as shown previously. With the tail being either 0 or $p_i - 1$, the solution has already been recovered, thus it is not necessary to compute another image of characteristic polynomial. Therefore we alter the algorithm into an incremental one to keep the number of required primes to a minimum.

Since we do not know in advance the minimum number of primes, we need to introduce check primes. Let $v_i = -1$ denote $v_i = p_i - 1$ for convenience. For l check primes, we require $v_{i+1} = \cdots = v_{i+l} \in \{0, -1\}$ for early termination. We use two check primes in our implementation, so the routine terminates for some $k > 1$ on the condition

$$v_k = v_{k+1} \in \{0, -1\}. \tag{3.15}$$

It is possible for the condition in Equation (3.15) to be true for some integer k, while k primes are still not sufficient. In that case the algorithm will return an incorrect answer, but this too has a low probability as we will show. There are many coefficients in $C(\lambda, x, y)$, so consider only one coefficient and the corresponding system of congruences for now.

Definition 12 (False Early Termination)**.** *Let $u \equiv c_i \pmod{p_i}$ for $1 \le i \le m$. Consider the solution in positive range mixed radix form, $u = v_1 + v_2p_1 + v_3p_1p_2 + \cdots + v_mp_1p_2 \ldots p_{m-1}$. For two check primes, false early termination occurs for some $k \in \mathbb{Z}$ when $v_k = v_{k+1} \in \{0, -1\}$ but $u^{(k)} \ne u \ne u^{(k)} - M_k$.*

Definition 13 (Number of Primes in an Interval)**.** *Let $x > y$ and $N(x, y)$ be the number of primes between the integers x and y, so*

$$N(x, y) = |\pi(x) - \pi(y)|. \tag{3.16}$$

Theorem 9 (False Early Termination Probability)**.** *Let $u \ne 0$ be the integer solution to a system of congruences $u \equiv c_i \pmod{p_i}$ for $1 \le i \le m$. Let the primes p_i be chosen randomly between 2^{q-1} and 2^q. Suppose $k < m$ primes are necessary to recover $u \in \mathbb{Z}$ in the positive range mixed radix representation as seen from Equation (3.10). False early termination with two check primes has probability*

$$\le \frac{k^2 q/(q-1)}{\binom{N(2^q, 2^{q-1})+1-k}{2}}.$$

Proof. Since k primes are necessary, $v_k \notin \{0, -1\}$. If $v_i = v_{i+1} \in \{0, -1\}$ for any $1 \le i \le k$ the algorithm will terminate and return an incorrect answer. Thus the probability of early termination is given by

$$\sum_{i=1}^{k} \Pr[v_i = v_{i+1} \in \{0, -1\}].$$

Recall that $v_1 \equiv c_1 \pmod{p_1}$ and the recurrence in Equation (3.11),

$$v_i \equiv (c_i - u^{(i-1)})M_{i-1}^{-1} \pmod{p_i}.$$

Consider when $v_1 = 0$, which happens if $c_1 \equiv u \equiv 0 \pmod{p_1} \iff p_1 \mid u$. The probability that $p_1 \mid u$ depends on their respective integer bit sizes. For convenience we define $u^{(0)} = 0$. For $1 \le i \le k$, $v_i = v_{i+1} \in \{0, -1\}$ implies either $v_i = v_{i+1} = 0$ or $v_i = v_{i+1} = -1$. Now when $v_i = v_{i+1} = 0$ we have

$$\begin{aligned} v_i = 0 &\iff u \equiv u^{(i-1)} \pmod{p_i} \iff p_i \mid u - u^{(i-1)} \quad \text{and} \\ v_{i+1} = 0 &\iff u \equiv u^{(i)} \pmod{p_{i+1}} \iff p_{i+1} \mid u - u^{(i)} = u - u^{(i-1)} - v_iM_{i-1} = u - u^{(i-1)}. \end{aligned}$$

Combining them gives

$$v_i = v_{i+1} = 0 \Longleftrightarrow p_i p_{i+1} \mid u - u^{(i-1)}. \tag{3.17}$$

Similarly, when $v_i = v_{i+1} = -1$ we have

$$\begin{aligned} v_i = -1 &\Longleftrightarrow u \equiv u^{(i-1)} - M_{i-1} \pmod{p_i} \Longleftrightarrow p_i \mid u - u^{(i-1)} + M_{i-1} \quad \text{and} \\ v_{i+1} = -1 &\Longleftrightarrow u \equiv u^{(i)} - M_i \pmod{p_{i+1}} \Longleftrightarrow p_{i+1} \mid u - u^{(i)} + M_i \\ &\qquad = u - (u^{(i-1)} + (p_i - 1)M_{i-1}) + M_i = u - u^{(i-1)} + M_{i-1}. \end{aligned}$$

Hence

$$v_i = v_{i+1} = -1 \Longleftrightarrow p_i p_{i+1} \mid u - u^{(i-1)} + M_{i-1}. \tag{3.18}$$

Consider the maximum number of divisors of the integer on the right side in Equation (3.18), which is

$$\begin{aligned} p_i p_{i+1} \mid u - u^{(i-1)} + M_{i-1} &= M_{i-1} + v_i M_{i-1} + v_{i+1} M_i + \cdots + v_k M_{k-1} \\ &= M_{i-1}(1 + v_i + v_{i+1} p_i + \cdots + v_k p_i \ldots p_{k-1}) \\ &= M_{i-1}\ S. \end{aligned}$$

Clearly $p_i p_{i+1} \nmid M_{i-1} = p_1 \ldots p_{i-1}$, so we look at whether or not

$$p_i p_{i+1} \mid S = 1 + v_i + v_{i+1} p_i + \cdots + v_k p_i \ldots p_{k-1} \leq p_i \ldots p_k < 2^{(k-i+1)q}.$$

Note that the case in Equation (3.17) is included, since the right-most side of Equation (3.18) is greater than that of (3.17). By standard arithmetic, the number of prime pairs $p_i p_{i+1} > 2^{2(q-1)}$ that divide an integer $S < 2^{(k-i+1)q}$ is

$$\leq \lfloor \log_{2^{2(q-1)}} 2^{(k-i+1)q} \rfloor = \frac{(k-i+1)q}{2q-2}.$$

The number of primes in the between 2^{q-1} and 2^q is given by $N(2^q, 2^{q-1})$ from Equation (3.16). With two check primes, the number of distinct prime pairs p_i, p_{i+1} is given by

$$\bar{N}_i = \binom{N(2^q, 2^{q-1}) + 1 - i}{2},$$

a binomial coefficient. The addition of $1 - i$ is to account for repeated primes. So the probability that $v_i = v_{i+1} \in \{0, -1\}$ satisfies

$$\Pr[v_i = v_{i+1} = 0] \leq \Pr[v_i = v_{i+1} = -1] \leq \frac{(k-i+1)q/(2q-2)}{\bar{N}_i} \leq \frac{1}{2}\ \frac{kq/(q-1)}{\bar{N}_k}.$$

Combining both cases gives

$$\Pr[v_i = v_{i+1} \in \{0, -1\}] = \Pr[v_i = v_{i+1} = 0] + \Pr[v_i = v_{i+1} = -1] \leq \frac{kq/(q-1)}{\bar{N}_k}.$$

Therefore the probability of false early termination is

$$\sum_{i=1}^{k} \Pr[v_i = v_{i+1} \in \{0, -1\}] \leq \frac{k^2 q/(q-1)}{\bar{N}_k}.$$

□

In our implementation we use primes between 2^{30} and 2^{31}. The number of primes by the prime number theorem is estimated to be over 50 million by Equation (2.9). We have implemented a primality test and found that the number of primes in that range is $N(2^{30}, 2^{31}) = 50,697,537$. For the size 256 matrix, the result was computed with $k = 12$ primes (excluding two check primes) with optimization instead of $m = 35$ (by Equation (3.2)). So for $q = 31$ and $k = 12$ the probability of false early termination is

$$\leq \left(\frac{k^2 q/(q-1)}{\bar{N}_k}\right) = 0.12 \times 10^{-12} < 0.1 \times 10^{-11}$$

for one system of congruences. Now we take into account all the integer coefficients in $C(\lambda, x, y)$. Recall the number of evaluation points e_x, e_y as defined on page 31. Their product $e_x e_y$ is an upper bound on the number terms in $c_i(x, y)$, the coefficient of λ^i. Thus the maximum number of terms in $C(\lambda, x, y)$ is bounded by $n e_x e_y$. For the size $n = 256$ matrix, the values are $e_x = 261$ and $e_y = 281$. So the overall probability of false early termination is

$$\leq n e_x e_y \left(\frac{k^2 q/(q-1)}{\bar{N}_k}\right) = 0.22 \times 10^{-5}.$$

Chapter 4

Implementation

In this chapter, we discuss implementation details such as data structures, parallel algorithm and memory usage. Our entire routine is implemented in C code, and the parallel version in Cilk C.

4.1 Data Structures

Matrix

The matrices of interest only contain monomials in two variables. To store the matrix, a simple way would be to use array of arrays, or double arrays. Arrays in C are closely related to pointers, as we use pointers to point to a location of memory where we can store array elements. The `malloc()` command allocates memory based on the input and `sizeof()` returns number of bytes for a type. The standard 32 bit integers have four bytes, and pointers typically have eight bytes. For dimension n square matrix, it is defined by

```
int ** M = malloc( n * sizeof( int * ) );
for( int i = 0 ; i < n ; i++ )
    M[i] = malloc( n * sizeof( int ) );
```

Since pointers in C start from zero, the element of the ith row and jth column will be stored in `M[i-1][j-1]`. The exponents in the matrices are relatively small, as each integer exponent can be stored with much less than 16 bits. So to prevent having another double pointer, we pack the exponents into one integer. We may store it by shifting one of the exponents by 16 bits, achieved by `«16` in C. Given the matrix element $x^a y^b$, the exponents are stored as

```
M[i-1][j-1] = (b<<16) + a;
```

Retrieving the exponents will require bitwise `AND` and shifting in the other direction. For convenience, we define the macro `H16` and show the extraction as follows.

```
#define H16 (1<<16)-1 // 16 tuple of 1s in binary
M[i-1][j-1] &H16; // to extract a
M[i-1][j-1] >>16; // to extract b
```

Characteristic Polynomial

The result $C(\lambda, x, y)$ involves three variables, and it is computed from m images (m primes). Thus we need to store m copies of the characteristic polynomial, one for each prime. We will need triple pointer, the first index for prime, second index for the degree of λ, and third index for coefficient. The degrees in x and y also fit within 16 bits, thus we may pack them into one integer size like the matrix entries. We use another double pointer to store the monomials, where first index corresponds to degree of λ. The second index stores the monomial that corresponds to the coefficient in the third index of the triple pointer. We show the data structure here, where `C` stores the coefficients and `D` stores the monomial degrees.

```
int *** C = malloc( m * sizeof( int ** ) ); /* m primes */
for( int k = 0 ; k < m ; k++ )
{
    C[k] = malloc( n * sizeof( int * ) ); /* degree n for lambda */
    for( int i = 0 ; i < n ; i++ )
    {   /* at most u terms for each lambda */
        C[k][i] = malloc( u * sizeof( int ) );
        for( int j = 0 ; j < u ; j++ )
            C[k][i][j] = ...;
    }       /* jth term coefficient of lambda^i, in kth prime */
}
int **  D = malloc( n * sizeof( int ** ) ); /* degree n for lambda */
for( int i = 0 ; i < n ; i++ )
{   /* at most u terms for each lambda */
    D[i] = malloc( u * sizeof( int ) );
    for( int j = 0 ; j < u ; j++ )
        D[i][j] = (deg_y << 16 ) + deg_x;
}       /* jth term monomial of lambda^i */
```

To print the jth coefficient and monomial of λ^i in image mod p_k in C, use the following command.

```
printf(" %d * x^%d * y^%d \n", C[k][i][j] , D[i][j]&H16, D[i][j]>>16 );
```

4.2 Parallelization

The modular algorithm was originally chosen since the computation for each prime can be done in parallel. Each evaluation, each Hessenberg call, each interpolation and each CRA may also be computed in parallel. We chose to run each prime sequentially and look to parallelize within each prime for two reasons. First, we do not know how many primes are necessary because of the loose bound from Section 3.4.3. Thus parallizing within each prime suits the incremental algorithm better. Second, memory may become an issue for computers with limited RAM.

The algorithm starts with two queries, which is done in parallel, as well as the work within each query. Within each prime in phase 2, the algorithm does the x computations in parallel, namely steps 1a) and 1b) (from Section 3.1). After each prime, the incremental CRA involves many coefficients, and they are computed in parallel in batches. Overall, the most significant speed up is with parallel Hessenberg algorithm in x. Queries, interpolation and incremental CRA only comprise of a relatively small amount of the total time. But for larger matrices, they give a good local speed up. So our implementation does the queries, x computations and CRA in parallel.

A portion of the code with main headers is shown starting on page 49. Note the locations of `spawn` and `sync` to see the parallelism. These are Cilk commands to start a new task and wait for all running tasks to finish before continuing, respectively.

For a glimpse of the impact of utilizing additional cores, please see the timings in Table 5.1 on page 56. For the smallest size 16 matrix, there is not enough work to see any significant speed up. The larger matrices show a speed up that is near the theoretical maximum, which will be discussed with more details in the next chapter.

4.3 Space

In this section we talk about how much space and memory are required to run the computations. We will start by counting all the storage required in terms of 32 bit integers, and then total them at the end. Figure 4.3 will often be referenced, and it is on page 51.

Matrix

The input square matrix are stored using two double array of integers, for coefficients and monomials. So the number of integers for a n by n matrix in this case is

$$2n^2. \tag{4.1}$$

Characteristic Polynomials

Suppose the modular algorithm terminates after m primes. We will need an array of m integers to store all the primes. Recall the number of evaluation points for two variables are e_x and e_y. Each coefficient of λ in $C(\lambda, x, y) \mod p$ has at most $u = e_x e_y$ integers. Each characteristic polynomial mod a prime will need $ne_x e_y$ integers. The corresponding monomials also require $ne_x e_y$ integers, but this does not depend on the prime number. Thus for a size n matrix and m primes, the number of integers needed is

$$m + mne_x e_y + ne_x e_y. \tag{4.2}$$

The allocation of the monomials can be found on line 31 in Figure 4.3. The allocation of the coefficients is stated on line 37.

Query

We make two queries, and each returns a characteristic polynomial over $\mathbb{Z}_p$. The polynomials will have degree n in λ. Since no optimizations are involved here, we must use the degree bound $D = n \max(d_x, d_y) + 1 \geq \max(D_x, D_y) + 1$ (previously defined on page 19).

In each query, space is needed for working memory which stores evaluation points, integer matrices and characteristic polynomial images. For evaluation points, D integers are needed.

Each integer matrix consists of n^2 integers. The Hessenberg algorithm involves a recurrence starting with one integer in $C_0(\lambda) = 1$, to $n+1$ integers in $C_n(\lambda)$. So the intermediate polynomials within the Hessenberg can be stored in a "triangle", and the number of integers is $1 + 2 + \cdots + (n+1) = \frac{1}{2}(n+1)(n+2)$. To store the "triangle" in a single array, we use two other integer arrays to index the position and degree. That is an additional $2n$ integers, which we choose to use the stack memory instead of allocating memory for more efficiency. $C_n(\lambda)$ is then copied else where so that the matrix and triangle space can be re-used for the next evaluation point. We may ignore the leading term of λ^n since it has unit coefficient. Thus we require $n \times D$ integers to store all images of $C_n(\lambda)$ prior to interpolation.

Next we interpolate in either variable, then store the polynomial in a double array so the number of integers is bounded by $(n+1)D$. For interpolation itself, we require an additional temporary space of $2D$ integers. As mentioned earlier, both queries along with work inside each query are done in parallel. If N cores are used, the total number of integers for two queries is given by

$$2D + 2(n+1)D + 2\left(n^2 + 2n + \frac{1}{2}(n+1)(n+2) + 2D\right)N. \tag{4.3}$$

Line 5 of Figure 4.3 contains some space allocations for the queries. After computing the factors, the query space is freed as indicated on line 11. The rest of the space allocations are omitted as they are in other function calls.

Factors

The query phase continues by computing f_i, g_i (the lowest degrees) and $\bar{f}_i, \bar{g}_i$ (the highest degrees) for each $c_i(x, y)$. They are computed in line 9 of Figure 4.3. These four values are then stored in an array for further processing. The largest degree is still under 16 bits, thus we pack these four values to save a bit of space. We store the degrees in x as is, and degrees in y shifted by 16. Two arrays of size n are used, so the number of integers is

$$2n. \tag{4.4}$$

Fixed Working Memory - Evaluation/Interpolation Points

We require e_x and e_y integers for evaluation. Our implementation also uses another e_x integers to store the square of each, to avoid re-computation. So the number of evaluation points uses $2e_x + e_y$ integers. This memory is allocated in the pointer `T0` on line 14 of Figure 4.3.

To recover $C(\lambda, x, y)$ mod a prime, we need to store the coefficients of $C(\lambda, \alpha_i, \beta_j)$, for $1 \leq i \leq e_x$ and $1 \leq j \leq e_y$. Ignoring the leading unit coefficient, the polynomial $C(\lambda, \alpha_i, \beta_j)$ mod a prime has n integers. Thus all the images of the characteristic polynomial take up ne_xe_y integers. Even though that space does not depend on the number of cores, it is computed in parallel because we evaluate x in parallel. Its allocation is in the triple pointer `T3` on line 28 in Figure 4.3. The integer count for the fixed working memory is

$$2e_x + e_y + ne_xe_y. \tag{4.5}$$

Parallel Working Memory - Hessenberg Algorithm

An integer matrix requires n^2 integers. Its characteristic polynomial has degree n, that is n integers if we ignore the leading term. As stated previously for the Hessenberg algorithm space required in the query phase, it involves an integer matrix, a "triangle" and some stack space. The same amount of space is needed here, and with N cores in parallel the integer amount is

$$\left(n^2 + 2n + \frac{1}{2}(n+1)(n+2)\right) N. \tag{4.6}$$

In Figure 4.3, the triangle space is allocated in the pointer `T1` on line 17. The matrix space is allocated in the pointer `T2` on line 22.

Parallel Working Memory - Interpolation Values

To interpolate y with each $x = \alpha_i$, we must store $C(\lambda, \alpha_i, \beta_j)$ where $1 \leq j \leq e_y$. That is e_y polynomials, or $(n+1)e_y$ integers which is needed to interpolate for $C(\lambda, \alpha_i, y)$. After interpolating with e_y points, the maximum number of terms/integers in $C(\lambda, \alpha_i, y)$ is $(n+1)e_y$. Ignoring the leading unit coefficient, the maximum number of integers becomes ne_y for every core. This space is allocated in line 22 of Figure 4.3.

Furthermore, interpolation on y requires e_y integers as temporary space. Interpolation on x also requires $3e_x$ additional integers in our implementation. The first set of e_x integers is required for optimization purposes. Another set stores the polynomial before it is copied to the answer space. The third set is temporary space for interpolation. So the total integer count for the images and interpolation (with N cores) is

$$(ne_y + 3e_x + e_y)\, N. \tag{4.7}$$

The allocation for $3e_x + ey$ is on line 17, where we take the maximum and multiply by four.

Total

The fixed memory includes the input matrix, primes with characteristic polynomial images, factors and evaluation values. Their respective integer count is given by Equations (4.1), (4.2), (4.4) and (4.5). So the combined total integer count comes to

$$2n^2 + m + mne_xe_y + ne_xe_y + 2n + 2e_x + e_y + ne_xe_y \in O(n^2 + mne_xe_y).$$

None of this amount is freed until the routine terminates. We label the rest as variable memory, as it includes the query space which is freed during the routine. It also includes parallel working memory, which depends on the number of cores used.

Theorem 10 (Variable Memory)**.** *The variable memory includes space needed for query and parallel working working memory. Combining Equations* (4.3)*,* (4.6) *and* (4.7)*, the integer count with N cores does not exceed*

$$2(n+2)D + \left(4D + 3(n^2 + 2n + \frac{1}{2}(n+1)(n+2)) + ne_y + 3e_x + e_y\right) N$$

$$\in O(nD) + NO(D + n^2 + ne_y).$$

For the largest matrix, we have dimension $n = 256$, $m = 14$ primes (including two check primes), $e_x = 261$ and $e_y = 281$. The integer size in C has 32 bits, or 4 bytes. The total fixed memory comes to about 1.2 gigabytes. The variable memory about 10 megabytes per core, including the stack memory of about 2 kilobytes per core.

4.4 Prime Numbers

Our routine generates random primes in the range $(2^{30}, 2^{31})$. First, we generate a random odd number r such that $0 < r < 2^{30}$. Then $p = r + 2^{30}$ is a random value in the desired range of $2^{30} < p = r + 2^{30} < 2^{31}$. Now we use the Miller-Rabin test to ensure p is a prime. Our algorithm is incremental, so we generate as many primes as needed until we terminate.

4.5 Partial Code

In the next three pages, we show the main functions of our routine in C code. The function calls start from bottom to top, as seen in the following diagram.

The routine begins when the function `findcharpoly` is called, in Figure 4.3. After making the queries and allocating sufficient memory space, the incremental algorithm begins in the loop on line 34 (of Figure 4.3). In Figure 4.2, there are two function calls in `bvmxhess`. One eventually goes to the Hessenberg algorithm and the other deals with interpolation on x. Continuing with the first function call of `uvmxhess` brings us to Figure 4.1, which also has two function calls. Similarly, the two function calls deal with he Hessenberg algorithm and interpolation on y. In `parchpy`, the double loop evaluates the matrix monomial entries with the square and multiply algorithm in $\mathbb{Z}_p$. After evaluation, the Hessenberg algorithm is applied in two stages with two function calls. We only show the function call `inthesschpy` for the second stage, as it involves additional stack memory.

The proper commands in Cilk Plus are `_Cilk_spawn` and `_Cilk_sync`. Another important Cilk command to to obtain the number of available cores. We rename them using the following macros which are the original names used in Cilk C.

```
#define  spawn  _Cilk_spawn
#define  sync   _Cilk_sync
#define  NCORE  __cilkrts_get_nworkers()
```

To see the parallelism, note their locations in Figure 4.2. Lines 29 and 38 contain `spawn`. Lines 31 and 40 contain `sync`.

The stack memory allocation is found on line 4 in Figure 4.1. The other memory allocations are shown in Figure 4.3. We call `malloc` through the following commands.

```
int *   intarray( int n ) { return malloc( n * sizeof( int    ) ); }
int **  ptrarray( int n ) { return malloc( n * sizeof( int *  ) ); }
int *** mtxarray( int n ) { return malloc( n * sizeof( int ** ) ); }
```

These commands are found on lines 5, and 14 through 37 (in Figure 4.3).

Figure 4.1: Main Functions in C Code (part 1/3).

```
 1 /* ##### Hessenberg algorithm part 2 - Recurrence ##### */
 2 int inthesschpy( int ** H, int n, int * C, int p )
 3 {
 4     int D[n], T[n]; /* Stack memory */
 5     ... /* Computes C(lambda, alpha_i, beta_j) mod p */
 6 }
 7
 8 /* ##### Integer Hessenberg - Compute characteristic polynomial ##### */
 9 void parchpy( int ** M, int ** H, int * T, int n, int e, int f,
10               int * S, int p )
11 {
12     for( int i = 0 ; i < n ; i++ )
13     for( int j = 0 ; j < n ; j++ )
14     {
15         int dx = powmod(e, M[i][j]&H16,p);
16         int dy = powmod(f, M[i][j]>>16,p);
17         H[i][j] = (long) dx * dy % p;
18     }
19     inthessform( H, n, p ); /* Hessenberg Part 1 - Decomposition */
20     int d = inthesschpy( H, n, T, p ); /* Part 2 - Recurrence */
21     for( int i = 0 ; i < n ; i++ ) S[i] = T[d+i];
22 }
23
24 /* ##### Univariate Hessenberg ##### */
25 void uvmxhess( struct mvpolmx M, int e, int * L, int ey, int ** C,
26                int p, int * T0, int * T1, int ** T2 )
27 {   ... /* Declarations */
28     int i, n, d;
29     n = M.n; d = M.d;
30 /*  Evaluate y (done previously) */
31     int * F = T0;
32 /*  Integer Hessenberg */
33     int ** H = T2; int ** S = T2+n; int *  T = T1;
34     for( i = 0 ; i < ey ; i++ )
35         parchpy( M.M, H, T, n, e, F[i], S[i], p );
36 /*  Interpolate y */
37     for( i = 0 ; i < n ; i++ )
38         parnerpy( S, F, C[i], ey, i, L[i]>>16, p, T1+ey );
39 }
```

Figure 4.2: Main Functions in C Code (part 2/3).

```
1 /* ##### Bivariate Hessenberg ##### */
2 void bvmxhess( struct mvpolmx M, int * L, int c, int ** C, int ** D,
3      int p, int * T0, int ** T1, int *** T2, int *** T3, int N )
4 {
5     int i, j, k, n, ex, ey;
6     n = M.n;
7     ex = c&H16;
8     ey = c>>16;
9
10 /*  Evaluation points: x */
11     int * E = T0;
12     int * F = T0+ex;
13     for( i = 0 ; i < ex ; i++ )
14     {
15         E[i] = i+2;
16         F[i] = (long) E[i] * E[i] % p;
17     }
18
19 /*  Evaluation points: y */
20     int * G = T0+2*ex;
21     for( i = 0 ; i < ey ; i++ )
22         G[i] = i+2;
23
24 /*  Univariate Hessenberg */
25     int *** I = T3;
26     for( i = 0 ; i < ex ; i+=N )
27     {
28         for( j = 0 ; j < N && i+j < ex ; j++ )
29             spawn uvmxhess( M, E[i+j], L, ey, I[i+j],
30                             p, G, T1[j], T2[j] );
31         sync;
32     }
33
34 /*  Interpolate x */
35     for( i = 0 ; i < n ; i+=N )
36     {
37         for( j = 0 ; j < N && i+j < n ; j++ )
38             spawn parnerpx( I, E, F, L[i+j], i+j,
39                             ex, ey, C[i+j], D[i+j], p, T1[j] );
40         sync;
41     }
42 }
```

Figure 4.3: Main Functions in C Code (part 3/3).

```
 1 /* ##### Combines everything ##### */
 2 int findcharpoly( struct mvpolmx M, int m, int*s , int*NC )
 3 {     ... /* Declarations */
 4     /* Query for compression */
 5     int ** X = ptrarray( M.n ); int ** Y = ptrarray( M.n );
 6     spawn uvmxhessxx( M, rng(2147483647), X, NCORE );
 7     spawn uvmxhessyy( M, rng(2147483647), Y, NCORE );
 8     sync;
 9     int t = cmps(X, Y, A.G, A.F, M.n, M.d); s[0] = t;
10     int ex = t&H16, ey = t>>16; int u = ex * ey;
11     ...  /* Free X and Y */
12     /* Working Memory */
13     int N = NCORE; NC[0] = N; /* Number of cores */
14     int * T0 = intarray(2*ex + ey);
15     int ** T1 = ptrarray(N); /* N x linear mem: triangle & interp temp */
16     for( int i = 0 ; i < N ; i++ )
17         T1[i] = intarray(4 * max(ex,ey) + (M.n+1)*(M.n+2)/2);
18     int *** T2 = mtxarray(N); /* N x rectangle mem: int mtx & charpoly */
19     for( int i = 0 ; i < N ; i++ )
20     {
21         T2[i] = ptrarray(M.n+ey);
22         for( int j = 0 ; j < M.n+ey ; j++ ) T2[i][j] = intarray(M.n+1);
23     }
24     int *** T3 = mtxarray(ex); /* Interpolate charpoly: ex * n * ey */
25     for( int i = 0 ; i < ex ; i++ )
26     {
27        T3[i] = ptrarray(M.n);
28        for( int j = 0 ; j < M.n ; j++ ) T3[i][j] = intarray(ey);
29     }
30     A.D = ptrarray(M.n); /* Answer Memory - Monomials */
31     for( int i = 0 ; i < M.n ; i++ ) A.D[i] = intarray(u);
32     /* Modular algorithm */
33     int k = 0;
34     for( int i = 0 ; i < m ; i++ )
35     {
36         A.I[i] = ptrarray(M.n); /* Answer Memory - Coefficients */
37         for( int j = 0 ; j < M.n ; j++ ) A.I[i][j] = intarray(u);
38         apprime(A.P, i, 30); /* Generate prime and append */
39     /* Bivariate Hessenberg */
40         bvmxhess( M, A.F, t, A.I[i], A.D, A.P[i], T0, T1, T2, T3, N );
41     /* Incremental CRA */
42         k = max( k, cramixrad(A.I, A.P, i, M.n, u) );
43         if( k == i ) break; /* Early termination (2 check primes) */
44     }
45 }
```

Chapter 5

Output

In this chapter, we show the correctness of our characteristic polynomials, compare our implementation with current software and generalize our routine to multivariate polynomial entries. Our routine has generated two characteristic polynomials which cannot be computed with current CAS. Due to their immense sizes, we only show part of the results in Appendix A3 and A4.

5.1 Validation

Let $C(\lambda, x, y) = \det(\lambda I_n - A(x, y))$. Suppose our routine outputs $S(\lambda, x, y) \in \mathbb{Z}[\lambda, x, y]$. Since we optimized our routine to involve queries and early termination, it may happen that $S(\lambda, x, y) \neq C(\lambda, x, y)$. Let the error be $E(\lambda, x, y) = C(\lambda, x, y) - S(\lambda, x, y)$. To detect $E(\lambda, x, y) \neq 0$, we add an additional validation phase to our routine after computing the stabilized solution.

Only for the smaller matrices, sizes $n \in \{16, 32, 64\}$, is Maple able to compute $C(\lambda, x, y)$. So we also verify the correctness of our output $S(\lambda, x, y)$ by reading it onto Maple, and checking that $C(\lambda, x, y) - S(\lambda, x, y) = 0$.

Our additional validation phase works as follows. In outline, we choose 10 verifying primes randomly from a large set, then for each verifying prime q we choose $\gamma, \alpha, \beta \in \mathbb{Z}_q$ randomly and check if

$$E(\lambda = \gamma, x = \alpha, y = \beta) \bmod q = \det(\gamma I_n - A(\alpha, \beta)) - S(\gamma, \alpha, \beta) \bmod q = 0.$$

We continue to use our random prime generator so that the verifying primes are in the range $2^{30} < q < 2^{31}$. Let k be the number of primes used to compute $S(\lambda, x, y)$ including the two check primes. The verifying primes are $q \in \{p_{k+1}, \ldots, p_{k+10}\}$, distinct from the

ones used to compute $S(\lambda, x, y)$.

Our routine computes $S(\lambda, x, y)$ in mixed radix form where each integer coefficient u is written as $u = v_1 + v_2 p_1 + v_3 p_1 p_2 + \cdots + v_k p_1 p_2 \ldots p_{k-1}$. After computing $S(\lambda, x, y)$, we evaluate it in C code to obtain $S(\gamma, \alpha, \beta) \bmod q$ from the mixed radix form. Then we compute $\det(\gamma I_n - A(\alpha, \beta)) \bmod q$ independently by Gaussian elimination, also with C code. We computed for the 10 verifying primes that $E(\gamma, \alpha, \beta) \bmod q = 0$. If our solution $S(\lambda, x, y)$ is incorrect, there are two possibilities where the error modulo q is zero. The first case is if the prime q divides all the integer coefficients in $E(\lambda, x, y)$. The second case is when $E(\lambda, x, y) \bmod q \neq 0$ and the random triplet is a root of $E(\lambda, x, y) \bmod q$.

For the first case, if every coefficient in $E(\lambda, x, y)$ is some multiple of the prime q, then $E(\gamma, \alpha, \beta) \bmod q$ will be zero regardless of the random evaluation points γ, α, β. Whether a prime q divides the error depends on the size of the prime q and the integer coefficients of $E(\lambda, x, y)$. Recall the coefficient bound $||C(\lambda, x, y)||_\infty \leq (n+3)^{n/2}$ in Equation (3.1) on page 25. Our routine terminates after computing with k 31-bit primes. So the coefficient bound on our output is $||S(\lambda, x, y)||_\infty \leq 2^{31k}\binom{h}{\lfloor h/2 \rfloor}$ where $h = \max h_i$ comes from the factor $(x^2-1)^{h_i}$. Then the bound on the error is

$$||E(\lambda, x, y)||_\infty \leq E_b = ||C(\lambda, x, y)||_\infty + ||S(\lambda, x, y)||_\infty = (n+3)^{n/2} + 2^{31k}\binom{h}{\lfloor h/2 \rfloor}. \quad (5.1)$$

Since our primes are $> 2^{30}$, there can be at most $\lfloor \log_{2^{30}} E_b \rfloor$ primes that divide the error.

For the largest $n = 256$ case, our routine used $k = 14$ primes to compute the solution and $h = \max h_i = 1024$. The maximum number of 31-bit primes which can divide the error is $\lfloor \log_{2^{30}} E_b \rfloor = 48$. Back in Chapter 3, we calculated the number of primes in our specified range, which is $N(2^{30}, 2^{31}) = 50,697,537$. So of the available number of primes, at most 48 primes cannot be chosen. Thus the probability of a verifying prime dividing the error is given by

$$\Pr[q \mid E(\lambda, x, y)] \leq \frac{48}{N(2^{30}, 2^{31}) - k} < 10^{-5}.$$

Repeated for the 10 verifying primes, the overall probability is

$$\prod_{i=1}^{10} \Pr[p_{k+i} \mid E(\lambda, x, y)] = \prod_{i=1}^{10} \left(\frac{48}{N(2^{30}, 2^{31}) - (k+i)} \right) < 10^{-59}.$$

For the second case, the error is zero if the random triplet happens to be a root of $E(\lambda, x, y) \bmod q$. We seek to bound $\Pr[E(\gamma, \alpha, \beta) \bmod q = 0]$ by using the Schwartz–Zippel Lemma.

Theorem 11 (Schwartz-Zippel Lemma [19])**.** *Let $f(x_1, \ldots, x_n)$ be a non-zero multivariate polynomial over a field F with total degree $d \geq 0$. Let $\alpha_1, \ldots, \alpha_n$ be n elements chosen randomly and independently from a finite subset $S \subseteq F$. Then*

$$\Pr[f(\alpha_1, \ldots, \alpha_n) = 0] \leq \frac{d}{|S|}.$$

Total degree bounds for the characteristic polynomials are given in Table 3.1 on page 25. Our field is $F = \mathbb{Z}_q$, where $2^{30} < q < 2^{31}$.

For the size $n = 256$ problem, we have total degree $\deg S(\lambda, x, y) = 4096$. Thus we have $\deg E(\lambda, x, y) \leq \max(\deg C, \deg S) = 4096$. So if our solution is incorrect, the probability that $E(\gamma, \alpha, \beta) \bmod q = 0$ is at most $\frac{4096}{p} < \frac{4096}{2^{30}} < 0.4^{-5}$. This is repeated for the 10 different verifying primes, thus the overall probability becomes at most

$$\left(\frac{4096}{q}\right)^{10} < \left(\frac{4096}{2^{30}}\right)^{10} < 10^{-53}.$$

The total time spent for validation in C code was about 7.6 seconds (see Appendix C).

Before adding a validation phase into our routine, our first attempt was to use Maple to read and validate our solutions $S(\lambda, x, y)$ for $n \in \{128, 256\}$. We picked the verifying primes $Q > \frac{1}{2}||E(\lambda, x, y)||_\infty$, so that it is not possible for the prime to divide the error. For the largest case, we verified that $E(\gamma, \alpha, \beta) \bmod Q = 0$ for 10 distinct primes $Q > 2^{1027}$. For these large primes Q, it took about three minutes each. If our solution is incorrect, the probability of obtaining zero is $\left(\frac{4096}{Q}\right)^{10} < \left(\frac{4096}{2^{1027}}\right) < 10^{-3000}$. Therefore, the output of our routine $S(\lambda, x, y)$ is correct with probability $> 1 - 10^{-3000}$.

Proof of Correctness

Our current validation phase with 10 runs only shows the output $S(\lambda, x, y)$ is correct with high probability. Here we outline a method to prove that the error $E(\lambda, x, y) = 0$, and hence the output is correct. We treat the error $E(\lambda, x, y)$ as a black box multivariate polynomial, where bounds on the coefficients and each degree are known. To prove that the error is zero, we interpolate the error for each variable to the corresponding degree bound. That also needs to be repeated for enough primes to account for the integer coefficient bound. If the errors at all the evaluation points and for all primes are zero, the interpolated result will also be zero and the error is zero.

In our validation phase, we compute the error for primes in the range $2^{30} < q < 2^{31}$ by

$$E(\lambda = \gamma, x = \alpha, y = \beta) \bmod \; = S(\gamma, \alpha, \beta) - \det(\gamma I_n - A(\alpha, \beta)) \bmod q.$$

For the size $n = 256$ characteristic polynomial, the degree bounds are $\deg_\lambda E(\lambda, x, y) \leq 256$, $\deg_x C(\lambda, x, y) \leq 3072$ and $\deg_y C(\lambda, x, y) \leq 1024$. Recall E_b, the bound on the error as defined in Equation (5.1). The error size is at most $\lceil \log_{2^{30}} E_b \rceil = 49$. Hence, 49 30-bit primes are sufficient to recover the integer coefficients in $E(\lambda, x, y)$ by the CRT. Thus to prove $E(\lambda, x, y) = 0$ for $n = 256$, we need to compute the error

$$(256 + 1) \times (3072 + 1) \times (1024 + 1) \times 49 \approx 40 \times 10^9 = 40 \text{ billion}$$

times and ensure they are all zero.

In Appendix C, the time to verify for 10 primes took 7.6 seconds, so each verification is about 0.76 seconds. To prove that the error is zero for the size $n = 256$ largest characteristic polynomial, we estimate the time to be $40 \times 10^9 \times 0.76$ seconds, which is more than 900 years.

The time may be reduced by a factor of $\deg_\lambda E(\lambda, x, y) + 1 = 257$ if we do not evaluate λ. Namely, we evaluate our output for $S(\lambda, x = \alpha, y = \beta) \bmod q$ and use the Hessenberg algorithm to compute $\det(\lambda I_n - A(\alpha, \beta)) \bmod q$. Then check whether or not

$$E(\lambda, x = \alpha, y = \beta) \bmod q = 0.$$

5.2 Benchmarks

On the next page, Table 5.1 consists of timings of our modular routine. Column `min` is the minimum number of 31 bit primes needed to recover the integer coefficients in $C(\lambda, x, y)$. Column `bnd` is the maximum number of primes needed based on Equation (3.2), which is calculated from the bound on $||C(\lambda, x, y)||_\infty$. The number of calls to the Hessenberg algorithm is $(e_x e_y)(\min +2)$ since our implementation uses two check primes. Table 5.2 includes data for Maple 2016 and Magma V2.22-2 (see Appendix B for code). Table 5.3 shows the time distribution of the algorithm for the case of $n = 256$. We also show the program output for the case of $n = 256$ in Appendix C, which contains more information on the timings, prime numbers and output from validation. The names and details of machines we ran our routine on are given below and they all run Fedora 24.

- sarah: Intel Core i5-4590 quad core at 3.3 GHz (3.7 GHz turbo), 8 GB RAM
- mark: Intel Core i5-4670 quad core at 3.4 GHz (3.8 GHz turbo), 16 GB RAM
- luke: AMD FX8350 eight core at 4.0 GHz (4.2 GHz turbo), 32 GB RAM
- ant: Intel Core i7-3930K six core at 3.2 GHz (3.8 GHz turbo), 64 GB RAM

Our New Bivariate Routine										
Size	#points	#primes	sarah		mark		luke		ant	
n	e_x, e_y	min,bnd	1 core	4 cores	1 core	4 cores	1 core	8 cores	1 core	6 cores
16	11,13	1, 2	0.06 s	0.02 s	0.06 s	0.03 s	0.05 s	0.02 s	0.10 s	0.03 s
32	28,31	1, 3	1.06 s	0.29 s	1.03 s	0.28 s	1.09 s	0.17 s	1.12 s	0.34 s
64	67,61	3, 7	36.5 s	9.9 s	35.5 s	9.6 s	42.1 s	6.3 s	37.2 s	6.8 s
128	131,141	5, 16	22.1 m	5.9 m	21.5 m	5.7 m	31.9 m	4.6 m	23.3 m	4.1 m
256	261,281	12, 35	21.6 h	5.7 h	20.9 h	5.6 h	34.5 h	4.9 h	22.7 h	3.9 h

Table 5.1: Bivariate routine timings in seconds (s), minutes (m) or hours (h).

	Maple								Magma
Size	sarah		mark		luke		ant		ant
n	real	cpu	real	cpu	real	cpu	real	cpu	cpu
16	0.28 s	0.30 s	0.21 s	0.23 s	0.46 s	0.53 s	0.32 s	0.36 s	0.32 s
32	34.1 s	45.2 s	30.2 s	41.1 s	50.3 s	83.7 s	32.7 s	46.3 s	99.7 s
64	19.9 h	32.6 h	12.1 h	23.2 h	3.63 h	5.42 h	2.86 h	3.91 h	15.1 h
128	Not available								
256	Not available								

Table 5.2: Maple and Magma timings in seconds (s), minutes (m) or hours (h).

Query	Evaluation	Hessenberg	Interpolation	CRA	Validation
$< 4\%$	$\ll 1\%$	$> 95\%$	$< 2\%$	$< 1\%$	$\ll 1\%$

Table 5.3: Time spent in parallel algorithm for $n = 256$.

Comparison

Recall from Chapter 2, the Bareiss fraction–free algorithm and the Berkowitz method are used by Magma and Maple, respectively. The Bareiss algorithm in Magma is implemented in C code [20], as is ours, thus providing a fair comparison to our routine. For the Berkowitz method on Maple, only some portions of its implementation are done in C code, namely for polynomial multiplication. But the matrix entries are polynomials, most of the computation time is used to multiply and expand polynomials. That is accomplished by the `expand()` command, which is implemented in C code. Thus comparing our routine to that of Maple's is still valid. It is also worth noting that Maple computes with some parallelism, as the cpu usage is occasionally greater than 100% during the program execution.

As reflected by the timings in Table 5.2, the quartic division–free algorithm is faster than the cubic fraction-free algorithm since the latter involves large polynomial divisions. The parallelism in Maple's Berkowitz implementation is also apparent as seen in the different times between cpu and real time. But given the available cores for use, the speed up is not very good. For example on `ant` with 6 cores available at 3.2 GHz, or one core turbo at 3.8 GHz, the theoretical maximum speed up is $\frac{6\times 3.2}{3.8} = 5.1$. Maple took 2.86 hours (in real

time) and 3.91 hours (cpu time) to compute for the size 64 characteristic polynomial, so the speed up does not reach a factor of two. Our implementation takes 32.6 seconds in serial, 5.9 seconds in parallel, giving a speed up of about 5.5x. The speed ups for computing the size 128 and 256 matrices are at least 5.7x.

Another aspect worth noting is the memory usage. Even with similar clock speed, the Maple timings are heavily affected by the amount of RAM, as seen in Table 5.2. Computing the size $n = 64$ characteristic polynomial takes 19.9 hours on `sarah` with 8 gigabytes of RAM but it takes only 2.86 hours on `ant` which has 64 gigabytes of RAM. After computing for the characteristic polynomial on `ant`, the memory usage came to a maximum of 19 gigabytes. Our routine solves the size 256 matrix with less than two gigabytes of memory, as calculated from the previous chapter.

5.3 General Routine

As mentioned in the Introduction, our code may be adapted to solve for the characteristic polynomial for any input matrices with bivariate polynomials. The primary differences will be in the method of evaluating and interpolation.

If the matrix entries are no longer monomials but polynomials, alternative methods for evaluation and interpolation should be considered. Multivariate polynomials tend to be sparse, and if a Kronecker substitution were to be used, the degrees would be rather large. In the case of large enough degrees, the FFT will be faster than evaluation by Horner's form. Similarly for interpolation, the inverse FFT will be faster than Newton's method. Another method for fast parallel multi-point evaluation [16] (a poster abstract) may also be considered, instead of repeated square and multiply.

As one can see in Table 5.3, the majority of computation power is in the Hessenberg algorithm. One method to reduce the total time is to use block decomposition [9]. This method is more cache friendly, thus an improvement to the decomposition stage of the Hessenberg algorithm.

To further generalize, consider matrices where the entries are polynomials with more than two variables. Let $A(z) = (a_{ij}(z))$ be a dimension n square matrix, where each entry is a polynomial in k variables (so $z = z_1, \ldots, z_k$). The characteristic polynomial in $k + 1$ variables is

$$C(\lambda, z) = \sum_{i=0}^{n} c_i(z)\lambda^i.$$

For simplicity, we will just outline the steps for computing characteristic polynomials of matrices of polynomials based from this thesis.

1. Using k queries, find the factors of the univariate images of $c_i(z)$, for $0 \leq i < n$. Each query evaluates $k-1$ variables in the matrix, giving two variables (including λ) in the characteristic polynomial. The query phase concludes after finding key values as mentioned from Chapter 3.3.

2. Solve for the characteristic polynomial images $C(\lambda, z)$ modulo primes with optimizations (if applicable) as shown from Chapter 3.4. Depending on the polynomial entries, alternate methods for evaluation or interpolation may be more efficient.

    - Apply the CRA incrementally on the image coefficients.

3. Validate solution independently.

5.4 Conclusion and Further Work

The original problem is to find the largest eigenvalues from the characteristic polynomials of specific matrices. Our code is able to efficiently generate those polynomials of the larger matrices, which would appear to be infeasible with current computer algebra systems. Furthermore, our implementation in C can be adapted so that it may be used as a general routine software for any matrices with bivariate polynomial entries.

The characteristic polynomials are very large, thus we only show partial solutions in Appendix A3 and A4. The text file containing the size 256 characteristic polynomial is 1.4 gigabytes. We also read it onto Maple to save it as a Maple file, which comes to 243 megabytes.

As stated in the Introduction, the next step is to solve for the largest $\lambda(x, y)$ from these polynomials which has the form

$$C(\lambda, x, y) = \lambda^n + \sum_{i=0}^{n-1} c_i(x, y)\lambda^i = 0.$$

Given the massive sizes of these polynomials, it is unclear whether any useful information may be extracted from them. Other approaches for solving the maximal eigenvalue which may not require characteristic polynomials should be considered. We leave it to the readers to further investigate how to solve for the largest eigenvalue and advance in this research topic.

Appendix

A1: $s_8(x, y)$ of 16 x 16 matrix

$$\left(2\,x^{18}+6\,x^{16}+2\,x^{14}\right)y^{12}+$$

$$\left(14\,x^{18}+46\,x^{16}+46\,x^{14}+14\,x^{12}\right)y^{11}+$$

$$\left(3\,x^{20}+54\,x^{18}+188\,x^{16}+218\,x^{14}+152\,x^{12}+42\,x^{10}+3\,x^{8}\right)y^{10}+$$

$$\left(16\,x^{20}+154\,x^{18}+482\,x^{16}+670\,x^{14}+526\,x^{12}+252\,x^{10}+92\,x^{8}+8\,x^{6}\right)y^{9}+$$

$$\left(54\,x^{20}+314\,x^{18}+896\,x^{16}+1350\,x^{14}+1266\,x^{12}+684\,x^{10}+262\,x^{8}+104\,x^{6}+20\,x^{4}\right)y^{8}+$$

$$\left(96\,x^{20}+512\,x^{18}+1256\,x^{16}+1988\,x^{14}+2028\,x^{12}+1284\,x^{10}+492\,x^{8}+168\,x^{6}+72\,x^{4}+24\,x^{2}\right)y^{7}+$$

$$\left(124\,x^{20}+576\,x^{18}+1417\,x^{16}+2274\,x^{14}+2365\,x^{12}+1496\,x^{10}+647\,x^{8}+230\,x^{6}+73\,x^{4}+28\,x^{2}+10\right)y^{6}+$$

$$\left(96\,x^{20}+512\,x^{18}+1256\,x^{16}+1988\,x^{14}+2028\,x^{12}+1284\,x^{10}+492\,x^{8}+168\,x^{6}+72\,x^{4}+24\,x^{2}\right)y^{5}+$$

$$\left(54\,x^{20}+314\,x^{18}+896\,x^{16}+1350\,x^{14}+1266\,x^{12}+684\,x^{10}+262\,x^{8}+104\,x^{6}+20\,x^{4}\right)y^{4}+$$

$$\left(16\,x^{20}+154\,x^{18}+482\,x^{16}+670\,x^{14}+526\,x^{12}+252\,x^{10}+92\,x^{8}+8\,x^{6}\right)y^{3}+$$

$$\left(3\,x^{20}+54\,x^{18}+188\,x^{16}+218\,x^{14}+152\,x^{12}+42\,x^{10}+3\,x^{8}\right)y^{2}+$$

$$\left(14\,x^{18}+46\,x^{16}+46\,x^{14}+14\,x^{12}\right)y+$$

$$\left(2\,x^{18}+6\,x^{16}+2\,x^{14}\right)y^{0}$$

A2: Some coefficients of $C(\lambda, x, y)$ for 16 by 16 matrix

$$c_0(x,y) = x^{32}y^{32}\left(x^2-1\right)^{32}$$

$$c_1(x,y) = -x^{32}y^{28}\left(x^2-1\right)^{28}\left(2\,x^4y^2+4\,x^2y^3+4\,x^2y^2+y^4+4\,x^2y+1\right)$$

$$c_2(x,y) = x^{24}y^{25}(x^2-1)^{25}\left(x^{14}y^3+8x^{12}y^4+9x^{12}y^3+8x^{10}y^5+8x^{12}y^2+\dots\right)$$

$$c_3(x,y) = -x^{26}y^{22}(x^2-1)^{22}\left(4x^{12}y^5+4x^{12}y^4+13x^{10}y^6+4x^{12}y^3+36x^{10}y^5+\dots\right)$$

$$\vdots$$

$$c_{14}(x,y) = y(x^2-1)\left(4x^{12}y^6+x^{14}y^3+4x^{12}y^5+4x^{12}y^4+x^{12}y^3+\dots\right)$$

$$c_{15}(x,y) = -x^4\left(x^4y^4+4x^2y^3+x^4+4x^2y^2+4x^2y+2y^2\right)$$

A3: $n = 128$ partial solution $C(\lambda, x, y) = \sum_{i=0}^{n} c_i(x,y)\lambda^i$

$$c_0(x,y) = x^{320}y^{448}(x^2-1)^{448}$$

$$c_1(x,y) = -x^{321}y^{441}(x^2-1)^{441}s_1(x,y)$$

$$s_1(x,y) = y^7+7\,x^2y^6+(14\,x^4+7\,x^2)y^5+(7\,x^6+21\,x^4+7\,x^2)y^4$$
$$+(7\,x^6+21\,x^4+7\,x^2)y^3+(14\,x^4+7\,x^2)y^2+7\,x^2y+1$$

$$\vdots$$

$$c_{127}(x,y) = -x^7s_{127}(x,y)$$

$$s_{127}(x,y) = x^6y^7+7\,x^4y^6+(7\,x^4+14\,x^2)y^5+(7\,x^4+21\,x^2+7)y^4$$
$$+(7\,x^4+21\,x^2+7)y^3+(7\,x^4+14\,x^2)y^2+7\,x^4y+x^6$$

A4: $n = 256$ partial solution $C(\lambda, x, y) = \sum_{i=0}^{n} c_i(x, y)\lambda^i$

$$c_0(x,y) = x^{1024}y^{1024}(x^2-1)^{1024}$$

$$c_1(x,y) = -x^{1024}y^{1016}(x^2-1)^{1016}s_1(x,y)$$

$$s_1(x,y) = y^8 + 8\,x^2y^7 + (20\,x^4 + 8\,x^2)y^6 + (16\,x^6 + 32\,x^4 + 8\,x^2)y^5$$

$$+(2\,x^8 + 24\,x^6 + 36\,x^4 + 8\,x^2)y^4 + (16\,x^6 + 32\,x^4 + 8\,x^2)y^3 + (20\,x^4 + 8\,x^2)y^2 + 8\,x^2y + 1$$

$$\vdots$$

$$c_{255}(x,y) = -x^8 s_{255}(x,y)$$

$$s_{255}(x,y) = x^8y^8 + 8\,x^6y^7 + (8\,x^6 + 20\,x^4)y^6 + (8\,x^6 + 32\,x^4 + 16\,x^2)y^5$$

$$+(8\,x^6 + 36\,x^4 + 24\,x^2 + 2)y^4 + (8\,x^6 + 32\,x^4 + 16\,x^2)y^3 + (8\,x^6 + 20\,x^4)y^2 + 8\,x^6y + x^8$$

B1: Time size 16 matrix on Maple

```
with(LinearAlgebra):
A := Matrix(16, 16, [ [x^8, x^5*y, ...], ... ]);
C := CodeTools[Usage]( CharacteristicPolynomial(A, lambda) );
```

B2: Time size 16 matrix on Magma

```
P<x,y> := PolynomialRing( IntegerRing(), 2);
A := Matrix(P, 16, 16, [ [x^8, x^5*y, ...], ... ]);
time C := CharacteristicPolynomial(A);
```

C: Program output from computing $C(\lambda, x, y)$ for $n = 256$

(Computed on ant: Intel Core i7-3930K six cores at 3.2 GHz, 64 GB RAM.)

```
Dimension n = 256
Cores: 6

Query time for x (Hess + Int): 210.658 + 59.535
Query time for y (Hess + Int): 210.735 + 60.156
Num points: ex = 261, ey = 281

Main incremental algorithm:
Run    Prime         Hess      Int-x      Total      CRA
 1, 1344204809     962.580     6.776     969.356    0.000
 2, 1312202393     964.785     6.777     971.562    0.530
 3, 1187854193     969.208     6.766     975.975    0.898
 4, 1214108377     970.880     6.786     977.667    0.789
 5, 1696107649     973.887     6.887     980.774    1.049
 6, 1465519721     973.596     6.789     980.386    1.309
 7, 1697017657     970.956     6.818     977.774    1.419
 8, 1460110273     971.978     6.785     978.764    1.573
 9, 1823289121     972.367     6.844     979.211    2.190
10, 1435338937     975.643     6.795     982.438    2.543
11, 1756678513     973.828     6.795     980.623    2.919
12, 1763888033     973.209     6.810     980.019    3.233
13, 1150366177     972.670     6.786     979.456    3.642
14, 1795534201     972.268     6.835     979.103    4.150
Terminate after 14 prime(s)
Total degree: 4096
Total routine time: 13990.312

Begin validation, evaluate E(lambda,x,y) mod q:
15, E(1038617224,  235486690,  789457208) mod 1409438473 = 0
16, E(1411215068,  296242329,  968697367) mod 1720266337 = 0
17, E( 262280286, 1344249662,  917504038) mod 1712817233 = 0
18, E(1275030987, 1065261693,  496887487) mod 1378074017 = 0
19, E(1429560951,   88392992, 1756787157) mod 1969923041 = 0
20, E(1253418645,  114601346,  386737691) mod 1609350401 = 0
21, E(1105842539, 1004604112,  601186927) mod 1405719401 = 0
22, E( 326184046,  574143466,   27482697) mod 1125705289 = 0
23, E(1413739028, 1269072384,  430579164) mod 1441551673 = 0
24, E( 869820978, 1148743685,  537441089) mod 1323179281 = 0
Validation time: 7.620

Total time: 13997.934
```

Bibliography

[1] J. Abdeljaoued. The Berkowitz Algorithm, Maple and Computing the Characteristic Polynomial in an Arbitrary Commutative Ring. *MapleTech* **5**(1): 21–32, Birkhauser, 1997.

[2] E.H. Bareiss. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of Computation* **22**(103): 565–578, 1968.

[3] S.J. Berkowitz. On Computing The Determinant in Small Parallel time using a Small Number of Processors. *Inf. Processing Letters* **18**(3): 147–150, 1984.

[4] C. Cohen. *A Course in Computational Algebraic Number Theory.* Graduate Texts in mathematics, **138**, Springer–Verlag, 1995.

[5] J.G. Dumas. Bounds on the Coefficients of the Characteristic and Minimal Polynomials. *Journal of Inequalities in Pure and Applied Mathematics* **8**(2): art. 31, pp.6, 2007.

[6] P. Dusart. Estimates of Some Functions Over Primes without R.H.. arXiv.org. 2007.

[7] J. von zur Gathen, J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, Cambridge, UK, 2003.

[8] K. Geddes, S. Czapor, G. Labahn. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Boston, Massachusetts, USA, 1992.

[9] V. Jonsson. Parallel Reduction from Block Hessenberg to Hessenberg using MPI. *Masters Thesis.* 2013.

[10] M. Kauers. Personal Communication.

[11] M. Law, M. Monagan. Computing Characteristic Polynomials of Matrices of Structured Polynomials. *Proceedings of CASC 2016.* Springer Verlag LNCS **9890**, 336–348, 2016.

[12] D. Leggett. "Fraction–Free Methods for Determinants" (2011). *Master's Theses.* Paper 1.

[13] The LinBox Group, LinBox – Exact Linear Algebra over the Integers and Finite Rings, Version 1.4.2; 2017. (http://linalg.org)

[14] S. Lo, M. Monagan, A. Wittkopf. A Modular Algorithm for Computing the Characteristic Polynomial of an Integer Matrix in Maple. *Proceedings of the 2005 Maple Conference*, MapleSoft, (2005), 369–376.

[15] O.P. Lossers. A Hadamard-Type Bound on the Coefficients of a Determinant of Polynomials. *SIAM Rev.*, **16**(3): 394–395 solution to an exercise by A. Jay Goldstein and Ronald L. Graham, 2006.

[16] M. Monagan, A. Wong. [Poster abstract] Fast Parallel Multi-Point Evaluation of Sparse Polynomials. *Proceedings of ISSAC 2016*, ACM Press. 2016.

[17] C. Pomerance, J. Selfridge, S. Wagstaff. The Pseudoprimes to $25 \cdot 10^{109}$. *Mathematics of Computation.* **35**(151): 1003–1026, 1980.

[18] M. Rabin. Probabilistic Algorithms in Finite Fields. *SIAM J. Comput.*, **9**(2): 273–280, 1979.

[19] J. Schwartz. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM*, **27**: 701–717, 1980.

[20] A. Steel. Personal Communication.

[21] C. Stover. “Ising Model.” From *MathWorld*–A Wolfram Web Resource, created by Eric W. Weisstein. Visited 12/29/2016. http://mathworld.wolfram.com/IsingModel.html

[22] P. Vrbik, M. Monagan. Lazy and Forgetful Polynomial Arithmetic and Applications. *Proceedings of CASC 2009.* Springer Verlag LNCS **5743**, 226–239, 2009.

[23] E. W. Weisstein. “Rabin-Miller Strong Pseudoprime Test.” From *MathWorld*–A Wolfram Web Resource. Visited 12/29/2016. http://mathworld.wolfram.com/Rabin-MillerStrongPseudoprimeTest.html

[24] Wikipedia contributors. “The Nine Chapters on the Mathematical Art.” *Wikipedia, The Free Encyclopedia.* Wikipedia, The Free Encyclopedia, 11 Jul. 2016. Web. 8 Dec. 2016.