FAST FOURIER TRANSFORM ALGORITHMS WITH APPLICATIONS

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Mathematical Sciences

---

by
Todd Mateer
August 2008

Accepted by:
Dr. Shuhong Gao, Committee Chair
Dr. Joel Brawley
Dr. Neil Calkin
Dr. Kevin James

ABSTRACT

This manuscript describes a number of algorithms that can be used to quickly evaluate a polynomial over a collection of points and interpolate these evaluations back into a polynomial. Engineers define the "Fast Fourier Transform" as a method of solving the interpolation problem where the coefficient ring used to construct the polynomials has a special multiplicative structure. Mathematicians define the "Fast Fourier Transform" as a method of solving the multipoint evaluation problem. One purpose of the document is to provide a mathematical treatment of the topic of the "Fast Fourier Transform" that can also be understood by someone who has an understanding of the topic from the engineering perspective.

The manuscript will also introduce several new algorithms that efficiently solve the multipoint evaluation problem over certain finite fields and require fewer finite field operations than existing techniques. The document will also demonstrate that these new algorithms can be used to multiply polynomials with finite field coefficients with fewer operations than Schönhage's algorithm in most circumstances.

A third objective of this document is to provide a mathematical perspective of several algorithms which can be used to multiply polynomials whose size is not a power of two. Several improvements to these algorithms will also be discussed.

Finally, the document will describe several applications of the "Fast Fourier Transform" algorithms presented and will introduce improvements in several of these applications. In addition to polynomial multiplication, the applications of polynomial division with remainder, the greatest common divisor, decoding of Reed-Solomon error-correcting codes, and the computation of the coefficients of a discrete Fourier series will be addressed.

DEDICATION

I dedicate this work to my wife Jennifer and my children Nathan, Laura, Jonathan, and Daniel. In terms of our family, the progress of my graduate research program has been measured through a collection of fifty quarters produced by the United States mint over roughly the same ten year period while I completed my graduate studies. I look forward to placing the final quarter on our "doctor school" map at the end of this year (2008) when I anticipate being finished with several publications related to the research presented in this manuscript.

I have really treasured this time to be at home with my family while my children were young and for their "company" and "support" while completing this project. Much of this dissertation was written with little children in my lap or by my side as my wife and I worked together to get through these first few years of parenthood. I consider this time to be more valuable than the degree for which this dissertation was written.

ACKNOWLEDGMENTS

There are many people who deserve recognition for their role in my education which has led me to this point in my academic career. I will attempt to be as complete as possible here, knowing that are likely several people that I have left out.

Obviously, my advisor Shuhong Gao and committee members deserve mention for being willing to mentor me through this process. This was an especially more challenging task given the fact that we were separated geographically during the entire writing of this dissertation. In fact, my advisor and I only saw each other two times in the three years it took to complete the research program. I would also like to thank Shuhong Gao for teaching a computer algebra class in 2001 which got me interested in this topic for my doctoral studies.

I would also like to thank several anonymous reviewers who read over this entire manuscript several times. There are likely more places where the document can be improved and if there is an interest, I will make revisions to the manuscript as these places are pointed out to me.

I would also like this opportunity to thank the teachers which taught me the subjects related to this research program. In particular, I would like to thank Rebecca Nowakowski, James Payne, Dale McIntyre, Yao-Huan Xu, Jenny Key, and Joel Brawley for equipping me with the algebra and complex variables background to take on this assignment. Also, Timothy Mohr, Frank Duda and Robert Mueller taught me the signals analysis which introduced me to the "engineering perspective" of the FFT. In classes taught by Joe Churm and Robert Jamison, I learned how the material of this dissertation is closely related to basic music theory. Through the instruction of Curt Frank, Jim Kendall, Fred Jenny, Michelle Claus, and James Peterson I learned

the computer science and programming skills needed for this research project. Finally, I would like to thank my father Robert Mateer who taught me the trigonometry that is so fundamental to the topic of this dissertation. While I turned out to be an algebraist instead of following in his footsteps and making analysis (i.e. Calculus) my specialty, I still hope to become as good of a teacher as my father someday.

...but there is a God in heaven who reveals mysteries.

Daniel 2:28a

TABLE OF CONTENTS

Table of Contents (Continued)

Page

Table of Contents (Continued)

Page

Table of Contents (Continued)

Page

LIST OF TABLES

LIST OF FIGURES

List of Figures (Continued)

Figure Page

CHAPTER 1

INTRODUCTION

Around 1805, Carl Friedrich Gauss invented a revolutionary technique for efficiently computing the coefficients of what is now called [1] a discrete Fourier series. Unfortunately, Gauss never published his work and it was lost for over one hundred years. During the rest of the nineteenth century, variations of the technique were independently discovered several more times, but never appreciated. In the early twentieth century, Carl Runge derived an algorithm similar to that of Gauss that could compute the coefficients on an input with size equal to a power of two and was later generalized to powers of three. According to Pierre Duhamel and M. Hollmann [22], this technique was widely known and used in the 1940's. However, after World War II, Runge's work appeared to have been forgotten for an unknown reason. Then in 1965, J. W. Cooley and J. W. Tukey published a short five page paper [16] based on some other works of the early twentieth century which again introduced the technique which is now known as the "Fast Fourier Transform." This time, however, the technique could be implemented on a new invention called a computer and could compute the coefficients of a discrete Fourier series faster than many ever thought possible. Since the publication of the Cooley-Tukey paper, engineers have found many applications for the algorithm. Over 2,000 additional papers have been published on the topic

[1] If the year of this discovery is accurate as claimed in [40], then Gauss discovered the Fourier series even before Fourier introduced the concept in his rejected 1807 work which was later published in 1822 as a book [27]. Nevertheless, Fourier's work was better publicized than that of Gauss among the scientific community and the term "Fourier series" has been widely adopted to describe this mathematical construction.

[39], and the Fast Fourier Transform (FFT) has become one of the most important techniques in the field of Electrical Engineering. The revolution had finally started.

In [25], Charles Fiduccia showed for the first time that the FFT can be computed in terms of algebraic modular reductions. As with the early FFT publications, this idea has been generally ignored. However, Daniel Bernstein recently wrote several unpublished works ([2], [3], [4]) which expand upon the observations of Fiduccia and show the algebraic transformations involved in this approach to computing the FFT.

The main purpose of this document is to provide a mathematical treatment of FFT algorithms, extending the work of Fiduccia and Bernstein. Many of the algorithms contained in this manuscript have appeared in the literature before, but not from this algebraic perspective. While it is unlikely that the completion of this thesis will trigger a revolution similar to that which followed the publication of the Cooley and Tukey paper, it is hoped that this document will help to popularize this mathematical perspective of FFT algorithms.

Another purpose of the document is to introduce a new algorithm originally invented by Shuhong Gao [30] that quickly evaluates a polynomial over special collections of finite field elements. Although it turned out that this algorithm is less efficient than existing techniques for all practical sizes, the careful study of the Cooley-Tukey algorithms through this research effort resulted in a new version of the algorithm that is superior to existing techniques for all practical sizes. The new version of this algorithm will also be introduced as part of this manuscript. We will then show how the new algorithm can be used to multiply polynomials with coefficients over a finite field more efficiently than Schönhage's algorithm, the most efficient polynomial multiplication algorithm for finite fields currently known.

Most FFT algorithms only work when the input size is the power of a small prime. This document will also introduce new algorithms that work for an arbitrary input size. We will then explore several applications of the FFT that can be improved using the new algorithms including polynomial division, the computation of the greatest common divisor, and decoding Reed-Solomon codes.

Another motivation for writing this document is to provide a treatment of the FFT that takes the perspective of both mathematicians and engineers into account so that these two communities may better communicate with each other. The engineering perspective of the FFT has been briefly introduced in these opening remarks. We will now consider the mathematician's perspective of the FFT.

1.1 The mathematician's perspective

It has already been mentioned that engineers originally defined the Fast Fourier Transform as a technique which efficiently computes the coefficients of a discrete Fourier series. As introduced by Fiduccia [25], mathematicians developed an alternative definition of the Fast Fourier Transform (FFT) as a method of efficiently evaluating a polynomial at the powers of a primitive root of unity. Unfortunately, this interpretation is completely the opposite of that of the engineer, who view the inverse of the Fast Fourier Transform as a solution to this multipoint evaluation problem using the discrete Fourier series. Similarly, the mathematician defines the "inverse FFT" as a method of interpolating a set of these evaluations back into a polynomial. We will see in Chapter 10 that this interpolation is the goal of what the engineers call the FFT. One of the challenges of studying the FFT literature is reading papers written by authors who view the FFT problem from a different perspective. Further distorting the engineer's original meaning of the phrase "Fast Fourier Transform",

the "additive FFT" has been defined [29] as an algorithm which exploits the additive vector space construction of finite fields to efficiently evaluate a polynomial at a special collection of these finite field elements. This technique has no relation to the discrete Fourier series at all.

In this manuscript, the FFT will be presented from the mathematician's point of view. In other words, we will define the FFT as "a technique which efficiently evaluates a polynomial over a special collection of points" and the inverse FFT will be defined as "a technique which efficiently interpolates a collection of evaluations of some polynomial at a special set of points back into this polynomial." Two types of FFT algorithms will be considered: (1) the "multiplicative FFT" which works with the powers of a primitive root of unity; and (2) the "additive FFT" which works over a special collection of finite field elements. Again, in Chapter 10, we will show how some of the algorithms developed in this document can be used to solve the engineering applications for which they were originally designed and put some of the "Fourier" back into the Fast Fourier Transform.

At this point, it might be appropriate to point out an additional difference of opinion between mathematicians and engineers relevant to the algorithms in this document. The complex numbers is a collection of elements of the form $A + \sqrt{-1} \cdot B$ where A and B are real numbers. Mathematicians typically use i to represent $\sqrt{-1}$ while the engineers typically use the symbol j. In this document, the symbol `I` will be used to represent $\sqrt{-1}$, following the convention used in several popular computer algebra packages.

1.2 Prerequisite mathematics

Much of the material in this document can be understood by a reader who has completed a curriculum in undergraduate mathematics. Specifically, one should

have completed a course in Discrete Mathematics based on material similar to [66], a course in Linear Algebra based on material similar to [51], and a introductory course in Modern Algebra based on material similar to [60]. In particular, one should have a basic understanding of binary numbers, trees, recursion, recurrence relations, solving linear systems of equations, inverses, roots of unity, groups, rings, fields, and vector spaces.

Additionally, some background in the algebraic structures used throughout this document is highly recommended. To understand the multiplicative FFTs, one needs to know the basic properties of complex numbers (see chapter 1 in [67]). To understand the more advanced material in the document, one should have an understanding of polynomial rings and finite fields, also known as Galois fields. One can study Chapter 2 of [52] or Chapters 1-6 of [80]) to learn this material.

1.3 Operation counts of the algorithms

Mathematically modeling the effort needed to implement any type of algorithm is a difficult topic that has changed several times over the years. Originally, one would only count the number of multiplication operations required by the computer and express the result as a function of the problem size represented by n. The "big-$\mathcal{O}$" notation was later invented to simplify these expressions when comparing different algorithms. A function $f(x)$ is said to be $\mathcal{O}(g(x))$ if there exists constants C and k such that

$$|f(x)| \leq C \cdot |g(x)| \tag{1.1}$$

whenever $x > k$. Unfortunately, this notation was misused over the years and was

later replaced with the "big-Θ" notation by Don Knuth [48]. [2] A function $f(x)$ is said to be $\Theta(g(x))$ if there exists constants C_1, C_2, and k such that

$$C_1 \cdot |g(x)| \leq |f(x)| \leq C_2 \cdot |g(x)| \tag{1.2}$$

whenever $x > k$. The "big-Θ" notation provides a better measure of the effort needed to complete a particular algorithm. In this manuscript, we will give a precise operation count for every FFT algorithm discussed in Chapters 2-4 and will also express an operation count using the "big-Θ" notation when appropriate. For the applications discussed in Chapters 5-10, it will often be difficult to obtain a precise operation count or even a lower bound of the number of operations required. In these cases, the "big-$\mathcal{O}$" notation will be used instead and the author will attempt to present as tight of an upper bound as possible.

In the early days of FFT analysis, only the number of multiplications required was considered significant and the number of additions needed was ignored. This led to an algorithm by Winograd [83] which provided a constructive lower bound on the number of multiplications needed to compute the FFT of size 2^k. Problems arose, however, when people attempted to implement the algorithm and only found it practical for computing FFTs of size up to $2^8 = 64$, much smaller than practical FFT sizes. It turned out that in order to achieve a modest reduction in the number of multiplications, a tradeoff of many more additions was required. As a result, Winograd's algorithm is only of theoretical interest and the number of additions is

[2] To demonstrate the misuse of the "big-$\mathcal{O}$" notation, one can show that any algorithm in this paper is $\mathcal{O}(x^{7399})$. This gives us no information whatsoever that may help us estimate how much effort a particular algorithm costs or that may help us compare two algorithms.

now computed as well as the number of multiplications. Even more advanced models of FFT algorithms also include the contribution from memory accesses and copies. However, these models are often dependent on the architecture of a computer and will not be used for the cost analyses presented in this document.

Instead, we will typically count the number of multiplications and the number of additions needed to implement a particular algorithm. Sometimes, the counts will be given in terms of the algebraic structure used in the algorithm and sometimes the counts will be given in terms of a component of the structure. [3] These results will usually not be combined, but occasionally an analysis will be presented that relates the two operations. In cases where a conservative operation count of the algorithm is desired, the number of copies will also be counted when data needs to be shuffled around. When this applies, a copy will be modeled as equivalent to the cost of an addition.

1.4 Multipoint polynomial evaluation

From the mathematician's perspective, the FFT is a special case of the multipoint evaluation problem. In the next few sections, we will explore algorithms for solving this more general problem.

Let $f(x)$ be a polynomial of degree less than n with coefficients in some ring R. We can express $f(x)$ as

$$f(x) = f_{n-1} \cdot x^{n-1} + f_{n-2} \cdot x^{n-2} + \cdots + f_1 \cdot x + f_0, \tag{1.3}$$

[3] For example, in the case of complex numbers ($\mathbb{C}$), we can either count the number of additions and multiplications in $\mathbb{C}$, or we can count the number of additions and multiplications in $\mathbb{R}$, the real numbers. We will see that there are several different strategies for computing complex number arithmetic in terms of the real numbers.

where $\{f_0, f_1, \ldots, f_{n-2}, f_{n-1}\} \in R$. We wish to evaluate f at some set of points $S = \{\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{n-1}\} \in R$.

Let ε_j be one of the points in S. Then

$$f(\varepsilon_j) \quad = \quad f_{n-1} \cdot {\varepsilon_j}^{n-1} + f_{n-2} \cdot {\varepsilon_j}^{n-2} + \cdots + f_1 \cdot \varepsilon_j + f_0. \tag{1.4}$$

If (1.4) is used to determine $f(\varepsilon_j)$ without seeking to minimize the computations, it would require $\frac{1}{2} \cdot n^2 - \frac{1}{2} \cdot n$ multiplications and $n - 1$ additions. The evaluation of ε_j for each point in S would require $\Theta(n^3)$ multiplications and $\Theta(n^2)$ additions.

In a typical high school algebra course (e.g. [5]), the technique of synthetic division is introduced. This method is based on the so-called "Remainder Theorem" which states that $f(\varepsilon)$ is equal to the remainder when $f(x)$ is divided by the polynomial $x - \varepsilon$. Synthetic division is equivalent to a technique called "Horner's method" which involves rewriting (1.4) as

$$f(\varepsilon_j) \quad = \quad ((\cdots((f_{n-1} \cdot \varepsilon_j) + f_{n-2} \cdot \varepsilon_j) + \ldots) + f_1 \cdot \varepsilon_j) + f_0. \tag{1.5}$$

Synthetic division and Horner's method require $n - 1$ multiplications and $n - 1$ additions to evaluate $f(\varepsilon_j)$. The evaluation of ε_j for each point in S now only requires $n^2 - n$ multiplications and $n^2 - n$ additions. These techniques are said to be $\Theta(n^2)$.

1.5 Fast multipoint evaluation

To introduce a number of concepts relevant to the study of FFT algorithms, we will now consider an algorithm based on material found in [25] and [57] which efficiently solves the multipoint evaluation problem. The presentation of the algorithm in this section is based on [34] and assumes that n is of the form 2^k. However, the algorithm can easily be adapted to let n be of the form p^k if desired.

The algorithm works by performing a series of "reduction steps" which are designed to efficiently compute the polynomial evaluations. Mathematicians view the reduction steps as transformations between various quotient rings, but use representative elements to compute in these algebraic structures. In this document, "$f(x) \bmod \mathcal{M}(x)$" will be interpreted as the remainder which results when $f(x)$ is divided by some other polynomial $\mathcal{M}(x)$ called the "modulus polynomial." Each reduction step receives some "residue polynomial" $f^\circ = f \bmod \mathcal{M}_A$ as input and produces as output the residue polynomials $f^\circ \bmod \mathcal{M}_B = f \bmod \mathcal{M}_B$ and $f^\circ \bmod \mathcal{M}_C = f \bmod \mathcal{M}_C$ where $\mathcal{M}_A = \mathcal{M}_B \cdot \mathcal{M}_C$.

The reduction steps can be organized into a binary tree with $k+1$ levels. On level i, there will be 2^{k-i} nodes for each i in $0 \leq i \leq k$. These nodes will be labeled using the notation (i, j) where i denotes the level of the node on the binary tree and j is used to label the nodes on each level from left to right. Here, $0 \leq j < 2^{k-i}$. In some of the algorithms discussed in later chapters, the nodes will be denoted by $(2^i, j)$.

Let us now express the reduction step of this algorithm in terms of the nodes of the binary tree. If the input to the reduction step is located at node $(i+1, j)$, then the two outputs can be stored in nodes $(i, 2j)$ and $(i, 2j+1)$ if the modulus polynomials are defined appropriately. Let $\mathcal{M}_{0,j}(x)$ be defined as $x - \varepsilon_j$ for each j in $0 \leq j < n$. Then use the recursive definition

$$\mathcal{M}_{i+1,j} \quad = \quad \mathcal{M}_{i,2j} \cdot \mathcal{M}_{i,2j+1} \tag{1.6}$$

to define $\mathcal{M}_{i,j}$ for all $i > 0$ and j in the range $0 \leq j < 2^{k-i}$. If $\mathcal{M}_{i,j}$ is used as the modulus polynomial, then node (i, j) of the binary tree will contain the intermediate result $f \bmod \mathcal{M}_{i,j}$. The reduction step with input at node $(i+1, j)$ transforms $f^\circ = f \bmod \mathcal{M}_{i+1,j}$ into $f^\circ \bmod \mathcal{M}_{i,2j} = f \bmod \mathcal{M}_{i,2j}$ and $f^\circ \bmod \mathcal{M}_{i,2j+1} = f \bmod \mathcal{M}_{i,2j+1}$, which are stored in nodes $(i, 2j)$ and $(i, 2j + 1)$ respectively.

To construct an algorithm based on this reduction step, initialize node $(k, 0)$ with $f(x)$, a polynomial of degree less than n. Since $\mathcal{M}_{k,0}$ has degree n, then $f = f \bmod \mathcal{M}_{k,0}$. Next, use the reduction step to compute the values in all of the nodes of the binary tree. At the end of this process, we will have $f \bmod \mathcal{M}_{0,j} = f \bmod (x - \varepsilon_j)$ for all j in $0 \leq j < n$. By the Remainder Theorem, $f(x) \bmod (x - \varepsilon_j) = f(\varepsilon_j)$ and we have obtained the desired multipoint evaluation of f.

There are two methods typically used to perform the reduction steps. The first method is called "breadth-first order" and proceeds by computing all of the reduction steps on a particular level at the same time. An algorithm which traverses the nodes of the tree in this manner is said to be "iterative". Pseudocode for an iterative implementation of the fast multipoint evaluation algorithm is given in Figure 1.1.

It is possible to construct a more general version of this algorithm that receives the contents of any node of the binary tree as input. Figure 1.2 presents the pseudocode for the second method of traversing the binary tree, called "depth-first order", in this more general form. The depth-first version of the fast multipoint evaluation algorithm proceeds by subdividing the original problem of size $2m$ into two problems of size m and then solving each subproblem individually. For this reason,

Algorithm : Fast multipoint evaluation (iterative implementation)
Input: $f = f \bmod \mathcal{M}_{k,0}$, a polynomial of degree less than $n = 2^k$ in a ring R.
Output: $f(\varepsilon_0), f(\varepsilon_1), \cdots, f(\varepsilon_{n-1})$.
0. If $n = 1$, then return $f(\varepsilon_0) = f$ (f is a constant). 1. **for** $i = k - 1$ downto 0 **do** 2. **for** $j = 0$ to 2^{k-i-1} **do** 3. Retrieve $f^\circ = f \bmod \mathcal{M}_{i+1,j}$ from previous computation or initial condition. 4. Compute $f \bmod \mathcal{M}_{i,2j} = f^\circ \bmod \mathcal{M}_{i,2j}$. 5. Compute $f \bmod \mathcal{M}_{i,2j+1} = f^\circ \bmod \mathcal{M}_{i,2j+1}$. 6. **end for** (Loop j) 7. **end for** (Loop i) 8. Return $f(\varepsilon_j) = f \bmod (x - \varepsilon_j) = f \bmod \mathcal{M}_{0,j}$ for all j in $0 \leq j < n$.

Figure 1.1 Pseudocode for fast multipoint evaluation (iterative implementation)

an algorithm which traverses the nodes of the binary tree using a depth-first approach is said to be "recursive". In this type of algorithm, the order that the reduction steps are completed is governed by common values of $i \cdot j$ in the binary tree.

There are advantages and disadvantages with both orderings of the reduction steps. The major advantage of the iterative approach is that it avoids recursion. When an extensive amount of recursion of small input sizes is involved in an algorithm, the overhead of implementing the recursive calls can dominate the total time needed to implement the algorithm. One major advantage of the recursive approach is that it better utilizes the computer's cache, a relatively small bank of memory in a computer that can be accessed more rapidly than storage outside the cache. The problem with the iterative approach is that for large problems, there will be significant movement of data into and out of the computer's cache. This swapping can have a devastating effect on the amount of time needed to implement an algorithm. In contrast, there will come a point in the recursive approach where a subproblem will fit entirely within

Algorithm : Fast multipoint evaluation (recursive implementation)
Input: $f^\circ = f \bmod \mathcal{M}_{i+1,j}$, a polynomial of degree less than $2m$ where $m = 2^i$ in a ring R.
Output: $f(\varepsilon_{j\cdot 2m+0}), f(\varepsilon_{j\cdot 2m+1}), \cdots, f(\varepsilon_{j\cdot 2m+2m-1})$.
0. If $(2m) = 1$ then return $f \bmod (x - \varepsilon_j) = f(\varepsilon_j)$. 1. Compute $f \bmod \mathcal{M}_{i,2j} = f^\circ \bmod \mathcal{M}_{i,2j}$. 2. Compute $f \bmod \mathcal{M}_{i,2j+1} = f^\circ \bmod \mathcal{M}_{i,2j+1}$. 3. Recursively call algorithm with input $f \bmod \mathcal{M}_{i,2j}$ to obtain $f(\varepsilon_{j\cdot 2m+0}), f(\varepsilon_{j\cdot 2m+1}), \cdots, f(\varepsilon_{j\cdot 2m+m-1})$. 4. Recursively call algorithm with input $f \bmod \mathcal{M}_{i,2j+1}$ to obtain $f(\varepsilon_{j\cdot 2m+m}), f(\varepsilon_{j\cdot 2m+m+1}), \cdots, f(\varepsilon_{j\cdot 2m+2m-1})$. 5. Return $f(\varepsilon_{j\cdot 2m+0}), f(\varepsilon_{j\cdot 2m+1}), \cdots, f(\varepsilon_{j\cdot 2m+2m-1})$.

Figure 1.2 Pseudocode for fast multipoint evaluation (recursive implementation)

the computer's cache. This subproblem can be solved without any swapping involving the cache memory. A second advantage of the recursive approach is that some people feel it better describes the reduction step involved in the algorithm. However, the tradeoff of this advantage is slightly more complicated expressions that have the same purpose as the loop variables in the iterative implementation.

In this document, we will present most of the algorithms in recursive form for readability purposes and will model the recursion involved in these algorithms as having no cost. In practice, what is typically done is to mix the two orderings. Specialized routines are written to implement an algorithm on an input size that will fit entirely in a computer's cache memory. These routines are sometimes written by special code-generators which remove all recursive calls from the routine. The FFTW package [28] uses this technique for computing the Fast Fourier Transforms that will be discussed in Chapter 2. As an example of this technique, pseudocode is provided in Figure 1.3 that computes the fast multipoint evaluation of a polynomial at eight points in depth-first order without using recursion.

Algorithm : Fast multipoint evaluation (8 points)
Input: $f = f \bmod \mathcal{M}_{3,0}$, a polynomial of degree less than 8 in a ring R.
Output: $f(\varepsilon_0), f(\varepsilon_1), \cdots, f(\varepsilon_7)$.
1. Compute $f^{\circ}_{2,0} = f \bmod \mathcal{M}_{2,0}$. 2. Compute $f^{\circ}_{2,1} = f \bmod \mathcal{M}_{2,1}$. 3. Compute $f^{\circ}_{1,0} = f^{\circ}_{2,0} \bmod \mathcal{M}_{1,0}$. 4. Compute $f^{\circ}_{1,1} = f^{\circ}_{2,0} \bmod \mathcal{M}_{1,1}$. 5. Compute $f(\varepsilon_0) = f^{\circ}_{0,0} = f^{\circ}_{1,0} \bmod \mathcal{M}_{0,0}$. 6. Compute $f(\varepsilon_1) = f^{\circ}_{0,1} = f^{\circ}_{1,0} \bmod \mathcal{M}_{0,1}$. 7. Compute $f(\varepsilon_2) = f^{\circ}_{0,2} = f^{\circ}_{1,1} \bmod \mathcal{M}_{0,2}$. 8. Compute $f(\varepsilon_3) = f^{\circ}_{0,3} = f^{\circ}_{1,1} \bmod \mathcal{M}_{0,3}$. 9. Compute $f^{\circ}_{1,2} = f^{\circ}_{2,1} \bmod \mathcal{M}_{1,2}$. 10. Compute $f^{\circ}_{1,3} = f^{\circ}_{2,1} \bmod \mathcal{M}_{1,3}$. 11. Compute $f(\varepsilon_4) = f^{\circ}_{0,4} = f^{\circ}_{1,2} \bmod \mathcal{M}_{0,4}$. 12. Compute $f(\varepsilon_5) = f^{\circ}_{0,5} = f^{\circ}_{1,2} \bmod \mathcal{M}_{0,5}$. 13. Compute $f(\varepsilon_6) = f^{\circ}_{0,6} = f^{\circ}_{1,3} \bmod \mathcal{M}_{0,6}$. 14. Compute $f(\varepsilon_7) = f^{\circ}_{0,7} = f^{\circ}_{1,3} \bmod \mathcal{M}_{0,7}$. 15. Return $f(\varepsilon_0), f(\varepsilon_1), \cdots, f(\varepsilon_7)$.

Figure 1.3 Pseudocode for fast multipoint evaluation (8 points)

For inputs of larger size, an iterative algorithm can be written to reduce the original problem into subproblems that can be solved by the specialized routines. These "depth-first" routines with the recursion removed then take over to implement the rest of the computation by efficiently using the computer's cache memory. All of these details will be hidden from the reader in the presentation of the algorithms discussed in this document, but these issues should be carefully considered by any reader who wishes to implement any of these algorithms on a computer.

We will now give a cost analysis of the recursive version of the algorithm. We will let $A(n)$ denote the number of additions or subtractions in the ring R needed to implement an algorithm with input size n under the assumption that an addition in R requires the same amount of effort to implement as a subtraction in R. Similarly,

we will let $M(n)$ denote the number of multiplications required in R to implement an algorithm of size n. [4]

Let us compute the cost of the recursive version of the algorithm which has input size $2m$. Line 0 simply ends the recursion and costs no operations. Lines 1 and 2 are each a division of a polynomial of degree less than $2m$ by a polynomial of degree m. Using the method of polynomial division typically learned in high school (e.g. [5]), this costs m^2 multiplications in R and m^2 subtractions in R. [5] Lines 3 and 4 each involve a recursive call to the algorithm with input size of m. It requires $M(m)$ multiplications and $A(m)$ additions / subtractions to implement each of these recursive calls. Line 5 requires no operations.

By combining these results, a total of

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{n^2}{2}, \tag{1.7}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{n^2}{2} \tag{1.8}$$

operations are needed to complete the algorithm with input size n. If $n = 1$, then the operation counts are given by $M(1) = 0$ and $A(1) = 0$.

In the appendix, closed-form solutions to provided for a number of general recurrence relations. By properly setting the parameters of these relations, closed-form

---

[4] Since R is only specified to be a ring, there is no guarantee that division is possible in this structure. In those cases where a division is required, we will specify what elements must be invertible in R.

[5] It will always be the case that the leading coefficient of any $\mathcal{M}_{i,j}$ is a 1. Thus, it is not necessary to invert the leading coefficient of this polynomial in the division algorithm and saves m multiplications in the polynomial division that would have otherwise been required.

formulas can be obtained for many of the algorithms presented in this manuscript. By setting $\mathcal{A} = 1/2$ and the rest of the parameters equal to 0 in Master Equation I, operation counts of the fast multipoint evaluation algorithm can be expressed by

$$M(n) = n^2 - n, \tag{1.9}$$

$$A(n) = n^2 - n. \tag{1.10}$$

This algorithm is also said to be $\Theta(n^2)$. It appears that we have not improved the number of operations needed to compute the multipoint evaluations compared to synthetic division. It turns out that we can improve the fast multipoint evaluation algorithm to $\mathcal{O}(n^{1.585} \cdot \log_2(n))$ using a more advanced method of computing the modular reductions. Even more efficient algorithms can be achieved if we can reduce the number of nonzero coefficients in the modulus polynomials. These ideas will be explored further in some of the later chapters.

1.6 Lagrangian interpolation

As mentioned earlier, the inverse of the task of evaluating a polynomial at a collection of points is to interpolate these evaluations back into the original polynomial. In a typical numerical analysis course (e.g. [11]), the technique of Lagrangian interpolation is introduced to accomplish this goal. In this section, we will briefly review this technique.

Let f be a polynomial of degree less than n and suppose that we are given the evaluation of some unknown polynomial f at n arbitrary points $\{\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{n-1}\}$ in some ring R. Thus, we are given $\{a_0, a_1, \ldots, a_{n-1}\}$ where $f(\varepsilon_j) = a_j$ for $0 \leq j < n$. Our task is to recover the unknown polynomial f.

Observe that

$$\begin{aligned}\mathcal{L}_i(x) &= \prod_{j=0, j\neq i}^{n-1} \frac{x-\varepsilon_j}{\varepsilon_i-\varepsilon_j} \\ &= \frac{(x-\varepsilon_1)\cdot(x-\varepsilon_2)\cdots(x-\varepsilon_{i-1})\cdot(x-\varepsilon_{i+1})\cdot(x-\varepsilon_{i+2})\cdots(x-\varepsilon_{n-1})}{(\varepsilon_i-\varepsilon_1)\cdot(\varepsilon_i-\varepsilon_2)\cdots(\varepsilon_i-\varepsilon_{i-1})\cdot(\varepsilon_i-\varepsilon_{i+1})\cdot(\varepsilon_i-\varepsilon_{i+2})\cdots(\varepsilon_i-\varepsilon_{n-1})}\end{aligned} \tag{1.11}$$

is a function such that

$$\mathcal{L}_i(\varepsilon_j) = \begin{cases} 0 & \text{if } j \neq i \\ 1 & \text{if } j = i \end{cases}. \tag{1.12}$$

Each $\mathcal{L}_i(x)$ is called a Lagrange interpolating polynomial and has degree $n-1$. The collection of interpolating polynomials can be computed in $\Theta(n^2 \cdot \log_2(n))$ operations using an algorithm that is provided in [34]. However, the interpolating polynomials can be precomputed and stored for a fixed collection of interpolation points. We will assume that this is the case here.

Given the Lagrange interpolating polynomials, then f can be easily recovered through the computation

$$f(x) = a_0 \cdot \mathcal{L}_0(x) + a_1 \cdot \mathcal{L}_1(x) + \cdots + a_{n-1} \cdot \mathcal{L}_{n-1}(x). \tag{1.13}$$

Using (1.12), it is easy to verify that $f(\varepsilon_i) = a_i$ for $0 \leq i < n$.

Since (1.13) has n terms, each of which consists of multiplying a constant by a polynomial of degree $n-1$, then a total of n^2 multiplications and $n^2 - n$ additions

are required to recover f. Thus, Lagrangian interpolation is said to require $\Theta(n^2)$ operations, provided that the interpolating polynomials are precomputed.

1.7 Fast interpolation

Earlier in this chapter, we considered a fast multipoint evaluation algorithm which is based on a reduction step that receives as input some polynomial $f^\circ = f \bmod \mathcal{M}_A$ and produces as output $f \bmod \mathcal{M}_B$ and $f \bmod \mathcal{M}_C$ where $\mathcal{M}_A = \mathcal{M}_B \cdot \mathcal{M}_C$. The fast interpolation algorithm discussed in this section is based on an interpolation step that essentially reverses this process.

The Chinese Remainder Theorem is a technique that originally appeared in an ancient book by Sun Zi [75] and was used to find integers that satisfied certain constraints based on the Chinese calendar. The technique was later generalized to other algebraic structures. The interpolation step of the algorithm considered in this section is based on the polynomial version of the Chinese Remainder Theorem. Although the technique can be made to work with an arbitrary number of inputs, we will only consider the two-input version of the Chinese Remainder Theorem in this section.

Let $\mathcal{M}_B$ and $\mathcal{M}_C$ be polynomials of degree m which do not have a common divisor. Using the Extended Euclidean Algorithm discussed in Chapter 8, [6] it is possible to find polynomials $u(x)$ and $v(x)$ such that

$$u \cdot \mathcal{M}_B + v \cdot \mathcal{M}_C = 1. \tag{1.14}$$

[6] The first part of Chapter 8 does not depend on the earlier chapters, so the interested reader can review this material at this point if desired.

Note that

$$u \cdot \mathcal{M}_B \bmod \mathcal{M}_C = 1, \tag{1.15}$$

$$v \cdot \mathcal{M}_C \bmod \mathcal{M}_B = 1. \tag{1.16}$$

Suppose that $b(x) = f \bmod \mathcal{M}_B$ and $c(x) = f \bmod \mathcal{M}_C$. The Chinese Remainder Theorem states that the polynomial $a(x)$ of smallest degree such that

$$a \bmod \mathcal{M}_B = b, \tag{1.17}$$

$$a \bmod \mathcal{M}_C = c \tag{1.18}$$

is given by

$$a = (b \cdot v \cdot \mathcal{M}_C + c \cdot u \cdot \mathcal{M}_B) \bmod M_B \cdot M_C. \tag{1.19}$$

The reader can easily verify that (1.19) satisfies (1.17) and (1.18).

If u and v are constant polynomials, then (1.19) simplifies to

$$a = b \cdot v \cdot \mathcal{M}_C + c \cdot u \cdot \mathcal{M}_B. \tag{1.20}$$

Otherwise, [19] provides the following alternative method of computing a with less effort than (1.19):

$$a \;=\; b + ((c - b) \cdot u \bmod \mathcal{M}_C) \cdot \mathcal{M}_B. \tag{1.21}$$

The reader can easily verify that (1.19) and (1.21) are equivalent and can also verify that $a = f \bmod \mathcal{M}_A$ where $\mathcal{M}_A = \mathcal{M}_B \cdot \mathcal{M}_C$. The interpolation step of the fast algorithm combines inputs $b = f \bmod \mathcal{M}_B$ and $c = f \bmod \mathcal{M}_C$ into $a = f \bmod \mathcal{M}_A$ for appropriately defined polynomials $\mathcal{M}_A$, $\mathcal{M}_B$, and $\mathcal{M}_C$.

Suppose that $\{\varepsilon_0, \varepsilon_1, \cdots, \varepsilon_{n-1}\}$ is a collection of n distinct points in a ring R. Let us define $\mathcal{M}_{0,j} = x - \varepsilon_j$ for $0 \le j < n$ and let $\mathcal{M}_{i+1,j} = \mathcal{M}_{i,2j} \cdot \mathcal{M}_{i,2j+1}$ for each $0 \le i \le k-1$ where $k = \log_2(n)$ and where $0 \le j \le 2^{k-i-1}$. The fast interpolation algorithm is initialized with $f(\varepsilon_j) = f \bmod \mathcal{M}_{0,j}$ for $0 \le j < n$. For given values of i and j, it can be easily shown that $\mathcal{M}_{i,2j}$ and $\mathcal{M}_{i,2j+1}$ do not have any common factors. Thus, $b = f \bmod \mathcal{M}_{i,2j}$ and $c = f \bmod \mathcal{M}_{i,2j+1}$ can be combined into $a = f \bmod \mathcal{M}_{i+1,j}$ using the Chinese Remainder Theorem given by either (1.20) or (1.21). We will assume that the polynomials u, v, $\mathcal{M}_{i,2j}$, and $\mathcal{M}_{i,2j+1}$ have been precomputed and stored in these formulas. The fast algorithm first performs the interpolation steps for $i = 0$ and $0 \le j < 2^{k-1}$. The algorithm then proceeds with the interpolation steps for $i = 1, 2, \ldots, k-1$. Since $\mathcal{M}_{k,0}$ is a polynomial of degree n and f is assumed to be a polynomial of degree less than n, then f is equal to the output of this final interpolation step.

The above description of the fast interpolation algorithm was given in iterative form. Pseudocode for an equivalent recursive implementation of the fast interpolation algorithm is given in Figure 1.4.

Algorithm : Fast interpolation (recursive implementation)
Input: The evaluations $f(\varepsilon_{j\cdot 2m+0}), f(\varepsilon_{j\cdot 2m+1}), \ldots, f(\varepsilon_{j\cdot 2m+2m-1})$ of some polynomial f with coefficients in a ring R where $m = 2^i$.
Output: $f \bmod \mathcal{M}_{i+1,j}$.
0. If $(2m) = 1$ then return $f \bmod \mathcal{M}_{0,j} = f \bmod (x - \varepsilon_j) = f(\varepsilon_j)$. 1. Recursively call algorithm with input $f(\varepsilon_{j\cdot 2m+0}), f(\varepsilon_{j\cdot 2m+1}), \ldots, f(\varepsilon_{j\cdot 2m+m-1})$ to obtain $b = f \bmod \mathcal{M}_{i,2j}$. 2. Recursively call algorithm with input $f(\varepsilon_{j\cdot 2m+m}), f(\varepsilon_{j\cdot 2m+m+1}), \ldots, f(\varepsilon_{j\cdot 2m+2m-1})$ to obtain $c = f \bmod \mathcal{M}_{i,2j+1}$. 3. Retrieve polynomial $u_{i+1,j}$ such that $u_{i+1,j} \cdot \mathcal{M}_{i,2j} + v_{i+1,j} \cdot \mathcal{M}_{i,2j+1} = 1$ for some $v_{i+1,j}$. 4. Return $f \bmod \mathcal{M}_{i+1,j} = b + ((c - b) \cdot u_{i+1,j} \bmod \mathcal{M}_{i,2j+1}) \cdot \mathcal{M}_{i,2j}$.

Figure 1.4 Pseudocode for fast interpolation (recursive implementation)

Let us now compute the operation count of this algorithm. Line 0 simply ends the recursion and costs no operations. The cost of lines 1 and 2 is equal to the number of operations needed to call the algorithm with input size m. We will assume that the cost of line 3 is no operations since the required polynomial was precomputed and stored. To compute line 4, we must first subtract two polynomials of size m and then multiply the result by a polynomial of size m. This polynomial of degree at most $2m - 1$ must be divided by a polynomial of degree m and the remainder of this computation must be multiplied by a polynomial of size $m + 1$ and then added to a polynomial of size m. These computations require $3m^2$ multiplications and $3m^2 + 2m$ additions / subtractions, assuming that the multiplications and divisions

are computed using the techniques typically learned in a high school algebra course (e.g. [5]).

By combining these results, a total of

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{3}{4} \cdot n^2 + \frac{1}{2} \cdot n, \tag{1.22}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{3}{4} \cdot n^2 + \frac{3}{2} \cdot n \tag{1.23}$$

operations are needed to complete the algorithm with input size n. If $n = 1$, then the operation counts are given by $M(1) = 0$ and $A(1) = 0$.

Master Equation I can be used to express these operation counts using the expressions

$$M(n) = \frac{3}{2} \cdot n^2 - \frac{3}{2} \cdot n + \frac{1}{2} \cdot n \cdot \log_2(n), \tag{1.24}$$

$$A(n) = \frac{3}{2} \cdot n^2 - \frac{3}{2} \cdot n + \frac{3}{2} \cdot n \cdot \log_2(n). \tag{1.25}$$

If $u_{i,j}$ is a constant for all i and j, then line 4 only requires $2m^2+m$ multiplications and $2m^2 + m$ additions. In this case, the total number of operations needed to implement the algorithm is

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n^2 + \frac{1}{2} \cdot n, \tag{1.26}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n^2 + \frac{3}{2} \cdot n, \tag{1.27}$$

or

$$M(n) = n^2 - n + \frac{1}{2} \cdot n \cdot \log_2(n), \tag{1.28}$$

$$A(n) = n^2 - n + \frac{3}{2} \cdot n \cdot \log_2(n). \tag{1.29}$$

In [43], Ellis Horowitz introduced another algorithm which can be used to perform the interpolation. The algorithm works by pre-scaling $f(\varepsilon_j)$ by

$$s_j = \frac{1}{(\varepsilon_j - \varepsilon_1) \cdot (\varepsilon_j - \varepsilon_2) \cdots (\varepsilon_j - \varepsilon_{j-1}) \cdot (\varepsilon_j - \varepsilon_{j+1}) \cdot (\varepsilon_j - \varepsilon_{j+2}) \cdots (\varepsilon_j - \varepsilon_{n-1})} \tag{1.30}$$

for all $0 \leq j < n$. By replacing u and v with 1 in (1.20), the reader can verify that recursively applying this interpolation step to the scaled inputs produces (1.13) at the end of the computations. This cost of the Horowitz algorithm is equivalent to the algorithm presented in Figure 1.4 for the case where $u_{i,j}$ and $v_{i,j}$ are constant polynomials for all i and j. However, the intermediate results of the Horowitz algorithm do not represent a modular reduction of the polynomial that we are trying to recover.

In any event, the "fast" algorithm in this section is not yet more efficient than Lagrangian interpolation for any of the cases. However, by using faster methods of computing the modular reductions discussed in the later chapters, we will see that the complexity of the fast interpolation algorithm can also be reduced to $\mathcal{O}(n^{1.585} \cdot \log_2(n))$. Even more efficient versions of the fast multipoint evaluation and fast interpolation algorithms can be constructed if the number of nonzero coefficients in $\mathcal{M}_{i,j}$ can be reduced for all i and j.

1.8 Concluding remarks

This chapter contained background material for a study of FFT algorithms. After presenting some historical information, mathematical prerequisites, and a brief discussion on algorithm complexity, several methods of solving the problems of multipoint polynomial evaluation and interpolation were explored. In both cases, a "fast" algorithm was presented based on a binary tree, but performed no better than other methods for solving the problems of multipoint evaluation and interpolation using the material covered thus far.

These algorithms will serve as templates for most FFT algorithms that will be presented in this manuscript. In Chapter 2, we will consider FFT algorithms that exploit the multiplicative structure of the powers of a primitive root of unity. In Chapter 3, we will consider FFT algorithms which exploit the additive structure of finite fields. The key to the performance of these algorithms is reducing the number of terms found in the $\mathcal{M}_{i,j}$'s and maximizing the number of the remaining terms that have a coefficient of 1 or -1. In Chapter 4, the companion inverse FFT algorithms for the algorithms presented in Chapters 2 and 3 will be considered.

After exploring how the FFT can be used to efficiently multiply two polynomials in Chapter 5, then Chapter 6 will give FFT algorithms that can be used to handle the case where n is not a prime power. Finally, in Chapters 7-10, we will explore additional applications of the FFT and will return to some of the problems discussed in this chapter.

CHAPTER 2

MULTIPLICATIVE FAST FOURIER TRANSFORM ALGORITHMS

Let us consider a ring R with primitive nth root of unity ω where $n = p^k$. Suppose that we wish to evaluate a polynomial $f \in R[x]$ of degree less than n at n points and the particular set of points used for the multipoint evaluation is not important. In this case, the number of operations needed to compute the multipoint evaluation can be significiantly reduced if f is evaluated at each of the powers of ω, i.e. $\{f(1), f(\omega), f(\omega^2), f(\omega^3), \ldots, f(\omega^{n-1})\}$. Each of the n points used for this computation is a root of $x^n - 1$. Mathematicians typically call an efficient algorithm for computing this particular multipoint evaluation a Fast Fourier Transform (FFT). In [29], Gao calls this operation the "multiplicative FFT" to distinguish it from the operation that will be discussed in Chapter 3. For the remainder of this chapter, it is to be understood that "FFT" will refer to this multiplicative FFT.

Nearly every presentation of the multiplicative FFT (for example see [22]) views the transformation which implements this multipoint evaluation as the matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \ldots & \omega^{2n-2} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \ldots & \omega^{3n-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2n-2} & \omega^{3n-3} & \cdots & \omega^{n^2-n} \end{pmatrix} \tag{2.1}$$

and shows how these computations can be completed using a "divide-and-conquer"

approach involving the factorization of this matrix. In this chapter, we will instead view the FFT as a special case of the multipoint evaluation algorithm discussed in Chapter 1. This view interprets the FFT as a series of modular reductions as introduced in the work of Fiduccia [25] and Bernstein [2]. As a result, it is possible to assign a mathematical interpretation to every intermediate result of the FFT computation.

In this chapter, we will first present two types of algorithms that can compute an FFT when $p = 2$ using algebraic descriptions of these algorithms found in [2]. Next, we will give algorithms with lower operation counts in the case where multiplication by certain roots of unity can be computed more efficiently than others. Finally, we will present multiplicative FFT algorithms that can be used when $p = 3$. With this background, the reader can develop FFT algorithms for other values of p if desired.

2.1 The bit reversal function

Before presenting the FFT algorithms, we should first mention that most of these algorithms typically produce the output in an order different from the "natural" order $\{f(1), f(\omega), f(\omega^2), f(\omega^3), \ldots, f(\omega^{n-1})\}$. If $n = 2^k$, the output is instead presented in the order $\{f(1), f(\omega^{\sigma(1)}), f(\omega^{\sigma(2)}), f(\omega^{\sigma(3)}), \ldots, f(\omega^{\sigma(n-1)})\}$ where $\sigma(j)$ is the "binary reversal" of j (with respect to n). That is to say, if j can be expressed in binary form as $(b_{k-1}b_{k-2}b_{k-3}\ldots b_2b_1b_0)_2$, then $\sigma(j) = (b_0b_1b_2\ldots b_{k-3}b_{k-2}b_{k-1})_2$. For example, if $j = 5$ and $n = 16$, then express $j = 0101_2$. So $\sigma(j) = 1010_2 = 10$. Note that leading zeros should be included in the binary representation of j and contribute to the value of $\sigma(j)$. The σ function is a permutation of the integers $\{0, 1, 2, \ldots n-1\}$. Since σ is a 1-1, onto function, the inverse of the σ function can be computed and used to rearrange the output of the FFT algorithms back into the "natural" order.

The σ function has several properties which will be important to the development of the FFT algorithms for the case where $p = 2$.

Theorem 1 *$\sigma(j) = 2 \cdot \sigma(2j)$ for $j < n/2$.*

Proof: Let $j < n/2$ and write j in binary form, i.e. $j = (0b_{k-2}b_{k-3}\ldots b_2b_1b_0)_2$. Here, $n/2 = 2^{k-1}$. Then $\sigma(j) = (b_0b_1b_2\ldots b_{k-3}b_{k-2}0)_2$. Now, $2j = (b_{k-2}b_{k-3}b_{k-4}\ldots b_1b_00)_2$ and $\sigma(2j) = (0b_0b_1b_2\ldots b_{k-3}b_{k-2})_2$. Multiplying this result by 2 gives $\sigma(j)$ and thus $\sigma(j) = 2 \cdot \sigma(2j)$. □

Theorem 2 *$\sigma(2j+1) = \sigma(2j) + n/2$ for $j < n/2$ and where $k = \log_2(n)$.*

Proof: If $2j = (b_{k-2}b_{k-3}b_{k-4}\ldots b_1b_00)_2$ and $\sigma(2j) = (0b_0b_1b_2\ldots b_{k-3}b_{k-2})_2$, then $2j+1 = (b_{k-2}b_{k-3}b_{k-4}\ldots b_1b_01)_2$ and $\sigma(2j+1) = (1b_0b_1b_2\ldots b_{k-3}b_{k-2})_2 = \sigma(2j)+n/2$, so $\sigma(2j+1) = \sigma(2j) + n/2$. □

2.2 Classical radix-2 FFT

Cooley and Tukey published a paper [16] which described an algorithm to efficiently compute the FFT for various radix sizes. In this section, we will describe the Cooley-Tukey algorithm for radix 2. Because we will encounter a different formulation of the FFT algorithm in the next section, we will call the Cooley-Tukey version of the FFT algorithm the "classical" FFT algorithm since it came first.

In [2], Bernstein observes that each reduction step of the classical radix-2 FFT algorithm can be viewed as a transformation of the form

$$\begin{aligned} R[x]/(x^{2m} - b^2) \quad &\rightarrow \quad R[x]/(x^m - b) \\ &\times \quad R[x]/(x^m + b). \end{aligned} \tag{2.2}$$

In this transformation, the modulus polynomials will have two nonzero terms in every case and one of the coefficients of each modulus polynomial is 1. By substituting these

polynomials into the multipoint evaluation algorithm discussed in Chapter 1, we will achieve a more efficient algorithm.

We are now going to present an implementation of the classical radix-2 FFT algorithm based on the above transformation. The input to the reduction step is given by $f \bmod (x^{2m} - b^2)$ and computes $f_Y = f \bmod (x^m - b)$ and $f_Z = f \bmod (x^m + b)$. It turns out that the modular reduction is simple to perform in this case. Split the input into two blocks of size m by writing $f \bmod (x^{2m} - b^2) = f_A \cdot x^m + f_B$. Then $f_Y = b \cdot f_A + f_B$ and $f_Z = -b \cdot f_A + f_B$. The reduction step can also be expressed in matrix form as

$$\begin{pmatrix} f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} b & 1 \\ -b & 1 \end{pmatrix} \cdot \begin{pmatrix} f_A \\ f_B \end{pmatrix}. \tag{2.3}$$

Engineers often represent this transformation as a picture and call it a "butterfly operation".

Suppose that we want to compute the FFT of a polynomial f of degree less than $n = 2^k$ over some ring R. We will recursively apply the reduction step with appropriate selections of m and b. Since $(\omega^{\sigma(2j)})^2 = \omega^{\sigma(j)}$ and $-\omega^{\sigma(2j)} = \omega^{\sigma(2j+1)}$ for all $j < n/2$, then b can easily be determined. Each reduction step receives as input $f \bmod (x^{2m} - \omega^{\sigma(j)})$ for some j and produces as output $f \bmod (x^m - \omega^{\sigma(2j)})$ and $f \bmod (x^m - \omega^{\sigma(2j+1)})$. After all of the reduction steps have been completed with input size $2m = 2$, then we have $f \bmod (x - \omega^{\sigma(j)}) = f(\omega^{\sigma(j)})$ for all $j < n$, i.e. the desired FFT of f. Pseudocode for this FFT algorithm is given in Figure 2.1.

It should be pointed out that the FFT algorithm is often applied to the roots of unity in the field of complex numbers. In this case, $\omega = e^{\mathrm{I} \cdot 2\pi/n}$ and z is traditionally used in place of x as the variable. Engineers typically select $\omega = e^{-\mathrm{I} \cdot 2\pi/n}$ as the

Algorithm : Classical radix-2 FFT

Input: $f \bmod (x^{2m} - \omega^{\sigma(j)})$, a polynomial of degree less than $2m$ in a ring R.
Here R has a nth root of unity ω, and m is a power of two where $2m \leq n$.

Output: $f(\omega^{\sigma(j \cdot 2m+0)}), f(\omega^{\sigma(j \cdot 2m+1)}), \ldots, f(\omega^{\sigma(j \cdot 2m+2m-1)})$.

0. If $(2m) = 1$ then return $f \bmod (x - \omega^{\sigma(j)}) = f(\omega^{\sigma(j)})$.
1. Split $f \bmod (x^{2m} - \omega^{\sigma(j)})$ into two blocks f_A and f_B each of size m
   such that $f \bmod (x^{2m} - \omega^{\sigma(j)}) = f_A \cdot x^m + f_B$.
2. Compute $f \bmod (x^m - \omega^{\sigma(2j)}) = f_A \cdot \omega^{\sigma(2j)} + f_B$.
3. Compute $f \bmod (x^m - \omega^{\sigma(2j+1)}) = -f_A \cdot \omega^{\sigma(2j)} + f_B$.
4. Compute the FFT of $f \bmod (x^m - \omega^{\sigma(2j)})$ to obtain
   $f(\omega^{\sigma(j \cdot 2m+0)}), f(\omega^{\sigma(j \cdot 2m+1)}), \ldots, f(\omega^{\sigma(j \cdot 2m+m-1)})$.
5. Compute the FFT of $f \bmod (x^m - \omega^{\sigma(2j+1)})$ to obtain
   $f(\omega^{\sigma(j \cdot 2m+m)}), f(\omega^{\sigma(j \cdot 2m+m+1)}), \ldots, f(\omega^{\sigma(j \cdot 2m+2m-1)})$.
6. Return $f(\omega^{\sigma(j \cdot 2m+0)}), f(\omega^{\sigma(j \cdot 2m+1)}), \ldots, f(\omega^{\sigma(j \cdot 2m+2m-1)})$.

Figure 2.1 Pseudocode for classical radix-2 FFT

primitive root of unity and use duality properties of the Fourier Transform to compute their version of the FFT using the same algorithm given in this section.

Let us now analyze the cost of this algorithm. Line 0 is just used to end the recursion and costs no operations. Line 1 just involves logicially partitioning the input into two blocks of size m, requiring no operations. In line 2, we first multiply the m components of f_A by $\omega^{\sigma(2j)}$. This requires m multiplications except when $j = 0$ in which case no multiplications are required. To complete line 2, we add the updated f_A to f_B at a cost of m additions. In line 3, notice that $f_A \cdot \omega^{\sigma(2j)}$ has already been computed, so all that is necessary is to subtract this result from f_B to obtain $f \bmod (x^m - \omega^{\sigma(2j+1)})$ at a cost of m subtractions in R. The cost of lines 4 and 5 is equal to the number of operations needed to compute two FFTs of size m. Line 6 requires no operations.

The total number of operations to compute the FFT of size n using the classical radix-2 algorithm is given by

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n, \tag{2.4}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + n, \tag{2.5}$$

where $M(1) = 0$ and $A(1) = 0$. Master Equation I can be used to solve these recurrence relations. We must also subtract multiplications to account for the cases where $j = 0$. The number of multiplications saved can be modeled by

$$M_s(n) = M_s\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n, \tag{2.6}$$

where $M_s(1) = 0$. This recurrence relation can be solved using Master Equation II. Combining the results of these closed-form formulas, the number of operations needed to compute the FFT using the classical radix-2 algorithm is given by

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{2.7}$$

$$A(n) = n \cdot \log_2(n). \tag{2.8}$$

This algorithm is said to be $\Theta(n \cdot \log_2(n))$.

2.3 Twisted radix-2 FFT

In [35], Gentleman and Saude presented a different algorithm to compute the FFT for various radices. [1] To compute the radix-2 FFT using this algorithm, Bernstein [2] observes that this approach always uses a reduction step with transformation

$$\begin{aligned} R[x]/(x^{2m}-1) &\rightarrow R[x]/(x^m-1) \\ &\times R[x]/(x^m+1). \end{aligned} \tag{2.9}$$

The mapping $x = \zeta \cdot \dot{x}$ is applied to $R[x]/(x^m+1)$ after each reduction step where ζ is some element of R that transforms $R[x]/(x^m+1)$ to $R[\dot{x}]/(\dot{x}^m-1)$ and allows (2.9) to be applied at the next reduction step. Bernstein calls this FFT the "twisted FFT" because the effect of the mapping $x = \zeta \cdot \dot{x}$ is to rotate the roots of unity according to an amount determined by ζ. We will adopt the terminology of the "twisted FFT" in this manuscript as well.

The mechanism that will be used to accomplish this transformation is called a "weighted" or "twisted" polynomial. Start with $f(x)$ and replace x with $\dot{x}$ to obtain the polynomial $f(\dot{x})$. The twisted polynomial $f(\zeta \cdot \dot{x})$ is computed by multiplying the coefficient of x^d in f by ζ^d for each d in $0 \leq d < n$. The $(\zeta^d)'s$ used in this computation are traditionally called "twiddle factors" in engineering literature. Note that the "twisted polynomial" is a linear transformation of f.

For example, let $f(x) = 3 \cdot x^3 + 2 \cdot x^2 + x + 1$. So $f(\dot{x}) = 3 \cdot \dot{x}^3 + 2 \cdot \dot{x}^2 + \dot{x} + 1$. Suppose that ω is a primitive 4th root of unity in the field of complex numbers, i.e. the

[1] It can be shown that the Gentleman-Saude algorithm is essentially equivalent to what engineers typically call the decimation-in-time FFT and that the Cooley-Tukey algorithm is essentially equivalent to what the engineers call the decimation-in-frequency FFT.

imaginary unit. Then $f(\omega \cdot \dot{x}) = (3\omega^3) \cdot \dot{x}^3 + (2\omega^2) \cdot \dot{x}^2 + \omega \cdot \dot{x} + 1 = -3 \cdot \mathtt{I} \cdot \dot{x}^3 - 2 \cdot \dot{x}^2 + \mathtt{I} \cdot \dot{x} + 1$. Similarly if $g(x) = 7 \cdot \dot{x}^3 + 5\dot{x} + 4$, then $g(\omega \cdot \dot{x}) = -7\mathtt{I} \cdot \dot{x}^3 + 5\mathtt{I} \cdot \dot{x} + 4$. One can verify that the linear transformation properties hold using these two results.

The following theorem provides the value of ζ necessary to achieve the desired transformation.

Theorem 3 *Let $f^{\circ}(x)$ be a polynomial of degree less than $2m$ in $R[x]$. Then $f^{\circ}(x) \bmod (x^m + 1) = f^{\circ}(\zeta \cdot \dot{x}) \bmod (\dot{x}^m - 1)$ where $\zeta = \omega^{\sigma(1)/m}$ and $x = \zeta \cdot \dot{x}$.*

Proof: Let $f^{\circ}(x)$ be a polynomial of degree less than $2m$ in $R[x]$ and let $r(x) = f^{\circ}(x) \bmod (x^m + 1)$. Then $f^{\circ}(x) = q(x) \cdot (x^m + 1) + r(x)$ for some $q(x)$. Now substitute $x = \zeta \cdot \dot{x}$ where $\zeta = \omega^{\sigma(1)/m}$ to rotate the roots of unity. Applying the linear properties of the twisted polynomial transformation gives

$$\begin{aligned} f^{\circ}(\zeta \cdot \dot{x}) &= q(\zeta \cdot \dot{x}) \cdot ((\zeta \cdot \dot{x})^m + 1) + r(\zeta \cdot \dot{x}) \\ &= q(\zeta \cdot \dot{x}) \cdot (\zeta^m \cdot \dot{x}^m + 1) + r(\zeta \cdot \dot{x}) \\ &= q(\zeta \cdot \dot{x}) \cdot (\omega^{\sigma(1)} \cdot \dot{x}^m + 1) + r(\zeta \cdot \dot{x}) \\ &= q(\zeta \cdot \dot{x}) \cdot (-1 \cdot \dot{x}^m + 1) + r(\zeta \cdot \dot{x}) \\ &= -q(\zeta \cdot \dot{x}) \cdot (\dot{x}^m - 1) + r(\zeta \cdot \dot{x}). \end{aligned} \tag{2.10}$$

Thus $r(\zeta \cdot \dot{x}) = f^{\circ}(\zeta \cdot \dot{x}) \bmod (\dot{x}^m - 1)$. □

It is simpler to ignore the distinctions between the various transformations involving the roots of unity and just use the variable x in all of the polynomials. However, whenever the notation $f(\omega^{\theta} \cdot x)$ is used throughout the rest of this chapter, it is to be understood that x is a dummy variable for some other unknown, say $\dot{x}$,

where $x = \omega^{\theta} \cdot \dot{x}$ and x is a variable that represents untransformed polynomials in this equation.

We can now present the reduction step of the twisted FFT algorithm which receives an input of some polynomial f° with degree less than $2m$. Split f° into two blocks of size m by writing $f^{\circ} = f_A \cdot x^m + f_B$. Then compute $f_Y = f^{\circ} \bmod (x^m - 1) = f_A + f_B$ and $f_Z = f^{\circ} \bmod (x^m + 1) = -f_A + f_B$. This reduction step can also be represented by the transformation

$$\begin{pmatrix} f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_A \\ f_B \end{pmatrix}. \tag{2.11}$$

The simplified reduction step can be directly applied to $f^{\circ} \bmod (x^m - 1)$ while we need to twist $f^{\circ} \bmod (x^m + 1)$ by ζ prior to using this result as input to the simplified reduction step.

It is not clear yet that an algorithm based on this reduction step will yield the FFT of some polynomial $f(x)$ of degree less than $n = 2^k$. The following theorems are intended to provide this clarification.

Theorem 4 *If* $f^{\circ}(x) = f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$, *then*
$f^{\circ}(x) \bmod (x^m - 1) = f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^m - 1) = f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m - 1)$.

Proof: Let $\theta = \sigma(j)/(2m) = 2\sigma(2j)/(2m) = \sigma(2j)/m$ and let $f^{\circ}(x) = f(\omega^{\theta} \cdot x) \bmod (x^{2m} - 1)$. Modularly reducing both sides of this equation by $x^m - 1$ produces the desired result. □

Theorem 5 *If* $f^{\circ}(x) = f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$, *then*
$f^{\circ}(\zeta \cdot x) \bmod (x^m - 1) = f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m - 1)$ *where* $\zeta = \omega^{\sigma(1)/m}$.

Algorithm : Twisted radix-2 FFT

Input: $f^\circ(x) = f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$, the modular reduction of some polynomial $f(x) \in R[x]$ that has been twisted by $\omega^{\sigma(j)/(2m)}$. Here R has a nth root of unity ω, and m is a power of two where $2m \leq n$.

Output: $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$.

0. If $(2m) = 1$ then return $f(\omega^{\sigma(j)} \cdot x) \bmod (x - 1) = f(\omega^{\sigma(j)})$.
1. Split $f^\circ(x)$ into two blocks f_A and f_B each of size m such that $f^\circ(x) = f_A \cdot x^m + f_B$.
2. Compute $f(\omega^{\sigma(2j)/(m)} \cdot x) \bmod (x^m - 1) = f^\circ(x) \bmod (x^m - 1) = f_A + f_B$.
3. Compute $f^\circ(x) \bmod (x^m + 1) = -f_A + f_B$.
4. Twist $f^\circ(x) \bmod (x^m + 1)$ by $\omega^{\sigma(1)/m}$ to obtain $f(\omega^{\sigma(2j+1)/m}) \bmod (x^m - 1)$.
5. Compute the FFT of $f(\omega^{\sigma(2j)} \cdot x) \bmod (x^m - 1)$ to obtain $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+m-1)})$.
6. Compute the FFT of $f(\omega^{\sigma(2j+1)} \cdot x) \bmod (x^m - 1)$ to obtain $f(\omega^{\sigma(j\cdot 2m+m)}), f(\omega^{\sigma(j\cdot 2m+m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$.
7. Return $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$.

Figure 2.2 Pseudocode for twisted radix-2 FFT

Proof: Let $\theta = \sigma(j)/(2m) = \sigma(2j)/m$ and observe that $\sigma(1)/m + \theta = \sigma(2j+1)/m$. Then $\zeta \cdot (\omega^\theta) = \omega^{\sigma(2j+1)/m}$. So $f^\circ(\zeta \cdot x) = f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^{2m} - 1)$. Modularly reducing both sides of this equation by $x^m - 1$ produces the desired result. □

So the reduction step receives as input $f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$, the modular reduction of some polynomial f that has been twisted by $\omega^{\sigma(j)/(2m)} = \omega^{\sigma(2j)/m}$. The reduction step produces as output $f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m - 1)$ and $f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m - 1)$ after the second result has been twisted by ζ.

The algorithm is initialized with $f(x)$ which equals $f(\omega^0 \cdot x) \bmod (x^n - 1)$ if f has degree less than n. By recursively applying the reduction step to $f(x)$, we obtain $f(\omega^{\sigma(j)} \cdot x) \bmod (x - 1) = f(\omega^{\sigma(j)} \cdot 1)$ for all j in the range $0 \leq j < n$, i.e. the FFT of $f(x)$ of size n. Pseudocode for this FFT algorithm is given in Figure 2.2.

Let us now analyze the cost of this algorithm. Line 0 is just used to end the recursion and costs no operations. Line 1 just involves logicially partitioning the input into two blocks of size m, requiring no operations. In line 2, we add f_A to f_B at a cost of m additions. In line 3, we subtract f_A from f_B at a cost of m subtractions. The cost of line 4 is $m-1$ multiplications in R and the cost of lines 5 and 6 is equal to the number of operations needed to compute two FFTs of size m. Line 7 requires no operations.

The total number of operations to compute the FFT of size n using the twisted radix-2 algorithm is given by

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n - 1, \tag{2.12}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + n, \tag{2.13}$$

where $M(1) = 0$ and $A(1) = 0$.

Using Master Equation I, these operation counts can also be expressed using

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{2.14}$$

$$A(n) = n \cdot \log_2(n). \tag{2.15}$$

This algorithm has the exact same operation count as the classical radix-2 FFT algorithm.

2.4 Hybrid radix-2 FFTs

It is not necessary to apply a twist after every reduction step. Suppose that we wish to evaluate f° at each of the roots of $x^{2^c m} - 1$. After c stages of reduction steps, we will have $f^\circ \bmod (x^m - \omega^{\sigma(j)})$ for each j in $0 \leq j < 2^c$. Each $\omega^{\sigma(j)}$ is a 2^cth root of unity.

The following generalization of Theorem 3 shows the transformation necessary to apply the simplified reduction step to each of these results.

Theorem 6 *Let $f^\circ(x)$ be a polynomial in $R[x]$ with degree d less than $2^c \cdot m$. Then $f^\circ(x) \bmod (x^m - \omega^{\sigma(j)}) = f^\circ(\zeta \cdot x) \bmod (x^m - 1)$ where $j < 2^c$ and $\zeta = \omega^{\sigma(j)/m}$.*

Proof: Similar to Theorem 3 □

Let us now more carefully examine the effect of twisting a polynomial at an arbitrary intermediate result in the computation of the FFT to see if it is possible to obtain an improved algorithm by combining the classical and twisted FFT algorithms. Suppose that we use the classical FFT algorithm to compute $f^\circ(x) \bmod (x^m - \omega^{\sigma(j)})$ and then at this point twist this polynomial to obtain $f^\circ(\zeta \cdot x) \bmod (x^m - 1)$ where $\zeta = \omega^{\sigma(j)/m}$. The benefit of this transformation is that no multiplications are needed for the reduction steps with inputs $f^\circ(\zeta \cdot x) \bmod (x^{2^{i-1}} - 1)$, $f^\circ(\zeta \cdot x) \bmod (x^{2^{i-2}} - 1), \cdots$, $f^\circ(\zeta \cdot x) \bmod (x - 1)$ where $2^i = m$. This results in a savings of

$$\sum_{d=0}^{i-1} 2^d \quad = \quad 2^i - 1 \tag{2.16}$$

multiplications. However, the cost of the twisting operation is also $2^i - 1$ multiplications.

Since this analysis was done for an arbitrary point in the computation of the FFT, there is never any overall improvement that results from twisting a polynomial. Thus, any hybrid algorithm involving the two FFT algorithms examined so far will neither improve nor increase the operation count. This observation may also help to explain why the two FFT algorithms discussed so far yielded the same number of multiplications.

2.5 Classical radix-4 FFT

Suppose that every element of R is expressed in the form $A + \mathtt{I} \cdot B$ where A and B are in a ring that does not contain $\mathtt{I}$ where $\mathtt{I}^2 = -1$. For example, R can be the ring of complex numbers and A and B are elements of the ring of real numbers. Then multiplication by $\mathtt{I} = \omega^{n/4} = \omega^{\sigma(2)}$ results in $-B + \mathtt{I} \cdot A$. Observe that no multiplications in R are needed to complete this operation. Similarly, multiplication by $-\mathtt{I} = \omega^{\sigma(3)}$ also involves no multiplications in R. We will now develop a radix-4 algorithm based on the Cooley-Tukey reduction step [2] which exploits this property to reduce the number of operations in the FFT. The algorithm that will be discussed in this section is based on the following transformation:

[2] Although the algorithm presented in this section is based on the Cooley-Tukey algorithm, it is not the algorithm presented in [16]. It turns out that Cooley and Tukey did not observe that multiplication by $\mathtt{I}$ could be implemented faster than other primitive roots of unity. Consequently, they erroneously concluded that radix-3 FFT algorithms were the most efficient in their paper. Although we will see that radix-3 algorithms are more efficient than radix-2 algorithms, this section will show that radix-4 algorithms are more efficient than either of these other cases. The paper by [35] first exploited the multiplications by $-\mathtt{I}$ to produce the improved radix-4 FFT algorithm. In this section, the observations given in [35] were applied to the Cooley-Tukey algorithm to obtain the "classical" radix-4 FFT algorithm.

$$
\begin{aligned}
R[x]/(x^{4m} - b^4) \quad &\rightarrow \quad R[x]/(x^m - b) \\
&\times \quad R[x]/(x^m + b) \\
&\times \quad R[x]/(x^m - \mathtt{I} \cdot b) \\
&\times \quad R[x]/(x^m + \mathtt{I} \cdot b).
\end{aligned} \tag{2.17}
$$

We are now going to present a radix-4 FFT algorithm based on this transformation. The reduction step of the algorithm receives as input $f^\circ = f \bmod (x^{4m} - b^4)$ and computes $f_W = f \bmod (x^m - b)$, $f_X = f \bmod (x^m + b)$, $f_Y = f \bmod (x^m - \mathtt{I} \cdot b)$, and $f_Z = f \bmod (x^m + \mathtt{I} \cdot b)$. It turns out that the modular reduction is simple to perform in this case as well. Split f° into four blocks of size m by writing $f^\circ = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.

Then

$$
\begin{aligned}
f_W &= f_A \cdot b^3 + f_B \cdot b^2 + f_C \cdot b + f_D, && (2.18) \\
f_X &= -f_A \cdot b^3 + f_B \cdot b^2 - f_C \cdot b + f_D, && (2.19) \\
f_Y &= -\mathtt{I} \cdot f_A \cdot b^3 - f_B \cdot b^2 + \mathtt{I} \cdot f_C \cdot b + f_D, && (2.20) \\
f_Z &= \mathtt{I} \cdot f_A \cdot b^3 - f_B \cdot b^2 - \mathtt{I} \cdot f_C \cdot b + f_D. && (2.21)
\end{aligned}
$$

Implementation of the radix-4 algorithm on a computer makes use of the following additional properties of the σ function which the reader can verify holds for $j < n/4$ where $n = 2^k$ using proofs similar to Theorems 1 and 2:

$$\sigma(j) = 4 \cdot \sigma(4j), \tag{2.22}$$

$$\sigma(4j+1) = \sigma(4j) + n/2, \tag{2.23}$$

$$\sigma(4j+2) = \sigma(4j) + n/4, \tag{2.24}$$

$$\sigma(4j+3) = \sigma(4j) + 3 \cdot n/4. \tag{2.25}$$

It follows from these properties that for any $j < n/4$:

$$\omega^{\sigma(4j+1)} = -\omega^{\sigma(4j)}, \tag{2.26}$$

$$\omega^{\sigma(4j+2)} = \mathtt{I} \cdot \omega^{\sigma(4j)}, \tag{2.27}$$

$$\omega^{\sigma(4j+3)} = -\mathtt{I} \cdot \omega^{\sigma(4j)}. \tag{2.28}$$

Suppose that we want to compute the FFT of a polynomial f of degree less than $n = 2^k$ over R as described at the beginning of this section. We will recursively apply the reduction step with appropriate selections of m and b. Since $(\omega^{\sigma(4j)})^4 = \omega^{\sigma(j)}$ and because of properties (2.26)-(2.28) above, then b can be easily determined. Each reduction step receives as input $f \bmod (x^{4m} - \omega^{\sigma(j)})$ for some j and produces as output $f_W = f \bmod (x^m - \omega^{\sigma(4j)})$, $f_X = f \bmod (x^m - \omega^{\sigma(4j+1)})$, $f_Y = f \bmod (x^m - \omega^{\sigma(4j+2)})$, and $f_Z = f \bmod (x^m - \omega^{\sigma(4j+3)})$. The reduction step can be implemented using the transformation

$$\begin{pmatrix} f_W \\ f_X \\ f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -\mathtt{I} & -1 & \mathtt{I} & 1 \\ \mathtt{I} & -1 & -\mathtt{I} & 1 \end{pmatrix} \cdot \begin{pmatrix} \omega^{3\sigma(4j)} \cdot f_A \\ \omega^{2\sigma(4j)} \cdot f_B \\ \omega^{\sigma(4j)} \cdot f_C \\ f_D \end{pmatrix}. \tag{2.29}$$

If k is even and all of the reduction steps have been completed with input size $4m = 4$, then we will have $f \bmod (x - \omega^{\sigma(j)}) = f(\omega^{\sigma(j)})$ for all $j < n$, i.e. the desired FFT of f. However, if k is odd, then one stage of reduction steps from either of the radix-2 algorithms discussed in the previous sections will be needed to complete the FFT computation. Pseudocode for this FFT algorithm is given in Figure 2.3.

Let us now compute the cost of this algorithm. Lines 0A and 0B are used to end the recursion. Line 0A costs no operations while line 0B costs one multiplication and two additions. Line 1 just involves logicially partitioning the input into two blocks of size m, requiring no operations. In line 2, we multiply each of the m components of f_A, f_B, and f_C by a power of ω unless $j = 0$ in which case no multiplications are required. Lines 3-10 describe a sequence of operations that efficiently computes the rest of the reduction step. Each of these instructions requires m additions or subtractions. Finally, lines 11-14 recursively calls the algorithm on the four indicated results to complete the FFT computation. Line 15 requires no operations.

The total number of operations to compute the FFT of size n using the classical radix-4 algorithm for $j \neq 0$ is given by

$$M(n) = 4 \cdot M\left(\frac{n}{4}\right) + \frac{3}{4} \cdot n, \tag{2.30}$$

$$A(n) = 4 \cdot A\left(\frac{n}{4}\right) + 2 \cdot n, \tag{2.31}$$

Algorithm : Classical radix-4 FFT

Input: $f \bmod (x^{4m} - \omega^{\sigma(j)})$, a polynomial of degree less than $4m$ in a ring R.
Here R has a nth root of unity ω, and m is a power of two where $4m \leq n$.

Output: $f(\omega^{\sigma(j \cdot 4m+0)}), f(\omega^{\sigma(j \cdot 4m+1)}), \ldots, f(\omega^{\sigma(j \cdot 4m+4m-1)})$.

0A. If $(4m) = 1$ then return $f \bmod (x - \omega^{\sigma(j)}) = f(\omega^{\sigma(j)})$.
0B. If $(4m) = 2$ then use a radix-2 algorithm to compute the desired FFT.
1. Split $f \bmod (x^{4m} - \omega^{\sigma(j)})$ into four blocks
   f_A, f_B, f_C, and f_D each of size m such that
   $f \bmod (x^{4m} - \omega^{\sigma(j)}) = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
2. Compute $f_A \cdot \omega^{3\sigma(4j)}$, $f_B \cdot \omega^{2\sigma(4j)}$, and $f_C \cdot \omega^{\sigma(4j)}$.
3. Compute $f_\alpha = f_A \cdot \omega^{3\sigma(4j)} + f_C \cdot \omega^{\sigma(4j)}$.
4. Compute $f_\beta = f_B \cdot \omega^{2\sigma(4j)} + f_D$.
5. Compute $f_\gamma = \mathtt{I} \cdot (-f_A \cdot \omega^{3\sigma(4j)} + f_C \cdot \omega^{\sigma(4j)})$.
6. Compute $f_\delta = -f_B \cdot \omega^{2\sigma(4j)} + f_D$.
7. Compute $f \bmod (x^m - \omega^{\sigma(4j)}) = f_\alpha + f_\beta$.
8. Compute $f \bmod (x^m - \omega^{\sigma(4j+1)}) = -f_\alpha + f_\beta$.
9. Compute $f \bmod (x^m - \omega^{\sigma(4j+2)}) = f_\gamma + f_\delta$.
10. Compute $f \bmod (x^m - \omega^{\sigma(4j+3)}) = -f_\gamma + f_\delta$.
11. Compute the FFT of $f \bmod (x^m - \omega^{\sigma(4j)})$ to obtain
    $f(\omega^{\sigma(j \cdot 4m+0)}), f(\omega^{\sigma(j \cdot 4m+1)}), \ldots, f(\omega^{\sigma(j \cdot 4m+m-1)})$.
12. Compute the FFT of $f \bmod (x^m - \omega^{\sigma(4j+1)})$ to obtain
    $f(\omega^{\sigma(j \cdot 4m+m)}), f(\omega^{\sigma(j \cdot 4m+m+1)}), \ldots, f(\omega^{\sigma(j \cdot 4m+2m-1)})$.
13. Compute the FFT of $f \bmod (x^m - \omega^{\sigma(4j+2)})$ to obtain
    $f(\omega^{\sigma(j \cdot 4m+2m)}), f(\omega^{\sigma(j \cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma(j \cdot 4m+3m-1)})$.
14. Compute the FFT of $f \bmod (x^m - \omega^{\sigma(4j+3)})$ to obtain
    $f(\omega^{\sigma(j \cdot 4m+3m)}), f(\omega^{\sigma(j \cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma(j \cdot 4m+4m-1)})$.
15. Return $f(\omega^{\sigma(j \cdot 4m+0)}), f(\omega^{\sigma(j \cdot 4m+1)}), \ldots, f(\omega^{\sigma(j \cdot 4m+4m-1)})$.

Figure 2.3 Pseudocode for classical radix-4 FFT

where $M(1) = 0$, $A(1) = 0$, $M(2) = 1$, and $A(2) = 2$. Master Equation I can be used to solve this recurrence relation. We must also subtract multiplications to account for the cases where $j = 0$. A recurrence relation which gives the multiplications saved is

$$M_s(n) = M_s\left(\frac{n}{4}\right) + \frac{3}{4} \cdot n, \tag{2.32}$$

where $M_s(1) = 0$ and $M_s(2) = 1$. Master Equation II can be used to solve this recurrence relation and obtain a savings of $n - 1$ operations.

The number of operations required to compute an FFT using the classical radix-4 algorithm is given by

$$M(n) = \begin{cases} \frac{3}{8} \cdot n \cdot \log_2(n) - n + 1 & \text{if } \log_2(n) \text{ is even} \\ \frac{3}{8} \cdot n \cdot \log_2(n) - \frac{7}{8} \cdot n + 1 & \text{if } \log_2(n) \text{ is odd} \end{cases}, \tag{2.33}$$

$$A(n) = n \cdot \log_2(n). \tag{2.34}$$

This algorithm has the same addition count as the classical radix-2 FFT algorithm, but the multiplication count has been significantly reduced.

2.6 Twisted radix-4 FFT

Now, we will describe the twisted version of the radix-4 algorithm. This is the algorithm that is described in [35] and is the first efficient radix-4 algorithm that appeared in the literature.

Observe that when $j = 0$, then (2.29) simplifies to

$$\begin{pmatrix} f_W \\ f_X \\ f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} +1 & +1 & +1 & +1 \\ -1 & +1 & -1 & +1 \\ -\mathtt{I} & -1 & +\mathtt{I} & +1 \\ +\mathtt{I} & -1 & -\mathtt{I} & +1 \end{pmatrix} \cdot \begin{pmatrix} f_A \\ f_B \\ f_C \\ f_D \end{pmatrix}. \tag{2.35}$$

The significance of this case is that no multiplications in R need to be computed in this transformation since multiplication by $\mathtt{I}$ or $-\mathtt{I}$ is implemented by simply swapping the components of an element of R. By applying Theorem 6 with $c = 2$, we can determine the necessary values of ζ to twist the results of this simplified radix-4 reduction step so they can be used as inputs to other simplified reduction steps:

$$f^\circ \bmod (x^m + 1) \;:\; \zeta = \omega^{\sigma(1)/m}, \tag{2.36}$$

$$f^\circ \bmod (x^m - \mathtt{I}) \;:\; \zeta = \omega^{\sigma(2)/m}, \tag{2.37}$$

$$f^\circ \bmod (x^m + \mathtt{I}) \;:\; \zeta = \omega^{\sigma(3)/m}. \tag{2.38}$$

At this point, we could mimic the steps used to convert the classical radix-2 algorithm to the twisted radix-2 algorithm and obtain a twisted radix-4 algorithm. We will not give the resulting pseudocode here.

The total number of operations to compute the twisted radix-4 FFT of size n is given by

$$M(n) = 4 \cdot M\left(\frac{n}{4}\right) + \frac{3}{4} \cdot n - \frac{3}{4}, \tag{2.39}$$

$$A(n) = 4 \cdot A\left(\frac{n}{4}\right) + 2 \cdot n, \tag{2.40}$$

where $M(1) = 0$, $A(1) = 0$, $M(2) = 0$, and $A(2) = 2$.

Master Equation I can be used to show that the operation counts of the twisted radix-4 algorithm are exactly the same as the classical radix-4 algorithm. Thus, from a theoretical point of view, there is no difference between the classical and twisted radix-4 algorithms.

2.7 Radix-8 FFT

Let ϕ be a primitive 8th root of unity in R. If R contains the element $\sqrt{2}/2$, then ϕ can be expressed by

$$\phi = \frac{\sqrt{2}}{2} + \mathtt{I} \cdot \frac{\sqrt{2}}{2} \tag{2.41}$$

and multiplication of an element $A + \mathtt{I} \cdot B$ in R by ϕ is given by

$$\phi \cdot (A + \mathtt{I} \cdot B) = \frac{\sqrt{2}}{2} \cdot (A - B) + \mathtt{I} \cdot \frac{\sqrt{2}}{2} \cdot (A + B). \tag{2.42}$$

Let R be the field of complex numbers for the remainder of this section. We will assume that a multiplication in $\mathbb{C}$ requires 4 multiplications and 2 additions [3] in $\mathbb{R}$, the real numbers, while a multiplication by ϕ requires 2 multiplications and 2

[3] In [10], Oscar Buneman introduced an alternative method of multiplication in $\mathbb{C}$ that requires 3 multiplications and 3 additions in $\mathbb{R}$ if the powers of ω are precomputed. However, because the time needed to implement floating point multiplication is about the same as the time needed to implement floating point addition on modern computers, it is generally felt that Buneman's somewhat more complicated method of multiplication in $\mathbb{C}$ no longer has the advantage that it once had.

additions in $\mathbb{R}$. Similar results hold for multiplication by $\phi^3, \phi^5 = -\phi$ and $\phi^7 = -\phi^3$ in $\mathbb{C}$ as well as other rings which contain the element $\sqrt{2}/2$.

As introduced in [1], a radix-8 algorithm can be constructed using the transformation

$$\begin{aligned} R[x]/(x^{8m} - b^8) \;\rightarrow\; & R[x]/(x^m - b) \\ \times\; & R[x]/(x^m + b) \\ \times\; & R[x]/(x^m - \mathtt{I} \cdot b) \\ \times\; & R[x]/(x^m + \mathtt{I} \cdot b) \\ \times\; & R[x]/(x^m - \phi \cdot b) \\ \times\; & R[x]/(x^m + \phi \cdot b) \\ \times\; & R[x]/(x^m - \phi^3 \cdot b) \\ \times\; & R[x]/(x^m + \phi^3 \cdot b). \end{aligned} \tag{2.43}$$

The radix-8 algorithm can be developed by duplicating the steps used to create the radix-2 or radix-4 algorithm at this point. This analysis will produce a transformation matrix given by

$$\begin{pmatrix} +1 & +1 & +1 & +1 & +1 & +1 & +1 & +1 \\ -1 & +1 & -1 & +1 & -1 & +1 & -1 & +1 \\ -\mathtt{I} & -1 & +\mathtt{I} & +1 & -\mathtt{I} & -1 & +\mathtt{I} & +1 \\ +\mathtt{I} & -1 & -\mathtt{I} & +1 & +\mathtt{I} & -1 & -\mathtt{I} & +1 \\ -\phi^3 & -\mathtt{I} & -\phi & -1 & +\phi^3 & +\mathtt{I} & +\phi & +1 \\ +\phi^3 & -\mathtt{I} & +\phi & -1 & -\phi^3 & +\mathtt{I} & -\phi & +1 \\ -\phi & +\mathtt{I} & -\phi^3 & -1 & +\phi & -\mathtt{I} & +\phi^3 & +1 \\ +\phi & +\mathtt{I} & +\phi^3 & -1 & -\phi & -\mathtt{I} & -\phi^3 & +1 \end{pmatrix} \tag{2.44}$$

which will be used to implement the reduction step. Pseudocode for the resulting algorithm similar to that introduced in [1] will not be given here. It can be shown that the number of operations needed to implement the classical version of the radix algorithm is governed by the recurrence relations

$$M(n) = 8 \cdot M\left(\frac{n}{8}\right) + \frac{9}{8} \cdot n, \tag{2.45}$$

$$A(n) = 8 \cdot A\left(\frac{n}{8}\right) + 3 \cdot n, \tag{2.46}$$

where $M(1) = 0$ and $A(1) = 0$, $M(2) = 1$, $A(2) = 2$, $M(4) = 3$, and $A(4) = 8$. These equations can be solved using Master Equation I. We must also subtract multiplications to account for the cases where $j = 0$. A recurrence relation which governs the multiplication savings in this case is given by

$$M_s(n) = M_s\left(\frac{n}{8}\right) + \frac{7}{8} \cdot n, \tag{2.47}$$

where $M_s(1) = 0$, $M_s(2) = 1$, and $M_s(4) = 3$. Master Equation II can be used to solve this recurrence relation.

The total number of operations needed to implement the algorithm is given by

$$M(n) = \begin{cases} \frac{3}{8} \cdot n \cdot \log_2(n) - n + 1 & \text{if } \log_2(n) \bmod 3 = 0 \\ \frac{3}{8} \cdot n \cdot \log_2(n) - \frac{7}{8} \cdot n + 1 & \text{if } \log_2(n) \bmod 3 = 1 \\ \frac{3}{8} \cdot n \cdot \log_2(n) - n + 1 & \text{if } \log_2(n) \bmod 3 = 2 \end{cases}, \tag{2.48}$$

$$A(n) = n \cdot \log_2(n). \tag{2.49}$$

This does not appear to be an improvement compared to the radix-4 algorithms, but we have not yet accounted for the special multiplications by the primitive 8th roots of unity. The recurrence relation

$$M_8(n) = 8 \cdot M_8\left(\frac{n}{8}\right) + \frac{1}{4} \cdot n \tag{2.50}$$

counts the number of multiplications in (2.48) which are special multiplications and should be subtracted from this count. Here, $M_8(1) = 0$ and $M_8(2) = 0$. Solving this recurrence relation using Master Equation I results in

$$M_8(n) = \begin{cases} \frac{1}{12} \cdot n \cdot \log_2(n) & \text{if } \log_2(n) \bmod 3 = 0 \\ \frac{1}{12} \cdot n \cdot \log_2(n) - \frac{1}{12} \cdot n & \text{if } \log_2(n) \bmod 3 = 1 \\ \frac{1}{12} \cdot n \cdot \log_2(n) - \frac{1}{6} \cdot n & \text{if } \log_2(n) \bmod 3 = 2 \end{cases} \tag{2.51}$$

special multiplications.

Modeling an addition in $\mathbb{C}$ by 2 additions in $\mathbb{R}$, a multiplication in $\mathbb{C}$ by 4 multiplications and 2 additions in $\mathbb{R}$, and a multiplication in $\mathbb{C}$ by a primitive 8th root of unity with 2 multiplications and 2 additions in $\mathbb{R}$, the total number of operations in $\mathbb{R}$ needed to implement the radix-8 algorithm for an input size which is a power of eight is given by

$$M_{\mathbb{R}}(n) = \frac{4}{3} \cdot n \cdot \log_2(n) - 4 \cdot n + 4, \tag{2.52}$$

$$A_{\mathbb{R}}(n) = \frac{11}{4} \cdot n \cdot \log_2(n) - 2 \cdot n + 2. \tag{2.53}$$

This represents a savings of $1/6 \cdot n \cdot \log_2(n)$ additions in $\mathbb{R}$ compared to the radix-4 algorithms. The savings for other input sizes are close to the above results, but not quite as attractive. The operation counts for the twisted radix-8 FFT algorithm are the same as the classical radix-8 algorithm.

Radix-16 algorithms have been proposed (e.g. [6]), but they do not improve upon the operation counts given in this section. This is because there does not appear to be anything special involved in multiplying by a primitive 16th root of unity that can be exploited to reduce the overall effort.

2.8 Split-radix FFT

It is possible to improve upon the counts of the radix-8 algorithm by constructing an algorithm which combines the radix-2 and radix-4 FFT reduction steps. This can be done for both the classical and twisted formulations of the algorithm. Here, we will show how to develop such an algorithm for the twisted case. This algorithm

was introduced in [84], but first clearly described and named over 15 years later in [21].

Suppose that we wish to compute the FFT of some polynomial f of degree less than n. It can be shown that this FFT can be computed by recursively applying the twisted radix-4 algorithm where the input to each reduction step is $f^\circ(x) = f(\omega^{\sigma(j)/(4m)} \cdot x) \bmod (x^{4m} - 1)$ for some $j < m$. The twisted radix-4 algorithm took advantage of the fact that it was possible to reduce f° into $f^\circ \bmod (x^m - 1)$, $f^\circ \bmod (x^m + 1)$, $f^\circ \bmod (x^m - \mathtt{I})$ and $f^\circ \bmod (x^m + \mathtt{I})$. All but the first of these results are then twisted in order to continue using the simplied reduction step. However, it is not necessary to twist $f^\circ \bmod (x^m + 1)$ at this point. If $m^* = 2m$, then it is possible to reduce $f^\circ \bmod (x^{2m^*} + 1)$ into $f^\circ \bmod (x^{m^*} - \mathtt{I})$ and $f^\circ \bmod (x^{m^*} + \mathtt{I})$ without any multiplications in R. The radix-4 algorithm does not exploit this situation and thus there is room for improvement. Even greater savings can be achieved if we reduce f° into $f^\circ \bmod (x^{4m^*} - 1)$ instead and then reduce this polynomial into $f^\circ \bmod (x^{m^*} - \mathtt{I})$ and $f^\circ \bmod (x^{m^*} + \mathtt{I})$.

The split-radix algorithm is based on a reduction step that transforms $f^\circ(x) = f(\omega^{\sigma(j)/(4m)} \cdot x) \bmod (x^{4m} - 1)$ into $f^\circ \bmod (x^{2m} - 1)$, $f^\circ \bmod (x^m - \mathtt{I})$, and $f^\circ \bmod (x^m + \mathtt{I})$. The algorithm is called "split-radix" because this reduction step can be viewed as a mixture of the radix-2 and radix-4 reduction steps. The two results of size m need to be twisted after the reduction step while the split-radix reduction step can be directly applied to $f^\circ \bmod (x^{2m} - 1)$.

The transformation used in the split-radix algorithm is given by

$$\begin{aligned} R[x]/(x^{4m}-1) &\rightarrow R[x]/(x^{2m}-1) \\ &\times R[x]/(x^m-\mathtt{I}) \\ &\times R[x]/(x^m+\mathtt{I}). \end{aligned} \tag{2.54}$$

Let $f^\circ = f(\omega^{\sigma(j)/(4m)} \cdot x) \bmod (x^{4m}-1)$, the input to the split-radix algorithm reduction step, be expressed as $f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$. We will use part of the radix-2 reduction step to obtain $f_W \cdot x^m + f_X = f^\circ \bmod (x^{2m}-1)$ and part of the radix-4 reduction step to obtain $f_Y = f^\circ \bmod (x^m - \mathtt{I})$ and $f_Z = f^\circ \bmod (x^m + \mathtt{I})$. These results are computed using

$$f_W = f_A + f_C, \tag{2.55}$$

$$f_X = f_B + f_D, \tag{2.56}$$

$$f_Y = -\mathtt{I} \cdot f_A - f_B + \mathtt{I} \cdot f_C + f_D, \tag{2.57}$$

$$f_Z = \mathtt{I} \cdot f_A - f_B - \mathtt{I} \cdot f_C + f_D. \tag{2.58}$$

After the reduction step, f_Y should be twisted by $\omega^{\sigma(2)/m}$ and f_Z can be twisted by $\omega^{\sigma(3)/m}$ so that the split-radix algorithm reduction step can be applied to these results as well as $f_W \cdot x^m + f_X$.

In 1989, an algorithm called the conjugate-pair split-radix algorithm was proposed [45]. The main difference in the reduction step of this algorithm compared to the split-radix FFT described above is that f_Z is twisted by $\omega^{-\sigma(2)/m}$ instead of $\omega^{\sigma(3)/m}$. Originally, it was claimed that this reduces the number of operations of the

split-radix algorithm, but it was later demonstrated ([37], [50]) that the two algorithms require the same number of operations. However, the conjugate-pair version requires less storage by exploiting the fact that $\omega^{\sigma(2)/m}$ and $\omega^{-\sigma(2)/m}$ are conjugate pairs. The conjugate-pair version of the algorithm requires a different "scrambling" function $\sigma'(j)$ which is defined according to the following formulas

$$\begin{aligned}
\sigma'(0) &= 0, && (2.59)\\
\sigma'(j) &= 2 \cdot \sigma'(2j) \text{ for } j < n/2, && (2.60)\\
\sigma'(4j+1) &= \sigma'(4j) + n/2 \text{ for } j < n/2, && (2.61)\\
\sigma'(4j+3) &= \sigma'(4j) + \rho(4j+3) \cdot n/4 \text{ for } j < n/4, && (2.62)
\end{aligned}$$

and $\rho(j)$ is defined as follows

$$\rho(j) = \begin{cases} -1 & \text{if } j \text{ is even} \\ 1 & \text{if } j \text{ is an integer of the form } 4\ell+1 \\ \rho(\ell) & \text{if } j \text{ is an integer of the form } 4\ell+3 \end{cases} . \quad (2.63)$$

Note that $\sigma'(j)$ is no longer the binary reversal function. If one wishes to view $\sigma'(j)$ as a permutation of the integers $\{0, 1, 2, \cdots n-1\}$, then one should add n to each negative value of $\sigma'(j)$ at the end of the construction. Pseudocode for the conjugate pair version of the split-radix algorithm based on a observations given in [4] is provided in Figure 2.4. The function $\sigma'(j)$ can simply be changed to $\sigma(j)$ to obtain the more common version of the split-radix algorithm.

Algorithm : Split-radix FFT (conjugate-pair version)

Input: $f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1)$ where m is a power of two such that $4m < n$.

Output: $f(\omega^{\sigma'(j \cdot 4m)}), f(\omega^{\sigma'(j \cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j \cdot 4m+4m-1)})$.

0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(j)}) = f(\omega^{\sigma'(j)} \cdot x) \bmod (x - 1)$.

0B. If $(4m) = 2$, then call a radix-2 algorithm to compute the FFT.

1. Split $f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1)$ into four blocks f_A, f_B, f_C, and f_D each of size m such that
   $f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1) = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
2. Compute $f_W \cdot x^m + f_X$
   $= f(\omega^{\sigma'(2j)/(2m)} \cdot x) \bmod (x^{2m} - 1) = (f_A + f_C) \cdot x^m + (f_B + f_D)$.
3. Compute $f_\alpha = -f_B + f_D$.
4. Compute $f_\beta = \mathtt{I} \cdot (-f_A + f_C)$.
5. Compute $f_Y = f_\alpha + f_\beta$.
6. Compute $f_Z = -f_\alpha + f_\beta$.
7. Compute $f(\omega^{\sigma'(4j+2)/m} \cdot x) \bmod (x^m - 1)$ by twisting f_Y by $\omega^{\sigma'(2)/m}$.
8. Compute $f(\omega^{\sigma'(4j+3)/m} \cdot x) \bmod (x^m - 1)$ by twisting f_Z by $\omega^{-\sigma'(2)/m}$.
9. Compute the FFT of $f(\omega^{\sigma'(2j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$ to obtain
   $f(\omega^{\sigma'(j \cdot 4m)}), f(\omega^{\sigma'(j \cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j \cdot 4m+2m-1)})$.
10. Compute the FFT of $f(\omega^{\sigma'(4j+2)/m} \cdot x) \bmod (x^m - 1)$ to obtain
    $f(\omega^{\sigma'(j \cdot 4m+2m)}), f(\omega^{\sigma'(j \cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma'(j \cdot 4m+3m-1)})$.
11. Compute the FFT of $f(\omega^{\sigma'(4j+3)/m} \cdot x) \bmod (x^m - 1)$ to obtain
    $f(\omega^{\sigma'(j \cdot 4m+3m)}), f(\omega^{\sigma'(j \cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma'(j \cdot 4m+4m-1)})$.
12. Return $f(\omega^{\sigma'(j \cdot 4m)}), f(\omega^{\sigma'(j \cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j \cdot 4m+4m-1)})$.

Figure 2.4 Pseudocode for split-radix FFT (conjugate-pair version)

Let us now analyze the cost of this algorithm. Lines 0A and 0B are used to end the recursion. Line 0A costs no operations while line 0B costs no multiplications and 2 additions. Line 1 just involves logicially partitioning the input into four blocks of size m, requiring no operations. In line 2, we add the f_A to f_C at a cost of m additions and f_B to f_D at a cost of m additions. Lines 3 through 6 show an efficient method of computing the rest of the reduction step. Each line costs m additions or subtractions. Then, lines 7 and 8 each involve a twisting of a polynomial of size m at a cost of $m-1$ multiplications. Finally, lines 9-11 recursively call the algorithm to complete the computation. Line 12 requires no operations.

The total number of operations to compute the FFT of size n using the split-radix algorithm is

$$M(n) = M\left(\frac{n}{2}\right) + 2 \cdot M\left(\frac{n}{4}\right) + \frac{1}{2} \cdot n - 2, \tag{2.64}$$

$$A(n) = A\left(\frac{n}{2}\right) + 2 \cdot A\left(\frac{n}{4}\right) + \frac{3}{2} \cdot n, \tag{2.65}$$

where $M(1) = 0$, $A(1) = 0$, $M(2) = 0$, and $A(2) = 2$. A solution to these recurrence relations is given in the appendix and is shown to be

$$M(n) = \begin{cases} \frac{1}{3} \cdot n \cdot \log_2(n) - \frac{8}{9} \cdot n + \frac{8}{9} & \text{if } \log_2(n) \text{ is even} \\ \frac{1}{3} \cdot n \cdot \log_2(n) - \frac{8}{9} \cdot n + \frac{10}{9} & \text{if } \log_2(n) \text{ is odd} \end{cases}, \tag{2.66}$$

$$A(n) = n \cdot \log_2(n). \tag{2.67}$$

If R contains the element $\sqrt{2}/2$, then the number of special multiplications by primitive 8th roots of unity is given by

$$M_8(n) = M_8\left(\frac{n}{2}\right) + 2 \cdot M_8\left(\frac{n}{4}\right) + 2, \tag{2.68}$$

where $M_8(1) = 0$, $M_8(2) = 0$ and $M_8(4) = 0$. The solution to this recursion is also given in the appendix and shown to be

$$M_8(n) = \begin{cases} \frac{1}{3} \cdot n - \frac{4}{3} & \text{if } \log_2(n) \text{ is even} \\ \frac{1}{3} \cdot n - \frac{2}{3} & \text{if } \log_2(n) \text{ is odd} \end{cases}. \tag{2.69}$$

If the coefficient ring of the input polynomial is the field of complex numbers, we can show that the number of operations in $\mathbb{R}$ needed to implement the split-radix algorithm is given by

$$M_{\mathbb{R}}(n) = \frac{4}{3} \cdot n \cdot \log_2(n) - \frac{38}{9} \cdot n + \frac{2}{9} \cdot (-1)^{\log_2(n)} + 6, \tag{2.70}$$

$$A_{\mathbb{R}}(n) = \frac{8}{3} \cdot n \cdot \log_2(n) - \frac{16}{9} \cdot n - \frac{2}{9} \cdot (-1)^{\log_2(n)} + 2. \tag{2.71}$$

The results for input sizes which are a power of 4 are slightly more attractive than other input sizes. In any event, the split-radix algorithm requires less operations than the radix-8 algorithm.

An "extended" split-radix [77] algorithm which mixes the radix-2 and radix-8 reduction steps has been proposed, but has been shown [7] to require the same total number of operations as the algorithm presented in this section. This paper [7] also claims that split-radix algorithms which mix the radix-2 and radix-16 reduction steps and radix-4 and radix-16 reduction steps also require the same number of operations.

Another claim given in [7] is that a split-radix algorithm combining any two different radix reduction steps from the 2-adic FFT algorithms requires the same total number of operations. [4]

2.9 Modified split-radix FFT

For many years, it was believed that the split-radix algorithm presented in the previous section required the fewest operations to compute an FFT of size 2^k. Yet around 2004, Van Buskirk was able to improve upon the split-radix algorithm in certain cases and posted his findings on the internet according to [44]. During the next several years, Johnson and Frigo [44] developed an algorithm which requires fewer operations than the split-radix algorithm for all input sizes when the coefficient ring for these polynomials is the field of complex numbers. A preprint of this article was available on the internet and used by the present author to develop a presentation of this algorithm using the mathematical perspective of the FFT. Meanwhile, Bernstein independently used the preprint to develop his own version of the algorithm from the mathematical perspective and called his algorithm the "Tangent FFT". Bernstein posted a description of the Tangent FFT on his web page [4] after the present author completed the work summarized in this section. The algorithm which follows can be viewed as a mixture of Johnson and Frigo's algorithm and Bernstein's algorithm. The same steps of Johnson and Frigo's algorithm will be used, but will be presented in terms of the mathematical perspective of the FFT.

---

[4] If a complex multiplication is assumed to require 3 multiplications and 3 additions in $\mathbb{R}$, then the multiplication operation counts and addition operation counts of the split-radix algorithms are all exactly the same. If a complex multiplication requires 4 multiplications and 2 additions, then some of the other split-radix algorithms require slightly fewer multiplications at the expense of the same number of extra addition operations in $\mathbb{R}$. Modern computers implement a multiplication in $\mathbb{R}$ in approximately the same amount of time as an addition in $\mathbb{R}$ so there does not appear to be any significant advantage to the other algorithms.

For the remainder of this section, we will again assume that R is the field of complex numbers. The new algorithms are based on the fact that any power of ω can be scaled into a special multiplication that requires 2 multiplications and 2 additions. The conjugate pair version of split-radix FFT algorithm is the foundation of the new algorithms, so all powers of ω involved (other than ω^0) will lie in the first and fourth quadrants of $\mathbb{C}$. If $\mathcal{W} = \omega^{\pm\sigma'(2)/m}$ is a primitive $(4m)$th root of unity, then multiplication by either $\mathcal{W}^d/\cos(\pi/2 \cdot d/m)$ or $\mathcal{W}^d/\sin(\pi/2 \cdot d/m)$ can be implemented using 2 multiplications and 2 additions (provided that the needed trigonometic function evaluations are precomputed and stored in memory).

The only place that we encounter multiplications by ω in the pseudocode for the split-radix FFT algorithm (Figure 2.4) is in the twisting of the polynomials found in lines 7 and 8 of the algorithm. The new split-radix algorithm will scale each of the powers of ω involved in these two lines to be one of the special multiplications that requires fewer multiplications in $\mathbb{R}$. For the moment, let us assume that all of these multiplications will be of the form $\mathcal{W}^d/\cos(\pi/2 \cdot d/m)$.

Instead of twisting f_Y and f_Z at the end of each reduction step as described in the previous sections, we will instead compute scaled versions of these twisted polynomials. For example, f_Y will be twisted according to

$$\widetilde{f_Y} = \sum_{d=0}^{m-1} \mathcal{W}^d/\cos(\pi/2 \cdot d/m) \cdot (f_Y)_d \tag{2.72}$$

$$= \sum_{d=0}^{m-1} (1 + \mathtt{I} \cdot \tan(\pi/2 \cdot d/m)) \cdot (f_Y)_d \tag{2.73}$$

at a savings of 2 multiplications in $\mathbb{R}$ per coefficient in $\widetilde{f_Y}$. Observe that a different

evaluation of the tangent function is used for each coefficient in the scaled twisting of $\widetilde{f}_Y$. Similar results hold for the computation of $\widetilde{f}_Z$.

Of course, scaling the twisted polynomials in this manner will distort the results that are used to compute the FFT and will ultimately produce an incorrect answer. To compensate for this, each coefficient of the input polynomial used in the reduction step must be viewed as a scaled version of the coefficient of the actual input polynomial and must be "unscaled" at a later reduction step. For a given reduction step, we will view the coefficient of degree d of the input polynomial as being scaled by c_d where

$$c_d \quad = \quad \cos(\pi/2 \cdot d_4/m) \tag{2.74}$$

and d_4 is the reminder when d is divided by 4, i.e. $d \bmod 4$.

In this next reduction step, $f_W \cdot x + f_X$ is now incorrectly scaled by $\cos(\pi/2 \cdot d_4/m)$. We could scale each coefficient of $f_W \cdot x + f_X$ by $1/\cos(\pi/2 \cdot d_4/m)$ to undo the scaling at this point, but each scaling would cost 2 multiplications in $\mathbb{R}$ per coefficient in this output polynomial and we will have lost all of the savings gained by working with the scaled twisted polynomials.

The key to the savings claimed by the new approach is to make adjustments to $f_W \cdot x + f_X$ only when absolutely necessary and push any other adjustments that need to be made to this result until later reduction steps of the algorithm. However, this will result in four different variations of the reduction step used in the new algorithm.

Another concern that needs to be addressed is the numerical stability of the scaling factors. Mathematicians who specialize in numerical analysis are concerned about situations where one divides by a number close to zero as this is the source of

roundoff errors in numerical calculations. To reduce the numerical concerns in the new algorithm, we will use $\cos(\pi/2 \cdot d_4/m)$ as the scaling factor when $0 \leq d_4/m \leq 1/2$ and $\sin(\pi/2 \cdot d_4/m)$ as the scaling factor when $1/2 \leq d_4/m < 1$. [5] Thus, the scaled twisted polynomials will be computed using

$$T_{4m,d} = \begin{cases} 1 \pm \mathtt{I} \cdot \tan(\pi/2 \cdot d_4/m) & \text{if } 0 \leq d_4/m \leq 1/2 \\ \cot(\pi/2 \cdot d_4/m) \mp \mathtt{I} & \text{if } 1/2 < d_4/m < 1 \end{cases} . \tag{2.75}$$

Since we will apply scaled twisted polynomials to every reduction step of the algorithm, we desire to select the scalings of the input to each reduction step in such a way as to compensate for the cumulative effect of all of the scaled twisted polynomials that will be encountered after that point in the algorithm.

Let $S_{4m,d}$ be the scaling factor that is applied to the degree d term of the input to a reduction step. Each value for $S_{4m,d}$ will be assigned recursively so that the desired cumulative effect is achieved.

If $(4m) \leq 4$, then $S_{4m,d} = 1$ for all $d < 4$. This is because we do not wish to perform any adjustments to $f_W \cdot x + f_X$ or the radix-2 reduction steps to achieve the desired FFT.

For all $(4m) > 4$, then $S_{4m,d} = \cos(\pi/2 \cdot d_4/m) \cdot S_{m,d_4}$ for all $d_4/m \leq 1/2$ and $S_{4m,d} = \sin(\pi/2 \cdot d_4/m) \cdot S_{m,d_4}$ for all $d_4/m > 1/2$. Again, the notation d_4 means to compute d mod 4 and is motivated by the need to match the cumulative scaling factors of the corresponding degree terms of f_A, f_B, f_C, and f_D with respect to the reduction

[5] Since all nonunity powers of ω used in the algorithm lie in the first quadrant, then the interval $0 \leq d_4/m < 1$ covers all cases.

steps of size m and lower. Observe that $S_{4m,d} = S_{4m,d+m} = S_{4m,d+2m} = S_{4m,d+3m}$ for all $d < m$, an important property that will be used in the reduction steps.

So $S_{4m,d}$ can be defined recursively using

$$S_{4m,d} = \begin{cases} 1 & \text{if } (4m) \leq 4 \\ \cos(\pi/2 \cdot d_4/m) \cdot S_{m,d_4} & \text{if } (4m) > 4 \text{ and } d_4/m \leq 1/2 \\ \sin(\pi/2 \cdot d_4/m) \cdot S_{m,d_4} & \text{if } (4m) > 4 \text{ and } d_4/m > 1/2 \end{cases} . \quad (2.76)$$

With the $S_{4m,d}$'s defined in this manner, the scaled twisted polynomials of the reduction step of size $4m$ will undo the $T_{4m,d}$ of $S_{4m,d}$ and will produce polynomials of degree less than m. Each of these polynomials becomes the input to a reduction step that has been scaled by $S_{m,d'}$ where $d' < m$.

Pseudocode for the four different reduction steps of the new split-radix algorithm are presented in the appendix along with an explanation of each version. Furthermore, analysis is provided which shows that the new algorithm requires

$$\begin{aligned} M_{\mathbb{R}}(n) = & \frac{6}{9} \cdot n \cdot \log_2(n) - \frac{10}{27} \cdot n + 2 \cdot \log_2(n) \\ & - \frac{2}{9} \cdot (-1)^{\log_2(n)} \cdot \log_2(n) + \frac{22}{27} \cdot (-1)^{\log_2(n)} + 2 \end{aligned} \quad (2.77)$$

multiplications in $\mathbb{R}$, roughly six percent lower than the number of multiplications required for the split-radix algorithm.

The reduced operation count of the new algorithm may be more of theoretical interest than a result that can produce an improved algorithm of practical value. In developing the FFTW package, Johnson and Frigo have observed that FFT algorithms with higher theoretical operation counts can sometimes run faster on modern

computers than split-radix algorithms. As mentioned in Chapter 1, this is because the actual running time of an FFT algorithm depends on many factors besides the number of multiplications and additions. These other factors have a variable influence from computer to computer and are difficult to model mathematically.

2.10 The ternary reversal function

We will now turn our attention to algorithms that can compute an FFT of size $n = 3^k$. Like the multiplicative FFT algorithms where $p = 2$, the algorithms that we will encounter in the following sections do not present the output in the natural order. For multiplicative FFTs with $p = 3$, the output will be presented in the order $\{f(1), f(\omega^{\Delta(1)}), f(\omega^{\Delta(2)}), f(\omega^{\Delta(3)}), \ldots, f(\omega^{\Delta(n-1)})\}$ where $\Delta(j)$ is the "ternary reversal" of j (with respect to $n = 3^k$). That is to say, if j can be expressed in ternary form as $(t_{k-1}t_{k-2}t_{k-3}\ldots t_2t_1t_0)_3$, then $\Delta(j) = (t_0t_1t_2\ldots t_{k-3}t_{k-2}t_{k-1})_3$. For example, if $j = 25$ and $n = 81$, then express $j = 0221_3$. So $\Delta(j) = 1220_3 = 51$. Note that leading zeros should be included in the ternary representation of j and will contribute to the value of $\Delta(j)$. The Δ function is a permutation of the integers $\{0, 1, 2, \ldots n-1\}$ and if the "natural" order of the evaluations is desired, then the inverse of the Δ function can be used to rearrange the output of the FFT algorithm into this order.

The Δ function has several properties which will be important to the development of the algorithms presented in the following sections. The proof of each of these theorems is similar to those used to prove Theorems 1 and 2.

Theorem 7 $\Delta(j) = 3 \cdot \Delta(3j)$ *for* $j < n/3$ *where* $n = 3^k$.

Theorem 8 $\Delta(3j+1) = \Delta(3j) + n/3$ *for* $j < n/3$ *and where* $n = 3^k$.

Theorem 9 $\Delta(3j+2) = \Delta(3j) + 2n/3$ *for* $j < n/3$ *and where* $n = 3^k$.

2.11 Classical radix-3 FFT

A radix-3 FFT algorithm evaluates a polynomial of degree less than n at each of the roots of $x^n - 1$ in some ring R where $n = 3^k$. Then R must have a primitive nth root of unity ω and a total of n evaluations will be computed. In [20], a radix-3 algorithm was introduced which required the inputs to be transformed into a different number system which the authors call the "slanted complex numbers". Here, we will present a new Cooley-Tukey formulation of the radix-3 algorithm which allows the inputs to remain in the traditional complex number system and reduces the operation counts of [20]. The new algorithm is similar to a technique introduced by Winograd as presented in [61], but is believed to be easier to understand than the Winograd algorithm.

Using the comments found in [3], a Cooley-Tukey radix-3 FFT is based on the transformation

$$\begin{aligned} R[x]/(x^{3m} - b^3) &\rightarrow R[x]/(x^m - b) \\ &\times R[x]/(x^m - \Omega \cdot b) \\ &\times R[x]/(x^m - \Omega^2 \cdot b), \end{aligned} \tag{2.78}$$

where $\Omega = \omega^{n/3}$ and $\Omega^3 = 1$. Note that Ω also has the property that $\Omega^2 + \Omega + 1 = 0$. Using these properties of Ω, it can be shown that $x^{3m} - b^3$ factors as $(x^m - b) \cdot (x^m - \Omega \cdot b) \cdot (x^m - \Omega^2 \cdot b)$.

The reduction step of the radix-3 FFT receives as input $f^\circ = f \bmod (x^{3m} - b^3)$ and computes $f_X = f \bmod (x^m - b)$, $f_Y = f \bmod (x^m - \Omega \cdot b)$, and $f_Z = f \bmod (x^m - \Omega^2 \cdot b)$. The modular reduction is once again simple to perform. Split f° into three blocks of size m by writing $f^\circ = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$. Then

$$f_X = b^2 \cdot f_A + b \cdot f_B + f_C, \tag{2.79}$$

$$f_Y = \Omega^2 \cdot b^2 \cdot f_A + \Omega \cdot b \cdot f_B + f_C, \tag{2.80}$$

$$f_Z = \Omega \cdot b^2 \cdot f_A + \Omega^2 \cdot b \cdot f_B + f_C. \tag{2.81}$$

Suppose that we want to compute the FFT of a polynomial f of degree less than $n = 3^k$ over some ring R. We will recursively apply the reduction step with appropriate selections of m and b. Since $(\omega^{\Delta(3j)})^3 = \omega^{\Delta(j)}$, $\Omega \cdot \omega^{\Delta(3j)} = \omega^{\Delta(3j+1)}$ and $\Omega^2 \cdot \omega^{\Delta(3j)} = \omega^{\Delta(3j+2)}$ for all $j < n/3$, then b can be easily determined. Each reduction step receives as input $f \bmod (x^{3m} - \omega^{\Delta(j)})$ for some j and produces as output

$$f_X = f \bmod (x^m - \omega^{\Delta(3j)}), \tag{2.82}$$

$$f_Y = f \bmod (x^m - \omega^{\Delta(3j+1)}), \tag{2.83}$$

$$f_Z = f \bmod (x^m - \omega^{\Delta(3j+2)}). \tag{2.84}$$

The reduction step can now be expressed as the transformation

$$\begin{pmatrix} f_X \\ f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} \omega^{2\Delta(3j)} & \omega^{\Delta(3j)} & 1 \\ \Omega^2 \cdot \omega^{2\Delta(3j)} & \Omega \cdot \omega^{\Delta(3j)} & 1 \\ \Omega \cdot \omega^{2\Delta(3j)} & \Omega^2 \cdot \omega^{\Delta(3j)} & 1 \end{pmatrix} \cdot \begin{pmatrix} f_A \\ f_B \\ f_C \end{pmatrix} \tag{2.85}$$

or

$$\begin{pmatrix} f_X \\ f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ \Omega^2 & \Omega & 1 \\ \Omega & \Omega^2 & 1 \end{pmatrix} \cdot \begin{pmatrix} \omega^{2\Delta(3j)} \cdot f_A \\ \omega^{\Delta(3j)} \cdot f_B \\ f_C \end{pmatrix}, \tag{2.86}$$

where $f_A \cdot x^{2m} + f_B \cdot x^m + f_C = f \bmod (x^{3m} - \omega^{\Delta(j)})$. After all of the reduction steps have been completed with input size $3m = 3$, then we have $f \bmod (x - \omega^{\Delta(j)}) = f(\omega^{\Delta(j)})$ for all $j < n$, i.e. the desired FFT of f.

Both Winograd's algorithm and the new algorithm exploit the fact that Ω and Ω^2 are complex conjugates. Express $\Omega = \Omega_R + \mathtt{I} \cdot \Omega_I$ and observe that $\Omega^2 = \Omega_R - \mathtt{I} \cdot \Omega_I$ where $\Omega_R = \cos(120^o)$ and $\Omega^I = \sin(120^o)$ or its equivalent in R. Now let f_R and f_I be defined by

$$f_R = \Omega_R \cdot (\omega^{2\Delta(3j)} \cdot f_A + \omega^{\Delta(3j)} \cdot f_B), \tag{2.87}$$

$$f_I = \mathtt{I} \cdot \Omega_I \cdot (\omega^{\Delta(3j)} \cdot f_B - \omega^{2\Delta(3j)} \cdot f_A). \tag{2.88}$$

Without much difficulty, it can be shown that $f_Y = f_R + f_C + f_I$ and $f_Z = f_R + f_C - f_I$. To efficiently perform the computations, we first compute $f_R + f_C$ and f_I. The two results are added to obtain f_Y and are subtracted to obtain f_Z. Finally, f_X is computed using $(\omega^{2\Delta(3j)} \cdot f_A + \omega^{\Delta(3j)} \cdot f_B) + f_C$ where $\omega^{2\Delta(3j)} \cdot f_A + \omega^{\Delta(3j)} \cdot f_B$ has already computed to determine f_R. The Winograd algorithm uses a somewhat different sequence of operations to produce the three outputs at the same cost. Pseudocode for the new classical radix-3 algorithm is provided in Figure 2.5.

Let us now analyze the cost of this algorithm. Line 0 is just used to end the recursion and costs no operations. Line 1 logicially partitions the input into three

Algorithm : New classical radix-3 FFT

Input: $f \bmod (x^{3m} - \omega^{\Delta(j)})$, a polynomial of degree less than $3m$ in a ring R.
Here R has a nth root of unity ω, and m is a power of three where $3m \leq n$.

Output: $f(\omega^{\Delta(j\cdot 3m+0)}), f(\omega^{\Delta(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta(j\cdot 3m+3m-1)})$.

0. If $(3m) = 1$ then return $f \bmod (x - \omega^{\Delta(j)}) = f(\omega^{\Delta(j)})$.
1. Split $f \bmod (x^{3m} - \omega^{\Delta(j)})$ into three blocks f_A, f_B, and f_C each of size m such that $f \bmod (x^{3m} - \omega^{\Delta(j)}) = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$.
2. Compute $f_\alpha = \omega^{2\Delta(3j)} \cdot f_A$ and $f_\beta = \omega^{\Delta(3j)} \cdot f_B$.
3. Compute $f_\gamma = f_\alpha + f_\beta$ and $f_\delta = f_\beta - f_\alpha$.
4. Compute $f_R + f_C = \Omega_R \cdot f_\gamma + f_C$ and $f_I = \mathtt{I} \cdot f_\delta$.
5. Compute $f \bmod (x^m - \omega^{\Delta(3j)}) = f_\gamma + f_C$.
6. Compute $f \bmod (x^m - \omega^{\Delta(3j+1)}) = (f_R + f_C) + f_I$.
7. Compute $f \bmod (x^m - \omega^{\Delta(3j+2)}) = (f_R + f_C) - f_I$.
8. Compute the FFT of $f \bmod (x^m - \omega^{\Delta(3j)})$ to obtain $f(\omega^{\Delta(j\cdot 3m+0)}), f(\omega^{\Delta(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta(j\cdot 3m+m-1)})$.
9. Compute the FFT of $f \bmod (x^m - \omega^{\Delta(3j+1)})$ to obtain $f(\omega^{\Delta(j\cdot 3m+m)}), f(\omega^{\Delta(j\cdot 3m+m+1)}), \ldots, f(\omega^{\Delta(j\cdot 3m+2m-1)})$.
10. Compute the FFT of $f \bmod (x^m - \omega^{\Delta(3j+2)})$ to obtain $f(\omega^{\Delta(j\cdot 3m+2m)}), f(\omega^{\Delta(j\cdot 3m+2m+1)}), \ldots, f(\omega^{\Delta(j\cdot 3m+3m-1)})$.
11. Return $f(\omega^{\Delta(j\cdot 3m+0)}), f(\omega^{\Delta(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta(j\cdot 3m+3m-1)})$.

Figure 2.5 Pseudocode for new classical radix-3 FFT

blocks of size m, requiring no operations. In line 2, we multiply the m components of f_A by $\omega^{2\Delta(3j)}$ and the m components of f_B by $\omega^{\Delta(3j)}$. Each of these operations costs m multiplications in R unless $j = 0$ in which case no multiplications are required. Line 3 requires $2m$ additions in R and the cost of line 4 is equivalent to m multiplications in R. Now, lines 5, 6, and 7 each require m additions in R to add the two terms in each instruction. The cost of lines 8, 9, and 10 is equal to the number of operations needed to compute three FFTs of size m. Line 11 costs no operations. The total number of operations to compute this radix-3 FFT of size n using the new algorithm is given by

$$M(n) = 3 \cdot M\left(\frac{n}{3}\right) + n, \tag{2.89}$$

$$A(n) = 3 \cdot A\left(\frac{n}{3}\right) + \frac{5}{3} \cdot n, \tag{2.90}$$

where $M(1) = 0$ and $A(1) = 0$. These recurrence relations can be solved by using Master Equation III. We must also subtract multiplications to account for the cases where $j = 0$. A recurrence relation giving the number of multiplications saved is

$$M_s(n) = M_s\left(\frac{n}{3}\right) + \frac{2}{3} \cdot n, \tag{2.91}$$

where $M_s(1) = 0$. By applying the same technique used to produce Master Equation II, it can be shown that $M_s(n) = n - 1$.

The number of operations needed to compute the FFT using this radix-3 algorithm is given by

$$
\begin{aligned}
M(n) &= n \cdot \log_3(n) - n + 1 && (2.92)\\
&= \frac{1}{\log_2(3)} \cdot n \cdot \log_2(n) - n + 1 \\
&\approx 0.631 \cdot n \cdot \log_2(n) - n + 1, \\
A(n) &= \frac{5}{3} \cdot n \cdot \log_3(n) && (2.93)\\
&= \frac{5}{3 \cdot \log_2(3)} \cdot n \cdot \log_2(n) \\
&\approx 1.051 \cdot n \cdot \log_2(n).
\end{aligned}
$$

This is the same number of operations required by the Winograd algorithm. It can be shown that the algorithm of [20] requires

$$
\begin{aligned}
M(n) &= \frac{2}{3} \cdot n \cdot \log_3(n) - n + 1, && (2.94)\\
A(n) &= \frac{10}{3} \cdot n \cdot \log_3(n) && (2.95)
\end{aligned}
$$

operations, provided that the complex numbers contained in each of the inputs and outputs of the algorithm are expressed as $A + \Omega \cdot B$. Provided that multiplication of two real numbers is less than twice as expensive as the addition of two real numbers, then the new algorithm will outperform the one presented in [20] and does not require conversions involving the slanted complex number system. However, the new algorithm is less efficient than the radix-2 algorithms discussed earlier in this chapter, both in terms of the number of multiplications and the number of additions.

2.12 Twisted radix-3 FFT

In [35], Gentleman and Saude gave an alternative algorithm to compute the radix-2 FFT. It is possible to develop a similar algorithm for the radix-3 FFT. In this section, we are going to present a new version of this algorithm which contains many of the features as an algorithm presented in [76]. In particular, the new algorithm exploits complex conjugate properties of complex numbers as well as the fact that $1 + \Omega + \Omega^2 = 0$.

The basic idea of the Gentleman-Saude radix-3 algorithm is to compute the reduction step using

$$\begin{aligned} R[x]/(x^{3m} - 1) &\rightarrow R[x]/(x^m - 1) \\ &\times R[x]/(x^m - \Omega) \\ &\times R[x]/(x^m - \Omega^2). \end{aligned} \tag{2.96}$$

The mapping $x = \zeta \cdot \dot{x}$ is always applied to $R[x]/(x^m - \Omega)$ after each reduction step where ζ is some element of R that transforms $R[x]/(x^m - \Omega)$ to $R[\dot{x}]/(\dot{x}^m - 1)$ by rotating the roots of unity and allows (2.96) to be applied at the next reduction step. A different value of ζ can be used in the transformation $x = \zeta \cdot \ddot{x}$ to convert $R[x]/(x^m - \Omega^2)$ to $R[\ddot{x}]/(\ddot{x}^m - 1)$. We will again adopt Bernstein's terminology and call this the "twisted" radix-3 FFT algorithm.

The "twisted" polynomial will again be used for the reduction step. As an example of an application of the twisted polynomial in this context, consider $f(x) = 3 \cdot x^3 + 2 \cdot x^2 + x + 1$. Suppose that ω is a primitive 3rd root of unity in the field of complex numbers, i.e. $\omega = \Omega$. Then $f(\omega \cdot \dot{x}) = (3\omega^3) \cdot \dot{x}^3 + (2\omega^2) \cdot \dot{x}^2 + \omega \cdot \dot{x} + 1 = 3 \cdot \dot{x}^3 + 2 \cdot \Omega^2 \cdot \dot{x}^2 + \Omega \cdot \dot{x} + 1$. Similarly, if $g(x) = 7 \cdot x^3 + 5x + 4$, then $g(\omega \cdot \dot{x}) = 7 \cdot \dot{x}^3 + 5 \cdot \Omega \cdot \dot{x} + 4$.

One can verify that the linear transformation properties of the twisted polynomial holds using these two results.

The following theorem provides the values of ζ necessary to achieve the desired transformations.

Theorem 10 *Let $f^\circ(x)$ be a polynomial of degree less than $3m$ in $R[x]$. Then $f^\circ(x) \bmod (x^m - \Omega) = f^\circ(\zeta \cdot \dot{x}) \bmod (\dot{x}^m - 1)$ where $\zeta = \omega^{\Delta(1)/m}$ and $x = \zeta \cdot \dot{x}$.*

Proof: Let $f^\circ(x)$ be a polynomial of degree less than $3m$ in $R[x]$ and let $r(x) = f^\circ(x) \bmod (x^m - \Omega)$. Then $f^\circ(x) = q(x) \cdot (x^m - \Omega) + r(x)$ for some $q(x)$. Now substitute $x = \zeta \cdot \dot{x}$ where $\zeta = \omega^{\Delta(1)/m}$ to rotate the roots of unity. Applying the linear properties of the twisted polynomial transformation gives

$$\begin{aligned} f^\circ(\zeta \cdot \dot{x}) &= q(\zeta \cdot \dot{x}) \cdot ((\zeta \cdot \dot{x})^m - \Omega) + r(\zeta \cdot \dot{x}) \\ &= q(\zeta \cdot \dot{x}) \cdot (\zeta^m \cdot \dot{x}^m - \Omega) + r(\zeta \cdot \dot{x}) \\ &= q(\zeta \cdot \dot{x}) \cdot (\omega^{\Delta(1)} \cdot \dot{x}^m - \Omega) + r(\zeta \cdot \dot{x}) \\ &= q(\zeta \cdot \dot{x}) \cdot (\Omega \cdot \dot{x}^m - \Omega) + r(\zeta \cdot \dot{x}) \\ &= \Omega \cdot q(\zeta \cdot \dot{x}) \cdot (\dot{x}^m - 1) + r(\zeta \cdot \dot{x}). \end{aligned} \tag{2.97}$$

Thus $r(\zeta \cdot \dot{x}) = f^\circ(\zeta \cdot \dot{x}) \bmod (\dot{x}^m - 1)$. □

Theorem 11 *Let $f^\circ(x)$ be a polynomial of degree less than $3m$ in $R[x]$. Then $f^\circ(x) \bmod (x^m - \Omega^2) = f^\circ(\zeta \cdot \ddot{x}) \bmod (\ddot{x}^m - 1)$ where $\zeta = \omega^{\Delta(2)/m}$ and $x = \zeta \cdot \ddot{x}$.*

Proof: Similar to proof of Theorem 10. □

As with the twisted radix-2 algorithms, it is simpler to ignore the distinctions between the various transformations of the roots of unity and just use the variable x in all of the polynomials. However, whenever the notation $f(\omega^{\theta} \cdot x)$ is used throughout the rest of this section, it is to be understood that x is a dummy variable for some other unknown, say $\dot{x}$, where $x = \omega^{\theta} \cdot \dot{x}$ and x is a variable used to represent untransformed polynomials in this equation.

We will now present the reduction step of the twisted radix-3 FFT algorithm which receives an input of some polynomial $f^{\circ}(x)$ with degree less than $3m$. Split f° into three blocks of size m by writing $f^{\circ} = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$. Then compute $f_X = f \bmod (x^m - 1) = f_A + f_B + f_C$, $f_Y = f \bmod (x^m - \Omega) = \Omega^2 \cdot f_A + \Omega \cdot f_B + f_C$, and $f_Z = f \bmod (x^m - \Omega^2) = \Omega \cdot f_A + \Omega^2 \cdot f_B + f_C$. This reduction step can also be represented by the transformation

$$\begin{pmatrix} f_X \\ f_Y \\ f_Z \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \\ \Omega^2 & \Omega & 1 \\ \Omega & \Omega^2 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_A \\ f_B \\ f_C \end{pmatrix}. \tag{2.98}$$

The simplified reduction step can be directly applied to $f^{\circ}(x) \bmod (x^m - 1)$. However, we need to twist $f^{\circ}(x) \bmod (x^m - \Omega)$ by $\omega^{\Delta(1)/m}$ and $f^{\circ}(x) \bmod (x^m - \Omega)$ by $\omega^{\Delta(2)/m}$ prior to using these results as inputs to the simplified reduction step.

The following theorem is provided to demonstrate that an algorithm based on this reduction step yields the FFT of some polynomial f of degree less than $n = 3^k$.

Theorem 12 *If* $f^{\circ}(x) = f(\omega^{\Delta(j)/(3m)} \cdot x) \bmod (x^{3m} - 1)$, *then* $f^{\circ}(\zeta \cdot x) \bmod (x^m - 1) = f(\omega^{\Delta(3j+d)/m} \cdot x) \bmod (x^m - 1)$ *where* $\zeta = \omega^{\Delta(d)/m}$ *for* $0 \leq d \leq 2$.

Proof: Let $0 \le d \le 2$, $\theta = \Delta(j)/(3m) = \Delta(3j)/m$ and let $f^\circ(x) = f(\omega^\theta \cdot x) \bmod (x^{3m}-1)$. Now, $\zeta \cdot (\omega^{\Delta(j)/(3m)}) = \omega^{\Delta(d)/m} \cdot \omega^{\Delta(3j)/m} = \omega^{\Delta(3j+d)/m}$. So $f^\circ(\zeta \cdot x) = f(\omega^{\Delta(3j+d)/m} \cdot x) \bmod (x^{3m}-1)$. Modularly reducing both sides of this equation by $x^m - 1$ produces the desired result. □

So the reduction step receives as input $f(\omega^{\Delta(j)/(3m)} \cdot x) \bmod (x^{3m}-1)$, the modular reduction of some polynomial f that has been twisted by $\omega^{\Delta(j)/(3m)} = \omega^{\Delta(3j)/m}$. The reduction step produces as output $f(\omega^{\Delta(3j)/m} \cdot x) \bmod (x^m - 1)$, $f(\omega^{\Delta(3j+1)/m} \cdot x) \bmod (x^m - 1)$ (after a "twist" by $\omega^{\Delta(1)/m}$), and $f(\omega^{\Delta(3j+2)/m} \cdot x) \bmod (x^m - 1)$ (after a "twist" by $\omega^{\Delta(2)/m}$).

Since $(\omega^{-\Delta(1)/m})^m = \omega^{-\Delta(1)} = \omega^{-n/3} = \omega^{2n/3} = \omega^{\Delta(2)} = (\omega^{\Delta(2)/m})^m$, then $\omega^{-\Delta(1)/m}$ is another value of ζ that can be used to twist $f^\circ(x) \bmod (x^m - \Omega^2)$ into $f^\circ(\zeta \cdot x) \bmod (x^m - 1)$. However, one would need to modify the ternary reversal function $\Delta(j)$ into a new function $\Delta'(j)$ which is compatible with the different twisting of the third output. This function shares most of the properties of $\Delta(j)$ except that $\Delta'(3j+2) = \Delta'(3j) - n/3$. The remaining details of converting $\Delta(j)$ to $\Delta'(j)$ are similar to the process of converting $\sigma(j)$ to $\sigma'(j)$ for the conjugate-pair split-radix algorithm and are left to the reader.

The improved algorithm makes use of the fact that $\omega^{\Delta'(1)/m}$ and $\omega^{-\Delta'(1)/m}$ are conjugate pairs and that Ω and Ω^2 are complex conjugate pairs, i.e. $\Omega^2 = \overline{\Omega}$. Several properties involving conjugates are proven in a section of the appendix and will be used to reduce the operation count of the algorithm.

Let $\zeta = \omega^{\Delta'(1)/m}$. The following formulas can be used to compute the coefficient of degree d in $f_X = f(\omega^{\Delta'(3j)/m}) \bmod (x^m - 1)$, $\widetilde{f_Y} = f(\omega^{\Delta'(3j+1)/m}) \bmod (x^m - 1)$, and $\widetilde{f_Z} = f(\omega^{\Delta'(3j+2)/m}) \bmod (x^m - 1)$: [6]

$$
\begin{aligned}
(f_X)_d &= (f_A)_d + (f_B)_d + (f_C)_d, && (2.99)\\
(\widetilde{f_Y})_d &= \left(\overline{\Omega} \cdot (f_A)_d + \Omega \cdot (f_B)_d + (f_C)_d\right) \cdot \zeta^d, && (2.100)\\
(\widetilde{f_Z})_d &= \left(\Omega \cdot (f_A)_d + \overline{\Omega} \cdot (f_B)_d + (f_C)_d\right) \cdot \overline{\zeta^d}. && (2.101)
\end{aligned}
$$

Since $\overline{\Omega} = -\Omega - 1$, then we can rewrite (2.100) as

$$
\begin{aligned}
(\widetilde{f_Y})_d &= (\Omega \cdot (f_B - f_A)_d + (f_C - f_A)_d) \cdot \zeta^d && (2.102)\\
&= (\Omega \cdot \zeta^d) \cdot (f_B - f_A)_d + \zeta^d \cdot (f_C - f_A)_d.
\end{aligned}
$$

By Theorem 44 found in the appendix, we can similarly rewrite (2.101) as

$$
(\widetilde{f_Z})_d = \overline{\Omega \cdot \zeta^d} \cdot (f_B - f_A)_d + \overline{\zeta^d} \cdot (f_C - f_A)_d. \quad (2.103)
$$

Once (2.102) has been computed, then Theorem 45 of the appendix can be used to compute (2.103) at a reduced cost. For both formulas, we will assume that $\Omega \cdot \zeta^d$ has

[6] The notation $\widetilde{f_Y}$ and $\widetilde{f_Z}$ is used to indicate that these are the outputs after the twisting has been applied.

Algorithm : Improved twisted radix-3 FFT

Input: $f^{\circ}(x) = f(\omega^{\Delta'(j)/(3m)} \cdot x) \bmod (x^{3m}-1)$, the modular reduction of some polynomial $f(x) \in R[x]$ that has been twisted by $\omega^{\Delta'(j)}$. Here R has a nth root of unity ω, and m is a power of three where $3m \leq n$.

Output: $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$.

0. If $(3m) = 1$ then return $f(\omega^{\Delta'(j)} \cdot x) \bmod (x-1) = f(\omega^{\Delta'(j)})$.
1. Split $f^{\circ}(x)$ into three blocks f_A, f_B, and f_C each of size m such that $f^{\circ}(x) = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$.
2. Compute $f(\omega^{\Delta'(3j)/m} \cdot x) \bmod (x^m-1) = f^{\circ}(x) \bmod (x^m-1) = f_A + f_B + f_C$.
3. Compute $f_B - f_A$ and $f_C - f_A$.
4. Let $\zeta = \omega^{\Delta'(1)/m}$
5. Compute $(\widetilde{f_Y})_d = (\Omega \cdot \zeta^d) \cdot (f_B - f_A)_d + \zeta^d \cdot (f_C - f_A)_d$ for all d in $0 \leq d < m$.
   Combine the $(\widetilde{f_Y})_d$'s to obtain $f(\omega^{\Delta'(3j+1)/m} \cdot x) \bmod (x^m - 1)$.
6. Compute $(\widetilde{f_Z})_d = \overline{\Omega \cdot \zeta^d} \cdot (f_B - f_A)_d + \overline{\zeta^d} \cdot (f_C - f_A)_d$ for all d in $0 \leq d < m$.
   Combine the $(\widetilde{f_Z})_d$'s to obtain $f(\omega^{\Delta'(3j+2)/m} \cdot x) \bmod (x^m - 1)$.
7. Compute the FFT of $f(\omega^{\Delta'(3j)/m} \cdot x) \bmod (x^m - 1)$ to obtain $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+m-1)})$.
8. Compute the FFT of $f(\omega^{\Delta'(3j+1)/m} \cdot x) \bmod (x^m - 1)$ to obtain $f(\omega^{\Delta'(j\cdot 3m+m)}), f(\omega^{\Delta'(j\cdot 3m+m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+2m-1)})$.
9. Compute the FFT of $f(\omega^{\Delta'(3j+2)/m} \cdot x) \bmod (x^m - 1)$ to obtain $f(\omega^{\Delta'(j\cdot 3m+2m)}), f(\omega^{\Delta'(j\cdot 3m+2m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$.
10. Return $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$.

Figure 2.6 Pseudocode for improved twisted radix-3 FFT

been precomputed and stored. By applying these concepts for all d in $0 \leq d < m$, the entire reduction step can be computed with fewer operations.

Regardless of which of the two twisted algorithms is used, the algorithm is initialized with $f(x)$ which equals $f(\omega^0 \cdot x) \bmod (x^n - 1)$ if f has degree less than n. By recursively applying the reduction step to f, we obtain $f(\omega^{\Delta'(j)} \cdot x) \bmod (x-1) = f(\omega^{\Delta'(j)} \cdot 1)$ for all j in the range $0 \leq j < n$, i.e. the FFT of f of size n. Pseudocode for the improved twisted radix-3 algorithm is provided in Figure 2.6.

Let us now analyze the cost of this algorithm. Line 0 is just used to end the recursion and costs no operations. Line 1 logicially partitions the input into three blocks of size m, requiring no operations. In line 2, we compute the sum $f_A + f_B + f_C$ at a cost of $2m$ additions and no multiplications. Next in line 3, we compute $f_B - f_A$ and $f_C - f_A$ at a cost of $2m$ subtractions. Line 4 is just a table lookup. We will assume that the computation of all powers of ζ can also be implemented by table lookups. Line 5 requires $2m-1$ multiplications [7] and m additions in R. By Theorem 45, line 6 only requires m additions in R. Finally, the cost of lines 7, 8, and 9 is equal to the number of operations needed to compute three FFTs of size m. Line 10 costs no operations. The total number of operations to compute the radix-3 FFT of size n using this twisted algorithm is given by

$$M(n) = 3 \cdot M\left(\frac{n}{3}\right) + \frac{2}{3} \cdot n - 1, \tag{2.104}$$

$$A(n) = 3 \cdot A\left(\frac{n}{3}\right) + 2 \cdot n, \tag{2.105}$$

where $M(1) = 0$ and $A(1) = 0$. Master Equation III can be used to solve these recurrence relations for the formulas given by

[7] We can subtract one multiplication for the case where $d = 0$ and so $\zeta^d = 1$.

$$
\begin{aligned}
M(n) &= \frac{2}{3} \cdot n \cdot \log_3(n) - \frac{1}{2} \cdot n + \frac{1}{2} && (2.106)\\
&= \frac{2}{3 \cdot \log_2(3)} \cdot n \cdot \log_2(n) - \frac{1}{2} \cdot n + \frac{1}{2} \\
&\approx 0.421 \cdot n \cdot \log_2(n) - 0.5 \cdot n + 0.5, \\
A(n) &= 2 \cdot n \cdot \log_3(n) && (2.107)\\
&= \frac{2}{\log_2(3)} \cdot n \cdot \log_2(n) \\
&\approx 1.262 \cdot n \cdot \log_2(n).
\end{aligned}
$$

Compared to the classical radix-3 algorithm, the number of multiplications has decreased while the number of additions has increased by roughly the same amount. As long as a multiplication is at least as expensive as an addition, then the algorithm presented in this section is more efficient than the classical radix-3 algorithm. Both of these algorithms, however, are less efficient than the radix-2 algorithms.

2.13 Hybrid radix-3 FFTs

It is not necessary to apply a twist after every reduction step. Suppose that we wish to evaluate f° at each of the roots of $x^{3^c m} - 1$. After c stages of reduction steps, we will have $f^\circ \bmod (x^m - \omega^{\Delta(j)})$ for each j in $0 \leq j < 3^c$. Each $\omega^{\Delta(j)}$ is a (3^c)th root of unity.

The following generalization of Theorem 10 shows the transformation necessary to apply the simplified reduction step to each of these results.

Theorem 13 *Let $f^\circ(x)$ be a polynomial in $R[x]$ with degree less than $3^c \cdot m$. Then $f^\circ(x) \bmod (x^m - \omega^{\Delta(j)}) = f^\circ(\zeta \cdot x) \bmod (x^m - 1)$ where $j < 3^c$ and $\zeta = \omega^{\Delta(j)/m}$.*

Proof: Similar to Theorem 10. □

By carefully analyzing the tradeoffs associated with performing a twist at any point in the algorithm, it can be shown that no overall savings or additional cost results from this action. The same result holds if $\Delta(j)$ is replaced by $\Delta'(j)$.

For the computation of an FFT of size 2^k, additional improved algorithms were possible by exploiting the fact that multiplications by 4th and 8th roots of unity could be implemented more efficiently than a general multiplication. In the case of the radix-3 FFT, it would be possible to develop a radix-9 algorithm, a twisted radix-9 algorithm, and a split-radix (3/9) algorithm if multiplication by the primitive 9th root of unity had some special characterisitics that could be exploited. In [61], a radix-9 algorithm is described that exploits the fact that $1 + \Omega + \Omega^2 = 0$ and that some of the roots of unity are complex conjugates of others. Additionally, a split-radix (3/9) algorithm is described in [79]. However, it can be shown that the new twisted radix-3 algorithm introduced earlier in this chapter is superior to each of these algorithms.

It is possible to combine the radix-3 algorithm with the 2-adic algorithms discussed at the beginning of this chapter. In [76], a radix-6 FFT algorithm and a radix-12 FFT algorithm are presented with a lower cost than the radix-3 algorithms already considered. Since neither of these algorithms has a lower operation count than the split-radix (2/4) algorithm [21] discussed earlier in this chapter, these algorithms will not be discussed here. However, the hybrid algorithms may be useful if someone has a need for efficiently computing an FFT of with size equal to a number of the form $2^a \cdot 3^b$.

2.14 Radix-3 FFT for symbolic roots of unity

The radix-3 FFT is an important module in an algorithm [68] which multiplies polynomials involving finite fields of characteristic 2. Such a finite field $\mathbb{F}$ does not have 2^kth roots of unity for any $k > 0$, nor can one work in an extension field of $\mathbb{F}$ to acquire the required roots of unity.

One method for multiplying two polynomials of degree less than $n = 3^k$ with coefficients in $\mathbb{F}$ is to transform these polynomials into polynomials with coefficients in D where $D = \mathbb{F}[x]/(x^{2m} + x^m + 1)$ and $m = 3^{\lceil k/2 \rceil}$. Here, D is the quotient ring of polynomials modulo $x^{2m} + x^m + 1$. Note that $x^{2m} = -x^m - 1 = x^m + 1$, and $x^{3m} = 1$. Let $t = n/m$. If $t = m$, then $\omega = x$ is a primitive $(3t)$th root of unity. Otherwise, $t = 3m$ and $\omega = x^3$ is a primitive $(3t)$th root of unity.

One version of the multiplication method involves evaluating each of the polynomials in $D[y]$ at $3t$ powers of ω. Another version involves evaluating each of the polynomials in $D[y]$ at $2t$ powers of ω. In either case, the classical or twisted version of the radix-3 FFT algorithm can be used to compute these evaluations. The improved versions of these algorithms discussed in the previous sections do not apply here because D has a special structure which allows each of the algorithms to be improved even more.

By observing that $x^{2m} = -x^m - 1$ in D, then multiplication by any power of ω in D can be implemented using just shifts and additions in $\mathbb{F}$ by applying the multiplication technique we discussed earlier from [20]. Suppose that we wish to multiply an element $\varepsilon = a_{2m-1} \cdot x^{2m-1} + a_{2m-2} \cdot x^{2m-2} + \cdots + a_1 \cdot x + a_0$. in D by ω^θ where $\theta < 3m/2$. Multiplication by x^θ simply shifts each of the coefficients in ε by θ positions to the left. For any resulting coefficient of degree d greater than $2m$, modular reduction must be performed. In this case, x^d can be replaced by $x^{d-m} + x^{d-2m}$. The

entire multiplication can be implemented by circularly shifting ε by θ positions and then performing θ additions in $\mathbb{F}$.

For example, suppose that $2m = 6$, and $\theta = 2$. Then if $\varepsilon = a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$, then $\omega^2 \cdot \varepsilon = x^2 \cdot \varepsilon = a_3 \cdot x^5 + (a^5 + a_2) \cdot x^4 + (a_4 + a_1) \cdot x^3 + a_0 \cdot x^2 + a_5 \cdot x + a_4$.

If $\theta > 3m/2$, then it is more advantageous to shift to the right. Consider ε to be a power series instead of a polynomial. Since $x^{3m} = 1$, then $x^\theta = x^{\theta - 3m}$. So divide ε by $x^{3m-\theta}$. This will produce a power series with negative exponents. For each coefficient of degree $d < 0$, then $x^d = x^{2m+d} - x^{m+d}$ can be used to eliminate the coefficients of negative degree. Note that the resulting power series is also a polynomial in $\mathbb{F}[x]$. So multiplication by ω^θ is implemented by circularly shifting ε by θ positions to the right and then performing $3m - \theta$ additions in $\mathbb{F}$.

For example, suppose that $2m = 6$ and $\theta = 7$. Then if $\varepsilon = a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$, then $\omega^7 \cdot \varepsilon = x^7 \cdot \varepsilon = x^{-2} \cdot \varepsilon = a_1 \cdot x^5 + a^0 \cdot x^4 + a_5 \cdot x^3 + (a_4 + a_1) \cdot x^2 + (a_3 + a_0) \cdot x + a_2$.

An FFT of size t in $D[y]$ can be computed in $2 \cdot t \cdot \log_3(t) - t + 1$ multiplications and $2 \cdot t \cdot \log_3(t)$ additions in D by using the operation count of the radix-3 algorithms without the improvements by [20] and [76]. Assuming that every power of ω is used roughly the same number of times in the FFT algorithm, then it can be shown that a multiplication in D by one of the powers of ω requires $3/4 \cdot m$ additions in $\mathbb{F}$ (plus copies which we will not count). Each addition in D requires $2m$ additions in $\mathbb{F}$. If $n = t \cdot m$, then an FFT of size t in $D[y]$ requires no multiplications and

$$\frac{11}{2} \cdot n \cdot \log_3(t) - \frac{3}{4} \cdot n + \frac{3}{4} \cdot m \tag{2.108}$$

additions in $\mathbb{F}$. [8] The multiplication algorithm which uses this FFT will be discussed in Chapter 5.

2.15 Concluding remarks

This chapter considered several different algorithms which compute the FFT of size a power of two and additional algorithms which compute the FFT of size a power of three. These algorithms fall into two major categories. The classical (Cooley-Tukey) algorithms modularly reduce the input by a different polynomial in each reduction step. The twisted (Gentleman-Saude) algorithms modularly reduce the input by the same polynomial in each reduction step and then "twist" one or more of the resulting outputs.

Ignoring any restrictions on algorithm selection on the basis of input size, the split-radix algorithms are the most efficient option for computing an FFT. The 2-adic algorithms (radix-4, radix-8, etc.) are the next most efficient category of algorithms. Finally, the 3-adic algorithms are the least efficient algorithms considered in this chapter. Even less efficient algorithms result by constructing radix-p^c algorithms for other values of $p > 3$.

The chapter also presented a new split radix algorithm recently introduced by Van Buskirk, Johnson and Frigo, and Bernstein. This algorithm reduces the theoretical asympotic number of multiplications in $\mathbb{R}$ by about six percent compared to the split-radix algorithm which previously required the fewest number of operations to compute an FFT of size 2^k, but requires R to be a field in order for the algorithm to work.

---

[8] The same result can be obtained by using the operation count of the algorithm introduced in [20]. The improved algorithm in [76] does not apply here because elements of D are not in the required form.

CHAPTER 3

ADDITIVE FAST FOURIER TRANSFORM ALGORITHMS

Finite fields possess an additive structure as well as a multiplicative structure. In [81], Yao Wang and Xuelong Zhu introduced an algorithm which exploits this additive structure to efficiently evaluate a polynomial at all of the points in a subspace of a finite field. Similar to the history of the multiplicative Fast Fourier Transform, this publication received little attention, was reinvented several years later by David G. Cantor [12], and is now usually referred to as "Cantor's Algorithm" in subsequent publications (e.g. [34]). The algorithm requires the finite field to be of the form $GF(p^K)$ where K is itself a power of p. If $N = p^K$ and n divides N, then the cost of the algorithm to compute an "additive FFT" of length n is given by $\Theta(n \log_p n)$ multiplications and $\Theta\left(n \cdot \left(\log_p n\right)^{1+\log_p((p+1)/2)}\right)$ additions. The algorithm was later generalized by von zur Gathen and Gerhard [32] to compute the additive FFT over any finite field with a greater operation cost. The first part of this chapter will review these algorithms in the case of finite fields with characteristic 2. The interested reader can study [12] to see how the techniques work for other finite fields.

3.1 Von zur Gathen-Gerhard additive FFT

A vector space V of dimension K over some field $\mathbb{F}$ is an algebraic structure such that every element of V can be represented as a linear combination of K basis elements $\{\beta_1, \beta_2, \ldots, \beta_K\} \in V$. That is to say, if ϖ is an element of V, then ϖ can be represented as

$$\varpi = a_K \cdot \beta_K + a_{K-1} \cdot \beta_{K-1} + \cdots + a_2 \cdot \beta_2 + a_1 \cdot \beta_1, \tag{3.1}$$

where each a_d is an element of $\mathbb{F}$.

Let $\mathbb{F}$ be a finite field of size $N = 2^K$. This finite field can be viewed as a vector space over $GF(2)$ of dimension K. If $\{\beta_1, \beta_2, \ldots, \beta_K\}$ is a basis for $\mathbb{F}$, then we may enumerate the N elements $\{\varpi_0, \varpi_1, \ldots, \varpi_{N-1}\}$ of $\mathbb{F}$ as follows. For $0 \leq j < N$, write j in binary form, i.e. $j = (b_{K-1}b_{K-2}\cdots b_1 b_0)_2$. Then ϖ_j is given by setting $a_d = b_{d-1}$ in (3.1) above for each d in $1 \leq d \leq K$.

Let us define subspace W_i of $\mathbb{F}$ to be all linear combinations of $\{\beta_1, \beta_2, \ldots, \beta_i\}$, i.e. all ϖ_j for $0 \leq j < 2^i$ where $i \leq K$. Observe that the subspaces $W_0 = \{0\}, W_1, W_2, \ldots, W_K$ form a strictly ascending chain as follows:

$$\{0\} = W_0 \subset W_1 \subset W_2 \subset \cdots \subset W_K = \mathbb{F}. \tag{3.2}$$

In this section, we will give an algorithm that can compute the FFT over the points of any $W_k \subseteq \mathbb{F}$ where $k \leq K$. The elements of W_k are given by

$$\varpi_j = b_{k-1} \cdot \beta_k + b_{k-2} \cdot \beta_{k-2} + \cdots + b_1 \cdot \beta_2 + b_0 \cdot \beta_1 \tag{3.3}$$

for $0 \leq j < 2^k$.

For any $0 \leq i < k$, we can decompose the elements of W_{i+1} into two pairwise disjoint cosets. One coset contains the elements of W_i and the other coset contains the elements $\beta_{i+1} + W_i$. In other words, the elements in this second coset are obtained by adding β_{i+1} to each element of W_i. Furthermore, if $i < k-1$ and ε is any linear combination of $\{\beta_{i+2}, \beta_{i+3}, \cdots, \beta_k\}$, then we can decompose the elements of $\varepsilon + W_{i+1}$ into the two pairwise disjoint cosets $\varepsilon + W_i$ and $\beta_{i+1} + \varepsilon + W_i$. Thus,

$$\varepsilon + W_{i+1} \;=\; (\varepsilon + W_i) \bigcup (\beta_{i+1} + \varepsilon + W_i). \tag{3.4}$$

Let us now define

$$s_i(x) \;=\; \prod_{a \in W_i} (x - a) \tag{3.5}$$

as the minimal polynomial of W_i. Observe that the minimal polynomial of $\varepsilon + W_i$ is given by

$$\prod_{a \in \varepsilon + W_i} (x - a) \;=\; \prod_{a \in W_i} ((x - \varepsilon) - a) = s_i(x - \varepsilon). \tag{3.6}$$

The minimal polynomial of $\beta_{i+1} + \varepsilon + W_i$ is similarly given by $s_i(x - \varepsilon - \beta_{i+1})$.

Because W_{i+1} can be partitioned into W_i and $\beta_i + W_i$, then (3.5) factors as follows:

$$\begin{aligned} s_{i+1}(x) \;&=\; \prod_{a \in W_{i+1}} (x - a) = \prod_{a \in W_i} (x - a) \cdot \prod_{a \in \beta_{i+1} + W_i} (x - a) \\ &=\; s_i(x) \cdot s_i(x - \beta_{i+1}). \end{aligned} \tag{3.7}$$

Similarly, because $\varepsilon + W_{i+1}$ can be partitioned into $\varepsilon + W_i$ and $\beta_{i+1} + \varepsilon + W_i$, then (3.6) factors as

$$s_{i+1}(x-\varepsilon) \;=\; s_i(x-\varepsilon)\cdot s_i(x-\varepsilon-\beta_{i+1}). \tag{3.8}$$

At this point, we recall the "Freshman's Dream Theorem" for finite fields. This result claims that for any a and b which are elements of a finite field of characteristic p and any q which is a power of p:

$$(a+b)^q = a^q + b^q. \tag{3.9}$$

The Freshman's Dream Theorem can be used to prove the following results which will be used in developing the reduction step of the additive FFT algorithms.

Lemma 14 *Let $L(x)$ be a linearized polynomial over $GF(q)$, i.e. $L(x)$ has the form*

$$L(x) \;=\; \sum_{d=0}^{i} a_d \cdot x^{q^d} \tag{3.10}$$

with coefficients in an extension field $GF(q^m)$. Then $L(x)$ is a linear map of $GF(q^m)$ over $GF(q)$, i.e.

$$L(a+b) \;=\; L(a)+L(b), \tag{3.11}$$

$$L(c\cdot a) \;=\; c\cdot L(a) \tag{3.12}$$

for all $a, b \in GF(q^m)$ and $c \in GF(q)$.

Proof: A proof of this theorem can be found after Definition 3.49 in [52]. □

Theorem 15 *Let $s_i(x)$ be the minimal polynomial of W_i. Then $s_i(x)$ is a linearized polynomial over $GF(2)$ for all i. Furthermore, $s_{i+1}(x) = (s_i(x))^2 - s_i(x) \cdot s_i(\beta_{i+1})$.*

Proof: We will prove this theorem by induction. First, observe that $s_0(x) = x$. Clearly, this is a linearized polynomial. Next, observe that $s_1(x) = x^2 - x \cdot \beta_1$ where β_1 be the nonzero element contained in W_1. Since $s_1(x) = x^2 - x \cdot \beta_1 = (s_0(x))^2 - s_0(x) - s_0(\beta_1)$, then the result holds for $i = 0$.

Assume that the result holds for some $i = \kappa \geq 0$. We need to show that the result holds for $i = \kappa + 1$. Consider $s_{\kappa+1}(x)$ which equals $s_\kappa(x) \cdot s_\kappa(x - \beta_{\kappa+1})$ by (3.7). By the induction hypothesis, s_κ is a linearized polynomial. So by Lemma 14 with $q = 2$, $s_\kappa(x - \beta_{\kappa+1}) = s_\kappa(x) - s_\kappa(\beta_{\kappa+1})$. Then, $s_{\kappa+1}(x) = s_\kappa(x) \cdot (s_\kappa(x) - s_\kappa(\beta_{\kappa+1})) = (s_\kappa(x))^2 - s_\kappa(x) \cdot s_\kappa(\beta_{\kappa+1})$. Also by the induction hypothesis, let us represent

$$s_\kappa(x) \;=\; \sum_{d=0}^{\kappa} \mathcal{A}_d \cdot x^{2^d}. \tag{3.13}$$

By the Freshman's Dream Theorem, it follows that

$$\begin{aligned} s_{\kappa+1}(x) \;=\; & \mathcal{A}_0 \cdot s_\kappa(\beta_{\kappa+1}) \cdot x + \sum_{d=1}^{\kappa} \left({\mathcal{A}_{d-1}}^2 + \mathcal{A}_d \cdot s_\kappa(\beta_{\kappa+1}) \right) \cdot x^{2^d} \\ & + {\mathcal{A}_\kappa}^2 \cdot x^{2^{\kappa+1}}. \end{aligned} \tag{3.14}$$

By setting $a_0 = \mathcal{A}_0 \cdot s_\kappa(\beta_{\kappa+1})$, $a_d = {\mathcal{A}_{d-1}}^2 + \mathcal{A}_d \cdot s_\kappa(\beta_{\kappa+1})$ for all d in $1 \leq d \leq \kappa$ and $a_{\kappa+1} = {\mathcal{A}_\kappa}^2$, then we have demonstrated that $s_{\kappa+1}$ has the form given in (3.10). Thus, $s_{\kappa+1}$ is a linearized polynomial.

By the principle of mathematical induction, the theorem has been proven. □

Corollary 16 *Let $s_{i+1}(x-\varepsilon)$ be the minimal polynomial of $\varepsilon + W_{i+1}$, let $s_i(x-\varepsilon)$ be the minimal polynomial of $\varepsilon + W_i$, and let $s_i(x-\varepsilon-\beta_{i+1})$ be the minimal polynomial of $\beta_{i+1}+\varepsilon+W_i$. Then*

$$\begin{aligned} s_{i+1}(x-\varepsilon) &= s_{i+1}(x) - s_{i+1}(\varepsilon), && (3.15)\\ s_i(x-\varepsilon) &= s_i(x) - s_i(\varepsilon), && (3.16)\\ s_i(x-\beta_{i+1}-\varepsilon) &= s_i(x) - s_i(\beta_{i+1}) - s_i(\varepsilon). && (3.17) \end{aligned}$$

Proof: Each of these results follows from the fact that s_{i+1} and s_i are linearized polynomials and Lemma 14. □

In [32], an algorithm is given which computes the additive FFT over W_k using an arbitrary basis. It is based on a reduction step which uses the transformation

$$\begin{aligned} R[x]/s_{i+1}(x-\varepsilon) &\rightarrow R[x]/s_i(x-\varepsilon) && (3.18)\\ &\times R[x]/s_i(x-\varepsilon-\beta_{i+1}). \end{aligned}$$

Suppose that we are given $f^\circ(x) = f(x) \bmod s_{i+1}(x-\varepsilon)$ where the degree of f° is less than the degree of $s_{i+1}(x-\varepsilon)$. Divide f° by $s_i(x-\varepsilon) = s_i(x) - s_i(\varepsilon)$ to produce quotient $q(x)$ and remainder $r(x)$, i.e.

$$\begin{aligned} f^{\circ} &= q \cdot (s_i - s_i(\varepsilon)) + r \\ &= q \cdot (s_i - s_i(\varepsilon) - s_i(\beta_{i+1})) + r + q \cdot s_i(\beta_{i+1}). \end{aligned} \tag{3.19}$$

So,

$$f \bmod s_i(x - \varepsilon) = r, \tag{3.20}$$

$$f \bmod s_i(x - \varepsilon - \beta_{i+1}) = r + q \cdot s_i(\beta_{i+1}). \tag{3.21}$$

Recall that ε is a linear combination of $\{\beta_{i+2}, \beta_{i+3}, \cdots, \beta_k\}$ whenever $i < k-1$. Then

$$\varepsilon = b_{k-1} \cdot \beta_k + b_{k-2} \cdot \beta_{k-1} + \cdots + b_{i+2} \cdot \beta_{i+3} + b_{i+1} \cdot \beta_{i+2}, \tag{3.22}$$

and $\varepsilon = \varpi_\theta$ where $\theta = (b_{i-1} b_{i-2} \cdots b_{i+1} 000 \cdots 0)_2$. So θ is a multiple of 2^{i+1} and $\varepsilon = \varpi_{j \cdot 2m}$ where $0 \leq j < 2^{k-i-1}$ and $m = 2^i$.

The reduction step for $\varepsilon = \varpi_\theta$ can be expressed using the transformation

$$\begin{aligned} R[x]/s_{i+1}(x - \varpi_\theta) &\rightarrow R[x]/s_i(x - \varpi_\theta) \\ &\times R[x]/s_i(x - \varpi_{\theta+m}). \end{aligned} \tag{3.23}$$

To implement the reduction step, divide the input polynomial $f^\circ(x) = f(x) \bmod s_{i+1}(x - \varepsilon)$ by $s_i(x - \varpi_\theta) = s_i(x) - s_i(\varpi_\theta)$ to produce quotient $q(x)$ and remainder $r(x)$. Then

$$f \bmod s_i(x - \varpi_\theta) = r, \tag{3.24}$$

$$f \bmod s_i(x - \varpi_{\theta+m}) = r + q \cdot s_i(\beta_{i+1}). \tag{3.25}$$

Suppose that we wish to evaluate a polynomial f of degree less than $n = 2^k$ at each of the points in W_k. Then $f = f \bmod s_k$ where $s_k(x)$ is the minimal polynomial of W_k. Von zur Gathen and Gerhard's algorithm recursively applies a reduction step which receives as input the polynomial $f^\circ = f \bmod (s_{i+1} - s_{i+1}(\varpi_{j \cdot 2m}))$ for some i and $0 \leq j < 2^{k-i-1}$. Here, $m = 2^i$. The reduction step divides this polynomial by $s_i - s_i(\varpi_{j \cdot 2m})$ to obtain output $f \bmod (s_i - s_i(\varpi_{j \cdot 2m}))$ and $f \bmod (s_i - s_i(\varpi_{j \cdot 2m+m}))$. The algorithm is initialized with values $i = k - 1$ and $j = 0$. After all of the reduction steps have been completed, then we will have $f \bmod (s_0 - s_0(\varpi_{j \cdot 1})) = f \bmod (x - \varpi_j) = f(\varpi_j)$, i.e. the desired additive FFT of f over W_k.

In Figure 3.1, pseudocode is provided for von zur Gathen and Gerhard's algorithm. Prior to the start of the algorithm, we must select a basis for $W_k \subseteq \mathbb{F}$, compute the s_i's using Theorem 15, compute the ϖ's using (3.3), and compute $s_i(\varpi_{j \cdot 2m})$ for each $0 \leq j < 2^{k-i-1}$ where $0 \leq i < k$. We will assume that it is possible to store each of these values.

Let us compute the cost of this algorithm. Line 0 is used to end the recursion and does not cost any operations. In line 1 we need to divide a polynomial of degree

Algorithm : Von zur Gathen-Gerhard additive FFT
Input: $f^\circ = f \bmod (s_{i+1} - s_{i+1}(\varpi_{j\cdot 2m}))$, a polynomial of degree less than $2m$ where $m = 2^i$ over $\mathbb{F}$, a field of characteristic 2 of dimension K.
Output: The multipoint evaluation of f over $\varpi_{j\cdot 2m} + W_{i+1} \subseteq \mathbb{F}$, i.e. $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$.
0. If $(2m) = 1$, then return $f \bmod (x - \varpi_j) = f(\varpi_j)$. 1. Divide f° by $(s_i - s_i(\varpi_{j\cdot 2m}))$. This results in two polynomials, q and r, each of size m such that $f^\circ = q \cdot (s_i - s_i(\varpi_{j\cdot 2m})) + r$. 2. Compute $f \bmod (s_i - s_i(\varpi_{j\cdot 2m})) = r$ and $f \bmod (s_i - s_i(\varpi_{j\cdot 2m+m})) = r + q \cdot s_i(\beta_{i+1})$. 3. Compute the additive FFT of $f \bmod (s_i - s_i(\varpi_{j\cdot 2m}))$ to obtain $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+m-1})$. 4. Compute the additive FFT of $f \bmod (s_i - s_i(\varpi_{j\cdot 2m+m}))$ to obtain $f(\varpi_{j\cdot 2m+m}), f(\varpi_{j\cdot 2m+m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$. 5. Return $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$.

Figure 3.1 Pseudocode for von zur Gathen-Gerhard additive FFT

less than $2m$ by $s_i - s_i(\varpi_{j\cdot 2m})$, a polynomial of degree m with $\log_2(m)+2$ coefficients. This costs $m \cdot (\log_2(m) + 2)$ multiplications, $m \cdot (\log_2(m) + 2)$ additions, plus at most 1 inversion in $\mathbb{F}$. If $j = 0$, then $\varpi_0 = 0$ and only $m \cdot (\log_2(m) + 1)$ operations of each type are required. In line 2, we need to perform m multiplications and m additions to obtain $f \bmod (s_i - s_i(\varpi_{j\cdot 2m+m}))$. The cost of lines 3 and 4 is equal to the number of operations needed to compute two additive FFTs of size m. Line 5 costs no operations.

If $j \neq 0$, then the total number of operations to compute the additive FFT of size n using the von zur Gathen-Gerhard algorithm is given by

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n \cdot \log_2(n) + n, \tag{3.26}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n \cdot \log_2(n) + n, \tag{3.27}$$

where $M(1) = 0$ and $A(1) = 0$. Master Equation I can be used to solve these recurrence relations. We must also subtract the number of operations saved when $j = 0$. A recurrence relation for the number of multiplications saved is

$$M_s(n) = M_s\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n, \tag{3.28}$$

where $M_s(1) = 0$. Master Equation II can be used to solve this recurrence relation to obtain $n - 1$, which is also the number of additions saved.

Combining the closed-form formulas, the number of operations required for this additive FFT algorithm is given by

$$M(n) = \frac{1}{4} \cdot n \cdot (\log_2(n))^2 + \frac{5}{4} \cdot n \cdot \log_2(n) - n + 1, \tag{3.29}$$

$$A(n) = \frac{1}{4} \cdot n \cdot (\log_2(n))^2 + \frac{5}{4} \cdot n \cdot \log_2(n) - n + 1. \tag{3.30}$$

3.2 Wang-Zhu-Cantor additive FFT

It was mentioned at the beginning of this chapter that von zur Gathen and Gerhard's algorithm was a generalization of an algorithm introduced independently by Wang and Zhu [81] and Cantor [12]. This earlier algorithm only works when the underlying finite field is of size p^K where K is itself a power of p. Here, we will show how to adapt the algorithm discussed in the previous section into the Wang-Zhu-Cantor algorithm for the case where $p = 2$.

In [32], it is claimed that the following set of elements can be constructed and that they form a basis for $GF(2^K)$, where K is a power of two.

$$\begin{aligned} \beta_1 &= 1, \\ \beta_i &= {\beta_{i+1}}^2 + \beta_{i+1} \quad \text{for } 1 \leq i < k. \end{aligned} \tag{3.31}$$

Although [32] claims that this was the basis used by Cantor in [12], this may not be obvious from reading Cantor's paper. In a section of the appendix, facts from Cantor's paper and standard results from finite field theory are used to show that the set of elements given in (3.31) can be constructed in $\mathbb{F}$ and that they form a basis for $\mathbb{F}$. In this proof, the functions

$$\varphi(x) = x^2 + x, \tag{3.32}$$

$$\varphi^k(x) = \varphi(\varphi^{k-1}(x)) \tag{3.33}$$

are introduced for $k > 1$ where $\varphi^1(x) = \varphi(x)$. By the associative property of the composition of functions, then $\varphi^k(x) = \varphi^{k-1}(\varphi(x))$. In the appendix, a nonrecursive formula for $\varphi^k(x)$ is derived and given by

$$\varphi^k(x) = \sum_{d=0}^{k} \binom{k}{d} x^{2^d}, \tag{3.34}$$

where $\binom{k}{d}$ is the modulo 2 reduction of the binomial coefficent $C(k, d)$. Here, we will also define $\varphi^0(x) = x$ which is consistent with the above formulas for $k > 0$.

The following result provides the key for the faster algorithm using this selection of basis elements.

Theorem 17 *For the special basis given by (3.31),* $s_i(x) = \varphi^i(x)$ *for all* $0 \leq i \leq k$. *Furthermore,* $s_i(\beta_{i+1}) = 1$ *for all* $0 \leq i < k$.

Proof: We will prove the theorem inductively. Since $s_0(x) = x = \varphi^0(x)$ and $s_0(\beta_1) = \beta_1 = 1$, then the case $i = 0$ has been satisfied.

Suppose that the theorem holds for $i = \kappa$ where $0 \leq \kappa < k - 1$. We need to show that the theorem holds for $i = \kappa + 1$. By Theorem 15, $s_{\kappa+1}(x) = (s_\kappa(x))^2 - s_\kappa(x) \cdot s_\kappa(\beta_{\kappa+1})$. Now by the inductive hypothesis,

$$\begin{aligned} s_{\kappa+1}(x) &= (\varphi^\kappa(x))^2 - \varphi^\kappa(x) \cdot 1 = (\varphi^\kappa(x))^2 + \varphi^\kappa(x) = \varphi(\varphi^\kappa(x)) \\ &= \varphi^{\kappa+1}(x), \end{aligned} \tag{3.35}$$

where addition and subtraction are equivalent in a finite field of characteristic 2. This establishes the first part of the theorem for $i = \kappa + 1$.

If $\kappa + 1 < k$, then by the inductive hypothesis,

$$\begin{aligned} s_{\kappa+1}(\beta_{\kappa+2}) &= \varphi^{\kappa+1}(\beta_{\kappa+2}) = \varphi^\kappa(\varphi(\beta_{\kappa+2})) = \varphi^\kappa({\beta_{\kappa+2}}^2 + \beta_{\kappa+2}) \\ &= s_\kappa(\beta_{\kappa+1}) = 1, \end{aligned} \tag{3.36}$$

proving the second part of the theorem for $i = \kappa + 1$.

By inductively repeating this argument, the theorem is proven. □

If $i < k-1$, then choose any ε that is a linear combination of $\{\beta_{i+2}, \beta_{i+3}, \cdots, \beta_k\}$ and let $\xi = s_i(\varepsilon)$. If $i = k - 1$, then let $\varepsilon = 0$ and $\xi = 0$. Now, $s_{i+1}(\varepsilon) = \varphi^{i+1}(\varepsilon) =$

$\varphi(\varphi^i(\varepsilon)) = \varphi(s_i(\varepsilon)) = \varphi(\xi) = \xi^2 + \xi$. Since $s_i(\beta_{i+1}) = 1$, substituting these results into (3.8) and applying Corollary 16 gives the simplified factorization

$$s_{i+1}(x) - (\xi^2 + \xi) \;=\; (s_i(x) - \xi) \cdot (s_i(x) - \xi - 1) \tag{3.37}$$

for the special basis.

The Wang-Zhu-Cantor additive FFT algorithm is based on the transformation

$$\begin{aligned} R[x]/(s_{i+1} - (\xi^2 + \xi)) \;&\rightarrow\; R[x]/(s_i - \xi) \\ &\times\; R[x]/(s_i - (\xi + 1)). \end{aligned} \tag{3.38}$$

Suppose that we are given $f^\circ(x) = f(x) \bmod (s_{i+1} - (\xi^2 + \xi))$. Divide f° by $s_i - \xi$ to produce quotient $q(x)$ and remainder $r(x)$, i.e.

$$\begin{aligned} f \;&=\; q \cdot (s_i - \xi) + r, \\ &=\; q \cdot (s_i - (\xi + 1)) + r + q. \end{aligned} \tag{3.39}$$

So,

$$\begin{aligned} f \bmod (s_i - \xi) \;&=\; r, \\ f \bmod (s_i - (\xi + 1)) \;&=\; r + q. \end{aligned} \tag{3.40}$$

Note that with the special basis, we have eliminated a multiplication by $s_i(\beta_{i+1})$ that was needed in (3.21).

The special basis allows for further improvements of the von zur Gathen-Gerhard algorithm. By (3.34), all of the nonzero coefficients of $s_i(x)$ are 1 compared to any element of $\mathbb{F}$ in the more general algorithm. This significantly reduces the number of multiplications in $\mathbb{F}$ for the Wang-Zhu-Cantor algorithm.

Also, it is proven in the appendix that the number of nonzero coefficients in $s_i(x) = \varphi^i(x)$ is given by 2^d where d is the number of ones in the binary expansion of i if the special basis is used. Clearly, this is less than the $i+1$ nonzero coefficients used in the polynomials of the von zur Gathen-Gerhard algorithm and will result in fewer operations required to perform the modular reductions in the algorithm.

Finally, the special basis allows one to constuct the elements of W_k without the use of the polynomial s_k. Recall that

$$\varpi_j \;=\; b_{k-1} \cdot \beta_k + b_{k-2} \cdot \beta_{k-1} + \cdots + b_1 \cdot \beta_2 + b_0 \cdot \beta_1 \tag{3.41}$$

for any j in $0 \leq j < 2^k$ where $j = (b_{k-1}b_{k-2} \cdots b_2 b_1 b_0)_2$. If the special basis is used, then the ϖ_j's have the following properties which can be proven in a manner similar to Theorems 1 and 2.

Lemma 18 *For any $j < n/2$, $\varpi_{2j+1} = \varpi_{2j} + 1$.*

Lemma 19 *For any $j < n/2$, $\varpi_j = (\varpi_{2j})^2 + \varpi_{2j}$.*

Substituting ϖ_{2j} for ξ in (3.37), then

$$s_{i+1} - \varpi_j \;=\; (s_i - \varpi_{2j}) \cdot (s_i - \varpi_{2j+1}) \tag{3.42}$$

and the reduction step for the Wang-Zhu-Cantor algorithm can be expressed as

$$\begin{aligned} R[x]/(s_{i+1} - \varpi_j) \;&\rightarrow\; R[x]/(s_i - \varpi_{2j}) \\ &\times\; R[x]/(s_i - \varpi_{2j+1}). \end{aligned} \tag{3.43}$$

Observe that $\xi = \varpi_{2j} \in W_{k-i}$ and $\xi^2 + \xi = \varpi_j \in W_{k-i-1}$.

Suppose that we wish to evaluate a polynomial f of degree less than $n = 2^k$ at each of the points in W_k. Then $f = f \bmod s_k$ where $s_k(x)$ is the minimal polynomial of W_k. The Wang-Zhu-Cantor algorithm recursively applies a reduction step which receives the polynomial $f \bmod (s_{i+1} - \varpi_j)$ as input for some i and j. The first reduction step uses values $i = k - 1$ and $j = 0$. Each reduction step divides the input polynomial by $s_i - \varpi_{2j}$ to obtain output $f \bmod (s_i - \varpi_{2j})$ and $f \bmod (s_i - \varpi_{2j+1})$. After all of the reduction steps have been completed, then we will have $f \bmod (s_0 - \varpi_j) = f \bmod (x - \varpi_j) = f(\varpi_j)$ for all $0 \leq j < n$, i.e. the desired additive FFT of f over W_k.

In Figure 3.2, pseudocode is provided for the Wang-Zhu-Cantor algorithm. Prior to the start of the algorithm, we must compute the β_i's using (3.31), the s_i's using (3.34), and the ϖ's using (3.41). We will assume that this represents one time work that can be precomputed and stored.

Let us compute the cost of this algorithm. Line 0 is used to end the recursion and does not cost any operations. In line 1 we need to divide a polynomial of degree

Algorithm : Wang-Zhu-Cantor additive FFT
Input: $f^\circ = f \bmod (s_{i+1} - \varpi_j)$, a polynomial of degree less than $2m$ where $m = 2^i$ over $\mathbb{F}$, a field of characteristic 2 of dimension K and K is a power of two.
Output: The multipoint evaluation of f over $\varpi_{j \cdot 2m} + W_{i+1} \subseteq \mathbb{F}$, i.e. $f(\varpi_{j \cdot 2m+0}), f(\varpi_{j \cdot 2m+1}), \ldots, f(\varpi_{j \cdot 2m+2m-1})$.
0. If $(2m) = 1$, then return $f \bmod (x - \varpi_j) = f(\varpi_j)$. 1. Divide f° by $(s_i - \varpi_{2j})$. This results in two polynomials, q and r, each of size m such that $f^\circ = q \cdot (s_i - \varpi_{2j}) + r$. 2. Compute $f \bmod (s_i - \varpi_{2j}) = r$ and $f \bmod (s_i - \varpi_{2j+1}) = q + r$. 3. Compute the additive FFT of $f \bmod (s_i - \varpi_{2j})$ to obtain $f(\varpi_{j \cdot 2m+0}), f(\varpi_{j \cdot 2m+1}), \ldots, f(\varpi_{j \cdot 2m+m-1})$. 4. Compute the additive FFT of $f \bmod (s_i - \varpi_{2j+1})$ to obtain $f(\varpi_{j \cdot 2m+m}), f(\varpi_{j \cdot 2m+m+1}), \ldots, f(\varpi_{j \cdot 2m+2m-1})$. 5. Return $f(\varpi_{j \cdot 2m+0}), f(\varpi_{j \cdot 2m+1}), \ldots, f(\varpi_{j \cdot 2m+2m-1})$.

Figure 3.2 Pseudocode for Wang-Zhu-Cantor additive FFT

less than $2m$ by $s_i - \varpi_{2j}$, a polynomial with $c_i + 1$ coefficients where c_i is the number of non-zero coefficients in s_i. The division is done "in-place" using the memory originally occupied by $f \bmod (s_{i+1} - \varpi_j)$. Since every nonzero element of s_i is 1, then line 1 costs m multiplications and $(c_i + 1) \cdot m$ additions in $\mathbb{F}$. However, if $j = 0$, then no multiplications and only $(c_i) \cdot m$ additions are required. In line 2, we need to perform m additions to obtain $f \bmod (s_i - \varpi_{2j+1})$. The cost of lines 3 and 4 is equal to the number of operations needed to compute two additive FFTs of size m. Line 5 costs no operations.

If $j \neq 0$, then the total number of operations to compute the additive FFT of size n using the Wang-Zhu-Cantor algorithm is given by

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n, \tag{3.44}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n \cdot c_{\log_2(n)-1} + n, \tag{3.45}$$

where $M(1) = 0$ and $A(1) = 0$.

We must also subtract $n - 1$ additions and $n - 1$ multiplications when $j = 0$, determined using the same recurrence relation used for the multiplicative FFT algorithms. Master Equation I can be used to solve the above recurrence relations. The resulting operation counts are given by

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{3.46}$$

$$\begin{aligned} A(n) &= \frac{1}{2} \cdot C_{\log_2(n)} \cdot n + n \cdot \log_2(n) - n + 1 \\ &\leq \frac{1}{2} \cdot n \cdot (\log_2(n))^{1.585} + n \cdot \log_2(n) - n + 1, \end{aligned} \tag{3.47}$$

where $C_{\log_2(n)} = c_0 + c_1 + \cdots + c_{\log_2(n)-1}$. The second formula for the addition count holds with equality when $\log_2(n)$ is a power of two and is an upper bound otherwise. So the Wang-Zhu-Cantor algorithm requires $\Theta(n \cdot (\log_2(n))^{1.585})$ operations.

3.3 Shifted additive FFT

In Chapter 2, we studied various "twisted" FFT algorithms which simplifed the FFT reduction step at the cost of computing a number of twisted polynomials. One may be wondering if the same thing is possible with the Wang-Zhu-Cantor algorithm. Since $\varpi_0 = 0$, this is an element that we would like to use as often as we can. A transformation for this case is given by

$$\begin{aligned} R[x]/(s_{i+1}) &\rightarrow R[x]/(s_i) \\ &\times R[x]/(s_i+1). \end{aligned} \tag{3.48}$$

Observe that the first output is already in the form needed to apply this transformation again as one progresses through the additive FFT computation. However, an adjustment is needed to put the second output in the proper form.

The mechanism that will be used to achieve the desired transformation is the Taylor shift of a polynomial at an element. In other words, if $f(x)$ is a polynomial in $R[x]$, then the Taylor shift of f at an element ξ in R is a polynomial $g(y)$ such that $g(y) = f(x-\xi)$ where $y = x-\xi$. An algorithm for computing the Taylor shift of f at ξ is discussed in a section of the appendix. Because R is a finite field of characteristic 2, then this Taylor shift is also equivalent to $f(x+\xi)$. The following theorem provides the value of ξ necessary to transform the second output of the simplified reduction step into the proper form for input to the next reduction step.

Theorem 20 *Let $f^\circ(x)$ be a polynomial of degree less than $2m$ in $R[x]$ where $m = 2^i$. Then $f^\circ(x) \bmod (s_i+1) = f^\circ(x+\beta_{i+1}) \bmod (s_i)$.*

Proof: Let $f^\circ(x)$ be a polynomial of degree less than $2m$ in $R[x]$ and let $r(x) = f^\circ(x) \bmod (s_i+1)$. Then $f^\circ(x) = q(x)\cdot(s_i+1) + r(x)$ for some $q(x)$. Let us shift both sides of this equation by β_{i+1}. Applying the linear properties of s_i and Theorem 17 gives:

$$
\begin{aligned}
f^{\circ}(x+\beta_{i+1}) &= q(x+\beta_{i+1})\cdot(s_i(x+\beta_{i+1})+1)+r(x+\beta_{i+1}) \qquad (3.49)\\
&= q(x+\beta_{i+1})\cdot(s_i(x)+s_i(\beta_{i+1})+1)+r(x+\beta_{i+1})\\
&= q(x+\beta_{i+1})\cdot(s_i(x)+1+1)+r(x+\beta_{i+1})\\
&= q(x+\beta_{i+1})\cdot(s_i(x))+r(x+\beta_{i+1}).
\end{aligned}
$$

Thus $r(x+\beta_{i+1}) = f^{\circ}(x+\beta_{i+1}) \bmod (s_i)$. □

We can now present the reduction step of the shifted additive FFT algorithm which receives an input of some polynomial f° of degree less than $2m$ where $m = 2^i$. Divide f° by s_i to obtain outputs $f_Y = f^{\circ} \bmod (s_i)$ and $f_Z = f^{\circ} \bmod (s_i + 1)$. The simplified reduction step can be directly applied to f_Y while we need to shift f_Z by β_i prior to using this result as input to the simplified reduction step.

It is not clear yet that an algorithm based on this reduction step will yield the additive FFT of some polynomial $f(x)$ of degree less than $n = 2^k$. The following theorem is intended to help provide this clarification.

Theorem 21 *If* $f^{\circ}(x) = f(x+\varpi_{\theta}) \bmod (s_{i+1})$ *where* θ *is a multiple of* 2^{i+1}*, then* $f^{\circ}(x+\beta_{i+1}) \bmod (s_i) = f(x+\varpi_{\theta+2^i}) \bmod (s_i)$.

Proof: Let θ be a multiple of 2^{i+1}. Since $s_i + 1$ divides s_{i+1}, then starting with $f^{\circ}(x) = f(x+\varpi_{\theta}) \bmod (s_{i+1})$ and modularly reducing both sides of this equation by $s_i + 1$ produces $f^{\circ}(x) \bmod (s_i + 1) = f(x+\varpi_{\theta}) \bmod (s_i + 1)$. Observe that $\varpi_{\theta} + \beta_{i+1} = \varpi_{\theta} + \varpi_{2^i} = \varpi_{\theta+2^i}$. Shifting both sides of the above equation by β_{i+1} yields $f^{\circ}(x+\beta_{i+1}) \bmod (s_i) = f(x+\varpi_{\theta+2^i}) \bmod (s_i)$ by Theorem 20. □

Algorithm : Shifted additive FFT

Input: $f^\circ(x) = f(x + \varpi_{j \cdot 2m}) \bmod (s_{i+1})$, a polynomial of degree less than $2m$ where $m = 2^i$, over $\mathbb{F}$, a field of characteristic 2 of dimension K and K is a power of two.

Output: The multipoint evaluation of f over $\varpi_{j \cdot 2m} + W_{i+1} \subseteq \mathbb{F}$, i.e. $f(\varpi_{j \cdot 2m+0}), f(\varpi_{j \cdot 2m+1}), \ldots, f(\varpi_{j \cdot 2m+2m-1})$.

0. If $(2m) = 1$, then return $f(x + \varpi_j) \bmod x = f(\varpi_j)$.
1. Divide f° by (s_i).
   This results in two polynomials, q and r, each of size m such that $f^\circ = q \cdot (s_i) + r$.
2. Compute $f_Y = f^\circ(x) \bmod (s_i) = f(x + \varpi_{j \cdot 2m}) \bmod (s_i) = r$ and
   $f_Z = f^\circ(x) \bmod (s_i + 1) = f(x + \varpi_{j \cdot 2m}) \bmod (s_i + 1) = q + r$.
3. Compute the Taylor shift of f_Z at β_{i+1} to obtain $f(x + \varpi_{j \cdot 2m+m}) \bmod (s_i)$.
4. Compute the additive FFT of $f(x + \varpi_{j \cdot 2m}) \bmod (s_i)$ to obtain $f(\varpi_{j \cdot 2m+0}), f(\varpi_{j \cdot 2m+1}), \ldots, f(\varpi_{j \cdot 2m+m-1})$.
5. Compute the additive FFT of $f(x + \varpi_{j \cdot 2m+m}) \bmod (s_i)$ to obtain $f(\varpi_{j \cdot 2m+m}), f(\varpi_{j \cdot 2m+m+1}), \ldots, f(\varpi_{j \cdot 2m+2m-1})$.
6. Return $f(\varpi_{j \cdot 2m+0}), f(\varpi_{j \cdot 2m+1}), \ldots, f(\varpi_{j \cdot 2m+2m-1})$.

Figure 3.3 Pseudocode for shifted additive FFT

So the reduction step receives as input $f(x + \varpi_{j \cdot 2m}) \bmod (s_{i+1})$, the modular reduction of some polynomial f by s_{i+1} that has been shifted by $\varpi_{j \cdot 2m}$ where $m = 2^i$ and $0 \leq j < 2^{k-i-1}$. So clearly $\theta = j \cdot 2m$ is a multiple of 2^{i+1}. The reduction step produces as output $f(x + \varpi_{j \cdot 2m}) \bmod (s_i)$ and $f(x + \varpi_{j \cdot 2m+m}) \bmod (s_i)$ by Theorem 21. Observe that both $j \cdot 2m$ and $j \cdot 2m + m$ are multiples of 2^i.

The algorithm is initialized with $f(x)$ which equals $f(x + \varpi_0) \bmod (s_k)$ if f has degree less than 2^k. By recursively applying the reduction step to $f(x)$, we obtain $f(x + \varpi_j) \bmod (x) = f(0 + \varpi_j)$ for all j in the range $0 \leq j < 2^k$, i.e. the additive FFT of $f(x)$ of size 2^k. Pseudocode for this shifted additive FFT algorithm is given in Figure 3.3.

Let us compute the cost of this algorithm. Line 0 is used to end the recursion and does not cost any operations. In line 1 we need to divide a polynomial of degree less than $2m$ by s_i, a polynomial with c_i coefficients since c_i is the number of non-zero coefficients in s_i. Since every nonzero element of s_i is 1, then line 1 costs no multiplications and $(c_i) \cdot m$ additions in $\mathbb{F}$. In line 2, we need to perform m additions to obtain $f \bmod (s_i - \varpi_{2j+1})$. Using the operation count derived in the appendix, the cost of line 3 is $1/2 \cdot m \cdot \log_2(m)$ multiplications and $1/2 \cdot m \cdot \log_2(m)$ additions in $\mathbb{F}$. Note that $1/2 \cdot m \cdot \log_2(m) = 1/4 \cdot 2m \cdot (\log_2(2m) - 1)$. The cost of lines 4 and 5 is equal to the number of operations needed to compute two additive FFTs of size m. Line 6 costs no operations.

The total number of operations to compute the additive FFT of size n using the shifted additive FFT algorithm is given by

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{4} \cdot n \cdot \log_2(n) - \frac{1}{4} \cdot n, \tag{3.50}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n \cdot c_{\log_2(n)-1} + \frac{1}{4} \cdot n \cdot \log_2(n) + \frac{1}{4} \cdot n, \tag{3.51}$$

where $M(1) = 0$ and $A(1) = 0$. Master Equation I can be used to solve these recurrence relations for the operation counts given by

$$M(n) = \frac{1}{8} \cdot n \cdot (\log_2(n))^2 - \frac{1}{8} \cdot n \cdot \log_2(n), \tag{3.52}$$

$$A(n) = \frac{1}{8} \cdot n \cdot (\log_2(n))^2 + \frac{1}{2} \cdot n \cdot (\log_2(n))^{\log_2(3)} + \frac{3}{8} \cdot n \cdot \log_2(n). \tag{3.53}$$

Observe that this algorithm is $\Theta(n \cdot (\log_2(n))^2)$, both in the number of multiplications and the number of additions. By comparing the steps of the shifted algorithm to

the Wang-Zhu-Cantor algorithm, one can see that more operations are required to implement the Taylor shift than those saved by using the simplified reduction step. In fact, it is unclear whether this algorithm is any better than the von zur Gathen-Gerhard algorithm used for the general case. So, unlike the multiplicative FFTs where the twisted version required the same number of operations as the classical version, a shifted version of an additive FFT algorithm requires significantly more operations than the unshifted version. Unless a faster method can be found to perform the Taylor shift, then shifted versions of additive FFT algorithms should be avoided.

3.4 Gao's additive FFT

In his computer algebra course notes [30], Gao introduced a new additive FFT algorithm which reduced the theoretical complexity needed to compute an additive FFT of a polynomial over $\mathbb{F}$ of degree less than n to $\Theta(n \cdot \log_2(n) \cdot \log_2 \log_2(n))$ for the first time. Here, $n = 2^k$, k is itself a power of two, and $\mathbb{F}$ has n elements. This algorithm is based on the factorization

$$x^\eta - x = \prod_{a \in W_t} (x^\tau - x - a), \tag{3.54}$$

where $\eta = \tau^2$ and $t = \log_2(\tau)$. Let us prove that this result holds before proceeding.

Theorem 22 *Suppose that $W_t \subset W_{2t} \subseteq \mathbb{F}$ where $\mathbb{F}$ is a finite field of characteristic 2, $\tau = 2^t$, and $\eta = \tau^2$. If $x^\tau - x$ is the minimal polynomial of W_t, and $x^\eta - x$ is the minimal polynomial of W_{2t}, then $x^\eta - x = \prod_{a \in W_t}(x^\tau - x - a)$.*

Proof: Suppose that $x^\tau - x$ is the minimal polynomial of W_t and W_{2t} is a quadratic extension of W_t so that $x^\eta - x$ is the minimal polynomial of W_{2t}. Then $x^\tau - x = \prod_{a \in W_t}(x - a)$. Observe that $x^\eta - x = x^{\tau^2} - x = x^{\tau^2} - x^\tau + x^\tau - x = (x^\tau - x)^\tau + (x^\tau - x)$.

Let $y = x^\tau - x$. Then $(x^\tau - x)^\tau + (x^\tau - x) = y^\tau + y = \prod_{a \in W_t}(y - a)$
$= \prod_{a \in W_t}(x^\tau - x - a)$. $\square$

Let $a_0, a_1, a_2, \ldots, a_{\tau-1}$ be the elements of W_t. We will assume that t is a power of two so that $x^\tau + x$ is the minimal polynomial of W_t and $x^\eta + x$ is the minimal polynomial of W_{2t} as proven in the appendix. The transformation used in Gao's algorithm is given by

$$\begin{aligned} R[x]/(x^\eta - x) \;\rightarrow\; & R[x]/(x^\tau - x - a_0) \\ \times\; & R[x]/(x^\tau - x - a_1) \\ \times\; & R[x]/(x^\tau - x - a_2) \\ & \cdots \\ \times\; & R[x]/(x^\tau - x - a_{\tau-1}). \end{aligned} \tag{3.55}$$

Let the input to the reduction step for Gao's algorithm be a polynomial f° of degree less than τ^2. We will first compute the Taylor expansion of f° at x^τ. This computation is discussed in a section of the appendix and involves finding "coefficients" $\{g_{\tau-1}(x), g_{\tau-2}(x), \ldots, g_1(x), g_0(x)\}$ such that

$$f^\circ(x) \;=\; g_{\tau-1} \cdot (x^\tau - x)^{\tau-1} + g_{\tau-2} \cdot (x^\tau - x)^{\tau-2} + \cdots + g_1 \cdot (x^\tau - x) + g_0. \tag{3.56}$$

Here, each "coefficient" is itself a polynomial of degree less than τ in x. The coefficients can be combined to form the polynomial

$$g(y) \;=\; g_{\tau-1} \cdot y^{\tau-1} + g_{\tau-2} \cdot y^{\tau-2} + \cdots + g_1 \cdot y + g_0. \tag{3.57}$$

Here, $y = x^\tau - x$ and $g(y) = f^\circ(x^\tau - x)$.

The goal of the reduction step is to compute $f^\circ \bmod (x^\tau - x - a)$ for every $a \in W_t$. Since $x^\tau - x \bmod (x^\tau - x - a) = a$, then this is equivalent to evaluating $g(y) = f(x^\tau - x)$ at every $a \in W_t$. For every $0 \leq \lambda < \tau$, let us define $h_\lambda(x)$ as the polynomial

$$h_\lambda(x) \;=\; (g_{\tau-1})_\lambda \cdot x^{\tau-1} + (g_{\tau-2})_\lambda \cdot x^{\tau-2} + \cdots + (g_{\tau-1})_\lambda \cdot x + (g_0)_\lambda \tag{3.58}$$

where the notation $(g_i)_d$ indicates the coefficient of degree d in the polynomial $g_i(x)$. So, $h_\lambda(x)$ is formed by extracting the coefficients of degree λ in $g_0(x), g_1(x), \ldots, g_{\tau-1}(x)$ and has degree less than τ. One way to conceptualize the construction of these polynomials is to write the coefficients of $g_0(x), g_1(x), \ldots, g_{\tau-1}(x)$ across the rows of a $\tau \times \tau$ matrix

$$\begin{pmatrix} (g_0)_0 & (g_0)_0 & \cdots & (g_0)_{\tau-1} \\ (g_1)_0 & (g_1)_1 & \cdots & (g_1)_{\tau-1} \\ \vdots & \vdots & \ddots & \vdots \\ (g_{\tau-1})_0 & (g_{\tau-1})_1 & \cdots & (g_{\tau-1})_{\tau-1} \end{pmatrix}. \tag{3.59}$$

The coefficients of the polynomial $h_\lambda(x)$ are determined by reading down column $\lambda+1$ of this matrix.

The evaluation of $g(y)$ at every $a \in W_t$ is equivalent to evaluating $h_\lambda(x)$ at every $a \in W_t$ for each λ in $0 \leq \lambda < \tau$. This in turn is equivalent to computing τ additive FFTs of polynomials of degree less than τ over W_τ. So the reduction step amounts to computing the Taylor expansion of f° at x^τ and then computing τ additive FFTs of size τ. Either Gao's reduction step or the Wang-Zhu-Cantor algorithm can be used recursively to compute these FFTs.

Most of the outputs of Gao's reduction step are not in the proper form for use as inputs to another application of the reduction step. In his notes, Gao instructs his students to compute the Taylor shift of each of these outputs following the reduction step. The following results provide the value of ξ that should be used for the Taylor shift.

Lemma 23 *For any $j < \tau \leq \sqrt{n}$, $\varpi_j = (\varpi_{\tau \cdot j})^\tau + \varpi_{\tau \cdot j}$.*

Proof: Let $\varphi(x) = \varphi^1(x) = x^2 + x$ and let $\varphi^m(x) = \varphi^{m-1}(\varphi(x))$. In the appendix, it is proven that $\varphi^t(x) = x^\tau + x$ where t is a power of two and $\tau = 2^t$. By repeated application of Lemma 19,

$$\begin{aligned}(\varpi_{2^t \cdot j})^\tau + \varpi_{2^t \cdot j} &= \varphi^t(\varpi_{2^t \cdot j}) = \varphi^{t-1}(\varphi(\varpi_{2^t \cdot j})) = \varphi^{t-1}({\varpi_{2^t \cdot j}}^2 + \varpi_{2^t \cdot j}) \quad (3.60)\\ &= \varphi^{t-1}(\varpi_{2^{t-1} \cdot j}) = \varphi^{t-2}(\varpi_{2^{t-2} \cdot j}) = \varphi^{t-3}(\varpi_{2^{t-3} \cdot j}) = \cdots \\ &= \varphi^1(\varpi_{2j}) = (\varpi_{2j})^2 + \varpi_{2j} = \varpi_j.\end{aligned}$$

□

Theorem 24 *If $r(x) = f^\circ(x) \bmod (x^\tau - x - \varpi_j)$ for some $\varpi_j \in W_t$, then $r(x + \varpi_{\tau \cdot j}) = f^\circ(x + \varpi_{\tau \cdot j}) \bmod (x^\tau - x)$.*

Proof: Let $r(x) = f^\circ(x) \bmod (x^\tau - x - \varpi_j)$ for some $\varpi_j \in W_t$. In the appendix, it is proven that $s_t(x) = \varphi^t(x) = x^\tau + x$. So, $r(x) = f^\circ(x) \bmod (s_t(x) - \varpi_j)$. Thus,

there exists a $q(x)$ such that $f^\circ(x) = q(x) \cdot (s_t(x) - \varpi_j) + r(x)$. Shifting both sides of this equation by $\varpi_{\tau \cdot j}$ gives

$$\begin{aligned} f^\circ(x + \varpi_{\tau \cdot j}) &= q(x + \varpi_{\tau \cdot j}) \cdot (s_t(x + \varpi_{\tau \cdot j}) - \varpi_j) + r(x + \varpi_{\tau \cdot j}) \\ &= q(x + \varpi_{\tau \cdot j}) \cdot (s_t(x) + s_t(\varpi_{\tau \cdot j})) - \varpi_j) + r(x + \varpi_{\tau \cdot j}) \\ &= q(x + \varpi_{\tau \cdot j}) \cdot (s_t(x) + (\varpi_{\tau \cdot j})^\tau + \varpi_{\tau \cdot j}) - \varpi_j) + r(x + \varpi_{\tau \cdot j}) \\ &= q(x + \varpi_{\tau \cdot j}) \cdot (s_t(x) + \varpi_j - \varpi_j) + r(x + \varpi_{\tau \cdot j}) \\ &= q(x + \varpi_{\tau \cdot j}) \cdot s_t(x) + r(x + \varpi_{\tau \cdot j}). \end{aligned} \tag{3.61}$$

Thus, $r(x + \varpi_{\tau \cdot j}) = f^\circ(x + \varpi_{\tau \cdot j}) \bmod (x^\tau + x)$. □

The following theorem is indended to show that an algorithm based on this reduction step yields the additive FFT of some polynomial $f(x)$ of degree less than $n = 2^k$ where k is a power of two.

Theorem 25 *If* $f^\circ(x) = f(x + \varpi_\theta) \bmod (x^\eta + x)$ *where* θ *is a multiple of* $\eta = \tau^2$, *then* $f^\circ(x + \varpi_{\tau \cdot j}) \bmod (x^\tau + x) = f(x + \varpi_{\theta + \tau \cdot j}) \bmod (x^\tau + x)$ *where* $0 \leq j < \tau$.

Proof: Let θ be a multiple of $\eta = \tau^2$ and let $j < \tau$. Since $x^\tau + x + \varpi_j$ divides $x^\eta + x$, then starting with $f^\circ(x) = f(x + \varpi_\theta) \bmod (x^\eta + x)$ and modularly reducing both sides of this equation by $x^\tau + x + \varpi_j$, we obtain $f^\circ(x) \bmod (x^\tau + x + \varpi_j) = f(x + \varpi_\theta) \bmod (x^\tau + x + \varpi_j)$. Since $j < \tau$, then $\tau \cdot j < \tau^2 \leq \theta$ and $\varpi_\theta + \varpi_{\tau \cdot j} = \varpi_{\theta + \tau \cdot j}$. Also recall that $x^\tau + x = s_t(x)$. Shifting the above equation by $\varpi_{\tau \cdot j}$, we obtain $f^\circ(x + \varpi_{\tau \cdot j}) \bmod (x^\tau + x) = f(x + \varpi_{\theta + \tau \cdot j}) \bmod (x^\tau + x)$ using the fact that $s_t(x + \varpi_{\tau \cdot j}) + \varpi_j = x^\tau + x$. □

So the reduction step receives as input $f(x+\varpi_{j\cdot\eta}) \bmod (x^{\eta}+x)$, the modular reduction of some polynomial f by s_{2t} that has been shifted by $\varpi_{j\cdot\eta}$ where $\eta = 2^{2t}$ and t is a power of two. The reduction step produces the outputs $f(x+\varpi_{j\cdot\eta}) \bmod (x^{\tau}-x)$, $f(x+\varpi_{j\cdot\eta+\tau}) \bmod (x^{\tau}-x)$, $f(x+\varpi_{j\cdot\eta+2\cdot\tau}) \bmod (x^{\tau}-x), \ldots, f(x+\varpi_{j\cdot\eta+(\tau-1)\cdot\tau}) \bmod (x^{\tau}-x)$ after the shifts have been applied where $\tau = \sqrt{\eta} = 2^{t}$. Observe that $j\cdot\eta+\phi\cdot\tau$ is a multiple of 2^{t} for $0 \leq \phi < \tau$.

The algorithm is initialized with $f(x)$ which equals $f(x+\varpi_0) \bmod s_k$ if f has degree less than 2^{k}. This reduction step is recursively applied until it is not possible to take the square root of the input size. The Wang-Zhu-Cantor algorithm is used to complete the computations. If k is a power of two, then the Wang-Zhu-Cantor algorithm is only used to resolve the recursive calls with input size of two. Pseudocode for Gao's algorithm is provided in Figure 3.4.

Let us now compute the cost of this algorithm for the case where K is a power of two and $n = 2^{K}$. Line 0 is used to end the recursion at a cost of no multiplications and one addition, i.e. $M(2) = 0$ and $A(2) = 1$. In line 1, a Taylor expansion of a polynomial of size η at x^{τ} is required. This requires no multiplications and $1/4\cdot\eta\cdot\log_2(\eta)$ additions. In theory, line 2 costs no operations. However, in practice it may be necessary to rearrange the results from line 1 so that the coefficients of each h_{λ} are adjacent to each other in memory. This costs $\tau = \sqrt{\eta}$ copies for each value of λ. We will assume that a copy operation requires the same amount of computational effort as an addition operation. Alternatively, the recursive calls in the algorithm can be expanded which results in a much longer implementation that operates on adjacent coefficients in the polynomials located τ cells apart. We will assume that when $\eta \leq 2^{16}$, then no rearrangement of the elements is necessary. Lines 3-5 involve τ recursive calls to the additive FFT algorithm. Each recursive call to the algorithm requires $M(\tau)$ multiplications and $A(\tau)$ additions. The input polynomial for each

Algorithm : Gao's additive FFT

Input: A polynomial $f^\circ = f(x + \varpi_{j\cdot\eta}) \bmod (x^\eta - x)$ over $\mathbb{F}$ where $\eta = 2^{2t}$.
Here, $\mathbb{F}$ has $N = 2^K$ elements where K is also a power of two.

Output: The FFT of f over $\varpi_{j\cdot\eta}+W_{2t}$, i.e. $f(\varpi_{j\cdot\eta}), f(\varpi_{j\cdot\eta+1}), \cdots, f(\varpi_{j\cdot\eta+\eta-1})$.

0. If $\eta = 2$, then return $\{f_0^\circ, f_0^\circ + f_1^\circ\}$ (from Wang-Zhu-Cantor algorithm).
1. Compute $g(y)$, the Taylor expansion of f° at x^τ where $\tau = \sqrt{\eta}$.
2. Construct $h_\lambda(x)$ using the coefficients of $g(y)$ for each λ in $0 \le \lambda < \tau$.
3. **for** $\lambda = 0$ to $\tau - 1$ **do**
4. Recursively call Gao's additive FFT algorithm
with input $h_\lambda(x) = h_\lambda(x) \bmod (x^\tau - x)$
to obtain $h_\lambda(\varpi_0), h_\lambda(\varpi_1), \cdots, h_\lambda(\varpi_{\tau-1})$.
5. **end for** (Loop λ)
6. Construct $f^\circ \bmod (x^\tau - x - \varpi_\phi)$ from the evaluations of $h_\lambda(x)$
for each ϕ in $0 \le \phi < \tau$.
7. **for** $\phi = 0$ to $\tau - 1$ **do**
8. Compute the Taylor shift of $f^\circ \bmod (x^\tau - x - \varpi_\phi)$ at $\varpi_{\phi\cdot\tau}$ to obtain
$f^\circ(x + \varpi_{\phi\cdot\tau}) \bmod (x^\tau - x) = f(x + \varpi_{j\cdot\eta+\phi\cdot\tau}) \bmod (x^\tau - x)$.
9. Recursively call Gao's additive FFT algorithm
with input $f(x + \varpi_{j\cdot\eta+\phi\cdot\tau}) \bmod (x^\tau - x)$
to obtain $f(\varpi_{j\cdot\eta+\phi\cdot\tau}), f(\varpi_{j\cdot\eta+\phi\cdot\tau+1}), \cdots, f(\varpi_{j\cdot\eta+\phi\cdot\tau+\tau-1})$.
10. **end for** (Loop ϕ)
11. Return $f(\varpi_{j\cdot\eta}), f(\varpi_{j\cdot\eta+1}), \cdots, f(\varpi_{j\cdot\eta+\eta-1})$.

Figure 3.4 Pseudocode for Gao's additive FFT

of these recursive calls is h_λ and has no relation to the original input polynomial f. In line 6, the results of the additive FFT computations are rearranged to obtain $f^\circ \bmod (x^\tau - x - \varpi_\phi)$ for each ϕ in $0 \leq \phi < \tau$. In theory, this requires no operations, but in practice a total of $\tau \cdot \tau = \eta$ copy operations may be involved when $\eta > 2^{16}$. Lines 7-10 involve recursively calling the additive FFT algorithm after performing a Taylor shift on the inputs to put them in the proper form. Note that no Taylor shift is required for the case where $\phi = 0$. Using the operation counts derived in the appendix, the Taylor shifts require $\frac{1}{2} \cdot (\tau - 1) \cdot \tau \cdot \log_2(\tau)$ multiplications and $\frac{1}{2} \cdot (\tau - 1) \cdot \tau \cdot \log_2(\tau)$ additions. [1] The recursive calls to the algorithm require $\tau \cdot M(\tau)$ multiplications and $\tau \cdot A(\tau)$ additions.

The total number of operations to compute the additive FFT of size n using Gao's algorithm is given by

$$M(n) = 2 \cdot \sqrt{n} \cdot M(\sqrt{n}) + 1/4 \cdot n \cdot \log_2(n) - 1/2 \cdot \sqrt{n} \cdot \log_2(\sqrt{n}), \quad (3.62)$$

$$A(n) = 2 \cdot \sqrt{n} \cdot A(\sqrt{n}) + 1/2 \cdot n \cdot \log_2(n) - 1/2 \cdot \sqrt{n} \cdot \log_2(\sqrt{n}), \quad (3.63)$$

where $M(2) = 0$ and $A(2) = 1$. Master Equation IV can be used to solve these recurrence relations for the formulas

$$M(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) - \frac{1}{4} \cdot \Lambda \cdot n \cdot \log_2(n), \quad (3.64)$$

$$A(n) = \frac{1}{2} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + \left(\frac{1}{2} - \frac{1}{4} \cdot \Lambda \right) \cdot n \cdot \log_2(n), \quad (3.65)$$

[1] Note that $\frac{1}{2} \cdot (\tau - 1) \cdot \tau \cdot \log_2(\tau) = \frac{1}{4} \cdot \eta \cdot \log_2(\eta) - \frac{1}{2} \cdot \tau \cdot \log_2(\tau)$.

where

$$\Lambda = \sum_{i=0}^{\log_2 \log_2(n)} (1/2)^{2^i} \tag{3.66}$$

and is bounded by $1/2 \leq \Lambda < 1$.

When $n > 2^{16}$, we can use the recurrence relation

$$A_c(n) = 2 \cdot \sqrt{n} \cdot A_c(\sqrt{n}) + 2 \cdot n \tag{3.67}$$

to model the number of copies required to compute an additive FFT of size n with the initial condition $A_c(2^{16}) = 0$. This recurrence relation is solved in the appendix, yielding

$$A_c(n) = \frac{1}{8} \cdot n \cdot \log_2(n) - 2 \cdot n \tag{3.68}$$

when $2^i > 2^{16}$. If a copy requires the same cost as an addition, then this increases the addition count slightly to

$$A(n) = \frac{1}{2} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + \left(\frac{5}{8} - \frac{1}{4} \cdot \Lambda \right) \cdot n \cdot \log_2(n) - 2 \cdot n. \tag{3.69}$$

Gao claims slightly higher operation counts of

$$M(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + 2 \cdot n \cdot \log_2(n) - n, \tag{3.70}$$

$$A(n) = \frac{1}{2} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + n \cdot \log_2(n) \tag{3.71}$$

in the course notes. Most of the discrepancy is accounted for by Gao's use of a different basis which requires additional computations throughout the algorithm. Additionally, Gao does not subtract operations to account for the cases where a Taylor shift is not required (the $\phi = 0$ cases in the algorithm, accounted for by the Λ terms in the operation counts).

In any event, the algorithm has a theoretical complexity of $\Theta(n \cdot \log_2(n) \cdot \log_2 \log_2(n))$. Unfortunately, the number of multiplications is $\Theta(n \cdot \log_2(n) \cdot \log_2 \log_2(n))$ and as a result the algorithm will be more expensive than the Wang-Zhu-Cantor algorithm for any practical size. This is a consequence of the use of the Taylor shifts throughout the algorithm. Gao's algorithm can be viewed as a "shifted" version of some other more efficient additive FFT algorithm which will be discussed in the next section. Although it is believed that Gao's algorithm will always be more expensive than the Wang-Zhu-Cantor algorithm, Gao reduced the theoretical complexity of the computation of the additive FFT and laid the groundwork for the following algorithm which is more efficient than the Wang-Zhu-Cantor algorithm as well.

3.5 A new additive FFT

A more efficient additive FFT algorithm that does not involve Taylor shifting is based on the factorization

$$x^\eta - x - (a^\tau + a) = \prod_{B \in W_t} (x^\tau - x - a - B), \tag{3.72}$$

where $\eta = \tau^2$, $t = \log_2(\tau)$, and a is a linear combination of $\{\beta_{t+1}, \beta_{t+2}, \ldots, \beta_{2t}\}$. The proof of this result is similar to the proof of the factorization used in Gao's algorithm.

For an a which is a linear combination of $\{\beta_{t+1}, \beta_{t+2}, \ldots, \beta_{2t}\}$ and a $B \in W_t$, there exists a $\delta \in (a + W_t)$ such that $\delta = a + B$. Thus, we can rewrite (3.72) as

$$x^\eta - x - (a^\tau + a) = \prod_{\delta \in (a+W_t)} (x^\tau - x - \delta). \tag{3.73}$$

Let $\delta_0, \delta_1, \delta_2, \ldots, \delta_{\tau-1}$ be the elements of $a + W_t$. The new additive FFT is based on the transformation

$$\begin{aligned} R[x]/\left(x^\eta - x - (a^\tau + a)\right) \rightarrow\ & R[x]/(x^\tau - x - \delta_0) \\ \times\ & R[x]/(x^\tau - x - \delta_1) \\ \times\ & R[x]/(x^\tau - x - \delta_2) \\ & \ldots \\ \times\ & R[x]/(x^\tau - x - \delta_{\tau-1}). \end{aligned} \tag{3.74}$$

Note that this reduction step is somewhat more complicated than the reduction step used in Gao's algorithm, but the use of this transformation will eliminate the need to compute the Taylor shifts throughout the algorithm.

The input to the reduction step of the new algorithm will be a polynomial f° of degree less than $\eta = \tau^2$. As with Gao's algorithm, we will perform the reductions by first computing the Taylor expansion $g(y)$ of f° at x^τ. However, instead of evaluating $g(y)$ at each of the elements in W_t, we will evaluate $g(y)$ at each of the elements in $a + W_t$. As a result, the transformation (3.74) can be used directly on the outputs of the reduction step without the need to compute the Taylor shift of these results.

Without some additional structure to the elements of $\mathbb{F}$, it would be difficult to implement an algorithm based on the transformation just described. Fortunately, the special basis developed by Cantor can be applied here to simplify the expressions in this new algorithm. The key result that will be used is $\varpi_j = (\varpi_{\tau \cdot j})^\tau + \varpi_{\tau \cdot j}$ for any $j < \tau$, proven in Lemma 23.

The algorithm will be initialized with some polynomial $f(x)$ of degree less than $n = 2^k$ and so $f \bmod (x^n - x - \varpi_J) = f$ for some J. Each reduction step will receive some polynomial $f^\circ = f \bmod (x^\eta - x - \varpi_j)$ for some j. A total of $\tau = \sqrt{\eta}$ additive FFTs of size τ will be computed to obtain $f \bmod (x^\tau - x - \varpi_\phi)$ for each ϕ in $\tau \cdot j \leq \phi < \tau \cdot (j + 1)$. The algorithm is recursively called until it is no longer possible to take the square root of the input size. As with Gao's algorithm, we apply the Wang-Zhu-Cantor algorithm at this point to complete the computation. The output of this process is the evaluation of f at each of the points in $\varpi_{n \cdot J} + W_k$.

If K is chosen to be itself a power of two, then the Wang-Zhu-Cantor algorithm is only needed to perform the reduction steps with input size of two. If K is not itself a power of two, then it will not be possible to constuct the special basis and the von zur Gathen-Gerhard algorithm must be used instead. Finite fields are typically selected in practice where K is a power of two to make full use of the number of positions available in standard data sizes on a computer, so this is usually not a concern. For the rest of this section, we will assume that K is a power of two.

Algorithm : New additive FFT

Input: A polynomial $f^\circ = f \bmod (x^\eta - x - \varpi_j)$ over $\mathbb{F}$ where $\eta = 2^{2t}$.
Here, $\mathbb{F}$ has $N = 2^K$ elements where K is a power of two.

Output: The FFT of f over $\varpi_{j\cdot\eta}+W_{2t}$, i.e. $f(\varpi_{j\cdot\eta}), f(\varpi_{j\cdot\eta+1}), \cdots, f(\varpi_{j\cdot\eta+\eta-1})$.

0. If $\eta = 2$, then return $\{f_0^\circ + \varpi_{2j} \cdot f_1^\circ, f_0^\circ + \varpi_{2j} \cdot f_1^\circ + f_1^\circ\}$.
1. Compute $g(y)$, the Taylor expansion of f° at x^τ where $\tau = \sqrt{\eta}$.
2. Construct h_λ using the coefficients of $g(y)$ for each λ in $0 \leq \lambda < \tau$.
3. **for** $\lambda = 0$ to $\tau - 1$ **do**
4. Recursively call the new additive FFT algorithm
   with input $h_\lambda = h_\lambda \bmod (x^\tau - x - \varpi_j)$
   to obtain $h_\lambda(\varpi_{j\cdot\tau}), h_\lambda(\varpi_{j\cdot\tau+1}), \cdots, h_\lambda(\varpi_{j\cdot\tau+\tau-1})$.
5. **end for** (Loop λ)
6. Construct $f \bmod (x^\tau - x - \varpi_\phi)$ from the evaluations of h_λ
   for each ϕ in $j \cdot \tau \leq \phi < (j+1) \cdot \tau$.
7. **for** $\phi = j \cdot \tau$ to $(j+1) \cdot \tau$ **do**
8. Recursively call the new additive FFT algorithm
   with input $f \bmod (x^\tau - x - \varpi_\phi)$
   to obtain $f(\varpi_{\tau\cdot\phi}), f(\varpi_{\tau\cdot\phi+1}), \cdots, f(\varpi_{\tau\cdot\phi+\tau-1})$.
9. **end for** (Loop ϕ)
10. Return $f(\varpi_{j\cdot\eta}), f(\varpi_{j\cdot\eta+1}), \cdots, f(\varpi_{j\cdot\eta+\eta-1})$.

Figure 3.5 Pseudocode for new additive FFT

If k is not itself a power of two, then let κ be the largest power of two such that $\kappa \leq k$. If we wish to evaluate f at each point in W_k, then reduction steps from the Wang-Zhu-Cantor algorithm can be used to compute $f \bmod (x^\kappa - x - \varpi_j)$ for all j in $0 \leq j < 2^{k-\kappa}$. The new algorithm can be used on each of these results to complete the additive FFT. For simplicity, the pseudocode given in Figure 3.5 assumes that k is a power of two. The reader can add the Wang-Zhu-Cantor algorithm reduction steps if desired for arbitrary k.

Let us now compute the cost of this algorithm for the case where κ is a power of two and $\eta = 2^\kappa$. Line 0 is used to end the recursion at a cost of one multiplication and two additions, i.e. $M(2) = 1$ and $A(2) = 2$. However, if $j = 0$, then this simplifies to

$M(2) = 0$ and $A(2) = 1$. In line 1, a Taylor expansion of a polynomial of size η at x^τ is required. This requires no multiplications and $1/4 \cdot \eta \cdot \log_2(\eta)$ additions. In theory, line 2 costs no operations. However, in practice it may be necessary to rearrange the results from line 1 so that the coefficients of each $h_\lambda(x)$ are adjacent to each other in memory. This costs $\tau = \sqrt{\eta}$ copies for each value of λ. We will assume that a copy operation requires the same amount of effort as an addition operation. Alternatively, the recursive calls in the algorithm can be expanded which results in a much longer implementation that operates on adjacent coefficients in the polynomials located $\sqrt{\eta}$ cells apart. We will assume that when $\eta \leq 2^{16}$, then no rearrangement of the elements is necessary. Lines 3-5 involve τ recursive calls to the additive FFT algorithm, each at a cost of $M(\tau)$ multiplications and $A(\tau)$ additions. The input polynomial for each of these recursive calls is h_λ and has no relation to the original input polynomial f. In line 6, the results of the additive FFT computations are rearranged to obtain $f \bmod (x^\tau - x - \varpi_\phi)$ for each ϕ in $j \cdot \tau \leq \phi < (j+1) \cdot \tau$. In theory, this requires no operations, but in practice a total of $\tau \cdot \tau = \eta$ copy operations may be involved when $\eta > 2^{16}$. Lines 7-9 involve recursively calling the additive FFT algorithm to complete the computation. These recursive calls costs $\tau \cdot M(\tau)$ multiplications and $\tau \cdot A(\tau)$ additions.

The total number of operations to compute the additive FFT of size n using the new algorithm is given by

$$M(n) = 2 \cdot \sqrt{n} \cdot M(\sqrt{n}), \tag{3.75}$$

$$A(n) = 2 \cdot \sqrt{n} \cdot A(\sqrt{n}) + \frac{1}{4} \cdot n \cdot \log_2(n), \tag{3.76}$$

where $M(2) = 1$ and $A(2) = 2$. These recurrence relations can be solved using Master Equation IV.

We must also subtract operations of both types to account for the cases where $j = 0$. A recurrence relation which gives the number of these cases is given by

$$M_s(n) = (\sqrt{n} + 1) \cdot M_s(\sqrt{n}), \tag{3.77}$$

where $M_s(2) = 1$. The number of additions saved is governed by the same recurrence relation and initial condition. This recurrence relation is solved in the appendix to yield $n - 1$ of each type of operation saved.

Combining the results, we obtain

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{3.78}$$

$$A(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + n \cdot \log_2(n) - n + 1. \tag{3.79}$$

By the same analysis used for Gao's algorithm, we will assume that a total of

$$A_c(n) = \frac{1}{8} \cdot n \cdot \log_2(n) - 2 \cdot n \tag{3.80}$$

copies are required throughout the new algorithm when $n > 2^{16}$. If a copy requires the same cost as an addition, then this increases the addition count slightly to

$$A(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + \frac{9}{8} \cdot n \cdot \log_2(n) - 3 \cdot n + 1. \quad (3.81)$$

Regardless of whether the algorithm is called with $j = 0$ or $j > 0$ and whether or not copies are required, the number of multiplications of the new algorithm is equivalent to the Wang-Zhu-Cantor algorithm, and the addition count has been reduced to $\Theta(n \cdot \log_2(n) \cdot \log_2 \log_2(n))$.

To determine the cost of the new algorithm when k is not a power of two, we will subtract the operations saved in levels κ and below from the cost of the Wang-Zhu-Cantor algorithm for size 2^k. The savings is $2^{k-\kappa}$ times the difference in the number of additions needed to compute one additive FFT of size 2^κ. If no copies are required, then the total number of additions when k is not a power of two is given by

$$\begin{aligned} A(n) &= \frac{1}{2} \cdot n \cdot \log_2(n)^{1.585} + n \cdot \log(n) - n + 1 \\ &\quad -2^{\log_2(n)-\kappa} \cdot \left(\frac{1}{2} \cdot 2^\kappa \cdot \kappa^{1.585} + 2^\kappa \cdot \kappa - 2^\kappa + 1 \right) \\ &\quad +2^{\log_2(n)-\kappa} \cdot \left(\frac{1}{4} \cdot 2^\kappa \cdot \kappa \cdot \log_2(\kappa) + 2^\kappa \cdot \kappa - 2^\kappa + 1 \right) \\ &= \frac{1}{2} \cdot n \cdot \log_2(n)^{1.585} + n \cdot \log(n) - n + 1 \\ &\quad -\frac{1}{2} \cdot n \cdot \kappa^{1.585} + \frac{1}{4} \cdot n \cdot \kappa \cdot \log_2(\kappa). \end{aligned} \quad (3.82)$$

In the case of multiplications, there is no savings because the multiplication count is the same for both algorithms.

3.6 Concluding remarks

This chapter presented several algorithms that can be used to compute the additive FFT of a polynomial defined over a finite field with $N = 2^K$ elements. In the general case, the von zur Gathen-Gerhard algorithm requiring $\Theta(n \cdot \log_2(n)^2)$ operations can be used to compute the additive FFT.

If K is a power of two, then several more efficient algorithms can be used to compute the FFT. The Wang-Zhu-Cantor algorithm is one option and requires $\Theta(n \cdot (\log_2(n))^{1.585})$ operations. A shifted version of this algorithm was also presented, but required more operations. Two new algorithms were also introduced which could compute the additive FFT using $\Theta(n \cdot \log_2(n) \cdot \log_2 \log_2(n))$ operations whenever the additive FFT size is 2^k where k is a power of two. Gao's algorithm is the first algorithm to achieve a theoretical complexity of $\Theta(n \cdot \log_2(n) \cdot \log_2 \log_2(n))$ and the new algorithm is the first algorithm to outperform the Wang-Zhu-Cantor algorithm for all practical sizes. We also discussed how the new algorithm could be combined with Wang-Zhu-Cantor algorithm to produce a hybrid algorithm that can efficiently compute the additive FFT for any k.

CHAPTER 4

INVERSE FAST FOURIER TRANSFORM ALGORITHMS

Chapter 2 examined several FFT algorithms which evaluate a polynomial $f \in R[x]$ of degree less than n at each of the powers of some primitive nth root of unity ω where n is a power of two or three. Chapter 3 examined additional algorithms which solve the multipoint evaluation problem when R is a finite field. In this chapter, several algorithms for computing the inverse Fast Fourier Transform (IFFT) will be considered. These algorithms interpolate the n evaluations produced by an FFT algorithm back into the polynomial f. An IFFT algorithm will be given for several of the FFT algorithms considered in the previous two chapters. The reader can construct the IFFT algorithms for the other cases using the techniques discussed in the following sections.

4.1 Classical radix-2 IFFT

The radix-2 inverse FFT algorithm interpolates $n = 2^k$ evaluations into a polynomial f of degree less than n over a ring R with a primitive nth root of unity. The n points used for the function evaluations are roots of $x^n - 1$, presented in the order determined by the σ function.

The inverse of Cooley-Tukey's FFT reduction step can be viewed as

$$\begin{aligned} R[x]/(x^m - b) \quad &\rightarrow \quad R[x]/(x^{2m} - b^2) \\ \times \; R[x]/(x^m + b) & \end{aligned} \tag{4.1}$$

by using the algebraic transformations given in [2].

Suppose that we wish to interpolate $f_{Y'} = f \bmod (x^m - b)$ and $f_{Z'} = f \bmod (x^m + b)$ into $f_{A'} \cdot x^m + f_{B'} = f \bmod (x^{2m} - b^2)$ where $f_{A'}$ and $f_{B'}$ are each polynomials of degree less than m. From the classical radix-2 FFT algorithm reduction step, we know that $f_{Y'} = b \cdot f_{A'} + f_{B'}$ and $f_{Z'} = -b \cdot f_{A'} + f_{B'}$. We can either solve this system of equations for $f_{A'}$ and $f_{B'}$ or we can apply the Chinese Remainder Theorem to determine that $f_{A'} = 1/2 \cdot b^{-1} \cdot (f_{Y'} - f_{Z'})$ and $f_{B'} = 1/2 \cdot (f_{Y'} + f_{Z'})$. Viewing this interpolation step as a special case of the fast interpolation algorithm, observe that the u and v polynomials are constants in every case.

The classical radix-2 IFFT algorithm uses a similar interpolation step, but saves the divisions by two in the formulas above until the end of the algorithm. In other words, the inverse FFT algorithm interpolation step receives as input $f_Y = m \cdot f \bmod (x^m - b)$ and $f_Z = m \cdot f \bmod (x^m + b)$. The formulas $f_A = b^{-1} \cdot (f_Y - f_Z)$ and $f_B = f_Y + f_Z$ are then used to interpolate these inputs into $f_A \cdot x^m + f_B = (2m) \cdot f \bmod (x^{2m} - b^2)$. [1]

The goal of this IFFT algorithm is to interpolate the set of polynomial evaluations $\{f(1), f(\omega^{\sigma(1)}), f(\omega^{\sigma(2)}), \ldots, f(\omega^{\sigma(n-1)})\}$ back into the polynomial f of degree less than $n = 2^k$. We will recursively apply the interpolation step with appropriate selections of m and b. Since $(\omega^{\sigma(2j)})^2 = \omega^{\sigma(j)}$ and $-\omega^{\sigma(2j)} = \omega^{\sigma(2j+1)}$ for all $j < n/2$, then b can be easily determined. Each interpolation step receives as input $m \cdot f \bmod (x^m - \omega^{\sigma(2j)})$ and $m \cdot f \bmod (x^m - \omega^{\sigma(2j+1)})$ for some $j < n/2$. By applying the transformation

---

[1] It is also possible to pre-scale each of the evaluations following the Horowitz approach using (1.30). This is somewhat more complicated than scaling at the end of the algorithm since different scaling factors are used for each of the inputs.

Algorithm : Classical radix-2 IFFT
Input: The evaluations $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$ of some polynomial with coefficients in a ring R with primitive nth root of unity ω. Here, m is a power of two where $2m \leq n$.
Output: $(2m) \cdot f \bmod (x^{2m} - \omega^{\sigma(j)})$.
0. If $(2m) = 1$ then return $f \bmod (x - \omega^{\sigma(j)}) = f(\omega^{\sigma(j)})$. 1. Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+m-1)})$ to obtain $f_Y = m \cdot f \bmod (x^m - \omega^{\sigma(2j)})$. 2. Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+m)}), f(\omega^{\sigma(j\cdot 2m+m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$ to obtain $f_Z = m \cdot f \bmod (x^m - \omega^{\sigma(2j+1)})$. 3. Compute $f_A = (\omega^{\sigma(2j)})^{-1} \cdot (f_Y - f_Z)$. 4. Compute $f_B = f_Y + f_Z$. 5. Return $(2m) \cdot f \bmod (x^{2m} - \omega^{\sigma(j)}) = f_A \cdot x^m + f_B$.

Figure 4.1 Pseudocode for classical radix-2 IFFT

$$\begin{pmatrix} f_A \\ f_B \end{pmatrix} = \begin{pmatrix} (\omega^{\sigma(2j)})^{-1} & -(\omega^{\sigma(2j)})^{-1} \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_Y \\ f_Z \end{pmatrix}, \tag{4.2}$$

then the output of the interpolation step is $f_A + m \cdot f_B = (2m) \cdot f \bmod (x^{2m} - \omega^{\sigma(j)})$. After all of the interpolation steps have been completed, then we will have $n \cdot f \bmod (x^n - 1)$. If f has degree less than n, then this output is equal to $n \cdot f$. By multiplying this result by $1/n$, then the desired polynomial f is recovered.

Pseudocode for this IFFT algorithm is given in Figure 4.1. As with the FFT algorithm, the IFFT algorithm is typically applied to the ring of complex numbers. In this case ω is often $e^{\mathrm{I}\cdot 2\pi/n}$ and z is traditionally used in place of x as the variable.

Let us now analyze the cost of this algorithm. Line 0 is just used to end the recursion and costs no operations. The cost of lines 1 and 2 is equal to the number of operations needed to compute two IFFTs of size m. In line 3, we first subtract f_Z

from f_Y at a cost of m subtractions in R. To complete the instruction, we multiply this result by $(\omega^{\sigma(2j)})^{-1}$ at a cost of m multiplications in R. If $j = 0$, however, then no multiplications are required. In line 4, we need to add f_Z to f_Y at a cost of m additions in R. Line 5 just involves logicially combining f_A and f_B into the desired result and costs no operations. The total number of operations to compute the IFFT of size n is

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{n}{2}, \tag{4.3}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + n, \tag{4.4}$$

where $M(1) = 0$ and $A(1) = 0$. We must also subtract multiplications to account for the cases where $j = 0$. The analysis is identical to that used to derive the classical radix-2 FFT operation counts and results in the formulas

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{4.5}$$

$$A(n) = n \cdot \log_2(n). \tag{4.6}$$

To recover the unscaled polynomial, we must multiply the final result of this algorithm by $1/n$ at a cost of n additional multiplications in R. With the extra multiplications included, this IFFT algorithm is said to be $\Theta(n \cdot \log_2(n))$.

4.2 Twisted radix-2 IFFT

We will now consider the companion IFFT for Gentleman and Saude's "twisted" radix-2 FFT algorithm introduced in [35]. The inverse of this twisted FFT algorithm transformation is

$$\begin{aligned} & R[x]/(x^m-1) \;\; \rightarrow \;\; R[x]/(x^{2m}-1). \\ \times \;\; & R[x]/(x^m+1) \end{aligned} \tag{4.7}$$

Initially, both inputs are polynomials in $R[x]/(x^m-1)$. The transformation $x \rightarrow \zeta^{-1} \cdot x$ is applied to one of these inputs prior to the interpolation step to convert it to be a polynomial contained in $R[x]/(x^m+1)$. Here, ζ is given by Theorem 3 and rotates the roots of unity by the inverse of the amount used in the twisted radix-2 FFT algorithm.

The interpolation step of the twisted radix-2 IFFT algorithm receives as input $f_Y = m \cdot f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m-1)$ and $m \cdot f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m-1)$ for some $j < m$. The second input is twisted by $\omega^{-\sigma(1)/m}$ to obtain $f_Z = m \cdot f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m+1)$. By inverting the transformation matrix of the twisted FFT algorithm or by using the Chinese Remainder Theorem, then $2m \cdot f(\omega^{\sigma(j)/2m} \cdot x) \bmod (x^{2m}-1) = 2m \cdot (f_A \cdot x^m + f_B)$ can be obtained by using the transformation

$$\begin{pmatrix} f_A \\ f_B \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_Y \\ f_Z \end{pmatrix}. \tag{4.8}$$

As with the classical radix-2 IFFT algorithm, there is a factor of 1/2 omitted from

Algorithm : Twisted radix-2 IFFT

Input: The evaluations $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$
of some polynomial with coefficients in a ring R.
Here R has a nth root of unity ω, and m is a power of two where $2m \leq n$.

Output: $2m \cdot f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$.

0. If $(2m) = 1$ then return $f(\omega^{\sigma(j)} \cdot x) \bmod (x - 1) = f(\omega^{\sigma(j)})$.
1. Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+0)}), f(\omega^{\sigma(j\cdot 2m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+m-1)})$
   to obtain $f_Y = m \cdot f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m - 1)$.
2. Compute the IFFT of $f(\omega^{\sigma(j\cdot 2m+m)}), f(\omega^{\sigma(j\cdot 2m+m+1)}), \ldots, f(\omega^{\sigma(j\cdot 2m+2m-1)})$
   to obtain $m \cdot f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m - 1)$.
3. Twist $m \cdot f(\omega^{\sigma(2j+1)/m} \cdot x) \bmod (x^m - 1)$ by $\omega^{-\sigma(1)/m}$
   to obtain $f_Z = m \cdot f(\omega^{\sigma(2j)/m} \cdot x) \bmod (x^m + 1)$.
4. Compute $f_A = f_Y - f_Z$.
5. Compute $f_B = f_Y + f_Z$.
6. Return $(2m) \cdot f(\omega^{\sigma(j)/(2m)} \cdot x) \bmod (x^{2m} - 1) = f_A \cdot x^m + f_B$.

Figure 4.2 Pseudocode for twisted radix-2 IFFT

each interpolation step. As a result, the result of each interpolation step has been multiplied by $2m$.

The algorithm is initialized with $f(\omega^{\sigma(j)}) = f(\omega^{\sigma(j)} \cdot x) \bmod (x - 1)$ for all j in the range $0 \leq j < n$. By recursively applying the interpolation step to these results, we obtain $n \cdot f(x) \bmod (x^n - 1) = n \cdot f(x)$ if f has degree less than n. By multiplying each coefficient of this result by $1/n$, the desired polynomial f is recovered. Pseudocode for this IFFT algorithm is given in Figure 4.2.

Let us now analyze the cost of this algorithm. Line 0 is just used to end the recursion and costs no operations. The cost of lines 1 and 2 is equal to the number of operations needed to compute two IFFTs of size m. The cost of the twisting operation in line 3 is $m - 1$ multiplications in R. In line 4, we add f_A to f_B at a cost of m additions and in line 5, we subtract f_A from f_B at a cost of m subtractions. Line 6 just involves logicially joining these two results, requiring no operations.

The total number of operations to compute the twisted radix-2 IFFT of size n is

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{n}{2} - 1, \tag{4.9}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + n, \tag{4.10}$$

where $M(1) = 0$ and $A(1) = 0$. The solution to the recurrence relations is the same as the solution to the twisted radix-2 FFT operation counts and results in the formulas

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{4.11}$$

$$A(n) = n \cdot \log_2(n). \tag{4.12}$$

This algorithm has the exact same operation count as the classical radix-2 IFFT algorithm. To recover the original polynomial, we must multiply the final result of the algorithm by $1/n$ at a cost of n additional multiplications in R. The operation counts of the IFFT algorithms considered so far differ from the companion FFT algorithms only by the n additional multiplication operations needed to undo the scaling.

4.3 Other multiplicative IFFTs

For any other multiplicative FFT algorithm, it is possible to construct an IFFT algorithm by determining the inverse of each line of the FFT algorithm and performing these instructions in reverse order. However, the instruction(s) used to end the recursion (typically the line 0's in each algorithm) will still be done first

in the inverse algorithm. Additionally, operations which do not depend on previous instructions in a particular FFT algorithm do not need to be performed in reverse order in the IFFT algorithm.

There will be a constant multiplicative factor that will be ignored by each interpolation step of the IFFT algorithm. To recover the desired polynomial at the end of the algorithm, an n additional multiplications are needed to multiply the final result by $1/n$ to compensate for these ignored factors. The resulting IFFT algorithm will require the same number of additions and n more multiplications compared to its companion FFT algorithm. This technique works for the radix-3 and radix-p algorithms as well.

As two additional examples of this process, the pseudocode for the conjugate pair version of the split-radix IFFT algorithm will be presented and then the pseudocode for the improved twisted radix-3 IFFT algorithm. These algorithms efficiently compute the inverse FFT of sizes $n = 2^k$ and $n = 3^{\mathbb{k}}$, respectively.

The inverse of the matrix transformation used in the split-radix FFT is

$$\begin{pmatrix} +1 & 0 & +1 & 0 \\ 0 & +1 & 0 & +1 \\ -\mathtt{I} & -1 & +\mathtt{I} & +1 \\ +\mathtt{I} & -1 & -\mathtt{I} & +1 \end{pmatrix}^{-1} = \begin{pmatrix} +\frac{1}{2} & 0 & +\frac{\mathtt{I}}{4} & -\frac{\mathtt{I}}{4} \\ 0 & +\frac{1}{2} & -\frac{1}{4} & -\frac{1}{4} \\ +\frac{1}{2} & 0 & -\frac{\mathtt{I}}{4} & +\frac{\mathtt{I}}{4} \\ 0 & +\frac{1}{2} & +\frac{1}{4} & +\frac{1}{4} \end{pmatrix}. \tag{4.13}$$

In the IFFT algorithm, the intermediate results are not scaled and the factors in front of f_W, f_X, f_Y and f_Z compensates for the denominators in this transformation. Thus, the transformation used in the split-radix IFFT algorithm is

Algorithm : Split-radix IFFT (conjugate-pair version)

Input: The evaluations $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$
of some polynomial with coefficients in a ring R.
Here R has a nth root of unity ω, and m is a power of two where $4m \leq n$.

Output: $(4m) \cdot f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1)$.

0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(j)} \cdot x) \bmod (x - 1) = f(\omega^{\sigma'(j)})$.
0B. If $(4m) = 2$, then call a radix-2 IFFT algorithm to compute the result.
1. Compute the IFFT of $f(\omega^{\sigma'(j\cdot 4m)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+2m-1)})$
to obtain $f_W \cdot x^m + f_X = (2m) \cdot f(\omega^{\sigma'(2j)/(2m)} \cdot x) \bmod (x^{2m} - 1)$.
2. Compute the IFFT of $f(\omega^{\sigma'(j\cdot 4m+2m)}), f(\omega^{\sigma'(j\cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+3m-1)})$
to obtain $m \cdot f(\omega^{\sigma'(4j+2)/m} \cdot x) \bmod (x^m - 1)$.
3. Compute the IFFT of $f(\omega^{\sigma'(j\cdot 4m+3m)}), f(\omega^{\sigma'(j\cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$
to obtain $m \cdot f(\omega^{\sigma'(4j+3)/m} \cdot x) \bmod (x^m - 1)$.
4. Compute $f_Y = m \cdot f(\omega^{\sigma'(4j)/m} \cdot x) \bmod (x^m - \mathtt{I})$
by twisting $m \cdot f(\omega^{\sigma'(4j+2)/m} \cdot x) \bmod (x^m - 1)$ by $\omega^{-\sigma'(2)/m}$.
5. Compute $f_Z = m \cdot f(\omega^{\sigma'(4j)/m} \cdot x) \bmod (x^m + \mathtt{I})$
by twisting $m \cdot f(\omega^{\sigma'(4j+3)/m} \cdot x) \bmod (x^m - 1)$ by $\omega^{\sigma'(2)/m}$.
6. Compute $f_\alpha = \mathtt{I} \cdot (f_Y - f_Z)$.
7. Compute $f_\beta = f_Y + f_Z$.
8. Compute $f_A = f_W + f_\alpha$.
9. Compute $f_B = f_X - f_\beta$.
10. Compute $f_C = f_W - f_\alpha$.
11. Compute $f_D = f_X + f_\beta$.
12. Return $(4m) \cdot f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m} - 1) = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.

Figure 4.3 Pseudocode for split-radix IFFT (conjugate-pair version)

$$\begin{pmatrix} f_A \\ f_B \\ f_C \\ f_D \end{pmatrix} = \begin{pmatrix} +1 & 0 & +\mathtt{I} & -\mathtt{I} \\ 0 & +1 & -1 & -1 \\ +1 & 0 & -\mathtt{I} & +\mathtt{I} \\ 0 & +1 & +1 & +1 \end{pmatrix} \cdot \begin{pmatrix} f_W \\ f_X \\ f_Y \\ f_Z \end{pmatrix} \tag{4.14}$$

and the results are twisted by $\sigma(j)$ for the traditional version of the split-radix algorithm or $\sigma'(j)$ for the conjugate-pair version after these reductions have been completed. Pseudocode that implements the conjugate-pair version of the split-radix IFFT algorithm is given in Figure 4.3. Note that operations which do not depend on previous results in the FFT algorithm are not performed in reverse order in the IFFT algorithm.

The recurrence relations for the split-radix IFFT algorithm are the same as those used for the split-radix FFT algorithm. Thus, the total number of operations to compute the inverse FFT of size n using this algorithm is given by

$$\begin{aligned} M(n) &= \frac{1}{3} \cdot n \cdot \log_2(n) + \frac{1}{9} \cdot n - \frac{1}{9} \cdot (-1)^n + 1 \qquad (4.15) \\ &\leq \frac{1}{3} \cdot n \cdot \log_2(n) + \frac{1}{9} \cdot n + \frac{10}{9}, \\ A(n) &= n \cdot \log_2(n). \qquad (4.16) \end{aligned}$$

As with the other multiplicative IFFT algorithms, an additional n multiplications in R are needed to recover the unscaled version of the output polynomial.

It is possible to add the scaling factors of Johnson and Frigo's modified split-radix FFT algorithm [44] to the above pseudocode and obtain the companion IFFT algorithm. The details of this process will not be given here.

Turning our attention to the twisted radix-3 IFFT, we first provide the pseudocode for this algorithm in Figure 4.4, obtained by following the general procedure for constructing an IFFT algorithm. The inverse of the transformation used in a twisted radix-3 FFT algorithm is given by

$$\begin{pmatrix} 1 & 1 & 1 \\ \Omega^2 & \Omega & 1 \\ \Omega & \Omega^2 & 1 \end{pmatrix}^{-1} = \frac{1}{3} \cdot \begin{pmatrix} 1 & \Omega & \Omega^2 \\ 1 & \Omega^2 & \Omega \\ 1 & 1 & 1 \end{pmatrix}. \tag{4.17}$$

Similar to the 2-adic multiplicative IFFTs, the factors of 1/3 are saved until the end of the algorithm. Thus, the transformation for a twisted radix-3 IFFT algorithm is given by

$$\begin{pmatrix} f_A \\ f_B \\ f_C \end{pmatrix} = \begin{pmatrix} 1 & \Omega & \Omega^2 \\ 1 & \Omega^2 & \Omega \\ 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} f_X \\ f_Y \\ f_Z \end{pmatrix}. \tag{4.18}$$

However, the result of line 2 of the algorithm is $\widetilde{f}_Y$, the polynomial which results from twisting f_Y by $\zeta = \omega^{\Delta'(1)/m}$. Similarly, the result of line 3 of the algorithm is $\widetilde{f}_Z$, the polynomial which results from twisting f_Z by $\overline{\zeta}$. These polynomials need to be "untwisted" before they can be used as input to (4.18). Instead of doing this computation explicitly before the interpolation step, it turns out that it is more efficient to combine the twisting with the interpolation step as described in the pseudocode.

Let us now count the operations required to implement this algorithm. Line 0 simply ends the recursion and costs no operations. The cost of lines 1, 2, and 3 is the number of operations needed to compute three IFFTs of size m. Line 4 is just a

Algorithm : New twisted radix-3 IFFT

Input: The evaluations $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$
of some polynomial with coefficients in a ring R.
Here R has a nth root of unity ω, and m is a power of three where $3m \leq n$.

Output: $(3m) \cdot f(\omega^{\Delta'(j)/(3m)} \cdot x) \bmod (x^{3m} - 1)$.

0. If $(3m) = 1$ then return $f(\omega^{\Delta'(j)} \cdot x) \bmod (x - 1) = f(\omega^{\Delta'(j)})$.
1. Compute the IFFT of $f(\omega^{\Delta'(j\cdot 3m+0)}), f(\omega^{\Delta'(j\cdot 3m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+m-1)})$
   to obtain $f_X = m \cdot f(\omega^{\Delta'(3j)/m} \cdot x) \bmod (x^m - 1)$.
2. Compute the IFFT of $f(\omega^{\Delta'(j\cdot 3m+m)}), f(\omega^{\Delta'(j\cdot 3m+m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+2m-1)})$
   to obtain $\widetilde{f}_Y = m \cdot f(\omega^{\Delta'(3j+1)/m} \cdot x) \bmod (x^m - 1)$.
3. Compute the IFFT of $f(\omega^{\Delta'(j\cdot 3m+2m)}), f(\omega^{\Delta'(j\cdot 3m+2m+1)}), \ldots, f(\omega^{\Delta'(j\cdot 3m+3m-1)})$
   to obtain $\widetilde{f}_Z = m \cdot f(\omega^{\Delta'(3j+2)/m} \cdot x) \bmod (x^m - 1)$.
4. Let $\zeta = \omega^{\Delta'(1)/m}$.
5. Compute $(f_\gamma)_d = \zeta^{-d} \cdot (\widetilde{f}_Y)_d + \overline{\zeta^{-d}} \cdot (\widetilde{f}_Z)_d$ for all d in $0 \leq d < m$.
   Combine the $(f_\gamma)_d$'s to obtain $f_\gamma = f_Y + f_Z$.
6. Compute $(f_\beta)_d = \Omega^2 \cdot \zeta^{-d} \cdot (\widetilde{f}_Y)_d + \overline{\Omega^2 \cdot \zeta^{-d}} \cdot (\widetilde{f}_Z)_d$ for all d in $0 \leq d < m$.
   Combine the $(f_\beta)_d$'s to obtain $f_\beta = \Omega^2 \cdot f_Y + \Omega \cdot f_Z$.
7. Compute $f_\alpha = f_\beta + f_\gamma = -\Omega \cdot f_Y - \Omega^2 \cdot f_Z$.
8. Compute $f_A = f_X - f_\alpha = f_X + \Omega \cdot f_Y + \Omega^2 \cdot f_Z$.
9. Compute $f_B = f_X + f_\beta = f_X + \Omega^2 \cdot f_Y + \Omega \cdot f_Z$.
10. Compute $f_C = f_X + f_\gamma = f_X + f_Y + f_Z$.
11. Return $(3m) \cdot f(\omega^{\Delta'(j)/(3m)} \cdot x) \bmod (x^{3m} - 1) = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$.

Figure 4.4 Pseudocode for new twisted radix-3 IFFT

table lookup and requires no operations. We will assume that all of the powers of ζ can also be implemented with table lookups. By Theorem 46 in the appendix, line 5 costs $m-1$ multiplications and $2m$ additions. This is the same technique used in the new classical radix-3 algorithm presented in Chapter 3. Note that when $d=0$, no multiplication is necessary in this instruction. Line 6 only costs m multiplications because we can reuse $\widetilde{f}_Y + \widetilde{f}_Z$ and $\widetilde{f}_Y - \widetilde{f}_Z$ which were already computed in line 5. Lines 7-10 each require m additions in R. Line 11 just involves logicially joining these three results, requiring no operations. The total number of operations to compute the unscaled IFFT of size n using this algorithm is given by

$$\begin{aligned} M(n) &= 3 \cdot M\left(\frac{n}{3}\right) + \frac{2}{3} \cdot n - 1, && (4.19) \\ A(n) &= 3 \cdot A\left(\frac{n}{3}\right) + 2 \cdot n, && (4.20) \end{aligned}$$

where $M(1) = 0$ and $A(1) = 0$. Solving these recurrence relations for closed-form formulas gives

$$\begin{aligned} M(n) &= \frac{2}{3} \cdot n \cdot \log_3(n) - \frac{1}{2} \cdot n + \frac{1}{2} && (4.21) \\ &= \frac{2}{3 \cdot \log_2(3)} \cdot n \cdot \log_2(n) - \frac{1}{2} \cdot n + \frac{1}{2} \\ &\approx 0.4206 \cdot n \cdot \log_2(n) - 0.5 \cdot n + 0.5, \\ A(n) &= 2 \cdot n \cdot \log_3(n) && (4.22) \\ &= \frac{2}{\log_2(3)} \cdot n \cdot \log_2(n) \\ &\approx 1.2619 \cdot n \cdot \log_2(n). \end{aligned}$$

As with the other multiplicative IFFT algorithms, an additional n multiplications in R are needed to recover the unscaled version of the output polynomial.

4.4 Wang-Zhu-Cantor additive IFFT

We now turn our attention to finding the companion IFFT for some of the additive FFT algorithms. Each IFFT receives as input $n = 2^k$ evaluations of some unknown function f at each of the elements of some subspace W_k of a finite field $\mathbb{F}$ of characteristic 2. The algorithms in the following sections can be generalized to finite fields of other characteristics if desired. We will assume that Cantor's special basis discussed in the appendix is used for each algorithm as well.

The inverse of the Wang-Zhu-Cantor additive FFT ([12], [81]) interpolation step can be viewed as the transformation

$$\begin{aligned} R[x]/(s_i - \varpi) \quad &\rightarrow \quad R[x]/\left(s_{i+1} - (\varpi^2 + \varpi)\right). \\ \times \; R[x]/\left(s_i - (\varpi + 1)\right) & \end{aligned} \tag{4.23}$$

Here, $s_i(x)$ is the minimal polynomial of $W_i \subseteq W_k$ and ϖ is an element of W_{k-i} that satisfies $\varpi = s_i(\varepsilon)$ for some ε that is a linear combination of $\{\beta_{i+2}, \beta_{i+3}, \cdots, \beta_k\}$ if $i < k-1$ and is zero when $i = k-1$. Then $s_i - \varpi$ is the minimal polynomial of $\varepsilon + W_i \subseteq W_k$. If the special basis is used, then $s_i(\beta_{i+1}) = 1$ for all $i < k$ and $s_i - (\varpi + 1)$ is the minimal polynomial of $\beta_{i+1} + \varepsilon + W_i \subseteq W_k$. Here, $(\varepsilon + W_i) \bigcup (\beta_{i+1} + \varepsilon + W_i) = \varepsilon + W_{i+1} \subseteq W_k$ and $s_{i+1} - (\varpi^2 + \varpi)$ is the minimal polynomial of $\varepsilon + W_{i+1}$.

The Wang-Zhu-Cantor additive FFT reduction step divides $f \bmod (s_{i+1} - (\varpi^2 + \varpi))$ by $(s_i - \varpi)$ to obtain $f \bmod (s_{i+1} - (\varpi^2 + \varpi)) = q \cdot (s_i - \varpi) + r$. Then

$$f \bmod (s_i - \varpi) = r, \tag{4.24}$$

$$f \bmod (s_i - (\varpi + 1)) = q + r. \tag{4.25}$$

Suppose that we wish to interpolate $f \bmod (s_i - \varpi)$ and $f \bmod (s_i - (\varpi + 1))$ into $f \bmod (s_{i+1} - (\varpi^2 + \varpi))$. Solving (4.24) and (4.25) for q and r, [2] we obtain

$$q = f \bmod (s_i - (\varpi + 1)) - f \bmod (s_i - \varpi), \tag{4.26}$$

$$r = f \bmod (s_i - \varpi). \tag{4.27}$$

Then $f \bmod (s_{i+1} - (\varpi^2 + \varpi))$ is computed using

$$f \bmod \big(s_{i+1} - (\varpi^2 + \varpi)\big) = q \cdot (s_i - \varpi) + r \tag{4.28}$$

and the interpolation step is completed.

Note that each interpolation step actually produces $f \bmod (s_{i+1} - (\varpi^2 + \varpi))$ instead of a scaled version of this result. So unlike the multiplicative IFFT algorithms, there is no need to multiply the final result of this additive IFFT algorithm by $1/n$.

---

[2] The Chinese Remainder Theorem can also be applied here, but simply solving the system of equations for q and r is somewhat easier.

Algorithm : Wang-Zhu-Cantor additive IFFT
Input: The evaluations $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$ of some polynomial f with coefficients in a finite field $\mathbb{F}$ with $n = 2^k$ elements. Here, $m = 2^i$ and $2m \leq n$.
Output: $f \bmod (s_{i+1} - \varpi_j)$.
0. If $(2m) = 1$, then return $f \bmod (x - \varpi_j) = f(\varpi_j)$. 1. Compute the IFFT of $f(\varpi_{j\cdot 2m+0}), f(\varpi_{j\cdot 2m+1}), \ldots, f(\varpi_{j\cdot 2m+m-1})$ to obtain $r = f \bmod (s_i - \varpi_{2j})$. 2. Compute the IFFT of $f(\varpi_{j\cdot 2m+m}), f(\varpi_{j\cdot 2m+m+1}), \ldots, f(\varpi_{j\cdot 2m+2m-1})$ to obtain $f \bmod (s_i - \varpi_{2j+1})$. 3. Compute $q = f \bmod (s_i - \varpi_{2j+1}) - f \bmod (s_i - \varpi_{2j})$. 4. Return $f \bmod (s_{i+1} - \varpi_j) = q \cdot (s_i - \varpi_{2j}) + r$.

Figure 4.5 Pseudocode for Wang-Zhu-Cantor additive IFFT

Like the companion additive FFT algorithm, the ϖ's are stored in an array with the properties that $\varpi_{2j+1} = 1 + \varpi_{2j}$ and $\varpi_j = {\varpi_{2j}}^2 + \varpi_{2j}$ for all $j < n/2$. One should substitute $\varpi = \varpi_{2j}$ into the interpolation step to use this array for the calculations. Pseudocode for Wang-Zhu-Cantor additive IFFT algorithm is given in Figure 4.5.

Let us compute the cost of this algorithm. Line 0 is used to end the recursion and does not cost any operations. The cost of lines 1 and 2 is equal to the number of operations needed to compute two IFFTs of size m. In line 3, we need to perform m subtractions to obtain q. Finally, the computation in line 4 requires m multiplications and $(c_i + 1) \cdot (m)$ additions in $\mathbb{F}$ where c_i is the number of nonzero coefficients in s_i. However, if $j = 0$, then no multiplications and only $c_i \cdot m$ additions are required to complete this instruction.

If $j \neq 0$, then the total number of operations to compute this IFFT of size n is

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n, \tag{4.29}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot c_{\log_2(n)-1} \cdot n + n, \tag{4.30}$$

where $M(1) = 0$ and $A(1) = 0$.

We must also subtract operations saved for the case when $j = 0$. The solution of the recurrence relations proceeds identically to those given for Cantor's additive FFT algorithm and results in operation counts of

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{4.31}$$

$$\begin{aligned} A(n) &= \frac{1}{2} \cdot C_{\log_2(n)} \cdot n + n \cdot \log_2(n) - n + 1 \\ &\leq \frac{1}{2} \cdot n \cdot \log_2(n)^{1.585} + n \cdot \log_2(n) - n + 1. \end{aligned} \tag{4.32}$$

No scaling is involved in this inverse FFT algorithm and the number of operations required is identical to those required to implement the companion FFT algorithm.

4.5 A new additive IFFT

Finally, we will construct the companion IFFT algorithm for the new additive FFT algorithm introduced in Chapter 3. Let $\mathbb{F}$ be a finite field of characteristic 2 of size $N = 2^K$ and let $\{\beta_1, \beta_2, \ldots \beta_K\}$ once again be the special basis introduced by Cantor. As usual, define subspace W_i of $\mathbb{F}$ to be all linear combinations of $\{\beta_1, \beta_2, \ldots, \beta_i\}$ for all $i \leq K$. We may enumerate the 2^k elements of W_k by

$$\varpi_j = b_{k-1} \cdot \beta_k + b_{k-2} \cdot \beta_{k-1} + \cdots + b_1 \cdot \beta_2 + b_0 \cdot \beta_1 \tag{4.33}$$

where $k \leq K$. Here, $0 \leq j < 2^k$ and the b's are given by the binary representation of j, i.e. $j = (b_{k-1}b_{k-2}\cdots b_1 b_0)_2$.

The algorithm discussed in this section computes the IFFT over the points of any $\varpi_{nJ} + W_k \subseteq \mathbb{F}$ where $k \leq K$, $n = 2^k$, and $J < 2^{K-k}$. That is to say, we are given n evaluations of some unknown polynomial f at $\{\varpi_{J\cdot n+0}, \varpi_{J\cdot n+1}, \ldots \varpi_{J\cdot n+n-1}\}$ and we wish to interpolate these evaluations back into f.

As with the companion algorithm, the interpolation step of the IFFT algorithm is based on the factorization

$$x^\eta - x - (a^\tau + a) \;=\; \prod_{\delta \in (a+W_t)} (x^\tau - x - \delta). \tag{4.34}$$

Here, $\eta = \tau^2$, $t = \log_2(\tau)$, and a is a linear combination of the basis elements $\{\beta_{t+1}, \beta_{t+2}, \ldots, \beta_{2t}\}$. The interpolation step can be interpreted as the transformation given by

$$\begin{aligned} R[x]/(x^\tau - x - \delta_0) \;&\rightarrow\; R[x]/(x^\eta - x - (a^\tau + a)), \\ \times\; R[x]/(x^\tau - x - \delta_1) & \\ \times\; R[x]/(x^\tau - x - \delta_2) & \\ \cdots & \\ \times\; R[x]/(x^\tau - x - \delta_{\tau-1}) & \end{aligned} \tag{4.35}$$

where $\delta_0, \delta_1, \delta_2, \ldots, \delta_{\tau-1}$ are the elements of $a + W_t$.

The input to the interpolation step is the collection of polynomials $f \bmod (x^\tau - x - \delta_0)$, $f \bmod (x^\tau - x - \delta_1), \cdots, f \bmod (x^\tau - x - \delta_{\tau-1})$ and the desired output is $f^\circ = f \bmod (x^\eta - x - (a^\tau + a))$. An intermediate result of the interpolation step is the computation of the Taylor expansion of f° at x^τ. That is, we will find "coefficients" $\{g_0(x), g_1(x), \ldots, g_{\tau-2}(x), g_{\tau-1}(x)\}$ such that

$$f^\circ = g_{\tau-1} \cdot (x^\tau - x)^{\tau-1} + g_{\tau-2} \cdot (x^\tau - x)^{\tau-2} + \cdots + g_1 \cdot (x^\tau - x) + g_0. \tag{4.36}$$

Here, each "coefficient" is itself a polynomial of degree less than τ in x. The Taylor expansion is equivalent to

$$g(y) = g_{\tau-1} \cdot x^{\tau-1} + g_{\tau-2} \cdot x^{\tau-2} + \cdots + g_1 \cdot x + g_0. \tag{4.37}$$

Since $x^\tau - x \bmod (x^\tau - x - \delta) = \delta$ for any $\delta \in a + W_t$, then the input to the interpolation step can also be viewed as the set of evaluations of $g(y)$ at every $\delta \in a + W_t$.

To find the Taylor expansion, we can compute τ IFFT's of size τ over $a + W_t$. As with the companion FFT algorithm, define $h_\lambda(x)$ as the polynomial

$$h_\lambda(x) = (g_{\tau-1})_\lambda \cdot x^{\tau-1} + (g_{\tau-2})_\lambda \cdot x^{\tau-2} + \cdots + (g_{\tau-1})_\lambda \cdot x + (g_0)_\lambda \tag{4.38}$$

formed by extracting the coefficients of degree λ from $g_0(x), g_1(x), \ldots, g_{\tau-1}(x)$. The coefficient of degree d in $h_\lambda(x)$ is given by the coefficient of degree λ in $g_d(x)$. One

way to conceptualize the construction of these polynomials is to write the coefficients of the g_d's across the rows of a $\tau \times \tau$ matrix

$$\begin{pmatrix} (g_0)_0 & (g_0)_0 & \dots & (g_0)_{\tau-1} \\ (g_1)_0 & (g_1)_1 & \dots & (g_1)_{\tau-1} \\ \vdots & \vdots & \ddots & \vdots \\ (g_{\tau-1})_0 & (g_{\tau-1})_1 & \dots & (g_{\tau-1})_{\tau-1} \end{pmatrix}. \tag{4.39}$$

The coefficients of the polynomial h_λ are determined by reading down column $\lambda + 1$ of this matrix. Since the known evaluations of $g(y)$ also give us the evaluation of each $h_\lambda(x)$ at every $\delta \in a + W_t$, then we can compute τ IFFTs of size τ to obtain $h_\lambda(x)$ for all $0 \leq \lambda < \tau$. The Taylor expansion $g(y)$ can be formed from these results.

An algorithm discussed in the appendix which computes the Taylor expansion of a polynomial at x^τ can be reversed to obtain a new algorithm which can transform $g(y)$ into f°. Pseudocode for the inverse Taylor expansion algorithm is not given in this manuscript, but requires the same number of operations as the Taylor expansion algorithm given in the appendix.

So the interpolation step amounts to computing τ IFFTs of size τ followed by one inverse Taylor expansion of a polynomial at x^τ. Either the above interpolation step can be used recursively or the Wang-Zhu-Cantor IFFT can be used to compute the IFFTs. If k is a power of two, then the Wang-Zhu-Cantor algorithm will be needed to resolve the IFFTs of size 2.

As with the companion FFT algorithm, Cantor's special basis simplifies the interpolation step of the new additive IFFT algorithm. Recall that for any $j < \tau$ where $\tau \leq \sqrt{n}$, then $\varpi_j = (\varpi_{j \cdot \tau})^\tau + \varpi_{j \cdot \tau}$. Also recall that if $c_1 < \tau$ and $c_2 \geq \tau$, then $\varpi_{c_1} + \varpi_{c_2} = \varpi_{c_1 + c_2}$. Using these results, the interpolation step can be expressed by

$$
\begin{aligned}
R[x]/(x^{\tau} - x - \varpi_{j\cdot\tau}) \quad &\rightarrow \quad R[x]/(x^{\eta} - x - \varpi_j). \qquad (4.40) \\
\times \; R[x]/(x^{\tau} - x - \varpi_{j\cdot\tau+1}) \\
\times \; R[x]/(x^{\tau} - x - \varpi_{j\cdot\tau+2}) \\
\cdots \\
\times \; R[x]/(x^{\tau} - x - \varpi_{j\cdot\tau+\tau-1})
\end{aligned}
$$

Here, $\{\varpi_{j\cdot\tau}, \varpi_{j\cdot\tau+1}, \varpi_{j\cdot\tau+2}, \ldots, \varpi_{j\cdot\tau+\tau-1}\} = \varpi_j + W_t$.

The algorithm will be initialized with $f \bmod (x - \varpi_{n\cdot J+d}) = f(\varpi_{n\cdot J+d})$ for every d in $0 \leq d < n$ for some $J < 2^{K-k}$. Assuming that k is a power of two, then one stage of interpolation steps from the Wang-Zhu-Cantor IFFT algorithm will be needed before we can start to use the interpolation step presented in this section. The interpolation step will receive the polynomials $f \bmod (x^{\tau} - x - \varpi_{\phi})$ for each ϕ in $j \cdot \tau \leq \phi < (j+1) \cdot \tau$ where $\tau = 2^t$ for some t and $j < t$. We will recursively call the IFFT algorithm τ times and then apply an inverse Taylor expansion of the result at x^{τ} to combine these results into $f \bmod (x^{\eta} - x - \varpi_j)$ where $\eta = \tau^2$. After all of the interpolation steps have been completed, we will obtain $f \bmod (x^n - x - \varpi_J)$. This final result equals f if f has degree less than n.

If K is not a power of two, then it will not be possible to constuct the special basis and the companion IFFT algorithm for the von zur Gathen-Gerhard additive FFT algorithm must be used instead. This algorithm is not discussed in this manuscript, but can be constructed using a technique similar to the other additive IFFT algorithms.

If k is not itself a power of two, then let κ be the largest power of two such that $\kappa \leq k$. The improved algorithm can used to compute $f \bmod (x^{\kappa} - x - \varpi_j)$ for all

Algorithm : New additive IFFT

Input: The evaluations $f(\varpi_{j\cdot\eta+0}), f(\varpi_{j\cdot\eta+1}), \ldots, f(\varpi_{j\cdot\eta+\eta-1})$
of some polynomial f with coefficients in a finite field $\mathbb{F}$
with $n = 2^k$ elements. Here, η is a power of two.

Output: $f \bmod (x^\eta - x - \varpi_j)$.

0. If $\eta = 2$, then return $(f(\varpi_{2j+1}) - f(\varpi_{2j})) \cdot (x - \varpi_{2j}) + f(\varpi_{2j})$.
1. Set $\tau = \sqrt{\eta}$.
2. **for** $\phi = j \cdot \tau$ to $j \cdot \tau + \tau - 1$ **do**
3. Recursively call the IFFT algorithm with input
$f(\varpi_{\phi\cdot\tau}), f(\varpi_{\phi\cdot\tau+1}), \cdots, f(\varpi_{\phi\cdot\tau+\tau-1})$
to obtain $f \bmod (x^\tau - x - \varpi_\phi)$.
4. **end for** (Loop ϕ)
5. Assign the coefficient of x^λ from $f \bmod (x^\tau - x - \varpi_\phi)$ to $h_\lambda(\varpi_\phi)$
for each ϕ in $j \cdot \tau \leq \phi < (j+1) \cdot \tau$ and each λ in $0 \leq \lambda \leq \tau - 1$.
6. **for** $\lambda = 0$ to $\tau - 1$ **do**
7. Recursively call the new IFFT algorithm with input
$h_\lambda(\varpi_{j\cdot\tau}), h_\lambda(\varpi_{j\cdot\tau+1}), \cdots, h_\lambda(\varpi_{j\cdot\tau+\tau-1})$
to obtain $h_\lambda(x) \bmod (x^\tau - x - \varpi_j) = h_\lambda(x)$.
8. **end for** (Loop λ)
9. Construct $g(y)$ using the coefficients of $h_\lambda(x)$ for each λ in $0 \leq \lambda < \tau$.
10. Recover $f \bmod (x^\eta - x - \varpi_j)$ by computing the inverse Taylor expansion
of $g(y)$ at x^τ.
11. Return $f \bmod (x^\eta - x - \varpi_j)$.

Figure 4.6 Pseudocode for the new additive IFFT

j in $0 \leq j < 2^{k-\kappa}$. Interpolation steps from the Wang-Zhu-Cantor IFFT algorithm can be used to complete the recovery of f. For the sake of simplicity, we will assume that k is a power of two in the pseudocode of the new IFFT algorithm given in Figure 4.6.

We now analyze the cost of this algorithm for the case where where κ is a power of two and $\eta = 2^\kappa$. Line 0 ends the recursion by calling the Wang-Zhu-Cantor algorithm with input size 2 at a cost of $M(2) = 1$ and $A(2) = 2$. We will assume that line 1 costs no operations as the work to complete this instruction is insignificant

compared to the rest of the algorithm. Lines 2-4 involve $\sqrt{\eta}$ recursive calls to the IFFT algorithm, each at a cost of $M(\sqrt{\eta})$ multiplications and $A(\sqrt{\eta})$ additions. In theory, line 5 costs no operations. However, in practice it may be necessary to rearrange the coeffficients of the polynomials determined in lines 2-4 so that each of the evaluations of h_λ are adjacent to each other in memory. This costs a total of η copies. As with the FFT algorithm, we will only assume that this is necessary when $\eta \geq 2^{32}$. Lines 6-8 involve another $\sqrt{\eta}$ recursive calls to the IFFT algorithm, each at a cost of $M(\sqrt{\eta})$ multiplications and $A(\sqrt{\eta})$ additions. In line 9, the elements of the h_λ's need to be reorganized into $g(y)$. In theory, this costs no operations, but in practice this will cost an additional τ copies when $\eta \geq 2^{32}$. Finally, in line 10, we need to perform one inverse Taylor expansion of $g(y)$ at x^τ at a cost of no multiplications and $1/4 \cdot \eta \cdot \log_2(\eta)$ additions. Line 11 costs no operations.

The total number of operations to compute this IFFT of size n using the new algorithm is

$$M(n) = 2 \cdot \sqrt{n} \cdot M(\sqrt{n}), \tag{4.41}$$

$$A(n) = 2 \cdot \sqrt{n} \cdot A(\sqrt{n}) + \frac{1}{4} \cdot n \cdot \log_2(n), \tag{4.42}$$

where $M(2) = 1$ and $A(2) = 2$. We must also subtract operations of both types to account for the cases where $j = 0$. The solution of these recurrence relations is the same as those used for the new additive FFT algorithm and results in operation counts of

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{4.43}$$

$$A(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + n \cdot \log_2(n) - n + 1, \tag{4.44}$$

when $n < 2^{32}$. If $n \geq 2^{32}$ then the addition count increases to

$$A(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + \frac{9}{8} \cdot n \cdot \log_2(n) - 3 \cdot n + 1 \tag{4.45}$$

to account for the copies required in lines 5 and 9.

If k is not a power of two, then let κ be the largest power of 2 such that $\kappa \leq k$. In this case, the number of operations when $n < 2^{32}$ is

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n) - n + 1, \tag{4.46}$$

$$A(n) = \frac{1}{2} \cdot n \cdot \log_2(n)^{1.585} + n \cdot \log(n) - n + 1 - \frac{1}{2} \cdot n \cdot \kappa^{1.585} + \frac{1}{4} \cdot n \cdot \kappa \cdot \log_2(\kappa), \tag{4.47}$$

and an additional $\frac{1}{8} \cdot n \cdot \kappa - 2n$ copies are needed for $n \geq 2^{32}$.

In any case, the new additive IFFT algorithm requires the same number of operations as the new additive FFT algorithm.

4.6 Concluding remarks

In this chapter, we examined a variety of IFFT algorithms. In the case of the multiplicative IFFT algorithms, the number of operations required was the number of

operations of the companion FFT algorithm, plus n additional multiplications. In the case of the additive IFFT algorithms, the number of operations required was exactly the same as the number of operations of the companion FFT algorithm.

CHAPTER 5

POLYNOMIAL MULTIPLICATION ALGORITHMS

5.1 Karatsuba multiplication

The classical algorithm for multiplying two polynomials of degree less than n into a product of degree less than $2n-1$ requires $\Theta(n^2)$ operations or is said to have a quadratic operation count with respect to the input polynomial size. This is the method learned in a typical high school introductory algebra course [5] and will not be discussed in any more detail here.

The first subquadratic multiplication algorithm was proposed by Karatsuba in the 1960's [46]. [1] This technique is based on the observation that multiplication is often a more expensive operation than addition in the coefficient ring of the polynomials. For example, in the ring of complex numbers $\mathbb{C}$, addition requires two arithmetic operations, whereas multiplication requires six operations. Karatsuba's algorithm is based on a technique that allows one to reduce the number of multiplications at the expense of extra additions. By recursively applying the basic idea, the overall result is a faster algorithm.

If $(f_1 \cdot x + f_0) \cdot (g_1 \cdot x + g_0) = f_1 g_1 \cdot x^2 + (f_1 g_0 + f_0 g_1) \cdot x + f_0 g_0$ is multiplied using classical multiplication, a total of 4 multiplications is required. Karatsuba's algorithm is based on the following alternative method of computing the product polynomial.

[1] The technique is sometimes called Karatsuba-Ofman multiplication because both of these individuals were authors of the paper which first introduced the algorithm. It appears [47] that Ofman only helped to write the paper and that Karatsuba was the sole inventor of this technique.

$$
\begin{aligned}
& (f_1 \cdot x + f_0) \cdot (g_1 \cdot x + g_0) \qquad (5.1)\\
= \;& f_1 g_1 \cdot x^2 + ((f_0 + f_1) \cdot (g_0 + g_1) - f_0 g_0 - f_1 g_1) \cdot x + f_0 g_0.
\end{aligned}
$$

Thus, the computation of the product polynomial requires 4 additions / subtractions, but only 3 multiplications.

Karatsuba's algorithm uses this idea recursively to further reduce the number of multiplications. Let $f(x)$ and $g(x)$ be polynomials of degree over any ring R of degree less than $2m$ where m is a power of two. By splitting f and g into two blocks of size m, then these polynomials can be written as

$$
\begin{aligned}
f &= f_A \cdot x^m + f_B, \qquad (5.2)\\
g &= g_A \cdot x^m + g_B. \qquad (5.3)
\end{aligned}
$$

Observe that f_A, f_B, g_A and g_B are each polynomials of degree less than m. The polynomials f and g can be multiplied into the product polynomial h using

$$
\begin{aligned}
h &= f \cdot g \qquad (5.4)\\
&= (f_A \cdot x^m + f_B) \cdot (g_A \cdot x^m + g_B)\\
&= f_A \cdot g_A \cdot x^{2m} + (f_A \cdot g_B + f_B \cdot g_A) \cdot x^m + f_B \cdot g_B.
\end{aligned}
$$

Observe that the middle term can be computed using

$$f_A \cdot g_B + f_B \cdot g_A \;=\; (f_A + f_B) \cdot (g_A + g_B) - f_A \cdot g_A - f_B \cdot g_B. \tag{5.5}$$

Thus, h can also be expressed as

$$\begin{aligned} h &= f_A \cdot g_A \cdot x^{2m} + (f_A \cdot g_B + f_B \cdot g_A) \cdot x^m + f_B \cdot g_B \qquad (5.6) \\ &= f_A \cdot g_A \cdot x^{2m} + ((f_A + f_B) \cdot (g_A + g_B) - f_A \cdot g_A - f_B \cdot g_B) \cdot x^m + f_B \cdot g_B. \end{aligned}$$

Computation of h now involves four additions / subtractions of polynomials of size m and three multiplications of polynomials of size m. The additions (subtractions) can be completed by pointwise adding (subtracting) the elements of the two polynomials. The multiplications are completed by recursively applying this technique to two polynomials of size m. When $(2m) = 2$, the computation is equivalent to (5.1). When $(2m) = 1$, then the multiplication is simply a pointwise product and no more recursion is necessary. Pseudocode for Karatsuba's algorithm is given in Figure 5.1.

Let $M(4m-1)$ be the number of multiplications in R required to compute the product of two polynomials with degree less than $2m$ and let $A(4m-1)$ be the number of additions in R required. If $(2m) = 1$, then $M(1) = 1$ and $A(1) = 0$.

If $(2m) > 1$, then h is computed using lines 2-8. Lines 2 and 3 simply consist of relabeling existing elements and are not assumed to require any operations. Lines 4, 5, and 8 each require $M(2m-1)$ multiplications and $A(2m-1)$ additions. Lines 6 and 7 are each additions of two polynomials of size m. Each polynomial addition costs no multiplications and m additions. In line 9, two polynomial subtractions need to be completed. Each of these computations requires no multiplications and $2m-1$

Algorithm : Karatsuba multiplication
Input: Polynomials f and g of degree less than $2m$ in any ring R where m is a power of two.
Output: Polynomial h of degree less than $4m-1$ in R.
1. If $(2m) = 1$, then return $h = f_0 \cdot g_0$. 2. Split f into two blocks f_A and f_B, each of size m, such that $f = f_A \cdot x^m + f_B$. 3. Split g into two blocks g_A and g_B, each of size m, such that $g = g_A \cdot x^m + g_B$. 4. Compute $f_A \cdot g_A$ by recursively calling Karatsuba multiplication. 5. Compute $f_B \cdot g_B$ by recursively calling Karatsuba multiplication. 6. Compute $f_A + f_B$. 7. Compute $g_A + g_B$. 8. Compute $(f_A + f_B) \cdot (g_A + g_B)$ by recursively calling Karatsuba multiplication. 9. Return $h = (f_A \cdot g_A) \cdot x^{2m} + ((f_A + f_B) \cdot (g_A + g_B) - f_A \cdot g_A - f_B \cdot g_B) \cdot x^m + (f_B \cdot g_B)$.

Figure 5.1 Pseudocode for Karatsuba multiplication

subtractions. Additionally, $2m-2$ elements of the middle term overlap with those of the first and last terms. These elements need to be combined as well.

Combining these results, the total number of operations needed to compute a product of degree less than $2n-1$ is given by

$$M(2n-1) = 3 \cdot M\left(2 \cdot \frac{n}{2} - 1\right), \tag{5.7}$$

$$A(2n-1) = 3 \cdot A\left(2 \cdot \frac{n}{2} - 1\right) + 4n - 4, \tag{5.8}$$

where $M(1) = 0$ and $A(1) = 0$. In a section of the appendix, these recurrences are solved for the closed-form formulas

$$M(2n-1) = n^{\log_2(3)} \approx n^{1.585}, \tag{5.9}$$

$$A(2n-1) = 6 \cdot n^{\log_2(3)} - 8n + 2 \approx 6 \cdot n^{1.585} - 8n + 2. \tag{5.10}$$

As reported in [82], the operation count provided in this section is based on work found in C. Paar's doctoral dissertation and is slightly lower than what is found in other sources. The improvement in the count is due to the fact that one of the elements in the middle term of line 8 does not need to be added to an element from the first term or last term of this expression. This is an improvement that can be implemented in practice as well as a theoretical observation.

5.2 Karatsuba's algorithm for other sizes

If n is not a power of two, one can "pad" the input polynomials with zeros so that these input polynomials are the required size to use the algorithm discussed in the previous section. However, this increases the amount of effort needed to compute the polynomial product.

Several papers ([58], [82]) have been written on computing Karatsuba's algorithm for other sizes. Of particular interest in this chapter will be the case where n is a power of 3 and a product of degree less than $2n-1$ is desired. The paper [82] describes how to construct a version of Karatsuba's algorithm that can compute a product of size $2 \cdot 6 - 1$ using 18 multiplications and 59 additions. The algorithm can also be used to compute a product of size $2 \cdot 18 - 1$ using 108 multiplications and 485 additions, and a product of size $2 \cdot 54 - 1$ using 648 multiplications and 3329 additions. Montgomery [58] reduced the number of multiplications to 102 and 646 for the two cases respectively, while preserving the number of additions. The

multiplication method to be discussed later in this chapter requires fewer operations than Karatsuba's algorithm when $n = 3^4$ or higher, so there is no need to present any additional results of these papers here.

5.3 FFT-based multiplication

Suppose that we have two polynomials $f(x)$ and $g(x)$ of degree less than n with coefficients over some ring R and we wish to compute $h = f \cdot g$. Another method of computing the product first evaluates f and g at n points in R using an FFT algorithm. Then these evaluations are multiplied pointwise using the fact that

$$h(\varepsilon) = f(\varepsilon) \cdot g(\varepsilon) \tag{5.11}$$

for any $\varepsilon \in R$. Finally, the n pointwise evaluations of h are interpolated into h using an IFFT algorithm.

If the degree of f and the degree of g sum to less than n, then the result of this process will be the desired product polynomial. Otherwise, the result will be h modulo a polynomial of degree n which is the minimal polynomial of the n points which were evaluated. If this is not desired, then n should be chosen to be a bigger value.

In order to multiply two polynomials based on a multiplicative FFT where $p = 2$, R must (1) possess some primitive nth root of unity ω where $n = 2^k$ and (2) n must be invertible in R. Unfortunately, a finite field $\mathbb{F}$ of characteristic 2 fails to meet both of these conditions. Cantor [13] provided an alternative algorithm to overcome the second requirement in general, but $\mathbb{F}$ still does not contain the required root of unity in order to use the multiplicative FFT algorithms. This is due in part to the

Algorithm : FFT-based multiplication

Input: Polynomials f and g of degree less than n in a ring R;
a set of points $S = \{\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_{n-1}\}$ in R that supports
some type of FFT algorithm.

Output: Polynomial h of degree less than n in R.
If $\deg(f) + \deg(g) < n$, then $h = f \cdot g$; otherwise $h = f \cdot g \bmod \mathcal{M}$
where $\mathcal{M}$ is the minimal polynomial of the points in S.

1. Evaluate f at each of the points in S using an FFT algorithm.
2. Evaluate g at each of the points in S using an FFT algorithm.
3. **for** $i = 0$ to $n - 1$ **do**
4. Compute $h(\varepsilon_i) = f(\varepsilon_i) \cdot g(\varepsilon_i)$.
5. **end for** (Loop i)
6. Interpolate $h(\varepsilon_0), h(\varepsilon_1), \ldots, h(\varepsilon_{n-1})$ into h using an IFFT algorithm.
7. Return $h = f \cdot g \bmod \mathcal{M}$.

Figure 5.2 Pseudocode for FFT-based multiplication

property that $-1 = 1$ in a finite field of characteristic 2. Instead, one of the additive FFT algorithms discussed in Chapter 3 must be used.

Pseudocode for FFT-based multiplication is given in Figure 5.2. Most of the work required of this algorithm is contained in the two evaluations (lines 1 and 2) and the interpolation (line 6). The time required to perform these operations depends on the structure of S. Any set S of points can be evaluated or interpolated in $\mathcal{O}(n^{1.585} \cdot \log_2(n))$ operations using the techniques discussed in Chapters 1 and 10. Thus, the cost of the multiplication algorithm in the pseudocode is also $\mathcal{O}(n^{1.585} \cdot \log_2(n))$. For large n, this somewhat improves the cost to compute a polynomial product compared to the classical method which requires $\Theta(n^2)$ operations. The polynomial product can be computed even more efficiently if S supports a multiplicative or additive FFT algorithm. Refer to Chapters 2 and 3 to determine the operations needed to complete these evaluations using the FFT and Chapter 4 to determine the operations needed for the required interpolation using the companion IFFT algorithm.

The remaining work in this algorithm is contained in the "pointwise multiplies" completed in lines 3-5 of the algorithm. In line 4, one pointwise multiply costs no additions and 1 multiplication. Since n pointwise multiplies are completed in the loop found in lines 3-5, then a total of n multiplications are needed.

We will say that the cost of computing the polynomial product is given by $3 \cdot M_F(n) + n$ multiplications [2] and $3 \cdot A_F(n)$ additions where $M_F(n)$ is the number of multiplications needed to compute one FFT of size n in R and $A_F(n)$ is the number of additions required.

We will further explore FFT-based multiplication using the new additive FFT algorithm in a later section.

5.4 Schönhage's algorithm

In 1971, Schönhage and Strassen presented an algorithm [70] that allows FFT-based multiplication to work over coefficient rings that do not possess a nth root of unity, but where 2 is a unit in the ring. The rational numbers $\mathbb{Q}$ is an example of such a coefficient ring. The algorithm transforms polynomials with coefficients in this ring to polynomials with coefficients over a quotient ring that contains the required root of unity. For example, polynomials with rational coefficients, $\mathbb{Q}[x]$, could be transformed to a polynomial with coefficients in the quotient ring $\mathbb{Q}[x]/(x^n + 1)$. In this case, $\omega = x$ is a $(2n)$th root of unity in $\mathbb{Q}[x]/(x^n + 1)$. We will not present the details of Schönhage and Strassen's algorithm here. The interested reader that is not fluent in German can refer to [34] for an explanation of how this algorithm works.

In 1977, Schönhage presented [68] a radix-3 variant of this algorithm. Here, the input polynomials are transformed to polynomials in a coefficient ring where there

[2] If an additive FFT is used, then the multiplication count is reduced to $3 \cdot M_F(n)$ since the output of the IFFT does not need to be scaled.

exists a primitive nth root of unity where $n = 3^k$. The algorithm then makes use of the radix-3 FFT and IFFT algorithms to perform the computations. In this section, we show how to use this algorithm to multiply polynomials with coefficients from a finite field $\mathbb{F}$ of characteristic 2.

Let f and g be polynomials of degree less than $n = 3^k$ for some positive integer $k > 1$ with coefficients that are elements of $\mathbb{F}$. The original presentation of Schönhage's algorithm transformed $f, g \in \mathbb{F}[x]$ to polynomials with coefficients over the quotient ring $D = \mathbb{F}[x]/(x^{3m} - 1)$ where $m = 3^{\lceil k/2 \rceil}$. A more recent version of the algorithm as described in [34] works over the quotient ring $D = \mathbb{F}[x]/(x^{2m} + x^m + 1)$ instead and yields a lower operation count. Let $t = n/m$. If $t = m$, then let $\omega = x$ and observe that $\omega^{2t} = x^m + 1$ and $\omega^{3t} = x^{2m} + x^m \bmod (x^{2m} + x^m + 1) = 1$. Otherwise, $t = m/3$ and we will let $\omega = x^3$. In this case, $\omega^{3t} = (x^3)^{3t} = (x^3)^m = x^{3m} = 1$. In both cases, $x^t \neq 1$, so ω is a primitive $(3t)$th root of unity.

To transform a polynomial $f \in \mathbb{F}[x]$ to a new polynomial f^* with coefficients in D, let us group the terms of f into blocks of size m where zero coefficients should be included in each block and factor out the largest multiple of x^m from each block. Then f^* is a polynomial in x^m with coefficients in D. To make it easier to distinguish between the elements of D and the powers of x^m, we will relabel the polynomial in terms of $y = x^m$. So, f^* is a polynomial of degree less than t in y with coefficients that are elements of $D = \mathbb{F}[x]/(x^{2m} + x^m + 1)$. This new polynomial will be said to be a member of $D[y]$.

To multiply two polynomials f and g in $\mathbb{F}[x]$, we first transform each of these to polynomials f^* and g^* in $D[y]$. Since ω is a $(3t)$th root of unity where t is a power of 3, then we could use the radix-3 FFT algorithm to evaluate f^* and g^* at $3t$ powers of ω, pointwise multiply these evaluations, and then use the radix-3 IFFT algorithm to obtain the product polynomial h^* in $D[y]$.

The improved version of Schönhage's algorithm takes advantage of the fact that since f^* and g^* have degree less than $t = 3^d$ for some $d < k$, then h^* will have degree less than $2t$. Therefore, it is only necessary to evaluate and interpolate at $2t$ points of D rather than the $3t$ points considered above.

If the full radix-3 FFT were to be computed, the first level of reduction steps would reduce $f^* \bmod (y^{3t} - 1)$ into

$$f^* \bmod (y^t - 1), \tag{5.12}$$

$$f^* \bmod (y^t - \omega^t), \tag{5.13}$$

$$f^* \bmod (y^t - \omega^{2t}). \tag{5.14}$$

Since the degree of f^* is less than t, then each of these three results is equal to f^*.

To implement the improved version of Schönhage's algorithm, we will compute a "truncated" FFT of f^* by only evaluating f^* at the points associated with the expressions (5.13) and (5.14). [3] Once this has been completed, we will have $2t$ evaluations which can be pointwise multiplied with $2t$ evaluations of g^*. Note that each of these "evaluations" is an element of D which is a polynomial in x. Pointwise multiplications in D involves multiplication of polynomials in x reduced modulo $x^{2m} + x^m + 1$. We will discuss pointwise multiplication of elements in D later in this section.

After the pointwise multiplies have been completed, the IFFT will interpolate the results into the two expressions

[3] In theory, one could use a different selection of two of these three expressions, but the selection of (5.13) and (5.14) is advantageous when the Schönhage's algorithm is called recursively.

$$h_Y = h^* \bmod (y^t - \omega^t), \tag{5.15}$$

$$h_Z = h^* \bmod (y^t - \omega^{2t}). \tag{5.16}$$

Now, the Extended Euclidean Algorithm discussed in Chapter 8 can be used to derive

$$\begin{aligned} (1) \cdot (y^t - \omega^t) - (1) \cdot (y^t - \omega^{2t}) &= -\omega^t + \omega^{2t} \\ &= -\omega^t + (\omega^t + 1) \ = \ 1. \end{aligned} \tag{5.17}$$

So by the Chinese Remainder Theorem,

$$\begin{aligned} h^* \bmod (y^{2t} + y^t + 1) &= (1) \cdot h_Z \cdot (y^t - \omega^t) + (-1) \cdot h_Y \cdot (y^t - \omega^{2t}) \\ &= (h_Z - h_Y) \cdot y^t - \omega^t \cdot h_Z + \omega^{2t} \cdot h_Y \\ &= (h_Z - h_Y) \cdot y^t - \omega^t \cdot h_Z + (\omega^t + 1) \cdot h_Y \\ &= (h_Z - h_Y) \cdot y^t - \omega^t \cdot (h_Z - h_Y) + h_Y. \end{aligned} \tag{5.18}$$

By reversing the transformation process described earlier, we obtain the desired polynomial h in $\mathbb{F}[x]$. Here, we substitute x^t in place of y and then simplify the resulting expression. However, since any of the $2m$ coefficients of an element of D can be nonzero in the product polynomial, some additional simplification is usually necessary.

The only detail that remains to be worked out is how to handle the pointwise products in D. To multiply two elements in D, each with degree less than $2m$, one multiplies the two polynomials considered as elements of $\mathbb{F}[x]$ and then computes the remainder of this result when divided by $x^{2m}+x^m+1$. Since the output of Schönhage's algorithm is modularly reduced by $y^{2t}+y^t+1$, then the result can be computed by simply recursively calling Schönhage's algorithm and combining like terms. In this case, the input polynomials to Schönhage's algorithm can be of degree at most $2t-1$ when transformed to $D[y]$.

For large values of n, the improved version of Schonhage's algorithm efficiently computes the product of two polynomials modulo $x^{2n}+x^n+1$, calling the algorithm recursively to handle the pointwise multiplications in D. For small values of n, it is more efficient to compute these products using Karatsuba's algorithm and then reduce the product modulo $x^{2n}+x^n+1$. The modular reductions are implemented by deleting each coefficient of the product polynomial with degree $d \geq 2n$ and adding this coefficient to the coefficients of $d-n$ and $d-2n$. The cost of this modular reduction is $2n$ additions. Pseudocode for an implementation of Schönhage's algorithm is given in Figure 5.3.

Let $M(2n)$ be the number of multiplications needed to multiply two polynomials of degree at most $2n = 2 \cdot 3^k$ modulo $x^{2n}+x^n+1$ and $A(2n)$ be the number of additions required. We will assume that it is more efficient to use Karatsuba's algorithm when we wish to multiply two polynomials with product degree of 54 or less. This can be verified by adjusting the condition in line 0 of the algorithm and comparing the operations required. So in line 0, if the input polynomials are of degree less than 54, then we need to use Karatsuba's algorithm to multiply two polynomials with input size $2 \cdot 3^k$ and reduce the result modulo $x^{2n}+x^n+1$. The operation counts

Algorithm : Schönhage's multiplication

Input: Polynomials $f, g \in \mathbb{F}[x]$ each of degree less than $2n$
where $\mathbb{F}$ is any finite field of characteristic 2 and $n = 3^k$ where $k > 1$.

Output: $h \bmod (x^{2n} + x^n + 1) = (f \cdot g) \bmod (x^{2n} + x^n + 1)$, a polynomial of degree less than $2n$ in $\mathbb{F}/(x^{2n} + x^n + 1)$; If the degrees of f and g sum to less than $2n$, then this result is also $h = f \cdot g$ where $h \in \mathbb{F}[x]$.

0. If $(2n) \leq 54$, then use Karatsuba's algorithm to compute $f \cdot g$ and then reduce this result modulo $(x^{2n} + x^n + 1)$.
1. Let $m = 3^{\lceil k/2 \rceil}$ and $t = n/m$ where $k = \log_2(n)$.
2. Transform $f, g \in \mathbb{F}[x]$ to polynomials f^*, g^* of degree less than $2t$ in $D[y]$ where $D = \mathbb{F}[x]/(x^{2m} + x^m + 1)$ using $y = x^t$ and grouping.
3. If $t = m$, let $\omega = x \in D$; otherwise, let $\omega = x^3 \in D$.
4. Compute radix-3 FFT to evaluate $f^*(y)$ at $\omega^{\sigma(t)}, \omega^{\sigma(t+1)}, \ldots, \omega^{\sigma(2t-1)}$.
5. Compute radix-3 FFT to evaluate $f^*(y)$ at $\omega^{\sigma(2t)}, \omega^{\sigma(2t+1)}, \ldots, \omega^{\sigma(3t-1)}$.
6. Compute radix-3 FFT to evaluate $g^*(y)$ at $\omega^{\sigma(t)}, \omega^{\sigma(t+1)}, \ldots, \omega^{\sigma(2t-1)}$.
7. Compute radix-3 FFT to evaluate $g^*(y)$ at $\omega^{\sigma(2t)}, \omega^{\sigma(2t+1)}, \ldots, \omega^{\sigma(3t-1)}$.
8. **for** $i = t$ **to** $3t - 1$ **do**
9. Recursively call Schönhage's Algorithm to compute $h^*(\omega^{\sigma(i)}) = f^*(\omega^{\sigma(i)}) \cdot g^*(\omega^{\sigma(i)})$.
10. **end for** (Loop i)
11. Compute radix-3 IFFT to compute $h_Y = h^*(y) \bmod (y^t - \omega^t)$.
12. Compute radix-3 IFFT to compute $h_Z = h^*(y) \bmod (y^t - \omega^{2t})$.
13. Compute $h^* \bmod (y^{2t} + y^t + 1) = (h_Z - h_Y) \cdot y^t - \omega^t \cdot (h_Z - h_Y) + h_Y$.
14. Transform $h^*(y) \bmod (y^{2t} + y^t + 1)$ to $h(x) \bmod (x^{2n} + x^n + 1)$ using $x^t = y$ and simplifying.
15. Return $h \bmod (x^{2n} + x^n + 1)$.

Figure 5.3 Pseudocode for Schönhage's multiplication

of Karatsuba's algorithm were given in a previous section and the result is a polynomial of degree less than $4n-1$. A total of $2 \cdot (2n-1) = 4n-2$ addition operations are required to implement the modular reduction. Combining these results, we obtain M(6) = 17, A(6) = 69, M(18) = 102, A(18) = 519, M(54) = 646, and A(54) = 3435.

Now suppose that $(2n) > 54$. Assume that the effort required to compute m and t is insignificant compared to an operation in $\mathbb{F}$, so line 1 does not contribute to the operation count. Line 2 is just a relabeling of the coefficients of f and g and we will assume that this does not contribute to the operation count either. An implementation of Schönhage's algorithm must be able to communicate to the recursive FFTs that only m coefficients in the input polynomials contribute to a "coefficient" in $D = \mathbb{F}[x]/(x^{2m} + x^m + 1)$, however. Line 3 is simply the assignment of the generator in D and this requires no operations in $\mathbb{F}$ or D. Lines 4-7 are four radix-3 FFTs in D, each of size t and lines 11-12 are two radix-3 IFFTs in D, each of size t as well. It can be shown that the total amount of work required for these six lines is no multiplications and

$$6 \cdot \left(\frac{11}{2} \cdot n \cdot \log_3(t) - \frac{3}{4} \cdot n + \frac{3}{4} \cdot m \right) = 33 \cdot n \cdot \log_3(t) - \frac{9}{2} \cdot n + \frac{9}{2} \cdot m \tag{5.19}$$

additions in $\mathbb{F}$.

In lines 8-10, we need to compute $2t$ "pointwise products", each a multiplication of two polynomials of size $2m$ modulo $x^{2m} + x^m + 1$. For each pointwise product, Schonhage's algorithm is called recursively with $n = m$. Thus, the amount of work required in these lines is $2t \cdot M(2m)$ multiplications and $2t \cdot A(2m)$ additions.

Next we need to compute the amount of work needed in line 13 of the algorithm. Computation of $h_Z - h_Y$ consists of t subtractions in D where each subtraction in D consists of $2m$ subtractions for a total of $2n$ subtractions in $\mathbb{F}$. This result is then multiplied by ω^t in D. The computation requires no multiplications in $\mathbb{F}$, but $t \cdot m = n$ additions in $\mathbb{F}$. Finally, this result is added to h_Y which will require t additions in D or $t \cdot 2m = 2n$ additions in $\mathbb{F}$. Combining these results gives a total of $5n$ additions / subtractions in $\mathbb{F}$ required to complete step 13.

Line 14 of the algorithm transforms polynomial in $D[y]$ back into a polynomial in $\mathbb{F}[x]$ at a cost of $n + m$ additions [4] in $\mathbb{F}$. We will assume that line 15 requires no operations to complete.

The total number of operations to compute a product of degree less than $2n$ using Schönhage's algorithm is

$$M(2n) = 2 \cdot t \cdot M(2m), \tag{5.20}$$

$$A(2n) = 2 \cdot t \cdot A(2m) + 33 \cdot n \cdot \log_3(t) + \frac{3}{2} \cdot n + \frac{11}{2} \cdot m \tag{5.21}$$

with the initial conditions stated above. Here, if $n = 3^k$, then $m = 3^{\lceil k/2 \rceil}$ and $t = 3^{\lfloor k/2 \rfloor}$.

If k is a power of two, then $t = m = \sqrt{n}$ and $\log_3(t) = \log_3(n)/2$. In this case, we can replace the above recurrence relations with

[4] If the input polynomials are each of size n, then only $n - m$ additions are required. When the input polynomials are each of size $2n$, then an additional $2m$ additions are needed to implement a modular reduction by $x^{2n} + x^n + 1$ on h.

$$M(2n) = 2 \cdot \sqrt{n} \cdot M(2\sqrt{n}), \tag{5.22}$$

$$A(2n) = 2 \cdot \sqrt{n} \cdot A(2\sqrt{n}) + \frac{33}{2} \cdot n \cdot \log_3(n) + \frac{3}{2} \cdot n + \frac{11}{2} \cdot \sqrt{n}, \tag{5.23}$$

and the initial conditions $M(18) = 102$, $A(18) = 519$. Solving these recurrence relations for closed-form solutions results in

$$\begin{aligned} M(2n) &= \frac{17}{3} \cdot n \cdot \log_3(n) \\ &\geq \frac{7}{2} \cdot n \cdot \log_2(n), \end{aligned} \tag{5.24}$$

$$\begin{aligned} A(2n) &\geq \frac{33}{2} \cdot n \cdot \log_3(n) \cdot \log_2 \log_3(n) + \frac{157}{12} \cdot n \cdot \log_3(n) - \frac{3}{2} \cdot n + \frac{11}{2} \cdot \sqrt{n} \\ &\geq \frac{52}{5} \cdot n \cdot \log_2(n) \cdot \log_2 \log_2(n) + \frac{3}{2} \cdot n \cdot \log_2(n) - \frac{3}{2} \cdot n + \frac{11}{2} \cdot \sqrt{n}. \end{aligned} \tag{5.25}$$

If k is not a power of two, then the results are somewhat greater. In this case, $M(2n) = 6 \cdot n \cdot \lceil \log_3(n) \rceil$ and (5.25) still gives a lower bound for $A(2n)$. However, the bound is not as tight as when k is a power of two.

5.5 FFT-based multiplication using the new additive FFT algorithm

Suppose that the new additive FFT algorithm is used to multiply two polynomials with product degree of less than the number of elements in the finite field. Using the formulas $M(n) = 3 \cdot M_F(n)$ and $A(n) = 3 \cdot A_F(n)$ derived in a previous section to compute the cost of this polynomial product, then the number of multiplications is given by

$$\begin{aligned} M(n) &= 3 \cdot \left(\frac{1}{2} \cdot n \cdot \log_2(n) - n + 1 \right) \\ &= \frac{3}{2} \cdot n \cdot \log_2(n) - 3n + 3 \end{aligned} \tag{5.26}$$

and the number of additions required is

$$\begin{aligned} A(n) &= 3 \cdot \left(\frac{1}{2} \cdot n \cdot \log_2(n)^{1.585} + n \cdot \log(n) - n + 1 \right. \\ &\quad \left. - \frac{1}{2} \cdot n \cdot \kappa^{1.585} + \frac{1}{4} \cdot n \cdot \kappa \cdot \log_2(\kappa) \right) \\ &= \frac{3}{2} \cdot n \cdot \log_2(n)^{1.585} + 3 \cdot n \cdot \log(n) - 3 \cdot n + 3 \\ &\quad - \frac{3}{2} \cdot n \cdot \kappa^{1.585} + \frac{3}{4} \cdot n \cdot \kappa \cdot \log_2(\kappa). \end{aligned} \tag{5.27}$$

One may have noticed that Schönhage's algorithm uses Karatsuba multiplication to handle the polynomial products of small degree. It is possible to use Karatsuba multiplication in a similar manner with FFT-based multiplication using the new additive FFT algorithm as discussed in the appendix. If this idea is used, then the number of multiplies in $\mathbb{F}$ needed to compute a polynomial product is reduced to

$$M(n) = \frac{3}{2} \cdot n \cdot \log_2(n) - \frac{15}{4} \cdot n + 8 \tag{5.28}$$

while the number of additions remains the same.

5.6 Comparison of the multiplication algorithms

We will now compare the costs of the algorithms presented in this chapter to determine if there are any additional crossover points in the timings for a possible hybrid algorithm combining several of the methods. This is somewhat difficult to do precisely because Schönhage's algorithm can only compute products of size $2 \cdot 3^{\mathbb{k}}$ and the other methods can only compute products of size 2^k. However, for the sake of comparing the algorithms, let us assume that the formulas given in the previous sections hold for any n. Because the algorithms produce different product sizes with respect to the input size, we must compare $M(2n)$ and $A(2n)$ for each of the algorithms.

By inspection of the formulas, it is obvious that there is no advantage to Schönhage's algorithm over FFT-based multiplication using the new additive FFT algorithm in terms of the multiplication count. Instead, the new approach improves the asymptotic multiplication count for polynomial multiplication over finite fields by over 16 percent. Because of the influence of the lower-order terms in the operation counts for FFT-based multiplication, the overall improvement is much greater on practical-sized problems ($n \leq 2^{32}$).

A comparison of the algorithms in terms of the number of additions requires a more careful analysis. Recall that when k is not a power of two, then the new additive FFT and IFFT algorithms involves reduction and interpolation steps from the Wang-Zhu-Cantor algorithms for FFT-based multiplication. A computer was used to show that FFT-based multiplication using the new additive FFT algorithm requires fewer additions than Schönhage's algorithm for all product sizes less than 2^{866}. At this point, the contribution from the Wang-Zhu-Cantor algorithms begins to dominate the overall cost up to a product size of 2^{1024}. Since 1024 is a power of two, then the reduction / interpolation step of the new additive FFT and IFFT algorithms once

again performs most of the computations. It turns out that the FFT-based algorithm requires less additions than Schönhage's algorithm up to product size of 2^{1476}. A comparison of the addition counts of the algorithms is mostly academic, for it will be many years before computers will be invented to handle the problem sizes just discussed.

To further support the claim that the FFT-based algorithm is more efficient than Schönhage's algorithm for all practical sizes, a range of actual product sizes that could be computed with Schönhage's algorithm was considered. For each product size up to $2 \cdot 3^{20}$, a computer was used to carefully evaluate the original recurrence relations of each algorithm and show that a product of greater size can be computed using the FFT-based multiplication algorithm. A summary of some of the results of these computations is given in Table 5.1.

Since the multiplication count for the FFT-based multiplication is also lower than the count for Schönhage's algorithm, then the overall cost for the FFT-based multiplication is lower than the overall cost for Schönhage's algorithm for all current practical sizes. When computers are invented to handle problem sizes of 2^{866} or greater, it might be more appropriate to consider the problem of trying to develop a hybrid multiplication algorithm that includes Schönhage's technique.

If implemented carefully, FFT-based multiplication using the new algorithm should outperform Schönhage's algorithm for all sizes when the product size is less than N where N is the number of elements in $\mathbb{F}$. If the product size is N or greater, then a multiplication method introduced by Reischert can be used. This technique is discussed in a section of the appendix and can be combined with truncated FFTs discussed in Chapter 6. If the product size is significantly larger than N, then a technique described in [32] may be used to map polynomials with coefficients in

Table 5.1 Addition cost comparison between Schönhage's algorithm and FFT-based multiplication using the new additive FFT algorithm

Product polynomial size	Schönhage's algorithm additions	New additive FFT algorithm additions
$2 \cdot 3^7 < 2^{13}$	1.02×10^6	8.26×10^5
$2 \cdot 3^8 < 2^{14}$	3.28×10^6	1.88×10^6
$2 \cdot 3^9 < 2^{16}$	1.53×10^7	6.09×10^6
$2 \cdot 3^{10} < 2^{17}$	4.79×10^7	1.41×10^7
$2 \cdot 3^{11} < 2^{19}$	1.55×10^8	7.34×10^7
$2 \cdot 3^{12} < 2^{21}$	4.83×10^8	3.63×10^8
$2 \cdot 3^{13} < 2^{22}$	1.81×10^9	8.00×10^8
$2 \cdot 3^{14} < 2^{24}$	5.59×10^9	3.80×10^9
$2 \cdot 3^{15} < 2^{25}$	1.77×10^{10}	8.22×10^9
$2 \cdot 3^{16} < 2^{27}$	5.45×10^{10}	3.79×10^{10}
$2 \cdot 3^{17} < 2^{28}$	2.35×10^{11}	8.12×10^{10}
$2 \cdot 3^{18} < 2^{30}$	7.19×10^{11}	3.63×10^{11}
$2 \cdot 3^{19} < 2^{32}$	2.23×10^{12}	9.14×10^{11}
$2 \cdot 3^{20} < 2^{33}$	6.81×10^{12}	2.01×10^{12}

$GF(N)$ to polynomials with coefficients in an extension field with more elements than the degree of the new polynomial.

A subject for debate is whether Schönhage's algorithm or FFT-based multiplication using the new additive FFT will perform better when we wish to multiply polynomials with coefficients in $GF(2)$. Two perspectives on this topic are summarized in another section of the appendix.

5.7 Concluding remarks

This chapter demonstrated how a new additive FFT algorithm can be applied to the problem of polynomial multiplication over finite fields of characteristic 2 to yield a new algorithm that performs faster than Schönhage's algorithm for most cases. In

order for the new algorithm to be faster, the finite field must be of size $N = 2^K$ where K is a power of two and the product polynomial must be of degree less than N. Most cases where products need to be computed over a finite field of characteristic 2 will satisfy these two restrictions.

CHAPTER 6

TRUNCATED FAST FOURIER TRANSFORM ALGORITHMS

Let R be a commutative ring and let $f(x), g(x)$ be two polynomials defined over $R[x]$. If R contains a primitive Nth root of unity where N is a power of two, then we learned in Chapter 5 that the product of these polynomials given by $h(x) \in R[x]$ can be computed in $\Theta(N \log_2 N)$ operations using the (multiplicative) Fast Fourier Transform (FFT) if the degree of h is less than N.

When the product polynomial has degree less than $n < N$ where n is not a power of two, we can still view this product as a polynomial of degree less than N. Here, FFT-based multiplication can be used to compute this product polynomial, but it will have zero coefficients in the highest degree terms. The operation count for computing a product of size n based on the FFT is constant in the range $1/2 \cdot N \leq n < N$. A graph of this operation count as a function of n will look like a staircase with a step corresponding to each new power of two. This technique wastes significant computational effort since the zero coefficients in the terms of degree n or higher are known before the computations are even started. The number of wasted operations is greatest when n is slightly greater than a power of two.

To reduce this waste, Crandall and Fagin [18] suggest selecting two powers of two, N_0 and N_1 such that $N_0 + N_1 \geq n$. As discussed in [2], one can compute the product modulo $x^{N_0} + 1$ and $x^{N_1} + 1$ using FFT-based multiplication and then combine these results using the Chinese Remainder Theorem. Note that this only works for values of n of the form $N_1 \cdot N_2$. In [3], this technique is generalized to handle additional values of n by working with d powers of two, $\{N_0, N_1, \cdots, N_{d-1}\}$ such that $N_0 + N_1 + \cdots + N_{d-1} \geq n$. If n is a fixed value, then this technique can

be used to make a custom FFT routine for computing over a particular length. For arbitrary n, there would be significant computational effort to derive the parameters needed by the Chinese Remainder Theorem and it would not be practical to compute these values each time the multiplication routine is called. Also, as d increases, the expressions for combining the results by the Chinese Remainder Theorem become more and more complicated. In summary, this is a good technique when n is fixed and d is small (like two or three), but not a good solution for computing the FFT of arbitrary n.

Another technique was introduced by van der Hoeven ([41], [42]) which computes the FFT using $\Theta(n \log_2 n)$ operations. In [41], it is noted that two of the paper referees wondered if there might be a connection between this "truncated Fast Fourier Transform" and Crandall and Fagin's technique, but an exploration of this question was left as an open problem. The referees also noted that no mention has been made of the number of operations required to implement Crandall and Fagin's technique.

This "truncated Fast Fourier Transform" does not apply for finite fields of characteristic 2 because this structure does not contain a primitive Nth root of unity. Instead, the "additive FFT" discussed in Chapter 3 can be used to evaluate and interpolate polynomials based on the roots of $x^N - x$ where N is a power of two. However, we are faced with the same problem of wasted computations discussed above for the multiplication of polynomials with finite field coefficients.

In this chapter, we give a modified version of van der Hoeven's algorithms which has also been generalized to work with both multiplicative and additive FFTs. In the case where the truncated algorithms are applied to FFTs involving a primitive Nth root of unity, we will show how the techniques are equivalent to that of Crandall and Fagin, but can be computed without the Chinese Remainder Theorem. We will

also show how the new algorithms can be used when R is a finite field of characteristic 2.

6.1 A truncated FFT algorithm

Let N be a power of two and let $K = \log_2(N)$. Define $s_0(x) = x$ and define $s_{i+1}(x)$ and $\bar{s}_i(x)$ [1] to be polynomials of degree 2^i such that $s_{i+1}(x) = s_i(x) \cdot \bar{s}_i(x)$ for all $0 \leq i \leq K-1$. Here, $\bar{s}_i(x) = s_i(x) + \mathcal{C}$ for some constant $\mathcal{C}$. We will assume that there exists an FFT algorithm that can efficiently evaluate a polynomial at each of the roots of $s_i(x)$ or $\bar{s}_i(x)$. The reduction step for this algorithm receives as input $f^\circ = f \bmod s_{i+1}$ and divides this polynomial by $s_i(x)$ to produce quotient q and remainder $r = f \bmod s_i$. Since $\bar{s}_i = s_i + \mathcal{C}$, then the other part of the reduction step computes $f \bmod \bar{s}_i$ with the formula $r - \mathcal{C} \cdot q$.

Suppose that f is a polynomial of degree less than $n < N$ that we wish to evaluate at n of the roots of $s_K(x)$. [2] To determine which n points will be selected for the evaluations, write n in binary form, i.e. $n = (b_{K-1}b_{K-2}\cdots b_1 b_0)_2$. We are going to evaluate f at each of the roots of $\bar{s}_i(x)$ where b_i is 1. In other words, we will evaluate f at the polynomial

$$\mathcal{M} = \prod_{i=0}^{K-1} (\bar{s}_i(x))^{b_i}. \tag{6.1}$$

The algorithm is initialized with $f = f \bmod s_K$ since f has degree less than N. A loop will iterate starting at $i = K-1$ and ending at $i = 0$. Iteration i of the loop

[1] In this chapter, $\bar{s}_i(x)$ is notation used to express a polynomial related to $s_i(x)$ and is not related to complex conjugation.

[2] If $n = N$, then we will assume that one can just use the FFT algorithm that evaluates f at each of the roots of $s_K(x)$.

will receive as input f mod s_{i+1} and reduce this polynomial into f mod s_i. If b_i is 1, then the loop will also reduce the polynomial into f mod $\bar{s}_i$ and call the existing FFT algorithm to evaluate f at each of the roots of f mod $\bar{s}_i$. At the end of the algorithm, we will have evaluated f at each of the n roots of (6.1). Pseudocode for the truncated FFT algorithm is provided in Figure 6.1.

Algorithm : Truncated FFT

Input: $f = f \bmod s_K(x) = f \bmod \mathcal{M}(x)$, a polynomial of degree less than n in a ring R. Here, $n = (b_{K-1}b_{K-2}\cdots b_1b_0)_2$ where $K = \log_2(N)$.

Output: The evaluation of f at each of the n roots of $\mathcal{M}(x) = (\bar{s}_0)^{b_0} \cdot (\bar{s}_1)^{b_1} \cdot (\bar{s}_2)^{b_2} \cdots (\bar{s}_{K-1})^{b_{K-1}}$.

```
1.  Determine L, the smallest integer such that b_L = 1.
2.  for i from K - 1 downto L do
3.      Reduce f mod s_{i+1} into f mod s_i.
4.      if b_i is 1
5.          Compute f mod s̄_i.
6.          Call the FFT algorithm to compute the
                evaluation of f mod s̄_i at each of the roots of s̄_i.
7.      end if
8.  end for
9.  Return the evaluation of f at each of the n roots of
        M(x) = (s̄_0)^{b_0} · (s̄_1)^{b_1} · (s̄_2)^{b_2} ··· (s̄_{K-1})^{b_{K-1}}.
```

Figure 6.1 Pseudocode for truncated FFT

Let us compute the cost of this algorithm. Assume that lines 1 and 9 cost no operations and all of the work takes place in a loop spanning lines 3 through 7 for all $L \leq i \leq K-1$. Note that the loop iterations can stop once all of the values of the output have been determined and L is the index where these computations are completed. Select some i in the above interval, let $m = 2^i$ and assume that $s_i(x)$ has $c_i + 1$ coefficients in it. Then line 3 requires $m \cdot c_i$ multiplications and $m \cdot c_i$

additions. If all of the nonzero coefficients of $s_i(x)$ are 1 or -1, then only $m \cdot c_i$ additions are required. Unless stated otherwise, we will assume that we are dealing with this simplified case for the rest of this chapter. Lines 5 and 6 are only executed if b_i is 1. Line 5 requires m multiplications and m additions. The cost of line 6 is the cost of the FFT algorithm with size 2^i. We will assume that the cost of this algorithm is $M_F(2^i)$ multiplications and $A_F(2^i)$ additions. If $M(n)$ is the number of multiplications needed to compute the truncated FFT for an input polynomial of size n and $A(n)$ is the number of additions or subtractions required, then formulas for the number of these operations is

$$M(n) = \sum_{i=L}^{K-1} \left(b_i \cdot \left(2^i + M_F(2^i) \right) \right), \tag{6.2}$$

$$A(n) = \sum_{i=L}^{K-1} \left(c_i \cdot 2^i + b_i \cdot \left(2^i + A_F(2^i) \right) \right). \tag{6.3}$$

It can be shown that the cost of the (untruncated) FFT algorithm is

$$M_F(N) = \sum_{i=0}^{K-1} \left(2^i + M_F(2^i) \right), \tag{6.4}$$

$$A_F(N) = \sum_{i=0}^{K-1} \left((c_i + 1) + A_F(2^i) \right). \tag{6.5}$$

By comparing these formulas, we can see that the truncated algorithm requires fewer operations. It can be shown that the truncated algorithm is an improvement over the untruncated algorithm by a factor of somewhere between 1 and 2.

We will provide further details of this cost of the truncated algorithm once c_i, $M_F(2^i)$, and $A_F(2^i)$ are known for particular cases.

6.2 An inverse truncated FFT algorithm

We now need an algorithm that can interpolate the collection of evaluations of f at each of the n roots of $\mathcal{M}(x)$ into $f \bmod s_K$. Since $\mathcal{M}(x)$ has degree n and f is assumed to have degree less than n, then $f \bmod s_K = f$ and is also equivalent to $f \bmod \mathcal{M}$.

Here, we will assume that there exists an inverse FFT algorithm that can efficiently interpolate the evaluations of some unknown polynomial at each of the roots of $s_i(x)$ or $\bar{s}_i(x)$. We will also assume that this inverse FFT algorithm contains an interpolation step that produces as output $f \bmod s_{i+1}$ given the two input polynomials $f \bmod s_i$ and $f \bmod \bar{s}_i$. Furthermore, we will assume that there exists the companion FFT algorithm used in the previous section.

Suppose that we are trying to determine $f \bmod s_{i+1}$, but already know the coefficients with degree δ_{i+1} or higher in this polynomial. In other words, we need to recover the lower δ_{i+1} coefficients of this result. Let $m = 2^i$. If $\delta_{i+1} \geq m$, then we are going to use the inverse FFT algorithm along with the given evaluations to compute $f \bmod \bar{s}_i$. Then we will combine $f \bmod \bar{s}_i$ and the known coefficients of $f \bmod s_{i+1}$ into the coefficients of $f \bmod s_i$ with degree $\delta_i = \delta_{i+1} - m$ or higher using the following result:

Theorem 26 ***(Operation A)*** *Given $f \bmod \bar{s}_i$ and the coefficients of $f \bmod s_{i+1}$ of degree $d + 2^i$ and higher, we can compute the coefficients of $f \bmod s_i$ with degree d and higher for any $d < 2^i$.*

Proof: Let $f^\circ = f \bmod s_{i+1}$ and let $m = 2^i$. Since s_{i+1} has degree $2m$, then f° will have degree less than $2m$. The given polynomial $\bar{r} = f \bmod \bar{s}_i$ satisfies

$$f^\circ = \bar{q} \cdot (\bar{s}_i) + \bar{r} \tag{6.6}$$

where the degree of $\bar{r}$ is less than the degree of $\bar{s}_i$. Since $\bar{s}_i(x) = s_i(x) + \mathcal{C}$, then

$$f^\circ \quad = \quad \bar{q} \cdot (s_i) + (\bar{r} + \mathcal{C} \cdot \bar{q}), \tag{6.7}$$

and $f \bmod s_i = \bar{r} + \mathcal{C} \cdot \bar{q}$. So all that is needed to recover $f \bmod s_i$ is to find a way to determine $\bar{q}$.

Now write f° as $f^\circ = f_\alpha \cdot x^{d+m} + f_\beta$ and $\bar{q}$ as $q_\alpha \cdot x^d + q_\beta$ where $d < m$. Then

$$f_\alpha \cdot x^{d+m} + f_\beta \quad = \quad (q_\alpha \cdot x^d + q_\beta) \cdot (\bar{s}_i) + \bar{r}. \tag{6.8}$$

By the hypothesis, the coefficients of f_α are known and the coefficients of f_β are unknown. Suppose that we divide $f_\alpha \cdot x^m$ by $\bar{s}_i$ to obtain quotient q^* and remainder r^*, i.e.

$$f_\alpha \cdot x^m \quad = \quad q^* \cdot (\bar{s}_i) + r^*. \tag{6.9}$$

Multiplying this equation by x^d and substituting this result into (6.8), we obtain

$$(q^* - q_\alpha) \cdot x^d \cdot (\bar{s}_i) \quad = \quad q_\beta \cdot (\bar{s}_i) + \bar{r} - r^* \cdot x^d - f_\beta. \tag{6.10}$$

Now,

$$\begin{array}{rcrcl} \deg(q_\beta \cdot \bar{s}_i) & = & \deg(q_\beta) + \deg(\bar{s}_i) & < & d+m, \\ \deg(\bar{r}) & & & < & d+m, \\ \deg(r^* \cdot x^d) & = & \deg(r^*) + d & < & d+m, \\ \deg(f_\beta) & & & < & d+m. \end{array} \tag{6.11}$$

Then $(q^* - q_\alpha) \cdot x^d \cdot (\bar{s}_i)$ must have degree less than $d+m$. Since $\deg(x^d \cdot \bar{s}_i) = d+m$, it must be the case that $q^* - q_\alpha = 0$, i.e. $q^* = q_\alpha$.

Thus, we can divide the known coefficients of f° of degree d and higher by $\bar{s}_i$ to obtain quotient q^*. This polynomial can be substituted into $\bar{r} + \mathcal{C} \cdot q^* \cdot x^d$ which matches $f \bmod s_i = f^\circ \bmod s_i$ in the terms of degree d and higher. □

If $\delta_{i+1} < m$, then we will combine the known evaluations of $f \bmod s_{i+1}$ into $f \bmod s_i$ with degree $\delta_i = \delta_{i+1}$ or higher. We will now prove that this can be accomplished with the reduction step of the companion FFT algorithm.

Theorem 27 *Given the coefficients of $f \bmod s_{i+1}$ of degree d and higher, we can compute the coefficients of $f \bmod s_i$ with degree d and higher for any $d < 2^i$.*

Proof: Let $f^\circ = f \bmod s_{i+1}$ and let $m = 2^i$. Since s_{i+1} has degree $2m$, then f° will have degree less than $2m$. If f° is written as $f^\circ = f_\alpha \cdot x^d + f_\beta$ for some $d < m$, then the desired polynomial $r = f \bmod s_i$ satisfies

$$f_\alpha \cdot x^d + f_\beta \;=\; q \cdot (s_i) + r, \tag{6.12}$$

where q and r each have degree less than m.

By hypothesis, the coefficients of f_α are known and the coefficients of f_β are unknown. Suppose we divide $f_\alpha \cdot x^d$ by s_i to obtain quotient q^* and remainder r^*. Then

$$f_\alpha \cdot x^d = q^* \cdot (s_i) + r^*. \tag{6.13}$$

Substituting this result into (6.12), we obtain

$$(q^* - q) \cdot s_i = r - r^* - f_\beta. \tag{6.14}$$

Since r, r^*, and f_β all have degree less than m, but the degree of s_i is equal to m, then $q^* - q = 0$ or $q^* = q$. It follows then that

$$r = r^* + f_\beta. \tag{6.15}$$

Since f_β with degree less than d is unknown, then we cannot determine the coefficients of degree less than d in r. However, this equation tells us that the coefficients of r^* with degree of d or higher are also the coefficients of $r = f \bmod s_i$ with degree d or higher. So, the coefficients of $f \bmod s_i$ with degree d or higher can be computed by dividing the known coefficients of $f \bmod s_{i+1}$ by s_i and reporting the terms of degree d and higher. Observe that this calculation is just the reduction step of the

companion FFT algorithm. □

Regardless of the value of δ_{i+1}, we then recursively consider the problem of computing $f \bmod s_{i-1}$ with the coefficients of degree δ_i or higher already known in this result.

Observe that $\delta_K = n$. Also, δ_i is simply the remainder resulting when n is divided by 2^i. So at some point in the recursion, δ_i will equal 2^{i-1}. This will occur for the value of i corresponding to the least significant bit in the binary expansion of n. For this value of i, we can combine $f \bmod \bar{s}_i$ and the known components of $f \bmod s_{i+1}$ into the remaining components of this result.

Theorem 28 ***(Operation B)*** *Given $f \bmod \bar{s}_i$ and the coefficients of $f \bmod s_{i+1}$ of degree 2^i and higher, we can recover all of the coefficients of $f \bmod s_{i+1}$.*

Proof: Let $f^\circ = f \bmod s_{i+1}$ and let $m = 2^i$. Since s_{i+1} has degree $2m$, then f° will have degree less than $2m$. If f° is written as $f^\circ = f_A \cdot x^m + f_B$, then the given polynomial $\bar{r} = f \bmod \bar{s}_i$ satisfies

$$f_A \cdot x^m + f_B \;=\; \bar{q} \cdot \bar{s}_i + \bar{r}, \tag{6.16}$$

where $\bar{q}$ and $\bar{r}$ each have degree less than m.

By hypothesis, the coefficients of f_A are known and the coefficients of f_B are unknown. Suppose that we divide $f_A \cdot x^m$ by $\bar{s}_i$ to obtain quotient q^* and remainder r^*. Then

$$f_A \cdot x^m \;=\; q^* \cdot \bar{s}_i + r^*. \tag{6.17}$$

Repeating the technique employed in the proofs of the previous theorems, we will again find that $\bar{q} = q^*$. In this case, this means that

$$f_B \;=\; \bar{r} - r^*. \tag{6.18}$$

So, to compute all of $f \bmod s_{i+1}$, we divide the known coefficients of $f \bmod s_{i+1}$ by $\bar{s}_i$ and subtract the result from the given expression $\bar{r} = f \bmod \bar{s}_i$. □

We can now undo the recursion. If $f \bmod \bar{s}_i$ is known, then we can use the known interpolation step to combine $f \bmod s_i$ and $f \bmod \bar{s}_i$ into $f \bmod s_{i+1}$. Another method is use (6.6) and (6.7) to solve for $\bar{q} = q_\alpha \cdot x^{d+m} + q_\beta$. Then solve (6.10) for f_β to obtain

$$f_\beta \;=\; q_\beta \cdot (\bar{s}_i) + \bar{r} - r^* \cdot x^d. \tag{6.19}$$

Here $\bar{r}$ and r^* can be reused from Operation A. The value of δ_{i+1} will determine which of these two methods is advantageous.

If $f \bmod \bar{s}_i$ is not known, then we can combine $f \bmod s_i$ and the known coefficients of $f \bmod s_{i+1}$ into the remaining coefficients of this result.

Theorem 29 ***(Operation C)*** *Given $f \bmod s_i$ and the coefficients of $f \bmod s_{i+1}$ with degree d or higher, we can compute all of the coefficients of $f \bmod s_{i+1}$ where $d < 2^i$.*

Proof: Let $f^\circ = f \bmod s_{i+1}$ and $r = f \bmod s_i$. Now write f° as $f^\circ = f_\alpha \cdot x^d + f_\beta$. If $r^* = (f_\alpha \cdot x^d) \bmod s_i$, then the proof of Theorem 27 tells us that $f_\beta = r - r^*$. Observe that r^* is already computed in this FFT reduction step and in this case all of $r = f \bmod s_i$ is assumed to be known. If r^* is saved from the earlier computation, it does not need to be recomputed here. So the remaining coefficients of $f \bmod s_{i+1}$ can be recovered by simply subtracting r^* from $f \bmod s_i$. □

After all of the recursion is undone, then we will have recovered $f \bmod \mathcal{M} = f$. Pseudocode for the inverse truncated FFT algorithm is given in Figure 6.2.

Let us determine the cost of the inverse algorithm. Assume that the algorithm is called for some value of $\kappa = i + 1$ where i is in the range $L \leq i \leq K - 1$. The algorithm has two cases depending on the value of b_i.

If $b_i = 1$, then instructions 2-9 are performed. In line 2, the inverse FFT algorithm is called with size $m = 2^i$. We will assume that this algorithm requires $M_F(2^i) + 2^i$ multiplications and $A_F(2^i)$ additions where $M_F(2^i)$ is the number of multiplications in the companion FFT algorithm and $A_F(2^i)$ is the number of additions required. The extra 2^i multiplications are due to scaling of the final output that is sometimes necessary in the inverse FFT. If $i > L$, then lines 4-6 are executed. Since δ_{i+1} is the number of known coefficients in $f^\circ = f \bmod s_{i+1}$, then the cost of line 4 is $c_i \cdot \delta_{i+1}$ additions where c_i is the number of coefficients in $s_i(x)$. Again, we will assume that all of the nonzero coefficients of $s_i(x)$ are 1 or -1. The cost of line 5 is the number of operations needed to implement the algorithm with $\kappa = i$. We will use the second method discussed above to implement line 6. The cost of this technique is at most m multiplications and $3m + (m - \delta_{i+1}) \cdot c_i$ additions. If $i = L$, then line 7 is executed. The cost of this instruction is at most $c_i \cdot m + m$ additions.

Algorithm : Inverse truncated FFT

Input: The coefficients with degree δ_κ or higher in some unknown polynomial $f^\circ = f \bmod s_\kappa \in R[x]$ of degree less than $2m = 2^\kappa$ where m is a power of two.
The evaluation of f° at the δ_κ roots of $\mathcal{M}_\kappa$ where $\mathcal{M}_\kappa = (\bar{s}_0)^{b_0} \cdot (\bar{s}_1)^{b_1} \cdot (\bar{s}_2)^{b_2} \cdots (\bar{s}_{\kappa-1})^{b_{\kappa-1}}$.
Here, $\delta_\kappa < 2m$ and $\delta_\kappa = (b_{\kappa-1} b_{\kappa-2} \cdots b_1 b_0)_2$.

Output: All of the coefficients of $f \bmod s_\kappa$.

1. **if** $b_{\kappa-1} = 1$ **do**
2. Call inverse FFT to compute $f \bmod \bar{s}_{\kappa-1}$.
3. **if** $\delta_\kappa < 2^{\kappa-1}$ **then**
4. Use "Operation A" to compute the coefficients of $f \bmod s_{\kappa-1}$ of degree $\delta_{\kappa-1} = \delta_\kappa - 2^{\kappa-1}$ or higher.
5. Recursively call algorithm to compute $f \bmod s_{\kappa-1}$ given the coefficients of degree $\delta_{\kappa-1}$ or higher.
6. Use inverse FFT interpolation step to combine $f \bmod s_{\kappa-1}$ and $f \bmod \bar{s}_{\kappa-1}$ into $f \bmod s_\kappa$.
7. **else**
8. Use "Operation B" to compute $f \bmod s_\kappa$ given the coefficients of this result of degree $2^{\kappa-1}$ or higher.
9. **end if**
10. **else**
11. Use FFT reduction step to compute the coefficients of $f \bmod s_{\kappa-1}$ of degree $\delta_{\kappa-1} = \delta_\kappa$ or higher.
12. Recursively call algorithm to compute $f \bmod s_{\kappa-1}$ given the coefficients of degree $\delta_{\kappa-1}$ or higher.
13. Use "Operation C" to compute $f \bmod s_\kappa$ given the coefficients of this result of degree δ_κ or higher and $f \bmod s_{\kappa-1}$.
14. **end if**
15. Return $f \bmod s_\kappa$.

Figure 6.2 Pseudocode for inverse truncated FFT

If $b_i = 0$, then lines 11-13 are performed. The cost of line 11 is at most $\delta_{i+1} \cdot c_i + m + m < m \cdot c_i + m$ additions. The cost of line 12 is the number of operations needed to implement the algorithm with $\kappa = i$. The cost of line 13 is at most m subtractions.

Combining these results, the cost of the inverse algorithm is at most

$$M(n) = \sum_{i=L}^{K-1} \left(b_i \cdot \left(2 \cdot 2^i + M_F(2^i) \right) \right), \tag{6.20}$$

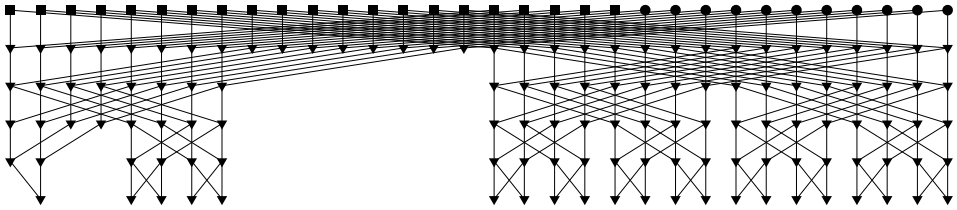
$$A(n) = \sum_{i=L}^{K-1} \left(2 \cdot 2^i + c_i \cdot 2^i + b_i \cdot \left(2^i + A_F(2^i) \right) \right) \tag{6.21}$$

operations. With these formulas, it can be easily shown that the cost of the inverse truncated FFT algorithm is the same as the cost of the truncated FFT algorithm plus at most n multiplications and $2N < 4n$ additions.

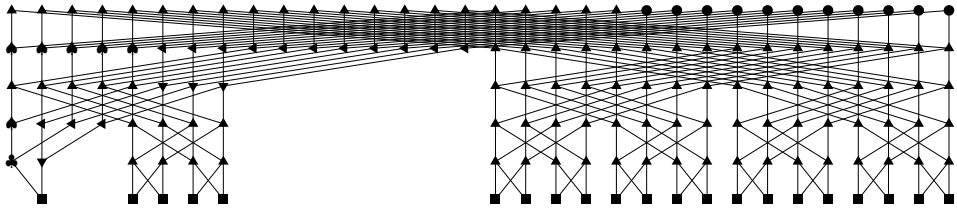
6.3 Illustration of truncated FFT algorithms

Figure 6.3 is provided to illustrate the truncated FFT and inverse truncated FFT algorithms for the case where $n = 21$. In the example, the truncated FFT computes the evaluation of some polynomial f at each of the roots of $\mathcal{M}(x) = \bar{s}_4(x) \cdot \bar{s}_2(x) \cdot \bar{s}_0(x)$ and the inverse truncated FFT interpolates these evaluations into f mod $\mathcal{M}$. Again, this selection for $\mathcal{M}$ was obtained by writing 21 in binary form, i.e. $(10101)_2$ and including the factor $\bar{s}_i(x)$ for each i such that $b_i = 1$ in this binary representation.

Figure 6.3 adopts the same representation of the input and output that is used in [41]. In particular, the top row of the figure represents a polynomial $f = a_{20} \cdot x^{20} + a_{19} \cdot x^{19} + \cdots + a_1 \cdot x + a_0$ of degree less than 21 as an array $[a_0, a_1, a_2, \ldots, a_{20}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$. The bottom row of the figure represents



(a) Truncated FFT for $n = 21$



(b) Inverse truncated FFT for $n = 21$

- • Known value at beginning of algorithm (zero value)
- ■ Known value at beginning of algorithm (not necessarily zero)
- ▼ Value computed using FFT step
- ◀ Value computed using Operation A
- ♣ Value computed using Operation B
- ▲ Value computed using inverse FFT step
- ♠ Value computed using Operation C

Figure 6.3 Illustration of truncated FFT algorithms

the evaluations of f where some of these evaluations are "truncated". These evaluations are presented in the order determined by the roots of s_0, $\bar{s}_0$, $\bar{s}_1, \cdots, \bar{s}_4$.

The legend at the bottom of the figure gives the methods used to obtain the results computed by the algorithm. The symbols used to represent the various operations generally illustrate the flow of the steps if the symbols are viewed as arrows. In the case of the inverse truncated FFT diagram, the algorithm works from the right side of the figure to the left, recovering as many values as possible. As the recursion of the algorithm is undone, the algorithm works from left to right in the figure, recovering the remaining values. Figure 6.3a illustrates the truncated FFT algorithm and Figure 6.3b illustrates the inverse truncated FFT algorithm.

6.4 Truncated algorithms based on roots of $x^N - 1$

In the classical form of the radix-2 version of the multiplicative FFT, R contains a primitive Nth root of unity ω and one evaluates a polynomial at each of N powers of ω. These N points used for the multipoint evaluation are roots of $s_K = x^N - 1$ where $K = \log_2(N)$. This algorithm is based on the factorization $x^{2m} - b^2 = (x^m - b) \cdot (x^m + b)$ where $m = 2^i$ for any $0 \leq i < K$ and contains a reduction step that converts $f^\circ = f \bmod (x^{2m} - b^2)$ into $f_Y = f \bmod (x^m - b)$ and $f_Z = f \bmod (x^m + b)$. In this case, $\mathcal{C} = 2$ and a simple way to implement the reduction step is to split f° into two polynomials of degree less than m, say $f^\circ = f_A \cdot x^m + f_B$. Then $f_Y = b \cdot f_A + f_B$ and $f_Z = -b \cdot f_A + f_B$. This evaluation step can be applied to $s_{i+1}(x) = x^{2m} - 1 = (x^m - 1) \cdot (x^m + 1) = s_i(x) \cdot \bar{s}_i(x)$ by letting $b = 1$. Observe here that $\bar{s}_i(x) = s_i(x) + 2$. It can be shown that the interpolation step of the companion IFFT is given by $f_A = 1/2 \cdot b^{-1} \cdot (f_Y - f_Z)$ and $f_B = 1/2 \cdot (f_Y + f_Z)$.

Some simplifications are possible with Operations A, B, and C because s_i will always have the form $x^m - 1$ and $\bar{s}_i$ will always have the form $x^m + 1$. Express

$f^\circ = f_{A_1} \cdot x^{d+m} + f_{A_2} \cdot x^m + f_{B_1} \cdot x^d + f_{B_2}$ where $d < m$. In Operation A, the division of $f_{A_1} \cdot x^m$ by $x^m + 1$ produces quotient f_{A_1} and remainder $-f_{A_1}$. So if $f \bmod (x^m + 1) = f_Z = f_{Z_1} \cdot x^d + f_{Z_2}$, then the upper coefficients of $f \bmod (x^m - 1)$ are simply given by $f_{Z_1} + 2 \cdot f_{A_1}$ since $\mathcal{C} = 2$. Similarly, in Operation B, the division of $f_A \cdot x^m$ by $x^m + 1$ has quotient f_A and remainder $-f_A$. So this operation simplifies to $f_B = f_A + f_Z$. Finally in Operation C, the division of $(f_{A_1} \cdot x^{m+d} + f_{A_2} \cdot x^m + f_{B_1} \cdot x^d)$ by $x^m - 1$ has remainder $(f_{A_1} + f_{B_1}) \cdot x^d + f_{A_2}$. Then if $f \bmod (x^{2^i} - 1) = f_{Y_1} \cdot x^d + f_{Y_2}$, the lower coefficients of $f \bmod (x^{2m} - 1)$ are simply given by $f_{B_2} = f_{Y_2} - f_{A_2}$.

In this case, $c_i = 1$ for all i, $M_F(2^i) = 1/2 \cdot 2^i \cdot i$ and $A_F(2^i) = 2^i \cdot i$. As discussed in the appendix, these results can be substituted into the operation count formulas derived earlier to show that the truncated algorithms have complexity $\Theta(n \cdot \log_2(n))$. The other algorithms from Chapter 2 can also be used to compute the FFTs of size 2^i in the truncated algorithms. In many cases, this will reduce the number of multiplications required.

The computations of the truncated algorithms involve the roots of

$$f \bmod \mathcal{M} \quad = \quad f \bmod \prod_{i=0}^{K-1} \left(x^{2^i} + 1\right)^{b_i}. \tag{6.22}$$

Observe that these are the same roots involved in the generalization of Crandall and Fagin's technique and that we can interpolate a collection of evaluations at these points without the use of the Chinese Remainder Theorem. The algorithms of van der Hoeven ([41], [42]) are based on other sets of n roots of $x^N - 1$, involve more cases than the algorithms discussed in this paper, and require slightly more operations.

6.5 Truncated algorithms based on roots of $x^N - x$

In Chapter 3, additive FFTs were discussed to efficiently compute the multipoint evaluation of a polynomial where the coefficient ring is a finite field $\mathbb{F}$ with 2^K elements. The most efficient FFT algorithms in this case represent each element of $\mathbb{F}$ as a vector space using the special basis discussed in Chapter 3. If this basis is used, then s_i is a polynomial such that all of its nonzero coefficients are 1 and are located in terms of degree that are a power of two. Furthermore, the coefficient of degree 2^d is determined by dividing the binomial coefficient $C(i, d)$ by 2 and recording the remainder of this computation. Also, $\bar{s}_i(x) = s_i(x) + 1$ so that $\mathcal{C} = 1$. The reduction step divides $f \bmod (s_{i+1} + (b^2 + b))$ by $s_i(x)$ to obtain quotient q and remainder $r = f \bmod (s_i + b)$. Then $f \bmod (\bar{s}_i) = f \bmod (s_i + (b + 1))$ is determined by the computation $r - q = r + q$ since $\mathbb{F}$ has characteristic 2. The interpolation step simply performs the inverse of these instructions in reverse order. This evaluation step can be applied to $s_{i+1}(x) = s_i(x) \cdot \bar{s}_i(x)$ by letting $b = 0$.

In this case, c_i is one less than the number of ones in the binary representation of 2^i, $M_F(2^i) = 1/2 \cdot 2^i \cdot i$ and $A_F(2^i) = 1/2 \cdot 2^i \cdot i^{1.585} + 2^i \cdot i$. By substituting these results into the operation count formulas derived earlier, we can show that the truncated algorithms require $\Theta(n \cdot (\log_2(n))^{1.585})$ operations using a computation similar to the multiplicative case. With some modification of the operation count derivations given earlier, we can apply the truncated algorithms to the more general form of the additive FFT also discussed in [32]. In this case, the truncated algorithms require $\Theta(n \cdot (\log_2(n))^2)$ operations. On the other hand, the new additive FFT algorithm introduced in this chapter can be used to reduce the number of operations required to compute the FFTs of size 2^i in the truncated algorithm.

6.6 Concluding remarks

It is possible to develop a truncated FFT based on an FFT of any size $N = p^K$ where p is a prime number. For example, if $p = 3$, then the truncated algorithm for the multiplicative FFT is based on the factorization

$$x^N - 1 = (x - 1) \cdot \prod_{i=0}^{K-1} (x^{3^i} - \Omega) \cdot (x^{3^i} - \Omega^2), \tag{6.23}$$

where Ω is a primitive 3rd root of unity and $\Omega^2 + \Omega + 1 = 0$. To compute the truncated FFT of size $n < N$, write n in ternary form, i.e. $(t_{K-1} t_{K-2} \cdots t_1 t_0)_3$. If $t_i = 1$, then we will use the radix-3 FFT to compute evaluations corresponding to the roots of $x^m - \Omega$ where $m = 3^i$. If $t_i = 2$, then we will use the radix-3 FFT to compute evaluations corresponding to the roots of $(x^m - \Omega) \cdot (x^m - \Omega^2) = x^{2m} + x^m + 1$ where again $m = 3^i$. One can mimic the techniques discussed in this chapter to construct the inverse truncated FFT algorithm for this case, using linear algebra to solve several systems of 3×3 equations for unknowns present on both sides of the equations.

In summary, this chapter presented a generalization of a truncated FFT algorithm introduced by van der Hoeven that can also be applied to polynomials where the coefficient ring is a finite field of characteristic 2. In the multiplicative case, the algorithms here differ from those of van der Hoeven because they are based on a factorization introduced by Crandall and Fagin and require slightly fewer operations. Depending on the input size, these algorithms require as few as half of the operations required to compute an FFT of size n by the common technique of selecting an input size $N = 2^K$ to perform the computation where $n \leq N$. This can significantly improve the operations needed to multiply two polynomials of degree less than n. With the new algorithm, this technique can also be applied to polynomials with finite field coefficients.

CHAPTER 7

POLYNOMIAL DIVISION WITH REMAINDER

Let $a(x)$ and $b(x) \neq 0$ be polynomials with coefficients in R, a commutative ring. The problem of computing division with remainder is to determine the unique polynomials $q(x)$ and $r(x)$ that satisfy

$$a = q \cdot b + r, \tag{7.1}$$

where the degree of r is less than the degree of b. Here, b has degree less than n, and a has degree $\deg(b) + \delta$ for some $\delta > 0$. If R is not a field, then b must have the further restriction that it is monic, i.e. it has a leading coefficient of 1.

If it is known at the beginning of the computation that r is 0, then the quotient can be computed using the technique of deconvolution if R supports division and some type of FFT. In this case, we would compute the FFT of a and b, compute the "pointwise quotients" $a(\varepsilon)/b(\varepsilon) = q(\varepsilon)$ for each ε in the set of elements used in the FFT, and finally compute the IFFT of this collection of evaluations. Such a computation would require roughly the same effort as multiplying the two polynomials with product degree less than $n + \delta$.

For the rest of this chapter, we will assume that $r \neq 0$. We will examine two techniques for computing this quotient, namely classical division and division based on Newton's method. Several improvements to Newton division will then be reviewed if R supports an FFT. The chapter will also show how Newton division can be further

improved with the truncated Fast Fourier Transform techniques covered in Chapter 6.

7.1 Classical division

Classical division is the technique typically learned in a high school algebra course (e.g. [5]). Assuming that R is a field or b is monic, the algorithm begins by dividing the leading coefficient of a by the leading coefficient of b to obtain the leading coefficient of the quotient q. If this leading coefficient is denoted by q_δ, then the next step of the process is to multiply $q_\delta \cdot x^\delta$ by b and subtract the result from a to obtain a polynomial of degree less than δ. The above process is repeated on this new polynomial to obtain $q_{\delta-1}$. Similarly, we can compute $q_{\delta-2}$, $q_{\delta-3}, \cdots, q_0$ and combine the results into the polynomial $q = q_\delta \cdot x^\delta + q_{\delta-1} \cdot x^{\delta-1} + \cdots + q_1 \cdot x + q_0$. At the end of the procedure, a has been reduced to a polynomial r with degree less than n. Thus, given polynomials b with degree less than n, and a with degree $\deg(b) + \delta$ for some $\delta > 0$, we have computed polynomials q with degree δ and r with degree less than b such that $a = q \cdot b + r$ if $a \neq 0$. If $a = 0$, then $q = r = 0$. The pseudocode in Figure 7.1 summarizes these steps used to implement classical division.

Let us now analyze the cost of this algorithm. Assume that the cost of line 1 is one inversion operation in R (if R is a field) and lines 7, 8, and 9 do not cost any operations. Thus, most of the work of the algorithm is containing in the loop spanning lines 2-7. We will now analyze a single iteration of this loop for some value of i.

In line 3, a decision is made which we will assume to cost no operations. If the condition was satisified, then line 4 involves one multiplication in R and line 5 involves n multiplications and n subtractions in R. If the condition was not satisifed,

Algorithm : Classical division

Input: Polynomials $b(x)$ with degree less than n
and $a(x)$ with degree $\deg(b) + \delta$ for some $\delta > 0$.

Output: Polynomials $q(x)$ with degree δ and $r(x)$ with degree less than $\deg(b)$
such that $a = q \cdot b + r$.

1. Let $r = a$ and let ρ be the inverse of the leading coefficient of b.
2. **for** $i = \delta$ downto 0 **do**
3. **if** $\deg(r) = n + i$ **then**
4. $q_i = r_i \cdot \rho$.
5. Set r equal to $r - q_i \cdot x^i \cdot b$.
6. **else** $q_i = 0$.
7. **end if**
8. **end for** (Loop i)
9. Return $q = q_\delta \cdot x^\delta + q_{\delta-1} \cdot x^{\delta-1} + \cdots + q_1 \cdot x + q_0$ and r.

Figure 7.1 Pseudocode for classical division

then line 6 requires an assignment which we will assume will cost no operations. Thus, the loop requires at most $n + 1$ multiplications and n subtractions.

Since the loop must iterate $\delta + 1$ times, then the cost of the algorithm is $\delta \cdot n + \delta + n + 1$ multiplications and $\delta \cdot n + n$ subtractions, plus one inversion in R.

Note that the cost of this algorithm is $\Theta(\delta \cdot n)$ where δ is the difference in degrees between the two input polynomials. If $\deg(a)$ is about $2n$, then $\delta = n$ and the algorithm is $\Theta(n^2)$.

7.2 Newton division

We are are now going to examine an algorithm that computes division with remainder more efficiently than classical division when the degree of a is much larger than the degree of b and the degree of b is reasonably large as well. The algorithm is based on Newton's method, a technique typically introduced in a Calculus course for

finding roots of equations. Surprisingly, the technique can also be applied to finite fields.

Classical division solves for the two unknown polynomials q and r at the same time. In constrast, Newton division first solves for q and then uses $a = q \cdot b + r$ to recover r. Suppose that $1/x$ is substituted for x in (7.1) and the result is multiplied by $x^{n+\delta}$. If the reversal of f, $\mathrm{rev}_d(f)$, is defined [1] as $x^d \cdot f(1/x)$ for any polynomial $f(x)$ and any integer d, then we obtain

$$\mathrm{rev}_{n+\delta}(a) \quad = \quad \mathrm{rev}_\delta(q) \cdot \mathrm{rev}_n(b) + \mathrm{rev}_{n+\delta}(r). \tag{7.2}$$

Since the degree of r is less than n, then $\mathrm{rev}_{n+\delta}(r) \bmod x^{\delta+1} = 0$ and

$$\mathrm{rev}_\delta(q) \cdot \mathrm{rev}_n(b) \quad = \quad \mathrm{rev}_{n+\delta}(a) - \mathcal{C} \cdot x^{\delta+1}. \tag{7.3}$$

Let $\mathcal{N}(x)$ be a polynomial such that $\mathrm{rev}_n(b) \cdot \mathcal{N} \bmod x^{\delta+1} = 1$. Multiplying (7.3) by $\mathcal{N}$ and modularly reducing both sides of the equation by $x^{\delta+1}$ we obtain

$$\mathrm{rev}_\delta(q) \quad = \quad \mathrm{rev}_{n+\delta}(a) \cdot \mathcal{N} \bmod x^{\delta+1}. \tag{7.4}$$

The problem of division with remainder essentially becomes the problem of finding $\mathcal{N}$, also called the "Newton inverse" of a power series modulo some x^m. In other words,

---

[1] Another interpretation of $\mathrm{rev}_d(f)$ is to create a polynomial of degree d by writing the coefficients of f in reverse order.

given a power series $f(x)$, we desire to compute another power series $g(x)$ such that $f \cdot g \bmod x^m = 1$. In this section, we will assume that $m = \delta + 1$ is a power of two. In exercise 9.6 of [34], an algorithm that works for arbitrary m is considered.

The technique used to find $\mathcal{N}$ is based on Newton's method which is typically introduced in a Numerical Analysis course (e.g. [11]) for solving nonlinear equations. Suppose that we wish to find a solution g to the equation $1/g - f = 0$ for some given f. Let $\Phi(g) = 1/g - f$. Newton's method receives some initial approximation $g_{(0)}$ as input and computes better approximations to g using the iteration

$$g_{(i)} = g_{(i-1)} - \frac{\Phi(g_{(i-1)})}{\Phi'(g_{(i-1)})}, \tag{7.5}$$

where $\Phi'(g_{(i-1)})$ is the derivative of $\Phi(g_{(i-1)})$. In the case of the problem at hand, $\Phi'(g_{(i-1)}) = -1/(g_{(i-1)})^2$. So then, (7.5) simplifies to

$$g_{(i)} = g_{(i-1)} - \frac{1/g_{(i-1)} - f}{-1/(g_{(i-1)})^2} = 2 \cdot g_{(i-1)} - f \cdot (g_{(i-1)})^2. \tag{7.6}$$

Newton's method can also be used to find a polynomial g such that $f \cdot g \bmod x^m = 1$ for some known polynomial f. In this case, $g_{(0)}$ is initialized to be the inverse of the constant term of f. Again, if R is not a field, then this term must be 1. Note that by the construction of $g_{(0)}$, then $f \cdot g_{(0)} \bmod x^1 = 1$. Now, for any $i \leq \log_2(m)$, then

$$g_{(i)} = 2 \cdot g_{(i-1)} - f \cdot (g_{(i-1)})^2 \bmod x^{2^i} \tag{7.7}$$

provides a polynomial that matches g in the lower 2^i coefficients if $g_{(i-1)}$ is already known. So starting with $g_{(0)}$, each call to (7.7) doubles the number of recovered coefficients of g. Thus, iteration (7.7) is said to possess "quadratic convergence". A proof of the quadratic convergence of this iteration is given in [34] for the case of polynomials by using Calculus concepts.

One may object to this procedure in the case where R is a finite field because the concept of limits does not exist in this structure. In [34], a derivation of the Newton inversion iterative step is provided using the "formal derivative" which does not depend on limits in its definition. It can be shown [34] that the "formal derivative" has the properties of the product rule and the chain rule which allow the derivative of $\Phi(g)$ to be computed using a algebraic definition.

Alternatively, [30] provides a derivation of the iterative step which does not require the use of derivatives at all in its presentation and clearly shows the quadratic convergence of the iterative step. A modified version of this derivation is given in the appendix.

Once $\mathcal{N}$ has been recovered using Newton's method, this result can be substituted into (7.4) to obtain $\text{rev}_\delta(q)$. The reversal of this polynomial can be computed to determine q. Then use

$$r = a - q \cdot b \tag{7.8}$$

to recover the remainder. The pseudocode provided in Figure 7.2 summarizes these procedures.

Let us compute the cost of this algorithm. Most of the work will take place in the loop spanning lines 2-4 which implements Newton's method. At iteration i, we

Algorithm : Newton division
Input: Polynomials $b(x)$ with degree less than n and $a(x)$ with degree $\deg(b) + \delta$ for some δ where $2^\kappa = \delta + 1$ for some κ.
Output: Polynomials $q(x)$ with degree δ and $r(x)$ with degree less than $\deg(b)$ such that $a = q \cdot b + r$.
1. Compute the reversal polynomial $f = \text{rev}_{n+\delta}(a)$; Let $g_{(0)} = (f_0)^{-1}$. 2. **for** $i = 1$ to κ **do** 3. Set $g_{(i)} = 2 \cdot g_{(i-1)} - f \cdot (g_{(i-1)})^2 \bmod x^{2^i}$. 4. **end for** (Loop i) 5. Compute $\text{rev}_\delta(q) = f \cdot g_{(\kappa)}$. 6. Reverse $\text{rev}_\delta(q)$ to obtain q. 7. Compute $r = a - q \cdot b$. 8. Return q and r.

Figure 7.2 Pseudocode for Newton division algorithm

only need to compute the upper 2^{i-1} coefficients of $g_{(i)}$ since the lower 2^{i-1} coefficients are given by $g_{(i-1)}$ computed in the previous iteration. Thus to compute $g_{(i)}$, we need to compute a product of degree less than 2^i to form ${g_{(i-1)}}^2$, compute a product of degree less than 2^{i+1} to form $f \cdot {g_{(i-1)}}^2$, and finally combine $2 \cdot g_{(i-1)}$ with the negative of the upper 2^{i-1} coefficients of this result. If $M_M(2n)$ and $A_M(2n)$ give the number of multiplications and additions to multiply two polynomials with product degree less than $2n$, then the cost of the iteration step for a given value of i is $M_M(2^i) + M_M(2^{i+1})$ multiplications and $A_M(2^i) + A_M(2^{i+1}) + 2^{i-1}$ additions. Summing over all values of i used in the algorithm gives an operation count of $\sum_{i=1}^{\kappa}(M_M(2^i) + M_M(2^{i+1}))$ multiplications and $\sum_{i=1}^{\kappa}(A_M(2^i) + A_M(2^{i+1}) + 2^{i-1})$ additions. If we assume that $2 \cdot M_M(2^i) \leq M_M(2^{i+1})$ and $2 \cdot A_M(2^i) \leq A_M(2^{i+1})$ for all $i \leq k$, then the total number of operations needed to compute the Newton inversion is at most $\sum_{i=1}^{\kappa}(3/2 \cdot M_M(2^{i+1}))$ multiplications and $\sum_{i=1}^{\kappa}(3/2 \cdot A_M(2^{i+1}) + 1/2 \cdot 2^i)$ additions. Using Master Equation V,

it can be shown that at most $3 \cdot M_M(2 \cdot (\delta+1))$ multiplications and $3 \cdot A_M(2 \cdot (\delta+1)) + \delta + 1$ additions are required.

Now let us compute the cost of the rest of the algorithm. We will assume that lines 1, 6, and 8 do not cost any operations. The multiplication in line 5 costs $M_M(2n)$ multiplications and $A_M(2n)$ additions if $n \geq \delta + 1$. If $\delta + 1 > n$, then line 5 costs $M_M(2 \cdot (\delta + 1))$ multiplications and $A_M(2 \cdot (\delta + 1))$ additions. The multiplication in line 7 costs $M_M(2 \cdot (\delta+1))$ multiplications and $A_M(2 \cdot (\delta+1))$ additions if $n \geq \delta+1$. If $\delta + 1 > n$, then line 7 costs $M_M(2n)$ multiplications and $A_M(2n)$ additions. In either case, at most n subtractions are required to recover r since $\deg(r) < \deg(b) = n$.

Combining the results, we find that the total number of operations required to implement this algorithm is at most $4 \cdot M_M(2 \cdot (\delta + 1)) + M_M(2n)$ multiplications and $4 \cdot A_M(2 \cdot (\delta + 1)) + A_M(2n) + n + \delta + 1$ additions. A number of researchers (e.g. Bernstein [3]) prefer to express the operations for the case where $\delta = n - 1$. Using this convention, one would say that the cost of dividing a polynomial of degree less than $2n - 1$ by a polynomial of degree less than n is roughly 5 times the cost of multiplying two polynomials of degree less than n using the above algorithm.

7.3 Newton divison using the multiplicative FFT

In [3], Bernstein summarizes a progression of improvements made to the computation of Newton inversion which exploits known coefficients in each $g_{(i)}$ computed in the iterations. These improvements require R to contain a (2^{k+1})th primitive root of unity ω for some k, i.e. R supports a multiplicative Fast Fourier Transform.

In the derivation of Newton's Method given in the appendix, $g_{(i)}$ is represented as $g_{(i)} = g_A \cdot x^{2^{i-1}} + g_{(i-1)}$ for some polynomial g_A. If ${g_{(i)}}^* = g_A$ in this expression, then

$$g_{(i)}{}^{*} \cdot x^{2^{i-1}} \;=\; g_{(i-1)} - f \cdot (g_{(i-1)})^2 \bmod x^{2^i} \tag{7.9}$$

using a result derived in the appendix. Let us compute

$$g_{(i)}{}^{\dagger} \;=\; g_{(i-1)} - f^{*} \cdot (g_{(i-1)})^2 \tag{7.10}$$

where $f^{*} = f \bmod x^{2^i}$. Here, we have truncated f to the lower 2^i coefficients because the terms of degree 2^i or higher will be truncated in (7.9) and are unnecessary to the computation. Observe that $g_{(i)}{}^{\dagger}$ is a polynomial of degree 2^{i+1} with zeros in the lower 2^{i-1} coefficients.

Since R contains a primitive (2^{k+1})th root of unity ω, then each of f^{*}, $g_{(i-1)}$ and $g_{(i)}{}^{\dagger}$ can be evaluated at each of the powers of ω. Since function evaluation is a linear operation, then

$$g_{(i)}{}^{\dagger}(\omega^{\theta}) = g_{(i-1)}(\omega^{\theta}) - f^{*}(\omega^{\theta}) \cdot g_{(i-1)}(\omega^{\theta}) \cdot g_{(i-1)}(\omega^{\theta}) \tag{7.11}$$

for any θ in $0 \leq \theta \leq 2^k$. So $g_{(i)}{}^{\dagger}$ can be determined by computing the FFT of size 2^{i+1} of f^{*}, computing the FFT of size 2^{i+1} of $g_{(i-1)}$, using (7.11) in place of the pointwise products, and then using an inverse FFT of size 2^{i+1} to recover the result.

We can further improve this computation by taking advantage of the 2^{i-1} known coefficients in $g_{(i)}{}^{\dagger}$. Instead of computing the FFTs of size 2^{i+1} for f^{*} and $g_{(i-1)}$, compute the FFTs to be of size $m = 3 \cdot 2^i$ where we will evaluate the two

polynomials at each of the roots of $(x^{2^i}+1)\cdot(x^{2^{i-1}}+1) = x^{3\cdot 2^{i-1}}+x^{2^i}+x^{2^{i-1}}+1$. The result of interpolating the evaluation of (7.11) at each of the roots of this polynomial will be ${g_{(i)}}^{\dagger\dagger} = {g_{(i)}}^{\dagger} \bmod (x^{3\cdot 2^{i-1}}+x^{2^i}+x^{2^{i-1}}+1)$. To recover ${g_{(i)}}^{\dagger}$, one could multiply the coefficients of degree less than 2^{i-1} in ${g_{(i)}}^{\dagger\dagger}$ by $x^{3\cdot 2^{i-1}}+x^{2^i}+x^{2^{i-1}}+1$ and then subtract the result from ${g_{(i)}}^{\dagger\dagger}$.

Another approach is to directly compute ${g_{(i)}}^{*}$ without first recovering ${g_{(i)}}^{\dagger}$. First, compute the reverse polynomials of f^* and $g_{(i-1)}$ with respect to degree $\nu = 3\cdot 2^i$. One then computes the product of these polynomials using the Truncated Fast Fourier Transform (TFFT) discussed in Chapter 6. This product will be the reverse polynomial of ${g_{(i)}}^{*}$ with respect to ν because the known zero coefficients in the upper positions of the reverse of ${g_{(i)}}^{\dagger}$ do not influence the results in the lower $3\cdot 2^i$ positions. So the reverse of this product can be computed to recover ${g_{(i)}}^{*}$. An advantage of this approach over the previous methods is that is can be easily adapted to the case where the size of $g_{(i)}$ is not a power of two. Specificially, if $g_{(i)}$ is of size n, then one can efficiently compute this result using a TFFT of size $\nu = 3/4\cdot n$.

The pseudocode provided in Figure 7.3 shows how to compute a quotient with remainder using these improvements to Newton division. To analyze this algorithm, we only need to consider the number of operations required for the Newton inversion (lines 2-11) and then add these results to those obtained from the previous section. We will first determine the cost of a single iteration of this loop (lines 3-10). Line 3 simply extracts some of the coefficients from f and costs no operations. We will assume that the cost of line 4 is no arithmetic operations, although copy operations are required to implement the instruction. The TFFT discussed in Chapter 6 can be used to compute the product represented by lines 5-8 in $0.75\cdot M_M(2^{i+1})$ multiplications and $0.75\cdot A_M(2^{i+1})$ additions. We will assume that the cost of line 9 is no arithmetic operations, although copy operations are required to implement the instruction. Line

Algorithm : Improved Newton division

Input: Polynomials $b(x) \in R[x]$ with degree less than n
and $a(x) \in R[x]$ with degree $\deg(b) + \delta$ for some δ where $2^k = \delta + 1$
for some k. R has a $(2^{k+1})th$ primitive root of unity ω.

Output: Polynomials $q(x)$ with degree δ and $r(x)$ with degree less than $\deg(b)$
such that $a = q \cdot b + r$.

1. Compute the reversal polynomial $f = \text{rev}_{n+\delta}(a)$;
   Let $g_{(0)} = (f_0)^{-1}$ where f_0 is the constant term of f.
2. **for** $i = 1$ to k **do**
3. Let $f^* = f \bmod x^{2^i}$.
4. Compute $f_r = \text{rev}_\nu(f^*)$ and $g_r = \text{rev}_\nu(g_{(i-1)})$ where $\nu = 3 \cdot 2^i$.
5. Evaluate f_r at each of the roots of $(x^{2^i} + 1) \cdot (x^{2^{i-1}} + 1)$.
6. Evaluate g_r at each of the roots of $(x^{2^i} + 1) \cdot (x^{2^{i-1}} + 1)$.
7. Compute $g_r^*(\omega^\theta) = g_r(\omega^\theta) - f_r(\omega^\theta) \cdot g_r(\omega^\theta) \cdot g_r(\omega^\theta)$ for each ω^θ
   that is a root of $(x^{2^i} + 1) \cdot (x^{2^{i-1}} + 1)$.
8. Compute $g_r^* \bmod (x^{2^i} + 1) \cdot (x^{2^{i-1}} + 1)$ by interpolating the evaluations
   from line 7.
9. Compute ${g_{(i)}}^* = \text{rev}_\nu(g_r^*)$.
10. Compute $g_{(i)} = {g_{(i)}}^* \cdot x^{2^{i-1}} + g_{(i-1)}$.
11. **end for** (Loop i)
12. Compute $\text{rev}_\delta(q) = f \cdot g_{(k)}$.
13. Reverse $\text{rev}_\delta(q)$ to obtain q.
14. Compute $r = a - q \cdot b$.
15. Return q and r.

Figure 7.3 Pseudocode for improved Newton division

10 involves concatenating two polynomials at no cost. Master Equation V can be used to show that the cost of this loop is at most $1.5 \cdot M_M(2 \cdot (\delta + 1))$ multiplications and $1.5 \cdot A_M(2 \cdot (\delta + 1))$ additions.

Combining these results with the rest of cost analysis for this algorithm given in the previous section, we find that the total number of operations required to implement this algorithm is at most $2.5 \cdot M_M(2 \cdot (\delta + 1)) + M_M(2n)$ multiplications and $2.5 \cdot A_M(2 \cdot (\delta + 1)) + A_M(2n) + n + \delta + 1$ additions. If $\delta = n - 1$, then the cost of dividing a polynomial of degree less than $2n - 1$ by a polynomial of degree less than n is roughly 3.5 times the cost of multiplying two polynomials of degree less than n using the above algorithm.

These performance results are not new. Schonhage [69] and Bernstein [3] each independently published an algorithm with a similar running time in 2000. Schonhage computes polynomial products using a technique similar to Reischert's Method [64], a multiplication technique discussed in a section of the appendix. Bernstein's method computes ${g_{(i)}}^{\dagger} \bmod (x^{2^i} + 1)$ and ${g_{(i)}}^{\dagger} \bmod (x^{2^{i-1}} + 1)$ which are combined using the Chinese Remainder Theorem. Note that the TFFT obtains a similar result without the use of the Chinese Remainder Theorem and allows more flexible input sizes. The application of the TFFT to this problem may also be viewed as simpler to work with than the other algorithms; indeed, Bernstein describes his own method [3] as a "rather messy algorithm".

7.4 Newton divison for finite fields of characteristic 2

The method described in the previous section does not apply to finite fields of characteristic 2 because they do not support a multiplicative FFT. However, the method can be adapted to work with finite fields by using the additive FFT described in Chapter 3 and the truncated version of the algorithm described in Chapter 6.

Furthermore, these finite fields possess a different property which allows an even more efficient algorithm to be generated.

Suppose that $\mathbb{F}$ is a finite field of characteristic 2 with N elements. To construct ${g_{(i-1)}}^2$ in such a field, simply square each of the coefficients of $g_{(i-1)}$ and assign each result to the coefficient of twice the degree in ${g_{(i-1)}}^2$. Thus, term $a \cdot x^i$ of $g_{(i-1)}$ becomes $a^2 \cdot x^{2^i}$ in ${g_{(i-1)}}^2$. Note that all of the terms of ${g_{(i-1)}}^2$ are of even degree. So it is possible to compute ${g_{(i)}}^\dagger$ using (7.10) by splitting f^* into a polynomial f_e consisting of even degree terms of f^* and f_o consisting of the odd degree terms. Then ${g_{(i)}}^\dagger$ is computed by combining

$$ {g_{(i)}}_e^\dagger = (g_{(i-1)})_e - f_e \cdot {g_{(i-1)}}^2, \tag{7.12} $$

$$ {g_{(i)}}_o^\dagger = (g_{(i-1)})_o - f_o \cdot {g_{(i-1)}}^2. \tag{7.13} $$

It may helpful to use the transformation $y = x^2$ for generating the polynomials ${g_{(i)}}_e^\dagger$ and ${g_{(i)}}_o^\dagger$.

To complete the derivation of the improved Newton division method for finite fields, replace each application of the multiplicative FFT in the previous section with an additive FFT. The details are left to the reader.

Following an analysis similar to the previous section, Master Equation V can be used to show that at most $2.25 \cdot M_M(2 \cdot (\delta + 1)) + M_M(2n)$ multiplications and $2.25 \cdot A_M(2 \cdot (\delta + 1)) + A_M(2n) + n + \delta + 1$ additions are required for the Newton inversion. Since Gao's algorithm and the new additive FFT algorithm only work

for certain input sizes, the Wang-Zhu-Cantor algorithm should be used for the FFT-based multiplication to achieve the above results. [2] If $\delta = n - 1$, then the cost of dividing a polynomial of degree less than $2n - 1$ by a polynomial of degree less than n is roughly 3.25 times the cost of multiplying two polynomials of degree less than n.

7.5 Concluding remarks

This chapter summarized a number of existing techniques in the literature to compute division with remainder using Newton inversion. We also showed how the truncated Fast Fourier Transform can be used to obtain a new algorithm that can be used to complete the division of a polynomial of degree less than $2n$ by a polynomial of degree less than n in roughly 3.5 times the cost of a multiplication of two polynomials of degree less than n. When the coefficients are elements of a finite field of characteristic 2, the factor can be reduced to 3.25.

One needs to carefully analyze the degrees of the two input polynomials when deciding with division algorithm to use. The Newton division methods are better than classical division for large sizes when one polynomial has degree roughly twice that of the other. However, if the polynomials are of small degree or have roughly the same degree, then one should consider classical division over Newton division. Since other factors contibute to the cost of the algorithms (e.g. the time needed for the copies required for the reversal of polynomials in Newton division) and will vary from computer platform to computer platform, one should implement both algorithms and develop criteria to select the best algorithm to use based on the degrees of the input polynomials.

---

[2] This reduction in operations is due to the reuse of the FFT of $g_{(i-1)}$ in (7.12) and (7.13).

CHAPTER 8

THE EUCLIDEAN ALGORITHM

The Euclidean Algorithm is one of the oldest techniques in mathematics that is still used today. Invented around 400 B.C., its purpose is to find the "largest" element of a given algebraic structure that divides each of two or more inputs to the algorithm. This element is called a "greatest common divisor" or GCD. Although it was originally invented to operate on integers, mathematicians have since specified conditions whereby the algorithm can work on other algebraic structures. Mathematicians call a structure where the Euclidean Algorithm can be applied a Euclidean Domain. In this chapter, we will first assume that the algebraic structure is univariate polynomials with coefficients over any field $\mathbb{F}$ and will later restrict $\mathbb{F}$ to finite fields.

8.1 The Euclidean Algorithm

Let $a(x), b(x) \in \mathbb{F}[x]$ and define the greatest common divisor of a and b, $\gcd(a, b)$, to be the common divisor of a and b which has the greatest degree and is monic. If $a = 0$ and $b = 0$, then the greatest common divisor is defined to be 0.

A Euclidean Domain $\mathcal{D}$ is an algebraic structure for which division with remainder is possible. In other words, if a and b are two elements of $\mathcal{D}$ with $b \neq 0$, then there exist q and r in $\mathcal{D}$ such that $a = q \cdot b + r$ where either $r = 0$ or $\delta(r) < \delta(b)$. Here, δ is a function called the "norm" that maps the nonzero elements of $\mathcal{D}$ to the nonnegative integers. In the case of $\mathbb{F}[x]$, then [34] explains that δ is defined to map an element of $\mathbb{F}[x]$ to the degree of the polynomial and that q, r are unique in this case.

Let us now prove the two key facts that are used to find $\gcd(a, b)$.

Theorem 30 *Let $f(x)$ and $g(x)$ be two polynomials in $\mathbb{F}[x]$ and let $f = q \cdot g + r$ where $q(x)$ and $r(x)$ are the quotient and remainder of dividing f by g. Then* $\gcd(g, r) = \gcd(f, g)$.

Proof: Let $G(x) = \gcd(f, g)$. Then G is the monic polynomial of maximum degree such that G divides both f and g. Represent $f = G \cdot p_1$ and $g = G \cdot p_2$ for some polynomials $p_1(x)$ and $p_2(x)$. Then $r = f - q \cdot g = G \cdot p_2 - q \cdot (G \cdot p_1) = G \cdot (p_2 - q \cdot p_1)$. So G is a divisor of r.

Suppose that there exists a divisor $G'(x)$ of both g and r such that the degree of G' exceeds the degree of G. Since $f = q \cdot g + r$, then G' divides f. So G' is a divisor of both f and g with degree that exceeds G. But this contradicts the definition of greatest common divisor.

Thus, G must be a monic divisor of g and r with maximum degree. We now need to show that it is unique. Suppose that there exists another monic polynomial $G^\dagger(x)$ with degree equal to G such that $G^\dagger$ divides both g and r. Since G and $G^\dagger$ are both monic polynomials with equal degree, it follows that $G^\dagger = G$. Thus, G is the unique monic divisor of g and r with maximum degree, i.e. $\gcd(g, r) = G = \gcd(f, g)$.

□

Theorem 31 *Let $f(x)$ be a polynomial in $\mathbb{F}[x]$ and let $g(x) = 0$. Then* $\gcd(f, g) = \mathcal{N}(f)$ *where the normalization function $\mathcal{N}$ makes the input polynomial monic by dividing each term by its leading coefficient.*

Proof: Compute $f' = \mathcal{N}(f)$ by dividing each coefficient of f by its leading coefficient ρ. Since $g = 0 \cdot f'$ and $f = \rho \cdot f'$, then f' is a divisor of f and g with leading coefficient of 1. Since the degree of f is the same as the degree of f', there cannot exist a divisor of f with greater degree. Thus, f' is the greatest common divisor of f and g, i.e.

$\gcd(f, g) = \mathcal{N}(f)$. $\square$

The Euclidean Algorithm for polynomials determines $\gcd(a, b)$ by computing a sequence of remainders $\{r_1, r_2, \ldots, r_{\ell-1}, r_\ell = 0\}$ based on the previous two theorems. For initialization purposes, we will define $r_{-1}(x) = a$ and $r_0(x) = b$. Then for each $i > 0$, divide r_{i-2} by r_{i-1} and obtain remainder r_i. Because $\mathbb{F}[x]$ is a Euclidean Domain, then either the degree of r_i will be less than r_{i-1} or else $r_i = 0$. In the first case, we can use Theorem 30 to reduce the problem of computing $\gcd(a, b)$ to the problem of computing $\gcd(r_i, r_{i-1})$. In the second case (when $i = \ell$), then $\gcd(a, b) = \mathcal{N}(r_{\ell-1})$ by Theorem 31 and we are done. The number of times for which Theorem 30 must be applied before we obtain the desired GCD is called the "Euclidean length" and is denoted by ℓ above and throughout this chapter.

An algorithm can be constructed for computing $\gcd(a, b)$ by recursively applying this reduction step. Since the degree of r_i decreases by at least one on each application of the reduction step, then we must encounter the second case after at most $m + 1$ iterations of this process. Thus, the algorithm would compute $\gcd(a, b)$ in a finite amount of time. Pseudocode which implements the Euclidean Algorithm is given in Figure 8.1.

Let us compute the cost of this algorithm. Line 0 performs some initialization and is assumed to have no cost. Lines 1-4 perform most of the work of the algorithm and consists of ℓ iterations of lines 2 and 3. We will assume that line 2 requires no operations. If we let $n_i = \deg(r_i)$ for each i in $-1 \leq i \leq \ell$ and use classical division, then the cost of line 3 is $(n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})$ multiplications, $(n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})$ additions, plus one inversion in $\mathbb{F}$. However, if r_i is monic for any i, then an inversion is not needed in that step. Finally, line 5 consists of computing one inversion in $\mathbb{F}$,

Algorithm : Euclidean Algorithm
Input: Polynomials $a(x), b(x) \in F[x]$ where $n = \deg(a)$ and $m = \deg(b) < n$.
Output: $G(x) = \gcd(a, b)$, i.e. the monic polynomial of greatest degree such that G divides a and G divides b.
0. Set $r_{-1}(x) = a$; Set $r_0(x) = b$; Set $i = 0$. 1. **While** $r_i \neq 0$ 2. Increment i. 3. Set $r_i = r_{i-2} \bmod r_{i-1}$. 4. **End While** 5. Return $G = \mathcal{N}(r_{i-1})$.

Figure 8.1 Pseudocode for Euclidean Algorithm

plus at most m multiplications by the inverse of the leading coefficient to normalize r_{i-1}.

Cumulating the operation counts for all of the applications of Theorem 30, we obtain a total of

$$\sum_{i=1}^{\ell} ((n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})) + m \tag{8.1}$$

multiplications,

$$\sum_{i=1}^{\ell} (n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1}) \tag{8.2}$$

additions, plus at most $\ell + 1$ inversions in $\mathbb{F}$.

For most applications of Theorem 30 in the Euclidean Algorithm, the division with remainder produces a quotient of degree 1 and a remainder of degree $n_i = n_{i-1} -$

1. In these cases, then the division only requires $2 \cdot (n_{i-1} + 1)$ multiplications and $2 \cdot (n_{i-1})$ additions, plus at most one inversion. Since the divisions are essentially linear-time operations using classical division, there is no advantage to applying Newton division in these cases.

The situation where $n_i = n_{i-1} - 1$ for all i in $1 \leq i \leq \ell$ is called the "normal case" by mathematicians. Here, the number of multiplications needed to implement the Euclidean Algorithm is given by

$$(n - m + 1) \cdot m + \sum_{i=2}^{\ell=m+1} (2 \cdot (m - i + 1)) + m \quad = \quad nm + m, \tag{8.3}$$

and the number of additions / subtractions is given by nm using a similar calculation. Since $m = n - k$ for some $k > 0$, then the number of multiplications is given by $n^2 - n \cdot k + n - k - 1$ and the number of additions is given by $n^2 - n \cdot k - 1$. In this case, the Euclidean Algorithm is $\Theta(n^2)$.

In [34], a proof is given that no other remainder sequence requires more operations than the "normal case". In other words, if $n_i < n_{i-1} - 1$ for one or more i, then the number of operations needed to implement the Euclidean Algorithm is bounded above by the number of operations needed for the normal case remainder sequence. Thus, the Euclidean Algorithm requires $\Theta(n^2)$ operations for any input polynomials a and b.

8.2 The Extended Euclidean Algorithm

The Extended Euclidean Algorithm not only computes $\gcd(a, b)$, but performs additional computations to express the result as a linear combination of a and b. That

is to say, if $G(x) = \gcd(a, b)$, then we will also find polynomials $u(x)$ and $v(x)$ such that

$$G = u \cdot a + v \cdot b. \tag{8.4}$$

We are going to express each intermediate result of the Euclidean Algorithm, $r_i(x)$, as a linear combination of a and b. Thus, we will find polynomials u_i and v_i such that

$$r_i = u_i \cdot a + v_i \cdot b. \tag{8.5}$$

Since the Euclidean Algorithm is initialized with $r_{-1}(x) = a$ and $r_0(x) = b$, then clearly $u_{-1} = 1$, $v_{-1} = 0$, $u_0 = 0$, and $v_0 = 1$. In matrix form, the linear combinations for the input polynomials can be expressed as

$$\begin{pmatrix} r_{-1} \\ r_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix}. \tag{8.6}$$

Now let us compute $r_1(x)$ using $r_1 = r_{-1} - q_1 \cdot r_0$ where $q_1(x)$ is the quotient that results when r_{-1} is divided by r_0. We can obtain the vector $(r_0, r_1)^T$ by using the linear transformation

$$\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix} \cdot \begin{pmatrix} r_{-1} \\ r_0 \end{pmatrix} \tag{8.7}$$
$$= \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} a \\ b \end{pmatrix}.$$

Let us define the matrices Q_i by

$$Q_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \tag{8.8}$$
$$Q_i = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \qquad \text{for all } i > 0, \tag{8.9}$$

and then define $P_{U,L}$ as

$$P_{U,L} = Q_U \cdot Q_{U-1} \cdot \cdots \cdot Q_{L+1} \cdot Q_L. \tag{8.10}$$

The Extended Euclidean Algorithm is based on the following results:

Lemma 32 *Select any i in $1 \leq i \leq \ell$ and let $P_{i,0}$ be defined as in (8.10). Then*

$$P_{i,0} = Q_i \cdot P_{i-1,0}. \tag{8.11}$$

Proof: Using the definition of $P_{i,0}$ as given by (8.10),

$$\begin{aligned} P_{i,0} &= Q_i \cdot Q_{i-1} \cdot \cdots \cdot Q_1 \cdot Q_0 \\ &= Q_i \cdot (Q_{i-1} \cdot Q_{i-2} \cdot \cdots \cdot Q_1 \cdot Q_0) \\ &= Q_i \cdot P_{i-1,0}. \end{aligned} \tag{8.12}$$

□

Theorem 33 *Select any i in $1 \leq i \leq \ell$ and let $P_{i,0}$ be defined as in (8.10). Then*

$$\begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = P_{i,0} \cdot \begin{pmatrix} a \\ b \end{pmatrix}. \tag{8.13}$$

Proof: We will prove this result by induction. The case $i = 1$ is proven by (8.6). Suppose that the result holds for $i = \kappa$ where $\kappa \geq 1$. We need to show that the result holds for $i = \kappa + 1$.

By the definition of $P_{i,0}$ as defined as in (8.10) and Lemma 32,

$$
\begin{aligned}
P_{\kappa+1,0} \cdot \begin{pmatrix} a \\ b \end{pmatrix} &= Q_{\kappa+1} \cdot P_{\kappa,0} \cdot \begin{pmatrix} a \\ b \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1 \\ 1 & -q_{\kappa+1} \end{pmatrix} \cdot \begin{pmatrix} r_{\kappa-1} \\ r_{\kappa} \end{pmatrix} \\
&= \begin{pmatrix} r_{\kappa} \\ r_{\kappa-1} - q_{\kappa+1} \cdot r_{\kappa} \end{pmatrix} \\
&= \begin{pmatrix} r_{\kappa} \\ r_{\kappa+1} \end{pmatrix}.
\end{aligned} \tag{8.14}
$$

Thus, the theorem is proven by mathematical induction. □

Corollary 34 *Select any i in $1 \leq i \leq \ell$ and let $P_{i,0}$ be defined as in (8.10). Then*

$$
P_{i,0} = \begin{pmatrix} u_{i-1} & v_{i-1} \\ u_i & v_i \end{pmatrix}. \tag{8.15}
$$

Proof: Since $P_{i,0}$ is a matrix that transforms $(a, b)^T$ into $(r_{i-1}, r_i)^T$ by Theorem 33, then the corollary is proven by (8.5). □

Corollary 35 *For any i in $0 \leq i \leq \ell$, then*

$$P_{i,0} = \begin{pmatrix} u_{i-1} & v_{i-1} \\ u_i & v_i \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \cdot \begin{pmatrix} u_{i-2} & v_{i-2} \\ u_{i-1} & v_{i-1} \end{pmatrix}. \tag{8.16}$$

Proof: By Lemma 32, $P_{i,0} = Q_i \cdot P_{i-1,0}$. Substituting (8.9) and (8.15) into this equation yields the result. □

Corollary 36 *Select any i in $0 \leq i \leq \ell$ and let Q_i be defined as in (8.9). Then*

$$\begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = Q_i \cdot \begin{pmatrix} r_{i-2} \\ r_{i-1} \end{pmatrix}. \tag{8.17}$$

Proof: By Lemma 32 and Theorem 33,

$$\begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = P_{i,0} \cdot \begin{pmatrix} a \\ b \end{pmatrix} = Q_i \cdot P_{i-1,0} \cdot \begin{pmatrix} a \\ b \end{pmatrix} = Q_i \cdot \begin{pmatrix} r_{i-2} \\ r_{i-1} \end{pmatrix}. \tag{8.18}$$

□

The Extended Euclidean Algorithm is initialized with $r_{-1} = a$, $r_0 = b$, and $P_{0,0}$ (the 2×2 identity matrix). On each reduction step i where $i > 0$, then r_{i-2} is divided by r_{i-1} to produce a quotient q_i and remainder r_i. In theory, Theorem 33 could be

used to compute r_i. However, Corollary 36 shows how this result could be computed using one reduction step of the Euclidean Algorithm. Based on the material covered so far, the polynomials involved with Theorem 33 are larger than the polynomials involved with Corollary 36 and so it is more efficient to simply apply a reduction step of the Euclidean Algorithm to obtain r_i.

The Extended Euclidean Algorithm also computes the polynomials $u_i(x)$ and $v_i(x)$ that express r_i as a linear combination of a and b. To do so, we will first construct Q_i using (8.9) and then compute $P_{i,0}$ using (8.16). In practice, only the second row of $P_{i,0}$ actually needs to be computed as the first row can be obtained by simply copying the second row of $P_{i-1,0}$. It can be easily verified that the formulas

$$u_i = u_{i-2} - q_i \cdot u_{i-1}, \tag{8.19}$$

$$v_i = v_{i-2} - q_i \cdot v_{i-1} \tag{8.20}$$

are equivalent to these matrix computations. The following theorem gives the degrees of each of these polynomials.

Theorem 37 *For all i in $1 \leq i \leq \ell$,*

$$\deg(u_i) = \deg(b) - n_{i-1}, \tag{8.21}$$

$$\deg(v_i) = \deg(a) - n_{i-1}, \tag{8.22}$$

where $n_{-1} = \deg(a)$, $n_0 = \deg(b)$, and $n_j = \deg(r_j)$ for all $1 \leq j \leq \ell$.

Proof: We will prove the second part of the theorem by the second principle of mathematical induction. [1] Observe that by (8.19), $v_i = v_{i-2} - q_i \cdot v_{i-1}$ for all $1 \leq i \leq \ell$. For $i = 1$, then $v_1 = v_{-1} - q_1 \cdot v_0 = -q_1$. Thus, $\deg(v_1) = \deg(q_1) = \deg(a) - \deg(b) = \deg(a) - n_0$. For $i = 2$, then $v_2 = v_0 - q_2 \cdot v_1 = 1 + q_1 \cdot q_2$. Thus, $\deg(v_2) = \deg(q_1) + \deg(q_2) = (\deg(a) - n_0) + (n_0 - n_1) = \deg(a) - n_1$. Assume that the result holds for $i = \kappa - 1$ and $i = \kappa$ where $2 \leq k < \ell$. We need to show that the result holds for $i = \kappa + 1$. Since the degrees of the remainders of the Euclidean Algorithm is a strictly decreasing sequence, then $n_{\kappa-1} > n_\kappa$. Now by the inductive hypothesis, $\deg(v_{\kappa-1}) = \deg(a) - n_{\kappa-1} < \deg(a) - n_\kappa = \deg(v_\kappa)$. Then clearly $\deg(v_{\kappa-1}) < \deg(q_{\kappa+1}) + \deg(v_\kappa) = \deg(q_{\kappa+1} \cdot v_\kappa)$ and thus,

$$\begin{aligned} \deg(v_{\kappa+1}) &= \deg(v_{\kappa-1} - q_{\kappa+1} \cdot v_\kappa) \\ &= \deg(q_{\kappa+1}) + \deg(v_\kappa) \\ &= (n_{\kappa-1} - n_\kappa) + (\deg(a) - n_{\kappa-1}) \\ &= \deg(a) - n_\kappa. \end{aligned} \tag{8.23}$$

So, the second part of the theorem is proven by the second principle of mathematical induction. A slightly simpler proof used the same approach can be used to establish the first part of the theorem. □

Since the Extended Euclidean Algorithm is essentially the Euclidean Algorithm with some additional computations, there will again exist an $\ell > 0$ such that

[1] The second part was chosen because this particular result will be used in Chapter 9.

Algorithm : Extended Euclidean Algorithm
Input: Polynomials $a(x), b(x) \in F[x]$ where $n = \deg(a)$ and $m = \deg(b) < n$.
Output: $G(x) = \gcd(a, b)$, i.e. the monic polynomial of greatest degree such that G divides a and G divides b. Also, $u(x)$, $v(x)$ such that $G = u \cdot a + v \cdot b$.
0. Set $r_{-1}(x) = a$, $u_{-1}(x) = 1$, $v_{-1}(x) = 0$. Set $r_0(x) = b$, $u_0(x) = 0$, $v_0(x) = 1$. Set $i = 0$. 1. **While** $r_i \neq 0$ 2. Increment i. 3. Set $q_i = r_{i-2} \operatorname{div} r_{i-1}$, $r_i = r_{i-2} \bmod r_{i-1}$. 4. Set $u_i = u_{i-2} - q_i \cdot u_{i-1}$. 5. Set $v_i = v_{i-2} - q_i \cdot v_{i-1}$. 6. **End While** 7. Let ρ be the leading coefficient of r_{i-1}. 8. Return $G(x) = r_{i-1}/\rho = \mathcal{N}(r_{i-1})$, $u(x) = u_{i-1}/\rho$, $v(x) = v_{i-1}/\rho$.

Figure 8.2 Pseudocode for Extended Euclidean Algorithm

$r_\ell = 0$. When this occurs, the Extended Euclidean Algorithm will terminate and the desired greatest common divisor will be determined by Theorem 31.

Pseudocode for the Extended Euclidean Algorithm is provided in Figure 8.2. Let us compute the cost of this algorithm. Line 0 performs some initialization and is assumed to have no cost. Lines 1-6 perform most of the work of the algorithm and consists of ℓ iterations of lines 2-5. We will assume that line 2 requires no operations. Line 3 is just one reduction step of the Euclidean Algorithm and requires $(n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})$ multiplications, $(n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})$ additions, plus one inversion in $\mathbb{F}$ from the discussion in the previous section. Again, if r_i is monic for any i, then an inversion is not needed in that step. Note that q_i is obtained as part of this calculation at no additional cost. If $i = 1$, then lines 4 and 5 can be computed at no cost. Otherwise, Theorem 37 can be used to show that the number of operations needed to implement line 4 is $(n_{i-2} - n_{i-1} + 1) \cdot (m - n_{i-2} + 1)$ multiplications and

$(n_{i-2}-n_{i-1})\cdot(m-n_{i-2}+1)+(m-n_{i-1})$ additions, assuming that classical multiplication is used in this step. Similarly, line 5 requires $(n_{i-2} - n_{i-1} + 1) \cdot (n - n_{i-2} + 1)$ multiplications and $(n_{i-2} - n_{i-1}) \cdot (n - n_{i-2} + 1) + (n - n_{i-1})$ additions. After the loop is completed, line 7 is assumed to require no operations and line 8 requires 1 inversion of ρ, plus at most $n + m - 1$ multiplications.

Cumulating the operation counts for all values of i in the algorithm, we obtain a total of

$$\begin{aligned} & \sum_{i=1}^{\ell} ((n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})) \\ + & \sum_{i=2}^{\ell} ((n_{i-2} - n_{i-1} + 1) \cdot (m - n_{i-2} + 1)) \\ + & \sum_{i=2}^{\ell} ((n_{i-2} - n_{i-1} + 1) \cdot (n - n_{i-2} + 1)) \\ + & n + m - 1 \end{aligned} \tag{8.24}$$

multiplications,

$$\begin{aligned} & \sum_{i=1}^{\ell} ((n_{i-2} - n_{i-1} + 1) \cdot (n_{i-1})) \\ + & \sum_{i=2}^{\ell} ((n_{i-2} - n_{i-1}) \cdot (m - n_{i-2} + 1) + (m - n_{i-1})) \\ + & \sum_{i=2}^{\ell} ((n_{i-2} - n_{i-1}) \cdot (n - n_{i-2} + 1) + (n - n_{i-1})) \end{aligned} \tag{8.25}$$

additions, plus at most $\ell + 1$ inversions in $\mathbb{F}$.

For the normal case, $n_{-1} = n$, and $n_i = m - i$ for all $i \geq 0$. In this case, the total number of operations is given by

$$\begin{aligned} & (n - m + 1) \cdot m + 2 \cdot \sum_{i=2}^{\ell} (n + i - 1) + n + m - 1 \qquad (8.26) \\ = \; & 3nm + 3m + n - 1 \end{aligned}$$

multiplications, and $3nm + 2m$ additions using a similar calculation. Since $m = n - k$ for some $k > 0$, then the number of each of these operations is $\Theta(n^2)$.

Note that if one does not need the greatest common divisor expressed as a linear combination of a and b, then the Euclidean Algorithm should be used instead of the Extended Euclidean Algorithm since less work is required. One situation where the Extended Euclidean Algorithm is useful is when $\gcd(a, b) = 1$ and we desire to compute b^{-1} modulo a. In this case $b^{-1} = v_{\ell-1}$. Note that we can compute this result by omitting line 4 from the Extended Euclidean Algorithm and can deduct $m^2 + m$ multiplications and additions each from the above operation counts.

8.3 Normalized Extended Euclidean Algorithm

In [34], von zur Gathen and Gerhard argue that when the Extended Euclidean Algorithm is applied to polynomials with coefficients in the field of rational numbers, the coefficients involved in the intermediate calculations have huge numerators and denominators even for inputs of moderate size. In the text, the authors demonstrate that the Extended Euclidean Algorithm works much better in this case if each $r_i(x)$ is normalized after each division step. The cost of the normalization is mostly compensated by a reduction in the operations needed for the polynomial divisions in the algorithm since the leading coefficient of the divisor is 1.

For the rest of this chapter, we will assume that the coefficients of the polynomials are from a field such that there is no advantage to normalizing each intermediate result of the Euclidean Algorithm. The interested reader can refer to [34] to learn how to adapt the techniques discussed in the remainder of this chapter to the case where the result of each division step is normalized.

8.4 The Fast Euclidean Algorithm

In this section, we will present an asymptotically faster algorithm for computing the greatest common divisor based on ideas introduced in [8] and [56]. The algorithm here is based on one found in [34] which more clearly presents these ideas and corrects some problems associated with these earlier algorithms. To simplify the presentation, the algorithm in this section is based on the Extended Euclidean Algorithm, whereas the algorithm in [34] is based on the Normalized Extended Euclidean Algorithm. Additionally, the algorithm presented in this section introduces some improvements to the algorithm resulting from the present author's solutions to some of the exercises proposed in [34].

Suppose that some polynomial f in $\mathbb{F}[x]$ is divided by another polynomial g in $\mathbb{F}[x]$ to obtain a quotient q and remainder r. The following theorem shows that obtaining this quotient only depends on the upper coefficients of f and g.

Theorem 38 *Let f be a polynomial of degree n_f in $\mathbb{F}[x]$ and let g be a polynomial of degree $n_g < n_f$ in $\mathbb{F}[x]$. The quotient $q = f \operatorname{div} g$ can be computed using only the terms of f and g with degree $n_f - \mathbb{k}$ or higher where $\mathbb{k}$ is any integer such that $2 \cdot deg(q) \leq \mathbb{k} \leq n_f$.*

Proof: Select $f, g \in F[x]$ where f has degree n_f and g has degree n_g and $n_f > n_g$. The desired quotient $f \operatorname{div} g$ is the unique polynomial q that satisfies

$$f = q \cdot g + r \tag{8.27}$$

and has degree $n_f - n_g$. Select any integer $\Bbbk$ that satisfies $2 \cdot \deg(q) \leq \Bbbk \leq n_f$. Next, partition each of f and g into two blocks such that

$$f = f_A \cdot x^{n_f - \Bbbk} + f_B, \tag{8.28}$$

$$g = g_A \cdot x^{n_f - \Bbbk} + g_B. \tag{8.29}$$

Here, f_B and g_B are polynomials of degree less than $n_f - \Bbbk$, f_A has degree less than $\Bbbk$, and g_A has degree less than $\Bbbk - (n_f - n_g)$. Now divide f_A by g_A to obtain quotient q^* and remainder r^*, i.e.

$$f_A = q^* \cdot g_A + r^* \tag{8.30}$$

By substituting (8.28) and (8.29) into (8.27), multiplying (8.30) by $x^{n_f - \Bbbk}$, and subtracting the results, we obtain

$$f_B = (q - q^*) \cdot g_A \cdot x^{n_f - \Bbbk} + q \cdot g_B + (r - r^* \cdot x^{n_f - \Bbbk}). \tag{8.31}$$

Now,

$$\begin{aligned}
\deg(f_B) &< n_f - \Bbbk \leq n_f - 2 \cdot \deg(q) \leq 2 \cdot n_g - n_f &< n_g, \\
\deg(q \cdot g_B) &< \deg(q) + n_f - \Bbbk \leq \deg(q) + 2 \cdot n_g - n_f & \\
&= \deg(q) + 2 \cdot n_g - (n_g + \deg(q)) &= n_g, \\
\deg(r) &< &n_g, \\
\deg(r^* \cdot x^{n_f - \Bbbk}) &< (\Bbbk - (n_f - n_g)) + (n_f - \Bbbk) &= n_g.
\end{aligned} \tag{8.32}$$

Thus, $(q - q^*) \cdot g_A \cdot x^{n_f - \Bbbk}$ must have degree less than $n_g < n_f - \Bbbk$. The only way that this can occur is for $q - q^* = 0$, i.e. $q = q^*$. Thus, the quotient of f divided by g is the same result as the quotient of f_A divided by g_A. In other words, the quotient can be computed using only the terms of f and g with degree $n_f - \Bbbk$ or higher. □

Corollary 39 *The coefficients of* $r = f \bmod g$ *with degree* $n_f - \Bbbk + deg(q)$ *or higher are the same as the coefficients of* $r^* = f_A \bmod g_A$ *with degree* $deg(q)$ *or higher.*

Proof: Since $q = q^*$, then (8.31) becomes

$$f_B = q \cdot g_B + (r - r^* \cdot x^{n_f - \Bbbk}). \tag{8.33}$$

Since $\deg(f_B) < n_f - \Bbbk$ and $\deg(q \cdot g_B) < \deg(q) + n_f - \Bbbk$, then $\deg(r - r^* \cdot x^{n_f - \Bbbk}) < \deg(q) + n_f - \Bbbk$. Thus, the coefficients of r with degree $n_f - \Bbbk + \deg(q)$ or higher are the same as the coefficients of $r^* \cdot x^{n_f - \Bbbk}$ with degree $n_f - \Bbbk + \deg(q)$ or higher and the coefficients of r^* with degree $\deg(q)$ or higher. □

These two results can be used to improve the Euclidean Algorithm.

Theorem 40 *Let* $\{r_1, r_2, r_3, \ldots, r_\ell\}$ *be the remainder sequence produced by the Extended Euclidean Algorithm to compute* $\gcd(a, b)$ *and let* $\{q_1, q_2, q_3, \ldots, q_\ell\}$ *be the associated quotient sequence. Select* $k \leq n$ *where* $n = \deg(a)$. *Then the coefficients of degree* $max\{n - 2k, 0\}$ *or higher of* a *and* b *can be used to compute* M *such that*

$$\begin{pmatrix} r_{s-1} \\ r_s \end{pmatrix} = M \cdot \begin{pmatrix} a \\ b \end{pmatrix}, \tag{8.34}$$

where $\deg(q_1) + \deg(q_2) + \deg(q_3) + \cdots + \deg(q_s) \leq k$.

Proof: Suppose that one uses the Extended Euclidean Algorithm to determine $\gcd(a, b)$ with the remainder sequence $\{r_1, r_2, r_3, \ldots, r_\ell\}$ and the quotient sequence $\{q_1, q_2, q_3, \ldots, q_\ell\}$. Select any $k < \deg(a)$ and any $s \leq \ell$ such that $\deg(q_1) + \deg(q_2) + \deg(q_3) + \cdots + \deg(q_s) \leq k$. We desire to compute $M = P_{s,0}$. Clearly, we can use the Extended Euclidean Algorithm to compute $P_{i,0}$ for any $i \leq \ell$. So, if $n - 2k \leq 0$, then use all of the coefficients of a and b to compute $M = P_{s,0}$.

Assume that $n - 2k > 0$. Let $f = r_{-1} = a$ and $g = r_0 = b$. Now subdivide f and g into two blocks as follows: $f = f_A \cdot x^{n-2k} + f_B$ and $g = g_A \cdot x^{n-2k} + g_B$. Here, $\deg(f_A) = 2k$ and $\deg(g_A) = 2k - \deg(q_1)$. Compute $q_1^* = f_A \operatorname{div} g_A$. By Theorem 38, $q_1 = q_1^*$. Then compute $r_1^* = f_A - q_1 \cdot g_A$ and observe that $\deg(r_1^*)$ has degree $2k - \deg(q_1) - \deg(q_2)$.

We will now inductively compute the sequence of polynomials $\{q_2, q_3, \ldots, q_s\}$. Choose some i that satisfies $2 \leq i \leq s$ and assume that $\{r_{-1}^*, r_0^*, \ldots, r_{i-1}^*\}$ are known. Let $f = r_{i-2}^*$ and $g = r_{i-1}^*$. If $D = \deg(q_1) + \cdots + \deg(q_{i-1})$, then $\deg(f) = 2k - D$, $\deg(g) = 2k - (D + \deg(q_i))$, $\deg(r_{i-2}) = n - D$, and $\deg(r_{i-1}) = n - (D + \deg(q_i))$.

By repeated application of Corollary 39, the coefficients of r_{i-2} and r_{i-1} with degrees $(n-D)-(2k-D)+D = n-2k+D$ or higher are the same as the coefficients of f and g with degree D or higher. Now subdivide f and g into two blocks as follows: $f = f_A \cdot x^{n-2k+D} + f_B$ and $g = g_A \cdot x^{n-2k+D} + g_B$. Here, $\deg(f_A) = 2k-2D$ and $\deg(g_A) = 2k-2D-\deg(q_i)$. Since $\deg(q_1)+\deg(q_2)+\deg(q_3)+\cdots+\deg(q_i) \leq k$, then $2k-2D-2\cdot\deg(q_i) \geq 0$. Since $2k-2D \geq 2\cdot\deg(q_i)$, then is possible to compute $q_i^* = f_A \operatorname{div} g_A$ and $q_i = q_i^*$, proven as part of Theorem 38. Now compute $r_i^* = f_A - q_i \cdot g_A$ and observe that $\deg(r_i^*)$ has degree $n-(D+\deg(q_{i+1}))$.

Since $\{q_1^*, q_2^*, \ldots, q_s^*\}$ have been computed and $q_i = q_i^*$ for all i in $1 \leq i \leq s$, then it is possible to construct

$$M = \prod_{i=1}^{s} \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \tag{8.35}$$

using only the terms of a and b of degree $n-2k$ or higher by the above discussion.

In either case, we have computed $M = P_{s,0}$. By Theorem 33, then

$$\begin{pmatrix} r_{s-1} \\ r_s \end{pmatrix} = M \cdot \begin{pmatrix} a \\ b \end{pmatrix} \tag{8.36}$$

as desired. □

Corollary 41 *If $k < n/2$, $a = a_A + x^{n-2k} + a_B$, $b = b_A \cdot x^{n-2k} + b_B$, and $r_1^*, r_2^*, \ldots, r_{\ell'}^*$ is the remainder sequence of $\gcd(a_A, b_A)$, then*

$$\begin{pmatrix} r_{s-1} \\ r_s \end{pmatrix} = \begin{pmatrix} r^*_{s-1} \\ r^*_s \end{pmatrix} \cdot x^{n-2k} + M \cdot \begin{pmatrix} a_B \\ b_B \end{pmatrix}. \tag{8.37}$$

Proof: Using the notation from Theorem 40, the computation of $\gcd(a_A, b_A)$ has quotient sequence $\{q^*_1, q^*_2, \cdots, q^*_{\ell'}\}$ and remainder sequence $\{r^*_1, r^*_2, \cdots, r^*_{\ell'}\}$. By Theorem 33,

$$\begin{pmatrix} r^*_{s-1} \\ r^*_s \end{pmatrix} = \prod_{i=1}^{s} \begin{pmatrix} 0 & 1 \\ 1 & -q^*_i \end{pmatrix} \cdot \begin{pmatrix} a_A \\ b_A \end{pmatrix}. \tag{8.38}$$

Since $q^*_i = q_i$ for all $1 \leq i \leq s$, then

$$\prod_{i=1}^{s} \begin{pmatrix} 0 & 1 \\ 1 & -q^*_i \end{pmatrix} = \prod_{i=1}^{s} \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} = M. \tag{8.39}$$

Thus,

$$\begin{pmatrix} r^*_{s-1} \\ r^*_s \end{pmatrix} = M \cdot \begin{pmatrix} a_A \\ b_A \end{pmatrix}. \tag{8.40}$$

Since

$$\begin{aligned}
\begin{pmatrix} r_{s-1} \\ r_s \end{pmatrix} &= M \cdot \begin{pmatrix} a \\ b \end{pmatrix} \\
&= M \cdot \begin{pmatrix} a_A \cdot x^{n-2k} + a_B \\ b_A \cdot x^{n-2k} + b_B \end{pmatrix} \\
&= M \cdot \begin{pmatrix} a_A \\ b_A \end{pmatrix} \cdot x^{n-2k} + M \cdot \begin{pmatrix} a_B \\ b_B \end{pmatrix} \\
&= \begin{pmatrix} r^*_{s-1} \\ r^*_s \end{pmatrix} \cdot x^{n-2k} + M \cdot \begin{pmatrix} a_B \\ b_B \end{pmatrix},
\end{aligned} \tag{8.41}$$

then the result has been proven. □

Suppose that we are given the polynomials a with degree n, b with degree m, and some parameter $k \leq n$. The Fast Euclidean Algorithm computes $(r_{s-1}, r_s)^T$ and the matrix

$$M = \begin{pmatrix} u_{s-1} & v_{s-1} \\ u_s & v_s \end{pmatrix} \tag{8.42}$$

where s is the largest integer such that $\deg(q_1) + \deg(q_2) + \deg(q_3) + \cdots + \deg(q_s) \leq k$. If $k = n$, then M gives the result after all ℓ steps of the Extended Euclidean Algorithm and $\gcd(a, b) = \mathcal{N}(r_{\ell-1})$. The basic idea of the Fast Euclidean Algorithm is to partition the s division steps of the Extended Euclidean Algorithm into two subproblems, each of which is solved through a recursive call to the Fast Euclidean Algorithm. For small values of k, then Theorem 38 will not apply and the Fast

Euclidean Algorithm will essentially solve the problem through one or more division steps of the Extended Euclidean Algorithm.

Prior to the computation of the first subproblem, we first choose some parameter $k_1 \leq k$. If $k_1/2 \leq n$, then split a and b into two blocks using $a = a_A \cdot x^{n-2k_1} + a_B$ and $b = b_A \cdot x^{n-2k_1} + b_B$. Otherwise, let $a_A = a$ and $b_A = b$. In either case, we now recursively call the Fast Euclidean Algorithm with a_A, b_A, and parameter k_1. If the Extended Euclidean Algorithm is used to compute $\gcd(a_A, b_A)$, we would obtain quotient sequence $\{q_1^*, q_2^*, \ldots, q_{\ell'}^*\}$ [2] and remainder sequence $\{r_1^*, r_2^*, \ldots, r_{\ell'}^*\}$ where $\ell' \leq \ell$. Instead, the recursive call to the Fast Euclidean Algorithm will compute α division steps of $\gcd(a_A, b_A)$ where $\deg(q_1) + \deg(q_2) + \deg(q_3) + \cdots + \deg(q_\alpha) = d_1 \leq k_1$ and $\alpha < s$. The algorithm will return $(r_{\alpha-1}^*, r_\alpha^*)^T$ and the matrix

$$R = \begin{pmatrix} u_{\alpha-1} & v_{\alpha-1} \\ u_\alpha & v_\alpha \end{pmatrix}. \tag{8.43}$$

If $a_A = a$ and $b_B = b$, then $(r_{\alpha-1}, r_\alpha)^T = (r_{\alpha-1}^*, r_\alpha^*)^T$. Otherwise, Corollary 41 can be used to efficiently compute $(r_{\alpha-1}, r_\alpha)^T$.

At this point, it may not be possible to compute any additional division steps in the computation of $\gcd(a, b)$. If this is the case, then set $s = \alpha$ and return $(r_{s-1}, r_s)^T = (r_{\alpha-1}, r_\alpha)^T$ along with matrix $M = R$. Otherwise, one division step of the Extended Euclidean Algorithm will be performed to compute $Q = Q_{\alpha+1}$. In this case, then set $\beta = \alpha + 1$, and $d' = d_1 + \deg(q_\beta)$. If the division step was not implemented then set Q to be the 2×2 identity matrix, $\beta = \alpha$, and $d' = d_1$. In either event,

[2] Recall that $q_i = q_i^*$ for all i in $1 \leq i \leq \ell'$

$$P_{\beta,0} = Q \cdot R \tag{8.44}$$

and we will set $f = r_{\beta-1}$, $g = r_\beta$, n' to be the degree of f, and the parameter to the second subproblem to be $k_2 = k - d'$. If $k_2/2 \leq n'$, then split f and g into two blocks using $f = f_A \cdot x^{n'-2k_2} + f_B$ and $g = g_A \cdot x^{n'-2k_2} + g_B$. Otherwise, let $f_A = f$ and $g_A = g$. In either case, we now recursively call the Fast Euclidean Algorithm with f_A, g_A, and parameter k_2. If the Extended Euclidean Algorithm is used to compute $\gcd(f_A, g_A)$, we would obtain quotient sequence $\{q^*_{\beta+1}, q^*_{\beta+2}, \ldots, q^*_\ell\}$ and remainder sequence $\{r^*_{\beta+1}, r^*_{\beta+2}, \ldots, r^*_\ell\}$. Instead, the recursive call to the Fast Euclidean Algorithm will compute $\gamma - \beta$ division steps of $\gcd(f_A, g_A)$ where $\deg(q_{\beta+1}) + \deg(q_{\beta+2}) + \deg(q_{\beta+3}) + \cdots + \deg(q_\gamma) = d_2 < k_2$. The algorithm will return $(r^*_{\gamma-1}, r^*_\gamma)^T$ and the matrix $S = P_{\gamma,\beta+1}$. If $f_A = f$, then $(r_{\gamma-1}, r_\gamma)^T = (r^*_{\gamma-1}, r^*_\gamma)^T$. Otherwise, Corollary 41 can be used to efficiently compute $(r_{\gamma-1}, r_\gamma)^T$.

It remains to determine $M = P_{s,0}$ where $s = \gamma$. Since $P_{\beta,0} = Q \cdot R$ by (8.44), then

$$\begin{aligned} M &= P_{\gamma,0} = Q_\gamma \cdot Q_{\gamma-1} \cdot \cdots \cdot Q_1 \cdot Q_0 \\ &= (Q_\gamma \cdot Q_{\gamma-1} \cdot \cdots \cdot Q_{\beta+2} \cdot Q_{\beta+1}) \cdot (Q_\beta \cdot Q_{\beta-1} \cdot \cdots \cdot Q_1 \cdot Q_0) \\ &= P_{\gamma,\beta+1} \cdot P_{\beta,0} \\ &= S \cdot Q \cdot R. \end{aligned} \tag{8.45}$$

Once M has been computed, the algorithm returns $(r_{\gamma-1}, r_\gamma)^T$ and M.

The number of division steps that a particular call to the Fast Euclidean Algorithm can implement is a function of k, not a function of the sizes of the polynomials passed to the algorithm. If we wish to subdivide a problem with parameter k into two subproblems of roughly equal size, then we should set $k_1 = \lfloor k/2 \rfloor$ and clearly the first subproblem has parameter less than or equal to $k/2$. If $d_1 = k/2$, then the second subproblem has parameter $k_2 = k - k/2 = k/2$ and a problem with parameter k has been subdivided into two subproblems with parameter $k/2$. If k is even and the quotient sequence of the Extended Euclidean Algorithm is normal, i.e. $\deg(q_i) = 1$ for all $1 \leq i \leq \ell$, then this will be indeed be the case. We will now discuss three different strategies for subdividing a problem in the general case.

The method discussed in [34] always performs one step of the Extended Euclidean Algorithm before starting the second subproblem. Since $d_1 + \deg(q_{\alpha+1}) > k/2$, then the second subproblem has parameter $k_2 = k - (d_1 + \deg(q_{\alpha+1})) < k/2$. Thus, a problem with parameter k is subdivided into two problems with parameter less than or equal to $k/2$.

Two improvements to this strategy will now be considered. In first improvement, the step of the Extended Euclidean Algorithm is omitted for those cases where $d_1 = k/2$. Here, a problem with parameter k can still be subdivided into two subproblems with parameter $k/2$, but $q_{\alpha+1}$ is now computed with the second subproblem. Hopefully, some of the lower coefficients $r_{\alpha-1}$ and r_α can be omitted from this computation using Theorem 38 and some operations can be saved in these cases. In this strategy, whenever $d_1 < k/2$, we still explicitly perform one step of the Extended Euclidean Algorithm as with the strategy of [34].

For the sake of developing a closed-form formula for the operation counts of the algorithm, it is desirable to reduce a problem with parameter k into two subproblems with parameter $k/2$ or less. However, if the goal of the algorithm is to compute

the greatest common divisor with the least number of operations, we should never explicitly perform one step of the Extended Euclidean Algorithm unless absolutely necessary. The second improvement to the strategy is to put the computation of $q_{\alpha+1}$ in the second subproblem as often as possible using the condition to omit the explicit step of the Extended Euclidean Algorithm when $\lfloor (k-d_1)/2 \rfloor < n_{\alpha-1} - n_\alpha$. If the Extended Euclidean Algorithm division step is omitted, then the second subproblem will have parameter $k - d_1$. In the subproblem, the Fast Euclidean Algorithm is called again with parameter $\lfloor (k-d_1)/2 \rfloor$. At this point, if $\lfloor (k-d_1)/2 \rfloor < n_{\alpha-1} - n_\alpha$, then Theorem 38 cannot be applied and no division steps are possible with the reduced number of coefficients in the second subproblem. Thus, there was no advantage to skipping the explicit step of the Extended Euclidean Algorithm.

On the other hand, if $\lfloor (k-d_1)/2 \rfloor \geq n_{\alpha-1} - n_\alpha$, then we may gain some division steps in the second subproblem. If not, then still get another chance to apply the division step of the Extended Euclidean Algorithm, now with the condition $\lfloor (k-d_1)/4 \rfloor < n_{\alpha-1} - n_\alpha$. At some recursive call to the algorithm, we may be forced to explictly perform a step of the Extended Euclidean Algorithm, but hopefully we have achieved some division steps through some of the other subproblems before this occurs.

Pseudocode for the Fast Euclidean Algorithm is provided below in Figure 8.3. Note that the pseudocode is given for the case of the first improvement to the algorithm. If the strategy of [34] is employed, then line 6A should be omitted and line 6B should always be executed. If the second improvement is implemented, then the condition for executing line 6A should be changed to "If $(k-d_1)/2 < n_{\alpha-1} - n_\alpha$". The pseudocode was written with the first improvement because it allows closed-form operation count formulas to be developed which is not possible with the second improvement strategy.

To simplify the pseudocode, the cases where $n - k_1 < 0$ or $n' - k_2 < 0$ are not handled. Due to the selection of the two subproblem parameters, it should nearly always be possible to split the polynomial as shown in the pseudocode. However, if it is not possible to split the polynomial, then all of the polynomial coefficients should be used as input to the subproblem. In this case, it not necessary to use Corollary 41 to recover the desired result of the division steps.

Let us now compute an upper bound for the cost of the algorithm provided in the pseudocode. We will compute the number of multiplications, $M(k)$ and the number of additions, $A(k)$, each of which is a function of the algorithm parameter k. Lines 0 and 5 simply end the recursion when k is too small to compute any results and do not cost any operations. We will assume that lines 1 and 7 also cost no operations. Lines 2 and 8 simply partition a polynomial into two blocks, requiring no operations. Line 3 requires at most $M(\lfloor k/2 \rfloor)$ multiplications and $A(\lfloor k/2 \rfloor)$ additions. Since $k - d' \leq \lfloor k/2 \rfloor$, then line 9 similarly requires at most $M(\lfloor k/2 \rfloor)$ multiplications and $A(\lfloor k/2 \rfloor)$ additions. If the condition for line 6A is satisfied, then the algorithm proceeds to line 7 with no operations in $\mathbb{F}$. It remains to analyze lines 4, 6B, 10, and 11 where all of the actual computations take place in the algorithm.

In line 4, the degree of a_B and b_B is at most $k-1$, the degrees of $R_{1,1}$, $R_{1,2}$, and $R_{2,1}$ are at most $k/2 - 1$, and the degree of $R_{2,2}$ is at most $k/2$. By breaking a_B and b_B into two blocks of size at most $k/2$, then the cost for line 4 is at most $8 \cdot M_M(k) + k$ multiplications and $8 \cdot A_M(k) + 3k/2$ additions. [3] The extra operations in each of these counts is due to the degree k coefficient of $R_{2,2}$ which is explicitly multiplied by each coefficient b_B and combined with the other results. It can be shown that

[3] Here the notation $M_M(k)$ denotes the number of multiplications needed to multiply two polynomials of size $k/2$ into a polynomial of size k. Similarly, $A_M(k)$ denotes the number of additions needed to multiply two polynomials of size $k/2$ into a polynomial of size k.

Algorithm : Fast Euclidean Algorithm

Input: Polynomials $a(x), b(x) \in F[x]$ where $n = \deg(a)$
and $m = \deg(b) < n$; An integer k such that $n/2 \leq k \leq n$.

Output: $\begin{pmatrix} r_{s-1} \\ r_s \end{pmatrix}$ and $M = \begin{pmatrix} u_{s-1} & v_{s-1} \\ u_s & v_s \end{pmatrix}$,
i.e. the result of s steps of the Extended Euclidean Algorithm.

Here, $0 \leq s \leq \ell$ and $\deg(q_1) + \deg(q_2) + \cdots + \deg(q_s) \leq k$,
where $\{q_1, q_2, \ldots, q_\ell\}$ is the quotient sequence that
would be produced by the Extended Euclidean Algorithm
in the computation of $\gcd(a, b)$.
Note: $r_{-1} = a$ and $r_0 = b$.

0. If $b = 0$ or $k < n - m$, then return $\begin{pmatrix} r_{-1} \\ r_0 \end{pmatrix}$ and $M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.
1. Let $k_1 = \lfloor k/2 \rfloor$.
2. Split a and b into two blocks such that
   $a = a_A \cdot x^{n-2k_1} + a_B$ and $b = b_A \cdot x^{n-2k_1} + b_B$.
3. Recursively call the Fast Euclidean Algorithm with input a_A, b_A, and k_1
   and output $\begin{pmatrix} r^*_{\alpha-1} \\ r^*_\alpha \end{pmatrix}$ and $R = \begin{pmatrix} u_{\alpha-1} & v_{\alpha-1} \\ u_\alpha & v_\alpha \end{pmatrix}$.
   Let $d_1 = \deg(R_{2,2})$. Then $\deg(q_1) + \deg(q_2) + \cdots + \deg(q_\alpha) = d_1 \leq k_1$.
4. Compute $\begin{pmatrix} r_{\alpha-1} \\ r_\alpha \end{pmatrix} = \begin{pmatrix} r^*_{\alpha-1} \\ r^*_\alpha \end{pmatrix} \cdot x^{n-2k_1} + R \cdot \begin{pmatrix} a_B \\ b_B \end{pmatrix}$.
5. If $r_\alpha = 0$ or $k < n - \deg(r_\alpha)$ then $s = \alpha$; return $\begin{pmatrix} r_{\alpha-1} \\ r_\alpha \end{pmatrix}$ and $M = R$.

6A. If $d_1 = k_1$, then let $\beta = \alpha$, Q be the 2×2 identity matrix, and $d' = d_1$.

6B. If $d_1 < k_1$, then $\beta = \alpha + 1$, $q_\beta = r_{\beta-2} \operatorname{div} r_{\beta-1}$, $r_\beta = r_{\beta-2} \operatorname{div} r_{\beta-1}$,
   $Q = \begin{pmatrix} 0 & 1 \\ 1 & -q_\beta \end{pmatrix}$, and $d' = d_1 + \deg(q_\beta)$.

7. Let $k_2 = k - d'$ and let $n' = \deg(r_{\beta-1})$.
8. Split $f = r_{\beta-1}$ and $g = r_\beta$ into two blocks such that
   $f = f_A \cdot x^{n'-2k_2} + f_B$ and $g = g_A \cdot x^{n'-2k_2} + g_B$.
9. Recursively call the Fast Euclidean Algorithm with input f_A, g_A, and k_2
   and output $\begin{pmatrix} r^*_{\gamma-1} \\ r^*_\gamma \end{pmatrix}$ and $S = \begin{pmatrix} u_{\gamma-1} & v_{\gamma-1} \\ u_\gamma & v_\gamma \end{pmatrix}$.
   Let $d_2 = \deg(S_{2,2})$. Then $\deg(q_{\beta+1}) + \cdots + \deg(q_\gamma) = d_2 = k_2$.
10. Compute $\begin{pmatrix} r_{\gamma-1} \\ r_\gamma \end{pmatrix} = \begin{pmatrix} r^*_{\gamma-1} \\ r^*_\gamma \end{pmatrix} \cdot x^{n'-2k_2} + S \cdot \begin{pmatrix} f_B \\ g_B \end{pmatrix}$.
11. Return $\begin{pmatrix} r_{\gamma-1} \\ r_\gamma \end{pmatrix}$ and $M = S \cdot Q \cdot R$ (Note: $s = \gamma$).

Figure 8.3 Pseudocode for Fast Euclidean Algorithm

$2 \cdot M_M(k) \leq M_M(2k)$ and $2 \cdot A_M(k) \leq A_M(2k)$ for any of the available methods used to perform polynomial multiplication. Then the cost of line 4 is at most $4 \cdot M_M(2k) + k$ multiplications and $4 \cdot A_M(2k) + k$ additions. Since none of the coefficients of this result will be of degree $2k_1$ or higher, this result can be combined with the result of line 3 at no cost.

In line 6B, $n_{\beta-2} \leq 2k$ and $n_{\beta-1} = n_{\beta-2} - c$ for some c in $1 \leq c \leq k$. The computation of the quotient and remainder in this line requires at most $4 \cdot M_M(2c) + M_M(4k - 2c)$ multiplications and $4 \cdot A_M(2c) + A_M(4k - 2c)$ additions, assuming that the computations are completed using Newton division without any use of the Fast Fourier Transform. Since the second derivative of M_M and A_M is strictly positive for all available multiplication methods in the range $1 \leq c \leq 2k$, then these functions are concave up and the maximum number of multiplications in this range must occur at $c = 1$ or $c = k$. An evaluation of each of the multiplication count functions at $c = 1$ and $c = k$ shows that the maximum occurs at $c = k$ for each case. So the total number of operations for line 6B is at most $5 \cdot M_M(2k)$ multiplications and $5 \cdot A_M(2k)$ additions. However, it is extremely unlikely that this step will require anywhere near this number of operations. While it will often be the case that $\deg(a) = 2k$, the only way that $r_{\beta-2}$ can be $2k$ is when k_1 is too small to make any progress in computing $\gcd(a_A, b_A)$. Another reason why line 6B usually requires fewer operations has to do with typical values for $\deg(q_\beta)$. Recall that most of the time, the degrees of each of the polynomials in the quotient sequence will be 1. In this case, classical division should be used instead of Newton division at a cost of $2k$ multiplications and $2k$ additions. In nearly all cases, $\deg(q_\beta) < 6 \cdot \log_2(k)$ and it is better to use classical division instead of Newton division for this step. To establish a upper bound for the cost of this algorithm that can be demonstrated mathematically, we will always assume the worst case, i.e. a cost of $5 \cdot M_M(2k)$ multiplications and $5 \cdot A_M(2k)$ additions, for this

step using Newton divison. The reader is cautioned, however, that this assumption greatly inflates the cost of this algorithm.

In line 10, the degrees of f_B and g_B is at most $k/2-1$, the degree of $S_{1,1}$, $S_{1,2}$, and $S_{2,1}$ is at most $k/2-1$, and the degree of $S_{2,2}$ is at most $k/2$. Following a technique similar to line 4, the cost of the premultiplication of $(f_B, g_B)^T$ by S is at most $2 \cdot M_M(2k) + k/2$ multiplications and $2 \cdot A_M(2k) + k/2$ additions. Since none of the coefficients of this result will be of degree $2k_2$ or higher, this result can be combined with the result of line 9 at no cost.

In line 11, we will compute the matrix product by first computing $QR = Q \cdot R$. If line 6B is executed, then the upper row of $Q \cdot R$ is equivalent to the lower row of R and can be determined with no operations. In the worst case scenario, $\deg(q_\beta) = k$ and the computation of $R_{1,1} - q_\beta \cdot R_{2,1}$ and $R_{1,2} - q_\beta \cdot R_{2,2}$ each requires at most $4 \cdot M_M(k) + \mathcal{O}(k)$ multiplications and $4 \cdot A_M(k) + \mathcal{O}(k)$ additions, following the technique of lines 4 and 9. Thus, at most $2 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and $2 \cdot A_M(2k) + \mathcal{O}(k)$ additions are required. As with line 6B, most of the time $\deg(q_\beta) = 1$. In this normal case, the computation of $Q \cdot R$ only requires at most $2 \cdot (2 \cdot (k/2+1)) = 2k+4$ multiplications and $2k$ additions. We will again assume the worst-case for this mathematical analysis, but caution the reader that this also greatly inflates the operation count.

To complete line 11, we must compute $M = S \cdot (QR)$. Recall that $\deg(QR_{1,1}) = \deg(R_{2,1}) < k/2$ and $\deg(QR_{1,2}) = \deg(R_{2,2}) \leq k/2$. Because $\deg(q_1) + \deg(q_2) + \cdots + \deg(q_\gamma) \leq k$, $\deg(QR_{2,1})$ is at most $k-1$ and $\deg(QR_{2,2})$ is at most k. The degrees of each of the components of S were given in the analysis of line 10. Using the technique of multiplying polynomials of different degrees, this operation requires $12 \cdot M_M(k) + \mathcal{O}(k) \leq 6 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and $6 \cdot A_M(2k) + \mathcal{O}(k)$ additions. Since each polynomial in M has degree k or less, there is no need to add any terms of

degree $k+1$ or greater in the products computed in this step. If the degree sequence is normal, then $\deg(QR_{2,1}) = k/2$ and $\deg(QR_{2,2}) = k/2+1$. It can be shown in this case that the number of operations drops to $4 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and $4 \cdot A_M(2k) + \mathcal{O}(k)$ additions.

Combining these results, we obtain

$$M(k) = 2 \cdot M(k/2) + 19 \cdot M_M(2k) + \mathcal{O}(k), \tag{8.46}$$

$$A(k) = 2 \cdot A(k/2) + 19 \cdot A_M(2k) + \mathcal{O}(k) \tag{8.47}$$

for the operation counts of the algorithm. Assuming that n is a power of two and letting $k = n$, i.e. we want to compute all of the steps of the Extended Euclidean Algorithm, we obtain operation counts of

$$M(n) = 19 \cdot M_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n), \tag{8.48}$$

$$A(n) = 19 \cdot A_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n) \tag{8.49}$$

by Master Equation VI. If we assume that the quotient sequence is normal, then the operation counts reduce to

$$M(n) = 10 \cdot M_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n), \tag{8.50}$$

$$A(n) = 10 \cdot A_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n). \tag{8.51}$$

While it is very unlikely that the degree of every polynomial in the quotient sequence

will be 1, it will typically be the case that most of these polynomials will have degree 1. So the operation count for a typical greatest common divisor calculation will typically be somewhere between these two results and closer to the normal case.

8.5 Algorithm improvements due to the Fast Fourier Transform

Recall that all of the actual computations in the Fast Euclidean Algorithm were contained in lines 4, 6B, 10, and 11. In this section, we will discuss how the Fast Euclidean Algorithm can be significantly improved if FFT-based multiplication is employed in these steps. For the first part of this section, assume that k is a power of two.

First, consider the division which takes place in line 6 of the algorithm. In Chapter 7, it was established that Newton division of a polynomial of degree $2k$ by a polynomial of degree k can implemented using at most $3.5 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and $3.5 \cdot A_M(2k) + \mathcal{O}(k)$ additions, including the computation of the remainder along with the quotient. Using a argument similar to the previous section, it can be shown that this case requires the maximum number of operations for this line of the algorithm. Again, if the degree sequence of the Euclidean Algorithm is normal, then line 6B only requires $\mathcal{O}(k)$ operations.

Now, let us consider the computation of the product required in line 4 of the algorithm. Let $a_B = a_{B1} \cdot x^{k/2} + a_{B2}$ and $b_B = b_{B1} \cdot x^{k/2} + b_{B2}$. Using the technique discussed in the previous section, the polynomial $r^*_{\alpha-1}$ is computed using $R_{1,1} \cdot a_{B1} + R_{1,2} \cdot b_{B1}$ and $R_{1,1} \cdot a_{B2} + R_{1,2} \cdot b_{B2}$ and the polynomial r^*_α is computed using $R_{2,1} \cdot a_{B1} + R'_{2,2} \cdot b_{B1}$ and $R_{2,1} \cdot a_{B2} + R'_{2,2} \cdot b_{B2}$. Here $R'_{2,2}$ denotes the coefficients of $R_{2,2}$ of degree less than k. For FFT-based multiplication, we will instead just use $R_{2,2}$ and compute the products modulo $\mathcal{M}(x)$ where $\mathcal{M}(x)$ is a polynomial of degree

k with a root at each of k points that will be used for the polynomial evaluations and interpolations.

Let $\mathcal{F}(p)$ represent the FFT of polynomial p and $\mathcal{F}^{-1}$ denote the inverse FFT. To compute one of the above 8 products modulo $\mathcal{M}(x)$ using FFT-based multiplication, for example $R_{1,1} \cdot a_{B1}$, one would compute two FFTs of size k, pointwise multiply k evaluations of the polynomials, and then compute one inverse FFT of size k. In the case of the example, we would compute $\mathcal{F}^{-1}\left(\mathcal{F}(R_{1,1}) \odot \mathcal{F}(a_{B1})\right)$ where $\odot$ denotes a pointwise product of the evaluation points. So the first polynomial used to construct $r^*_{\alpha-1}$ could be computed as

$$\mathcal{F}^{-1}(\mathcal{F}(R_{1,1}) \odot \mathcal{F}(a_{B1})) + \mathcal{F}^{-1}(\mathcal{F}(R_{1,2}) \odot \mathcal{F}(b_{B1})). \tag{8.52}$$

Again, this computation only gives the resulting polynomial modulo $\mathcal{M}(x)$. However, since the degree of any of the polynomials contained in R is at most $k/2$ and the maximum degree of any of $\{a_{B1}, a_{B2}, b_{B1}, b_{B2}\}$ is at most $k/2-1$, then the maximum degree of any of the polynomial products is $k/2+(k/2-1) = k-1$. Thus, computing any of the products modulo $\mathcal{M}(x)$ is the same as computing the product directly.

Because the Fourier Transform and its inverse are linear functions, then this result can also be computed as

$$\mathcal{F}^{-1}\left(\mathcal{F}(R_{1,1}) \odot \mathcal{F}(a_{B1}) \oplus \mathcal{F}(R_{1,2}) \odot \mathcal{F}(b_{B1})\right), \tag{8.53}$$

where $\oplus$ denotes a pointwise addition of the product of the evaluations. So this polynomial can be computed using only 4 FFTs and 1 inverse FFT, rather than with

2 inverse FFTs above. Note that $\mathcal{F}(R_{1,1})$, $\mathcal{F}(R_{1,2})$, $\mathcal{F}(R_{2,1})$, $\mathcal{F}(R_{2,2})$, $\mathcal{F}(a_{B1})$, $\mathcal{F}(a_{B2})$, $\mathcal{F}(b_{B1})$, and $\mathcal{F}(b_{B2})$ are reused in the computations of the 4 polynomials and these are the only 8 FFTs required. Additionally, one inverse FFT is required for each of the 4 desired polynomials in this step. Since one multiplication with product of size k consists of 2 forward FFTs plus 1 inverse FFT, the work of this line of the algorithm is $4 \cdot M_M(k) + \mathcal{O}(k) \leq 2 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and at most $2 \cdot A_M(2k) + \mathcal{O}(k)$ additions.

A similar technique can be used to implement line 10 of the algorithm. Because f_B and g_B are each of degree at most $k/2-1$, it is not necessary to split each of these polynomials into two blocks. In this case, only 6 forward FFTs and 2 inverse FFTs are needed to complete this line of the algorithm. Thus, the work of this line of the algorithm is $8/3 \cdot M_M(k) + \mathcal{O}(k) \leq 4/3 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and at most $4/3 \cdot A_M(2k) + \mathcal{O}(k)$ additions.

In line 11 of the algorithm, we need to compute $M = S \cdot Q \cdot R$. Here, we will consider the case where Q is not the identity matrix and note that the operation count derived here is an upper bound for the case where Q is the identity matrix. We need to compute

$$\begin{pmatrix} M_{1,1} & M_{1,2} \\ M_{2,1} & M_{2,2} \end{pmatrix} = \begin{pmatrix} S_{1,1} & S_{1,2} \\ S_{2,1} & S_{2,2} \end{pmatrix} \cdot \begin{pmatrix} 0 & 1 \\ 1 & -q_\beta \end{pmatrix} \cdot \begin{pmatrix} R_{1,1} & R_{1,2} \\ R_{2,1} & R_{2,2} \end{pmatrix}. \quad (8.54)$$

Using the technique discussed in line 4, then $M_{1,1}$ can be computed using

$$M_{1,1} = \mathcal{F}^{-1}\left(\mathcal{F}(S_{1,1}) \odot \mathcal{F}(R_{2,1}) \oplus \mathcal{F}(S_{1,2}) \odot (\mathcal{F}(R_{1,1}) \ominus \mathcal{F}(q_\beta) \odot \mathcal{F}(R_{2,1}))\right), \quad (8.55)$$

where $\ominus$ denotes a pointwise difference. This will compute $M_{1,1}$ modulo $\mathcal{M}(x) = M_{1,1}$ since $M_{1,1}$ has degree less than k. The technique can also be used to produce $M_{1,2}$ and $M_{2,1}$. Since the degree of $M_{2,2}$ is at most k, then

$$M^*_{2,2} = \mathcal{F}^{-1}\left(\mathcal{F}(S_{2,1}) \odot \mathcal{F}(R_{2,2}) \oplus \mathcal{F}(S_{2,2}) \odot (\mathcal{F}(R_{1,2}) \ominus \mathcal{F}(q_\beta) \odot \mathcal{F}(R_{2,2}))\right) \quad (8.56)$$

may not actually be $M_{2,2}$. Compute $\deg(S_{2,2}) + \deg(q_\beta) + \deg(R_{2,2})$. If this sum is less than k, then $M^*_{2,2} = M_{2,2}$. Otherwise, let $M^*_{2,2} = M_{2,2} \bmod \mathcal{M}(x)$ and

$$M_{2,2} \quad = \quad M^*_{2,2} + \mathcal{C}(x) \cdot \mathcal{M}(x) \quad (8.57)$$

for some polynomial $\mathcal{C}(x)$. Since $M_{2,2}$ has degree k and $\mathcal{M}(x)$ has degree k, then $\mathcal{C}(x)$ must be a constant and is the leading coefficient of $M_{2,2}$. Multiply the leading coefficients of $S_{2,2}$, q_β, and $R_{2,2}$ together to determine $\mathcal{C}(x)$. Then $M_{2,2}$ can be recovered with T multiplications and T additions where T is the number of nonzero coefficients in $\mathcal{M}(x)$.

Note that in (8.54), the FFTs of each of the components in R were already computed in line 4, and the FFTs of each of the components in S were already computed in line 10. Furthermore, if the remainder in line 6B was computed using FFT-multiplication, then $\mathcal{F}(q_\beta)$ has already been computed as well. These FFTs can be saved from the earlier lines of the algorithm and reused in line 11. Thus, the computation of (8.54) requires only 4 inverse FFTs of size k, plus $\mathcal{O}(k)$ pointwise

multiplications, additions and subtractions. Thus, the work of this line of the algorithm is $4/3 \cdot M_M(k) + O(k) \leq 2/3 \cdot M_M(2k) + \mathcal{O}(k)$ multiplications and at most $2/3 \cdot A_M(2k) + \mathcal{O}(k)$ additions.

Letting $k = n$ and combining these operation counts, a total of

$$M(n) = 7.5 \cdot M_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n), \tag{8.58}$$

$$A(n) = 7.5 \cdot A_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n) \tag{8.59}$$

operations are required in the general case using Master Equation VI. If we assume that the quotient sequence is normal, then the operation counts reduce to

$$M(n) = 4 \cdot M_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n), \tag{8.60}$$

$$A(n) = 4 \cdot A_M(2n) \cdot \log_2(n) + \mathcal{O}(n) \cdot \log_2(n). \tag{8.61}$$

It was mentioned above that these operation counts were derived for the case where k is a power of two so that FFTs of size a power of two could be used in the multiplications. In practice, one can use multiplication based on the truncated FFT discussed in Chapter 6 to reduce the number of multiplications. In this case, the formulas above with arbitrary k can be used to provide reasonable estimates for upper bounds of the operation counts.

It appears difficult to improve upon the above upper bound formulas, but additional practical improvements are possible with the algorithms. When k is "small", it will often be faster to implement the multiplications using Karatsuba's algorithm

or classical multiplication rather than FFT-based multiplication. It was already mentioned that classical division should be used instead of Newton division for the computation of all quotients and remainders except when the input polynomials are of large degree and the difference between the degrees of the input polynomials is also large. Furthermore, the above operation counts do not take into account the reduction observed by implementing the two improvements to the Fast Euclidean Algorithm discussed earlier. The author implemented the Fast Euclidean Algorithm presented in the previous section and compared the number of operations required for the various strategies. While the exact amount of improvement depends on the sizes of the input polynomials and the degrees of the polynomials in the quotient sequence, the author observed around a 15 percent reduction in operations when the first improvement was implemented and a reduction by about 20 percent when the second improvement was implemented, compared to the strategy of explicitly performing a division step after the completion of every first subproblem.

8.6 Concluding remarks

This chapter presented several algorithms which efficiently compute the greatest common divisor of two polynomials. Solutions for several exercises related to the Fast Euclidean Algorithm proposed in [34] were also provided along with several additional ideas for improving the algorithm. Several improvements in the analysis of the Fast Euclidean Algorithm were also given. Although the algorithm presented here has the same asymptotic complexity as the algorithm in [34], the research summarized in this chapter reduced the multiplicative constant from 24 to 19 in the general case when FFT-multiplication was not applied, compared to the results given in [34]. If FFT-based multiplication is permitted, then this chapter showed how to further

reduce the multiplicative constant to 7.5 in the general case and to 4 when the degree sequence is normal.

The chapter also presented several improvements to the Fast Euclidean Algorithm which reduce the number of operations by about 15-20 percent in practice, but could not be established mathematically. The reason for this is the presence of a computational step between the two Fast Euclidean Algorithm subproblems that cannot be accurately modeled. The goal of the practical improvements was to skip the computational step when the algorithm parameter was large, deferring it to recursive calls where the algorithm parameter is smaller and would require less effort to complete. The reason why it is difficult to derive mathematical formulas to describe the improvements is because one cannot know when to skip this computational step without prior knowledge of the degrees of each of the polynomials in the quotient sequence needed to compute the GCD. Furthermore, the second improvement typically subdivides a problem into two subproblems of unequal size. Because the second subproblem typically has parameter greater than half of the parameter to the original call to the Fast Euclidean Algorithm, the techniques discussed in this chapter cannot be used to mathematically analyze the effect of these improvements.

Because the actual number of operations needed to implement the Fast Euclidean Algorithm is so dependent on the degrees of each of the polynomials in the quotient sequence, it appears difficult to develop any further improvements to the Fast Euclidean Algorithm. One should be aware of a paper by Strassen [72] which uses an entropy argument to claim that the asymptotic number of operations needed to solve the greatest common divisor problem has a lower bound which coincides with the Fast Euclidean Algorithm presented in this chapter. If Strassen's claim is true, then it appears that there is little additional improvement that can be made to the

algorithm presented in this chapter, other than further improvements to the multiplicative constants of the operation counts and other practical improvements which cannot be established mathematically.

CHAPTER 9

REED-SOLOMON ERROR-CORRECTING CODES

Reed-Solomon codes are a powerful method of correcting errors introduced when a message is transmitted in a noisy environment. These codes are very popular and can be found in compact disc players and NASA satellites used for deep-space exploration. The power of these codes resides in algebraic properties of finite fields which allows for multiple errors to be corrected in each codeword. Unfortunately, most algorithms used for decoding Reed-Solomon codes are somewhat complicated and require an advanced knowledge of algebra to understand how the decoding process works.

In 1988, Shiozaki [71] published a simple algorithm for decoding Reed-Solomon codewords. Unaware of the earlier publication, Gao [29] reinvented essentially the same algorithm some 15 years later. Both of these algorithms decode a Reed-Solomon codeword which is nonsystematically encoded, which means that the message does not appear in the transmitted codeword. This is the approach of Reed and Solomon in their original presentation of the code [63]. Most practical applications of Reed-Solomon codes, however, use the systematic encoding popularized by Gorenstein and Zierler [38] which uses generator polynomials to encode each codeword. A 2005 paper by Fedorenko [23] [1] claims that the method for decoding Reed-Solomon codewords does not depend on whether the codeword was encoded systematically or nonsystematically. However, Fedorenko does not provide any proof of this claim in the

[1] A 2006 correction to this paper [24] was also published in which Shiozaki is given credit for also discovering the algorithm.

paper, nor any details about how Gao's decoding algorithm could be applied to the systematic encoding case.

In this chapter, we explicitly show how to apply the simple decoding algorithm to Reed-Solomon codewords that are systematically encoded for the most popular version of the codes where the underlying finite field is characteristic 2. The reader can generalize the techniques presented here to other Reed-Solomon codes if desired.

9.1 Systematic encoding of Reed-Solomon codewords

Let $\mathbb{F}$ be a finite field with q elements where q is a power of two. A standard result of finite field theory states that the nonzero elements of $\mathbb{F}$ can generated by some $\alpha \in \mathbb{F}$. Thus, $\mathbb{F}$ consists of the zero element as well as the set of elements $\{1, \alpha, \alpha^2, \alpha^3, \ldots, \alpha^{n-1}\}$ where $n = q - 1$. Note that $\alpha^n = 1$ where $\alpha \neq 1$.

A (n, k, d) Reed-Solomon codeword is a vector consisting of n elements of $\mathbb{F}$ and can also be expressed as a polynomial. This codeword is used to transmit a message consisting of $k < n$ elements of $\mathbb{F}$. In a standard coding theory textbook (e.g. [59]), it is shown that the minimum distance for this code is $d = n - k + 1$. This means that the code is capable of correcting up to $t = (n-k)/2$ errors in the received codeword.

To systematically encode the message $\{m_0, m_1, m_2, \ldots, m_{k-1}\} \in \mathbb{F}$, we will first construct the message polynomial

$$m(x) \quad = \quad m_{k-1} \cdot x^{k-1} + m_{k-2} \cdot x^{k-2} + \cdots + m_1 \cdot x + m_0 \tag{9.1}$$

and generating polynomial

$$g(x) = (x - \alpha) \cdot (x - \alpha^2) \cdot \cdots \cdot (x - \alpha^{n-k}). \tag{9.2}$$

We will divide the polynomial $m \cdot x^{n-k}$ by g and obtain quotient $Q(x)$ and remainder $\mathcal{R}(x)$. Thus, $m \cdot x^{n-k} = g \cdot Q + \mathcal{R}$. The Reed-Solomon codeword $C(x) = C_{n-1} \cdot x^{n-1} + C_{n-2} \cdot x^{n-2} + \cdots + C_1 \cdot x + C_0$ is given by

$$C \;\; = \;\; m \cdot x^{n-k} + \mathcal{R}, \tag{9.3}$$

that is

$$C \;\; = \;\; g \cdot Q. \tag{9.4}$$

Thus, C is a multiple of g for any Reed-Solomon codeword C.

Since g has degree $n - k$, then the Division Algorithm implies that $\mathcal{R}(x)$ has degree less than $n - k$. Thus the coefficients of C of degree $n - k$ or higher simply consist of the coefficients of m. Since all of the elements of m appear in C, then C is systematic.

Let us consider the polynomial $F(x)$ defined by

$$F(x) \;=\; \sum_{i=0}^{n-1} C_i \cdot \mathcal{L}_i(x), \tag{9.5}$$

where $\mathcal{L}_i(x)$ is the Lagrange interpolating polynomial given by

$$\mathcal{L}_i(x) \;=\; \alpha^i \cdot \frac{x^n - 1}{x - \alpha^i} \;=\; \alpha^i \cdot x^{n-1} + \alpha^{2i} \cdot x^{n-2} + \cdots + \alpha^{(n-1)\cdot i} \cdot x + 1 \tag{9.6}$$

and $\{C_0, C_1, C_2, \cdots, C_{n-1}\}$ is a set of elements extracted from the corresponding coefficients of $C(x)$. Recall from Chapter 1 that $\mathcal{L}_i(x) = 0$ when $x \neq \alpha^i$ and that $\mathcal{L}_i(x) = 1$ when $x = \alpha^i$. Note that $F(x)$ has the property that $F(\alpha^i) = C_i$ for all i in $0 \leq i \leq n-1$. Thus, $F(x)$ is said to be an interpolating polynomial for the set of evaluations $\{C_0, C_1, C_2, \cdots, C_{n-1}\}$ at the set of points $\{1, \alpha, \alpha^2, \ldots, \alpha^{n-1}\}$.

The formulas (9.5) and (9.6) represent a rather complicated method of generating the polynomial F. Fortunately, there is a much easier method. Note that

$$\begin{aligned}
F \;=\;& C_0 \cdot x^{n-1} + C_0 \cdot x^{n-2} + \cdots + C_0 \cdot x + C_0 \\
+\;& C_1 \cdot \alpha \cdot x^{n-1} + C_1 \cdot \alpha^2 \cdot x^{n-2} + \cdots + C_1 \cdot \alpha^{n-1} \cdot x + C_1 \\
+\;& C_2 \cdot \alpha^2 \cdot x^{n-1} + C_2 \cdot \alpha^4 \cdot x^{n-2} + \cdots + C_2 \cdot \alpha^{2(n-1)} \cdot x + C_2 \\
& \vdots \\
+\;& C_{n-1} \cdot \alpha^{n-1} \cdot x^{n-1} + C_{n-1} \cdot \alpha^{2(n-1)} \cdot x^{n-2} \\
& \quad + \cdots + C_{n-1} \cdot \alpha^{(n-1)(n-1)} \cdot x + C_{n-1}.
\end{aligned} \tag{9.7}$$

Collecting all common terms of x^j together, the coefficient of x^j in F, denoted by F_j, is given by

$$\begin{aligned} F_j &= C_{n-1} \cdot \alpha^{(n-1)(n-j)} + C_{n-2} \cdot \alpha^{(n-2)(n-j)} + \cdots + C_2 \cdot \alpha^{2(n-j)} + C_1 \cdot \alpha^{n-j} + C_0 \\ &= C(\alpha^{n-j}). \end{aligned} \tag{9.8}$$

In other words, F_j can be determined by simply evaluating C at α^{n-j} and F is obtained by computing the multipoint evaluation of C at $\{\alpha^n = 1, \alpha^{n-1}, \alpha^{n-2}, \ldots, \alpha^2, \alpha\}$.[2] In coding theory, the "Galois Field Fourier Transform" is defined by

$$V(x) = \mathcal{F}[v(x)] = V_{n-1} \cdot x^{n-1} + V_{n-2} \cdot x^{n-2} + \cdots + V_1 \cdot x + V_0 \tag{9.9}$$

where

$$V_j = \sum_{i=0}^{n-1} \omega^{i \cdot j} \cdot v_i = v(\omega^j) \tag{9.10}$$

for $v(x) = v_{n-1} \cdot x^{n-1} + v_{n-2} \cdot x^{n-2} + \cdots + v_2 \cdot x^2 + v_1 \cdot x + v_0$ and the Inverse Galois Field Fourier Transform is defined by

$$\mathcal{F}^{-1}[V(x)] = v_{n-1} \cdot x^{n-1} + v_{n-2} \cdot x^{n-2} + \cdots + v_1 \cdot x + v_0, \tag{9.11}$$

[2] This is a consequence of the duality property of the multiplicative FFT.

where

$$v_i \;=\; \sum_{j=0}^{n-1} \omega^{-i \cdot j} \cdot V_j \;=\; V(\omega^{-i}). \tag{9.12}$$

Here, ω is a generator of the multiplicative subgroup of $\mathbb{F}$ and is a primitive nth root of unity. Then F can be determined by computing the Inverse Galois Field Fourier Transform of C with $\omega = \alpha$. Since α^{n-1} also generates the multiplicative subgroup of $\mathbb{F}$, one can also determine F by computing the Galois Field Fourier Transform of C with $\omega = \alpha^{n-1}$. So depending on one's perspective, we may regard F as either the Galois Field Fourier Transform of C or the inverse Galois Field Fourier Transform of C.

Efficient algorithms for computing the multipoint evaluation of a function at all of the elements of $\mathbb{F}$ were explored in Chapter 3. Interpolation algorithms to reverse the process were discussed in Chapter 4. In order to apply these algorithms to the computation of the Galois Field Fourier Transform and its inverse, one must precompute and store an array which will permute the output of these fast algorithms into the order of the coefficients of F. It is not possible to specify the elements of the permutation array in general because it is dependent upon the polynomial used to construct the finite field $\mathbb{F}$.

We conclude this section by proving a property of the polynomial F which is important for the decoding algorithm of systematic Reed-Solomon codewords.

Theorem 42 *If F is the Inverse Galois Field Fourier Transform of C, then F has degree less than k.*

Proof: By (9.8), the coefficient F_j of x^j in F is given by $C(\alpha^{n-j})$. By (9.4), $C = g \cdot Q$ where g is given by (9.2). Choose any j such that $k \leq j \leq n-1$. Then $n-j$ falls in the range $1 \leq n-j \leq n-k$ and

$$\begin{aligned} F_j &= C(\alpha^{n-j}) \\ &= g(\alpha^{n-j}) \cdot Q(\alpha^{n-j}) \\ &= 0 \cdot Q(\alpha^{n-j}) \\ &= 0 \end{aligned} \tag{9.13}$$

since α^{n-j} is a root of g by (9.2). Thus, $F_j = 0$ for all $j \geq k$ and the theorem is proven. □

9.3 Decoding of systematic Reed-Solomon codewords

In an ideal environment, the codeword polynomial C would be transmitted with no errors introduced and we could recover the message polynomial m by simply extracting the coefficients of degree $n-k$ or higher from C by using (9.3). Unfortunately, in a practical transmitting environment, it possible for errors represented by $E(x) = E_{n-1} \cdot x^{n-1} + E_{n-2} \cdot x^{n-2} + \cdots + E_1 \cdot x + E_0$ to be introduced to the codeword and the receiver instead receives the collection of elements of $\mathbb{F}$ represented by the polynomial $R(x) = R_{n-1} \cdot x^{n-1} + R_{n-2} \cdot x^{n-2} + \cdots + R_1 \cdot x + R_0$. Here, R could also be represented as

$$R = C + E. \tag{9.14}$$

Our task now is to recover C given R and the properties of Reed-Solomon codewords.

Instead of solving this problem directly, we are going to compute the Inverse Galois Field Fourier Transform with generator α of $\{R, C, E\}$ and do our decoding in this transformed domain. Following the procedure used to construct $F(x)$, we construct a polynomial whose coefficients are equal to the evaluations of $R(x)$ at each of the primitive roots of unity and obtain

$$F'(x) = R(\alpha) \cdot x^{n-1} + R(\alpha^2) \cdot x^{n-2} + \cdots + R(\alpha^{n-1}) \cdot x^1 + R(1) \qquad (9.15)$$

which has the property that $F'(\alpha^i) = R_i$ for all i in $0 \leq i \leq n-1$. Similarly, we can construct the polynomial

$$\mathcal{E}(x) = E(\alpha) \cdot x^{n-1} + E(\alpha^2) \cdot x^{n-2} + \cdots + E(\alpha^{n-1}) \cdot x^1 + E(1) \qquad (9.16)$$

which has the property that $\mathcal{E}(\alpha^i) = E_i$ for all i in $0 \leq i \leq n-1$. Note that unless F' is a valid Reed-Solomon codeword (and thus a multiple of g), the degree of F' can be at most $n-1$. Similarly, $\mathcal{E}$ may have degree at most $n-1$.

We are now going to give an alternative representation of $\mathcal{E}$ necessary for the decoding algorithm. Let $\mathcal{X}$ be the set of positions where there is a discrepancy between C and R. Then define the error locator polynomial by

$$W(x) = \prod_{\varepsilon \in \mathcal{X}} (x - \alpha^{\varepsilon}). \qquad (9.17)$$

Note that W has a value of 0 for each α^j such that R_j was received incorrectly and returns a nonzero value for each α^j such that R_j was received correctly. We can also define the function

$$W^*(x) = \frac{x^n - 1}{W(x)} \tag{9.18}$$

which has the property that W^* has a nonzero value for each α^j such that R_j is incorrect and returns 0 for each α^j such that R_j was received correctly. Note that if W has degree at most $t = (n-k)/2$, then W^* will have degree at least $n-t = (n+k)/2$.

Next, define the error corrector polynomial according to

$$Y(x) = \frac{F(x) - F'(x)}{W^*(x)}. \tag{9.19}$$

A typical Reed-Solomon decoding algorithm first explicitly computes an error locator polynomial similar to W, [3] factors the error locator polynomial to determine the locations of the errors in R, computes an error-corrector polynomial similar to Y, and concludes by evaluating this error-corrector polynomial at each of the roots of the error-locator polynomial to determine the necessary adjustments of R.

The simple algorithm will instead directly compute the polynomial F using the Extended Euclidean Algorithm which was discussed in Chapter 8. Observe that

[3] Typically, the error locator used is given by $\prod_{\varepsilon \in \mathcal{X}}(1 - \alpha^{\varepsilon} \cdot x)$. This polynomial is zero for each $\alpha^{-\varepsilon}$ such that $\varepsilon \in \mathcal{X}$.

$$\mathcal{E}(x) \;=\; F(x) - F'(x) = Y(x) \cdot W^*(x) = \frac{Y(x) \cdot (x^n - 1)}{W(x)}. \tag{9.20}$$

By multiplying (9.20) by W and rearranging the result, we obtain

$$Y \cdot (x^n - 1) + W \cdot F' \;=\; W \cdot F. \tag{9.21}$$

Theorem 43 below is a modified version of a result given in [54] which states that F can be computed using the Extended Euclidean Algorithm.

Theorem 43 *Suppose that the degree of W is less than or equal to $(n-k)/2$. Let $\{r_1, r_2, \ldots, r_\ell\}$ be the remainder sequence produced by the Extended Euclidean Algorithm to compute $gcd(x^n - 1, F')$. If s is the smallest integer such that $deg(r_s) < (n+k)/2$, then*

$$F \;=\; \frac{r_s}{v_s} \tag{9.22}$$

where v_s is the polynomial produced by the Extended Euclidean Algorithm such that

$$u_s \cdot (x^n - 1) + v_s \cdot F' \;=\; r_s. \tag{9.23}$$

Proof: By (9.21),

$$Y \cdot (x^n - 1) + W \cdot F' \;=\; P \tag{9.24}$$

where $P = W \cdot F$. Since $\deg(W) \leq (n-k)/2$ and $\deg(F) < k$, then $\deg(P) < (n+k)/2$. Suppose that the Extended Euclidean Algorithm is used to determine $\gcd(x^n - 1, F')$. At step s of these calculations, we have

$$u_s \cdot (x^n - 1) + v_s \cdot F' \;=\; r_s \tag{9.25}$$

where $\deg(r_s) < (n+k)/2$, $\deg(r_{s-1}) \geq (n+k)/2$ and by Theorem 37, $\deg(v_s) = n - \deg(r_{s-1}) \leq (n-k)/2$. Now, multiply (9.21) by v_s and multiply (9.25) by W to obtain

$$v_s \cdot Y \cdot (x^n - 1) + v_s \cdot W \cdot F' \;=\; v_s \cdot P, \tag{9.26}$$

$$W \cdot u_s \cdot (x^n - 1) + v_s \cdot W \cdot F' \;=\; W \cdot r_s. \tag{9.27}$$

Thus,

$$v_s \cdot P \;=\; W \cdot r_s + \mathcal{K} \cdot (x^n - 1) \tag{9.28}$$

for some polynomial $\mathcal{K}(x)$. Since $\deg(v_s \cdot P) = \deg(v_s) + \deg(P) < n$ and $\deg(W \cdot r_s) = \deg(W) + \deg(r_s) < n$, then it must be the case that $\mathcal{K} = 0$ and

$$v_s \cdot P \quad = \quad W \cdot r_s. \tag{9.29}$$

Then,

$$F \quad = \quad \frac{W \cdot F}{W} = \frac{P}{W} = \frac{r_s}{v_s}. \tag{9.30}$$

□

The Galois Field Fourier Transform can be used to efficiently recover F if every possible received vector differs in at most $(n-k)/2$ positions from some valid codeword. Once r_s and v_s have been determined, one can can compute the Galois Field Fourier Transform of r_s and v_s, compute the pointwise quotients $r_s(\varepsilon)/v_s(\varepsilon) = F(\varepsilon)$ for each ε in the multiplicative subgroup of $\mathbb{F}$, and then use the inverse Galois Field Fourier Transform to interpolate the evaluations into $F(x)$. Recall that this process is called deconvolution and is only valid when the remainder of a division computation is known to be 0 in advance. If there exists a received vector that differs from every valid codeword in at least $(n-k)/2$ positions, then Theorem 43 does not apply. In this case, the division of r_s by v_s will produce a quotient with degree k or higher and a nonzero remainder. One should use classical division or Newton division to efficiently perform the computation in this case.

Once $F(x)$ has been determined, one can evaluate $F(x)$ at each of the roots of $x^n - 1$ using the Galois Field Fourier Transform to recover the transmitted codeword $C(x)$ and easily recover $m(x)$ as described at the beginning of this section. If more than $(n + k)/2$ errors were introduced during the transmission process, then the algorithm instead returns another valid Reed-Solomon codeword that is closer to $R(x)$. In this case, the number of errors is beyond the capacity of the code.

9.4 Pseudocode and operation count of the simple decoding algorithm

Given a received vector represented by $R(x)$, the decoding algorithm proceeds by interpolating R into the polynomial F'. The Extended Euclidean Algorithm is used to start the computation of $\gcd(x^n - 1, F')$. When a remainder r_s is encountered with degree less than $(n+k)/2$, the computations stop. At this point, we also have v_s as part of the computations of the Extended Euclidean Algorithm. If $r_s \bmod v_s = 0$, then compute r_s/v_s; otherwise report "Decoding Failure".

If R has less than $(n-k)/2$ differences compared to the transmitted codeword C, then $F = r_s/v_s$. One can evaluate F at each of the roots of $x^n - 1$ to recover C. The desired message m is contained in the coefficients of C with degree $n - k$ or higher.

If R and C differ in at least $d = (n - k)/2$ positions, then the algorithm will return some other message $m^\dagger$. Again, the error-correcting capability of the (n, k) Reed-Solomon code is at most $d = (n - k)/2$ and it is not possible to recover m in this case regardless of the decoding algorithm used.

The pseudocode given in Figure 9.1 can be used to implement the simple decoding algorithm. By using the basic techniques for implementing lines 1-4 of this algorithm discussed in the previous chapters, it can be shown that each of these instuctions requires at most $\mathcal{O}(n^2)$ additions and $\mathcal{O}(n^2)$ multiplications. Line 5 only

Algorithm : Simple Reed-Solomon decoding algorithm

Input: The polynomial $R(x)$ of degree less than n which represents
the received vector of a (n, k, d) Reed-Solomon codeword
transmitted through a noisy environment where $d = (n - k)/2$.

Output: A message polynomial $m^\dagger(x)$ of degree less than k
which can be encoded with the Reed-Solomon codeword $C^\dagger(x)$
such that $C^\dagger$ and R differ in at most $(n - k)/2$ positions
or "Decoding Failure".

1. Interpolate $R(x)$ into $F'(x)$ using the inverse Galois Field Fourier Transform.
2. Apply the Extended Euclidean Algorithm to $x^n - 1$ and F'.
   Stop after s steps where s is the first division step such that
   the degree of r_s is less than $(n + k)/2$.
   At this point, we will have $r_s = u_s \cdot (x^n - 1) + v_s \cdot F'$.
3. Divide r_s by v_s.
   If the remainder is nonzero, then return "Decoding Failure";
   Otherwise, let $F^\dagger = r_s/v_s$.
4. Evaluate $F^\dagger$ at each of the roots of $x^n - 1$ to form $C^\dagger$
   using the Galois Field Fourier Transform.
5 Extract $m^\dagger$ from the coefficients of $C^\dagger$ of degree $n - k$ and higher.
6. Return $m^\dagger$.

Figure 9.1 Pseudocode for simple Reed-Solomon decoding algorithm

requires $n-k$ copy instructions to transfer the appropriate components of $C^{\dagger}$ into $m^{\dagger}$. Line 6 requires no operations. So the overall complexity of the decoding algorithm is $\mathcal{O}(n^2)$, the same as other Reed-Solomon decoding algorithms discussed in [59].

Using the techniques discussed in Chapters 3 and 4 for computing the Fast Fourier Transform over all of the elements of a Galois Field, lines 1 and 4 can each be computed using $\mathcal{O}(n \log_2(n))$ multiplications and $\mathcal{O}(n \log_2(n) \log_2 \log_2(n))$ additions. This algorithm can be used in conjunction with Newton division discussed in Chapter 7 to compute line 3 in $\mathcal{O}(n \log_2(n))$ multiplications and $\mathcal{O}(n \log_2(n) \log_2 \log_2(n))$ additions as well. Finally, the Fast Euclidean Algorithm discussed in Chapter 8 can be used to show that line 2 can be computed in $\mathcal{O}(m(\log_2(m))^2)$ multiplications and $\mathcal{O}(m(\log_2(m))^2 \log_2 \log_2(m))$ additions where $m = (n+k)/2$. This is because the Fast Euclidean Algorithm only uses the upper coefficients of $x^n - 1$ and $F'(x)$ to recover $F(x)$. It should be noted that Gao similarly shows how to adapt the partial GCD computation in line 2 to compute $F(x)$ using only the upper coefficients of $x^n - 1$ and $F'(x)$ in [29]. Without specific knowledge of n and k, one cannot determine the overall complexity of the algorithm using the asymptotically faster techniques, but in any case the overall complexity will be better than $\mathcal{O}(n^2)$.

Although the techniques mentioned in the previous paragraph require fewer operations than traditional techniques for large values of n, Reed-Solomon codeword sizes are not typically large in practice. For example, the Reed-Solomon codes used in CD players use $n \leq 256$. In this case, one needs to carefully compare the various methods of performing each of the steps. For such small sizes, the overall cost of each of these operations will likely depend on factors other than the number of additions and the number of multiplications and will likely vary from computer platform to computer platform. One must carefully implement the decoding algorithm described above and then compare the overall time with existing techniques to determine the

most efficient method for decoding a Reed-Solomon codeword. Although the decoding method described here is expected to be comparible to existing techniques, it is not known which is the most efficient method for a particular Reed-Solomon code on a particular machine. The author plans to experiment with timing comparisons of the simple decoding algorithm with other methods at some point in the future using Reed-Solomon codes typically encountered in practice. These results, however, will likely have little value for other readers because the timing results will only be valid on the author's computer. Anyone interested in the most efficient Reed-Solomon decoding algorithm for a particular application should carefully implement the available algorithms on his or her computer and make this decision based on performance tests of these methods.

9.5 Concluding remarks

This chapter introduced a simple algorithm for decoding systematic Reed-Solomon codewords based on the nonsystematic decoding algorithms presented in [71], [29], [23], and [24]. It should be mentioned at this point that a nonsystematic encoder essentially lets $F(x) = m(x)$ and evaluates $F(x)$ at each of the roots of $x^n - 1$ to obtain the transmitted codeword $C(x)$. The decoding algorithm in each of these cases proceeds by only implementing lines 1-3 of the pseudocode provided in the previous section.

The algorithm presented in his chapter only decodes the most commonly encountered Reed-Solomon codewords where $\mathbb{F}$ is a finite field of characteristic 2 with q elements and $n = q - 1$. Once the reader understands the principles described in this chapter, he or she can read [29] to understand how to adapt the algorithm to work with other possible Reed-Solomon codes. In these cases, one must replace the polynomial $x^n - 1$ used throughout this chapter with a polynomial that characterizes the

roots used in the other codes. One must also derive different Lagrange interpolating polynomials for this case and develop a method of efficiently computing multipoint evaluation and interpolation for this case. Gao's paper also briefly mentions how the algorithm can be adapted for erasure decoding of the Reed-Solomon codewords.

Although the algorithm described here is believed to have a simpler presentation and proof than existing decoding methods, it is not clear yet how it performs compared to existing techniques. The simple decoding algorithm has similar asymptotic operation counts compared to existing decoding methods, but these operation counts have little practical value because typical Reed-Solomon codewords have relatively small sizes and work over small finite fields. One must carefully implement the various decoding algorithms to determine the most efficient method of recovering a message from a received Reed-Solomon codeword.

CHAPTER 10

FURTHER APPLICATIONS AND CONCLUDING REMARKS

In this final chapter, we will first return to some topics left unresolved from the first chapter. In particular, we will discuss how the FFT algorithms developed in this manuscript can be used to compute the coefficients of a discrete Fourier series and how to improve multipoint evaluation and interpolation over an arbitrary collection of points. Next, we will briefly describe other applications of the FFT which can be explored as further areas of research. Finally, some concluding remarks will be given, including a summary of the highlights of this manuscript.

10.1 Computing the coefficients of a discrete Fourier series

Recall that in Chapter 1, we mentioned that engineers defined the "Fast Fourier Transform" as an algorithm which can be used to compute the coefficients of a discrete Fourier series given a collection of samples of some signal. In this section, we will more carefully define this problem and show how the FFT algorithms developed in the earlier chapters can be used to solve the problem.

Let $f_a(t)$ be some continuous periodic real-valued function with fundamental period T_0, i.e. $f_a(t) = f_a(t + T_0)$ for all t. Suppose that $f_a(t)$ is sampled uniformly at n points over each period of $f_a(t)$ to obtain the discrete-time function $f(\tau)$. The sampling rate T is given by T_0/n. The sample at $\tau = 0$ can be chosen to correspond to any t, but we will assume here that sample τ will correspond to $t = \tau \cdot T$ for all integer values of τ. Note that the fundamental period of $f(\tau)$ is n samples.

As presented in [62], $f_a(t)$ can be approximated by interpolating the values of the discrete signal $f(\tau)$ into a function called the discrete Fourier series which is a summation of harmonically related sinusoidal functions given by

$$f(t) = a_0 + \sum_{k=1}^{n_2} \left(a_k \cdot \cos(k \cdot 2\pi f_0 \cdot t) + b_k \cdot \sin(k \cdot 2\pi f_0 \cdot t)\right), \tag{10.1}$$

where $f_0 = 1/T_0$. Also, $n_2 = n/2$ if n is even and $(n-1)/2$ if n is odd.

Our goal is to determine the coefficients $\{a_0, a_1, a_2, \cdots, a_{n_2}, b_1, b_2, \cdots, b_{n_2}\} \in \mathbb{R}$ in (10.1). To do so, let us instead consider the related discrete-time function

$$f(\tau) = a_0 + \sum_{k=1}^{n_2} \left(a_k \cdot \cos(2\pi k/n \cdot \tau) + b_k \cdot \sin(2\pi k/n \cdot \tau)\right) \tag{10.2}$$

which matches $f(t)$ at all values of t of the form $t = \tau \cdot (T_0/n)$ where τ is any integer. Observe that if n is even, then $\sin(2\pi k/n \cdot \tau) = 0$ for all integer values of τ and the related term in $f(t)$ will not contribute to the discrete Fourier series. So (10.1) will always have exactly n terms in it.

We desire to interpolate the evaluations $\{f(\tau = 0), f(1), \cdots, f(n-1)\}$ into $f(\tau)$. At first glance, it does not appear that the FFTs presented in Chapters 2 and 4 can be used to solve this problem. However, if the following results learned in a course in complex variables (e.g. [67])

$$\cos(2\pi k/n \cdot \tau) = \frac{1}{2} \cdot (e^{\mathtt{I} \cdot 2\pi k/n \cdot \tau} + e^{-\mathtt{I} \cdot 2\pi k/n \cdot \tau}), \tag{10.3}$$

$$\sin(2\pi k/n \cdot \tau) = \frac{1}{2\mathtt{I}} \cdot (e^{\mathtt{I} \cdot 2\pi k/n \cdot \tau} - e^{-\mathtt{I} \cdot 2\pi k/n \cdot \tau}) \tag{10.4}$$

are substituted into (10.1) for each k in $1 \leq k \leq n_2$, we obtain

$$\begin{aligned} f(\tau) &= a_0 + \frac{1}{2} \cdot \sum_{k=1}^{n_2} \left((a_k - \mathtt{I} \cdot b_k) \cdot e^{\mathtt{I} \cdot 2\pi k/n \cdot \tau} \right) \\ &\quad + \frac{1}{2} \cdot \sum_{k=1}^{n_2} \left((a_k + \mathtt{I} \cdot b_k) \cdot e^{-\mathtt{I} \cdot 2\pi k/n \cdot \tau} \right). \end{aligned} \tag{10.5}$$

Now since $e^{-\mathtt{I} \cdot 2\pi k/n \cdot \tau} = e^{\mathtt{I} \cdot 2\pi (n-k)/n \cdot \tau}$ for all integer values of τ, then

$$\begin{aligned} f(\tau) &= a_0 + \sum_{k=1}^{n_2} \left(\frac{a_k - \mathtt{I} \cdot b_k}{2} \cdot e^{\mathtt{I} \cdot 2\pi k/n \cdot \tau} \right) \\ &\quad + \sum_{k=n-n_2}^{n-1} \left(\frac{a_{n-k} + \mathtt{I} \cdot b_{n-k}}{2} \cdot e^{\mathtt{I} \cdot 2\pi k/n \cdot \tau} \right). \end{aligned} \tag{10.6}$$

If n is odd, then let us now define $c_0 = a_0$, $c_k = (a_k - \mathtt{I} \cdot b_k)/2$ for all k in $1 \leq k < n_2$, and $c_k = (a_{n-k} + \mathtt{I} \cdot b_{n-k})/2$ for all k in $n_2 + 1 \leq k < n$. If n is even, then define the c_k's in the same manner except that $c_{n_2} = a_{n_2}$. In either case, the unknown polynomial will have the form

$$f(\tau) = \sum_{k=0}^{n-1} c_k \cdot e^{\mathtt{I} \cdot 2\pi k/n \cdot \tau} \tag{10.7}$$

where each c_k is an element of $\mathbb{C}$, but $f(\tau)$ is a real-valued function. This is a consequence of the fact that c_k and c_{n-k} are complex conjugates.

Finally, apply the transformation $x = e^{\mathtt{I} \cdot 2\pi/n \cdot \tau}$. Then we obtain

$$f(x) = \sum_{d=0}^{n-1} f_d \cdot x^d \tag{10.8}$$

where $f_d = c_d$ for all $0 \leq d < n$. If we let $\omega = e^{\mathtt{I} \cdot 2\pi/n}$, then (10.8) can be used to compute $\{f(x = 1), f(\omega), f(\omega^2), \cdots, f(\omega^{n-1})\}$. If n is a power of two, then the coefficients of $f(x)$ can be recovered by applying any power-of-two inverse FFT algorithm found in Chapter 4. The coefficients of the discrete Fourier series $f(t)$ can then be determined using the formulas

$$a_0 = f_0 \tag{10.9}$$

$$a_k = f_k + f_{n-k} \tag{10.10}$$

$$a_{n/2} = f_{n/2} \tag{10.11}$$

$$b_k = \mathtt{I} \cdot (f_k - f_{n-k}) \tag{10.12}$$

for $0 \leq k \leq n/2 - 1$.

Because of the duality property of multiplicative FFT algorithms, $f(x)$ can also be recovered by any forward FFT algorithm with ω replaced by ω^{-1}. This accounts for the use of $W = e^{-\mathtt{I} \cdot 2\pi/n}$ as the primitive nth root of unity which frequently appears in engineering literature.

Assuming that the signal inputs are all elements of the real number system, it turns out that we can exploit the fact that $f(t)$ is a real-valued function to reduce the number of computations. Several algorithms for computing a real FFT in roughly half of the operations of an FFT with complex input are given in [53] and [14].

It should also be pointed out that the result of the engineer's version of the FFT gives the coefficients of the discrete Fourier series, but scaled by n/T_0. The most frequent use of the FFT in engineering is to perform the operation of convolution, essentially equivalent to the mathematician's operation of polynomial multiplication. The engineering version of the FFT is computed for two sequences of time samples to be convoluted. The result of these computations is two scaled discrete Fourier series. The coefficients of these discrete Fourier series can be pointwise multiplied to form the scaled discrete Fourier series of the desired output sequence. The engineering version of the inverse FFT is then computed to evaluate the Fourier series at the desired points in time. The result of this computation is the desired output, but scaled by n. Each component of the inverse FFT output is then multiplied by $1/n$ to undo the scaling and obtain the original signal samples. So convolution and polynomial multiplication essentially follow an identical sequence of steps, but with a different selection of the primitive root of unity. For this reason, engineering and mathematical FFT literature will appear very similar, but there will be subtle differences because the mathematician defines the FFT differently than the engineer.

10.2 Fast multipoint evaluation: revisited

Recall that in Chapter 1, we also presented an algorithm that could efficiently evaluate a polynomial f of degree [1] less than $n = 2^k$ with coefficients in some ring R at any set S consisting of n points in R. The reduction step of this algorithm consists of computing two remainders of a polynomial of degree less than $2m$ by a polynomial of degree m. From the discussion on the fast division algorithms presented in Chapter 7, we can reduce the number of multiplications needed to obtain each of these modular

[1] Again, the results in this section can be generalized to other selections of n if desired.

reductions from m^2 to $5 \cdot M_M(2m)$ where $M_M(2m)$ is number of multiplications in R needed to compute a product of polynomials of degree less than $2m$. Similarly, the number of additions needed to compute a modular reduction used in the algorithm is reduced from m^2 to $5 \cdot A_M(2m)$.

It turns out that in the case of this fast multipoint evaluation algorithm, we can reduce the operation counts associated with the remainder computations even more. Recall that we assumed that set S was fixed and that $\mathcal{M}_{i,j}$ could be precomputed for all $0 \leq i < k$ and $0 \leq j \leq 2^{k-i}$. In this case, all of the Newton inverses needed in the fast division algorithm can be precomputed and the cost of each modular reduction is decreased to $2 \cdot M_M(2m)$ multiplications and $2 \cdot A_M(2m)$ additions.

We are going to use these improvements to reduce the operation counts of the fast multipoint evaluation algorithm. Substituting the reduced operation counts for the modular reductions into the recurrence relations derived in Chapter 1, we obtain new recurrence relations given by

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + 4 \cdot M_M(n), \tag{10.13}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + 4 \cdot A_M(n). \tag{10.14}$$

As before, the operation counts for the case where $n = 1$ are given by $M(1) = 0$ and $A(1) = 0$. Master Equation VI can be used to solve these recurrence relations for the formulas given by:

$$M(n) \leq 4 \cdot M_M(n) \cdot \log_2(n), \tag{10.15}$$

$$A(n) \leq 4 \cdot A_M(n) \cdot \log_2(n). \tag{10.16}$$

For most practical cases where a multiplicative FFT can be used to implement the modular reduction, then these results imply that the multipoint evaluation can be implemented using no more than $\mathcal{O}(n \cdot (\log_2(n))^2)$ operations. If R is a finite field with 2^k elements where k is a power of two, then the multipoint evaluation can be implemented using no more than $\mathcal{O}(n \cdot (\log_2(n))^{2.585})$ operations. [2] If k is not a power of two, then $\mathcal{O}(n \cdot (\log_2(n))^3)$ operations are required. In the worst-case scenario where R has no special structure, then Karatsuba's algorithm can be used to implement the modular reductions. In this case, the multipoint evaluation can be implemented in $\mathcal{O}(n^{1.585} \cdot \log_2(n))$ operations.

10.3 Fast interpolation: revisited

In Chapter 1, a fast interpolation algorithm was also presented. This algorithm interpolates an collection of n evaluations of some function f at each of the points in some set S into the original function f. Each interpolation step receives as input two polynomials of degree less than m which were computed by two recursive calls to the fast interpolation algorithm. The interpolation step produces a polynomial of degree less than $2m$ using two multiplications of a polynomial of degree less than m by a polynomial of degree exactly m and one addition of two polynomials of degree less than $2m$. Replacing the multiplication method used in Chapter 1 with the improved multiplication methods covered in Chapter 5, we obtain new recurrence relations for the operation count of the fast interpolation algorithm given by

[2] These results are for the evaluation of a polynomial at an arbitrary collection of points. The results of Chapters 2 and 3 are for specially chosen collections of points.

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + 2 \cdot M_M(n), \tag{10.17}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + 2 \cdot A_M(n) + n \tag{10.18}$$

where $M(1) = 0$ and $A(1) = 0$. Master Equation VI can be used to solve these recurrence relations for the formulas given by:

$$M(n) \leq 2 \cdot M_M(n) \cdot \log_2(n), \tag{10.19}$$

$$A(n) \leq 2 \cdot A_M(n) \cdot \log_2(n) + n \cdot \log_2(n). \tag{10.20}$$

These results tell us that the number of operations to implement the fast interpolation algorithm is roughly equal to the number of operations required to complete the fast multipoint evaluation algorithm. So the fast interpolation algorithm can be implemented in at most $\mathcal{O}(n^{1.585} \cdot \log_2(n))$ operations by using Karatsuba multiplication to help perform the modular reductions.

10.4 Other research areas involving FFT algorithms

As we prepare to conclude this manuscript, this section presents some additional areas for research involving FFT algorithms besides trying to further improve upon the methods already presented. Many of the following topics have already been considered in the literature before, but some can be improved upon using the methods discussed in the previous chapters. To prevent this manuscript from becoming a two volume work, we will only briefly mention these areas here and leave the details for a later time or another author.

Factorization of polynomials over finite fields is covered in [34] and could be a next logical choice for a topic to continue in this research. The case where the coefficient ring is finite fields of characteristic 2 is covered in a separate paper [32]. The factorization technique covered in these sources works in three phases: (1) Square-free factorization which removes repeated factors in the polynomials; (2) Distinct-degree factorization which splits polynomials into products of factors of equal degree; and (3) Equal-degree factorization which completes the factorization of the polynomial. Each of these three phases replies upon computing the greatest common divisor. By incorporating the improvements to the greatest common divisor operation covered in Chapter 8 into the factorization routines discussed in [32] and [34], it is reasonable to expect a modest improvement in the performance of these algorithms. It may also be possible to improve upon the algorithms used to complete the three phases of the factorization process.

Another area of future research is multiplication of polynomials with finite field coefficients that do not support one of the FFT algorithms discussed in this manuscript. The most common example of this situation is GF(2) which only has two elements and cannot be used to multiply polynomials of product degree 2 or higher. As discussed in Chapter 5, the technique of clumping is used to map polynomials over GF(2) to polynomials over larger finite fields to handle this case. If one is forced to work over a finite field of size 2^k and k is not a power of two, a technique similar to clumping may be used to map polynomials defined over such a finite field to another finite field which can use the more efficient additive FFT routines. This may be an area to explore further if one needs to compute over one of these other finite fields.

Yet another area for future research involves multivariate polynomials. One technique for handling this situation is to map such polynomials into one-dimensional polynomials with sufficient zero-padding. Many of the coefficients in this polynomial

will be zero and there will be significant waste involved in FFTs which compute with these polynomials. One of the applications of the truncated Fast Fourier Transform presented in [41] and [42] is to efficiently compute over such multivariate polynomials, exploiting the known zeros. Since the author did not have an interest in computing over multivariate polynomials, he did not explore this topic any further. Additional algorithms for computing over multidimensional data structures are covered in [31].

If one wishes to compute over large data structures, he or she may consider researching methods of implementing any of the FFT algorithms presented in the previous chapters in parallel. A number of efficient parallel FFT algorithms are discussed in [14]. These algorithms can also be used to efficiently multiply polynomials using FFT-based multiplication. Open problems are how to compute division with remainder and the greatest common divisor in parallel. The present author started to investigate these problems, but has not yet made any progress in developing these parallel algorithms.

Finally, there are many applications of the FFT algorithm that can be improved upon by using the techniques discussed in this manuscript. Bernstein summarizes many of the applications of interest to mathmaticians in [3]. Brigham [9] contains an extensive bibliography which gives many other applications of the FFT algorithm to areas of science and engineering.

10.5 Concluding remarks

This chapter returned to the topics discussed at the beginning of this document and showed how the FFT algorithms discussed in the earlier chapters could be used to efficiently solve each of these problems. Topics for additional research were also discussed that may be further investigated by the author at a later date or by another individual.

The purpose of this manuscript was to explore several types of FFT algorithms and to explore several common applications of the FFT algorithm. The multiplicative FFT algorithms are extensively covered in the literature and this research effort was unable to improve any of these algorithms. However, the research did extend the work of Bernstein and presented each of these algorithms in terms of modular reductions, several for the first time. This study of the multiplicative FFT algorithms also allowed insights to be drawn which resulted in new algorithms for computing the additive FFT that can be applied to certain finite fields. The additive FFT was then used to derive a new multiplication method that is superior to Schönhage's algorithm for multiplying many polynomials over characteristic 2. Next, we introduced new truncated FFT algorithms which can be used to efficiently multiply polynomials with finite field coefficients of arbitrary degree. We then used the results of the earlier chapters to improve the performance of polynomial division, computation of the greatest common divisor, and decoding Reed-Solomon codes. It is expected that the performance of many other applications of the FFT can be improved as well, but this seems to be an appropriate stopping point for the current research effort.

One additional goal of this manuscript is to allow mathematicians and engineers to more easily translate each other's work so that both communities may be able to benefit from one another. While mathematicians will likely still use "i" and engineers will likely still use "j" to represent the imaginary element, mathematicians, engineers, and computer scientists would better appreciate each other's work if a common treatment of the Fast Fourier Transform was taught to these students in colleges and universities. The author hopes that this manuscript and future revisions of the material presented in this document will ultimately become one step toward the realization of this vision.

APPENDICES

Appendix A

Master equations for algorithm operation counts

Many of the algorithms in this document are governed by recurrence relations which are special cases of a few general recurrence relations. In this section of the appendix, we will develop closed-form equations for each of these general recurrence relations.

A technique called iteration will be used to develop the closed-form solutions. As discussed in [17], this method recursively substitutes the formula into itself with the appropriate input size. Initial conditions for the recurrence relation are used to end the recursion and obtain the closed-form formula.

Let us first consider the recurrence relation

$$F_1(n) = 2 \cdot F_1\left(\frac{n}{2}\right) + \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n + \mathcal{C} + \mathcal{D} \cdot n \cdot \log_2(n) + \mathcal{E} \cdot n \cdot c_{\log_2(n)-1} \tag{A.1}$$

where $F_1(1) = 0$. Here, $c_{\log_2(n)-1}$ is defined by 2^d where d is the number of ones in the binary expansion of $\log_2(n) - 1$. We are going to solve this recurrence relation using iteration:

$$\begin{aligned} F_1(n) &= 2 \cdot F_1\left(\frac{n}{2}\right) + \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n + \mathcal{C} + \mathcal{D} \cdot n \cdot \log_2(n) + \mathcal{E} \cdot n \cdot c_{\log_2(n)-1} \qquad \text{(A.2)} \\ &= 2 \cdot \left(2 \cdot F_1\left(\frac{n}{4}\right) + \mathcal{A} \cdot \left(\frac{n}{2}\right)^2 + \mathcal{B} \cdot \frac{n}{2} + \mathcal{C} + \mathcal{D} \cdot \left(\frac{n}{2}\right) \cdot \log_2\left(\frac{n}{2}\right) \right. \\ &\quad \left. + \mathcal{E} \cdot \frac{n}{2} \cdot c_{\log_2(n)-2}\right) + \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n + \mathcal{C} + \mathcal{D} \cdot n \cdot \log_2(n) + \mathcal{E} \cdot n \cdot c_{\log_2(n)-1} \end{aligned}$$

$$
\begin{aligned}
&= 4 \cdot F_1\left(\frac{n}{4}\right) + \left(1+\frac{1}{2}\right)\cdot\mathcal{A}\cdot n^2 + 2\cdot\mathcal{B}\cdot n + (1+2)\cdot\mathcal{C} \\
&\quad +\mathcal{D}\cdot n\cdot\left(\log_2(n)+\log_2\left(\frac{n}{2}\right)\right) + \mathcal{E}\cdot n\cdot\left(c_{\log_2(n)-1}+c_{\log_2(n)-2}\right) \\
&= 8\cdot F_1\left(\frac{n}{8}\right) + \left(1+\frac{1}{2}+\frac{1}{4}\right)\cdot\mathcal{A}\cdot n^2 + 3\cdot\mathcal{B}\cdot n + (1+2+4)\cdot\mathcal{C} \\
&\quad +\mathcal{D}\cdot n\cdot\left(\log_2(n)+\log_2\left(\frac{n}{2}\right)+\log_2\left(\frac{n}{4}\right)\right) \\
&\quad +\mathcal{E}\cdot n\cdot\left(c_{\log_2(n)-1}+c_{\log_2(n)-2}+c_{\log_2(n)-3}\right) \\
&= \ldots \\
&= n\cdot F_1(1) + \left(1+\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{n/2}\right)\cdot\mathcal{A}\cdot n^2 + \mathcal{B}\cdot n\cdot\log_2(n) \\
&\quad +\mathcal{C}\cdot\left(1+2+4+\cdots+\frac{n}{2}\right) \\
&\quad +\mathcal{D}\cdot n\cdot\left(\log_2(n)+\log_2\left(\frac{n}{2}\right)+\log_2\left(\frac{n}{4}\right)+\cdots+1\right) \\
&\quad +\mathcal{E}\cdot n\cdot\left(c_{\log_2(n)-1}+c_{\log_2(n)-2}+c_{\log_2(n)-3}+\cdots+c_{\log_2(n)-\log_2(n)}\right) \\
&= 2\cdot\left(1-\frac{1}{n}\right)\cdot\mathcal{A}\cdot n^2 + \mathcal{B}\cdot n\cdot\log_2(n) + \mathcal{C}\cdot(n-1) \\
&\quad +\frac{1}{2}\cdot\mathcal{D}\cdot n\cdot\left((\log_2(n))^2+\log_2(n)\right) \\
&\quad +\mathcal{E}\cdot n\cdot\left(c_{\log_2(n)-1}+c_{\log_2(n)-2}+c_{\log_2(n)-3}+\cdots+c_0\right) \\
&= 2\mathcal{A}\cdot(n^2-n) + \mathcal{B}\cdot n\cdot\log_2(n) + \mathcal{C}\cdot(n-1) + \frac{1}{2}\cdot\mathcal{D}\cdot n\cdot\left((\log_2(n))^2+\log_2(n)\right) \\
&\quad +\mathcal{E}\cdot n\cdot(c_{\log_2(n)-1}+c_{\log_2(n)-2}+c_{\log_2(n)-3}+\cdots+c_0).
\end{aligned}
$$

The final line of (A.2) is referred to as Master Equation I throughout the manuscript and is used to derive operation count formulas of the form (A.1). Here, we made use of the identity

$$
\sum_{i=0}^{k} a^i = \frac{a^{k+1}-1}{a-1} \tag{A.3}
$$

which holds for all $a \neq 1$. In the derivation above, the identity was used with $a = 2$ and $a = 1/2$.

In [12], Cantor explains that $c_{\log_2(n)-1} + c_{\log_2(n)-2} + c_{\log_2(n)-3} + \cdots + c_0$ is equal to $(\log_2(n))^{\log_2(3)} \approx (\log_2(n))^{1.585}$ when $n = 2^k$ and k is a power of two. This can be seen by applying the Binomial Theorem to $(1+2)^{\log_2(n)}$ and then using properties of logarithms to show that $3^{\log_2(n)} = \log_2(n)^{\log_2(3)}$. If k is not a power of two, then $\log_2(n)^{\log_2(3)}$ is an upper bound for $c_{\log_2(n)-1} + c_{\log_2(n)-2} + c_{\log_2(n)-3} + \cdots + c_0$.

By properly setting the constants of Master Equation I, this formula can also be used to solve recurrence relations of the form

$$F_1(n) = 4 \cdot F_1\left(\frac{n}{4}\right) + \left(1 + \frac{1}{2}\right) \cdot \mathcal{A} \cdot n^2 + 2 \cdot \mathcal{B} \cdot n + (1+2) \cdot \mathcal{C}. \tag{A.4}$$

In this case, then the solution is given by (A.2) or

$$\begin{aligned} F_1(n) &= \frac{n}{2} \cdot F_1(2) + \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n/4}\right) \cdot \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n \cdot (\log_2(n) - 1) \\ &\quad + \mathcal{C} \cdot \left(1 + 2 + 4 + \cdots + \frac{n}{4}\right) \\ &= \frac{n}{2} \cdot F_1(2) + \left(2 - \frac{4}{n}\right) \cdot \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n \cdot (\log_2(n) - 1) + \mathcal{C} \cdot \left(\frac{n}{2} - 1\right) \end{aligned} \tag{A.5}$$

if n is not divisible by a power of four.

Similarly, Master Equation I can be used to solve recurrence relations of the form

$$F_1(n) \;=\; 8 \cdot F_1\left(\frac{n}{8}\right) + \left(1 + \frac{1}{2} + \frac{1}{4}\right) \cdot \mathcal{A} \cdot n^2 + 3 \cdot \mathcal{B} \cdot n + (1 + 2 + 4) \cdot \mathcal{C}. \tag{A.6}$$

In this case, then the solution can be given by (A.2), (A.5) or

$$\begin{aligned} F_1(n) \;&=\; \frac{n}{4} \cdot F_1(4) + \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n/8}\right) \cdot \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n \cdot (\log_2(n) - 2) \\ &\quad + \mathcal{C} \cdot \left(1 + 2 + 4 + \cdots + \frac{n}{8}\right) \qquad (A.7) \\ &=\; \frac{n}{4} \cdot F_1(4) + \left(2 - \frac{8}{n}\right) \cdot \mathcal{A} \cdot n^2 + \mathcal{B} \cdot n \cdot (\log_2(n) - 2) + \mathcal{C} \cdot \left(\frac{n}{4} - 1\right) \end{aligned}$$

if n is not divisible by a power of 8.

Next, let us consider the recurrence relation

$$F_2(n) \;=\; F_2\left(\frac{n}{2}\right) + \mathcal{A} \cdot n \tag{A.8}$$

where $F_2(1) = 0$. We are now going to solve this recurrence relation using iteration:

$$
\begin{aligned}
F_2(n) &= F_2\left(\frac{n}{2}\right) + \mathcal{A}\cdot n \\
&= \left(F_2\left(\frac{n}{4}\right) + \mathcal{A}\cdot\frac{n}{2}\right) + \mathcal{A}\cdot n \\
&= F_2\left(\frac{n}{4}\right) + \left(1+\frac{1}{2}\right)\cdot\mathcal{A}\cdot n \\
&= F_2\left(\frac{n}{8}\right) + \left(1+\frac{1}{2}+\frac{1}{4}\right)\cdot\mathcal{A}\cdot n \\
&= \cdots \\
&= F_2(1) + \left(1+\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{n/2}\right)\cdot\mathcal{A}\cdot n \\
&= 2\cdot\mathcal{A}\cdot\left(1-\frac{1}{n}\right)\cdot n \\
&= 2\cdot\mathcal{A}\cdot(n-1).
\end{aligned}
\tag{A.9}
$$

The final line of (A.9) is referred to as as Master Equation II throughout the manuscript and used to solve operation count formulas of the form (A.8). Here, we made use of (A.3) with $a = 1/4$. By properly setting the constants of Master Equation II, this formula can also be used to solve recurrence relations of the form

$$
F_2(n) = F_2\left(\frac{n}{4}\right) + \left(1+\frac{1}{2}\right)\cdot\mathcal{A}\cdot n. \tag{A.10}
$$

The solution is given by (A.9) or

$$
\begin{aligned}
F_2(n) &= F_2(2) + \left(1+\frac{1}{2}+\frac{1}{4}+\cdots+\frac{1}{n/4}\right)\cdot\mathcal{A}\cdot n \\
&= F_2(2) + 2\cdot\mathcal{A}\cdot(n-2)
\end{aligned}
\tag{A.11}
$$

if n is not a power of four.

Similarly, equations of the form

$$F_2(n) = F_2\left(\frac{n}{8}\right) + \left(1 + \frac{1}{2} + \frac{1}{4}\right) \cdot \mathcal{A} \cdot n \tag{A.12}$$

can be solved by using (A.9), (A.11) or

$$\begin{aligned} F_2(n) &= F_2(4) + \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n/8}\right) \cdot \mathcal{A} \cdot n \\ &= F_2(4) + 2 \cdot \mathcal{A} \cdot (n-4) \end{aligned} \tag{A.13}$$

if n is not a power of eight.

Now, let consider the recurrence relation

$$F_3(n) = 3 \cdot F_3\left(\frac{n}{3}\right) + \mathcal{A} \cdot n + \mathcal{B} \tag{A.14}$$

where $F_3(1) = 0$. We are going to solve this recurrence relation using iteration:

$$
\begin{aligned}
F_3(n) &= 3 \cdot F_3\left(\frac{n}{3}\right) + \mathcal{A} \cdot n + \mathcal{B} \qquad \text{(A.15)} \\
&= 3 \cdot \left(3 \cdot F_3\left(\frac{n}{9}\right) + \mathcal{A} \cdot \frac{n}{3} + \mathcal{B}\right) + \mathcal{A} \cdot n + \mathcal{B} \\
&= 9 \cdot F_3\left(\frac{n}{9}\right) + 2 \cdot \mathcal{A} + (1+3) \cdot \mathcal{B} \\
&= 27 \cdot F_3\left(\frac{n}{27}\right) + 3 \cdot \mathcal{A} + (1+3+9) \cdot \mathcal{B} \\
&= \cdots \\
&= n \cdot F_3(1) + \log_3(n) \cdot \mathcal{A} \cdot n + \left(1+3+9+\cdots+\frac{n}{3}\right) \cdot \mathcal{B} \\
&= n \cdot \log_3(n) \cdot \mathcal{A} + \frac{1}{2} \cdot (n-1) \cdot \mathcal{B}.
\end{aligned}
$$

The final line of (A.15) is referred to as Master Equation III throughout the manuscript and is used to solve recurrence relations of the form (A.14). Here, we made use of (A.3) with $a = 3$.

The next recurrence relation to be considered is given by

$$
F_4(n) = 2 \cdot \sqrt{n} \cdot F_4(\sqrt{n}) + \mathcal{A} \cdot n \cdot \log_2(n) + \mathcal{B} \cdot \sqrt{n} \cdot \log_2(\sqrt{n}) \qquad \text{(A.16)}
$$

where $F_4(2)$ is a given initial condition. We will also solve this recurrence relation using iteration. Let $n = 2^{2^I}$. Then

$$
\begin{aligned}
F_4\left(2^{2^I}\right) &= 2\cdot 2^{2^{I-1}}\cdot F_4\left(2^{2^{I-1}}\right)+\mathcal{A}\cdot 2^{2^I}\cdot 2^I+\mathcal{B}\cdot 2^{2^{I-1}}\cdot 2^{I-1} \qquad \text{(A.17)}\\
&= 2\cdot 2^{2^{I-1}}\cdot\left(2\cdot 2^{2^{I-2}}\cdot F_4\left(2^{2^{I-2}}\right)+\mathcal{A}\cdot 2^{2^{I-1}}\cdot 2^{I-1}\right.\\
&\quad \left.+\mathcal{B}\cdot 2^{2^{I-2}}\cdot 2^{I-2}\right)+\mathcal{A}\cdot 2^{2^I}\cdot 2^I+\mathcal{B}\cdot 2^{2^{I-1}}\cdot 2^{I-1}\\
&= 2^2\cdot 2^{2^{I-1}+2^{I-2}}\cdot F_4\left(2^{2^{I-2}}\right)+2\cdot\mathcal{A}\cdot 2^{2^I}\cdot 2^I\\
&\quad +\mathcal{B}\cdot 2^{I-1}\cdot\left(2^{2^{I-1}+2^{I-2}}+2^{2^{I-1}}\right)\\
&= 2^3\cdot 2^{2^{I-1}+2^{I-2}+2^{I-3}}\cdot F_4\left(2^{2^{I-3}}\right)+3\cdot\mathcal{A}\cdot 2^{2^I}\cdot 2^I\\
&\quad +\mathcal{B}\cdot 2^{I-1}\cdot\left(2^{2^{I-1}+2^{I-2}+2^{I-3}}+2^{2^{I-1}+2^{I-2}}+2^{2^{I-1}}\right)\\
&\quad \cdots\\
&= 2^I\cdot 2^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots+2^0}\cdot F_4(2)+\mathcal{A}\cdot 2^{2^I}\cdot 2^I\cdot I\\
&\quad +\mathcal{B}\cdot 2^{I-1}\cdot\left(2^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots+1}+2^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots+2}+\cdots+2^{2^{I-1}}\right)\\
&= 2^I\cdot 2^{2^I-1}\cdot F_4(2)+\mathcal{A}\cdot 2^{2^I}\cdot 2^I\cdot I\\
&\quad +\mathcal{B}\cdot 2^{I-1}\cdot\left(\frac{1}{2^{2^0}}\cdot 2^{2^I}+\frac{1}{2^{2^1}}\cdot 2^{2^I}+\cdots+\frac{1}{2^{2^{I-1}}}\cdot 2^{2^I}\right)\\
&= 2^I\cdot 2^{2^I-1}\cdot F_4(2)+\mathcal{A}\cdot 2^{2^I}\cdot 2^I\cdot I\\
&\quad +\mathcal{B}\cdot 2^{I-1}\cdot 2^{2^I}\cdot\left(\frac{1}{2^{2^0}}+\frac{1}{2^{2^1}}+\cdots+\frac{1}{2^{2^{I-1}}}\right).
\end{aligned}
$$

So Master Equation IV is given by

$$
\begin{aligned}
F_4(n) &= \frac{1}{2}\cdot n\cdot\log_2(n)\cdot F_4(2)+\mathcal{A}\cdot n\cdot\log_2(n)\cdot\log_2\log_2(n)\\
&\quad +\frac{1}{2}\cdot\mathcal{B}\cdot\Lambda\cdot n\cdot\log_2(n) \qquad \text{(A.18)}
\end{aligned}
$$

where

$$\Lambda \;=\; \sum_{i=0}^{I-1} \left(\frac{1}{2}\right)^{2^i} \tag{A.19}$$

and Λ is bounded by $1/2 \leq \Lambda < 1$.

The next Master Equation is not recursively defined, but involves iteration to determine its solution. Let $G(n)$ be a function such that $2 \cdot G(n) \leq G(2n)$. Thus, $G(n) \leq 1/2 \cdot G(2n)$. Then let $F_5(n)$ be defined by

$$F_5(n) \;=\; \sum_{i=1}^{k} \left(\mathcal{A} \cdot G(2^{i+1}) + \mathcal{B} \cdot 2^i\right) \tag{A.20}$$

where $k = \log_2(n)$. An upper bound for the solution to this equation is given by

$$\begin{aligned}
F_5(n) &= \sum_{i=1}^{k} \left(\mathcal{A} \cdot G(2^{i+1}) + \mathcal{B} \cdot 2^i\right) \qquad (A.21)\\
&= \mathcal{A} \cdot \sum_{i=1}^{k} G(2^{i+1}) + \mathcal{B} \sum_{i=1}^{k} \cdot 2^i \\
&= \mathcal{A} \cdot \left(G(2^{k+1}) + \sum_{i=1}^{k-1} G(2^{i+1})\right) + \mathcal{B} \cdot (2^{k+1} - 1) \\
&\leq \mathcal{A} \cdot \left(G(2^{k+1}) + \sum_{i=1}^{k-1} \frac{1}{2} \cdot G(2^{i+2})\right) + \mathcal{B} \cdot (2^{k+1} - 1) \\
&= \mathcal{A} \cdot \left(G(2^{k+1}) \cdot \left(1 + \frac{1}{2}\right) + \sum_{i=1}^{k-2} \frac{1}{2} \cdot G(2^{i+2})\right) + \mathcal{B} \cdot (2^{k+1} - 1)
\end{aligned}$$

$$
\begin{aligned}
F_5(n) &\leq \mathcal{A} \cdot \left(G(2^{k+1}) \cdot \left(1 + \frac{1}{2} + \frac{1}{4} \right) + \sum_{i=1}^{k-3} \frac{1}{4} \cdot G(2^{i+3}) \right) + \mathcal{B} \cdot (2^{k+1} - 1) \\
&\quad \cdots \\
&\leq \mathcal{A} \cdot G(2^{k+1}) \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{n} \right) + \mathcal{B} \cdot (2^{k+1} - 1) \\
&< 2 \cdot \mathcal{A} \cdot G(2n) + 2 \cdot \mathcal{B} \cdot n
\end{aligned}
$$

where the last row of this analysis is Master Equation V.

Finally, consider the recurrence relation given by

$$
F_6(n) = 2 \cdot F_6\left(\frac{n}{2}\right) + \mathcal{A} \cdot G(n) + \mathcal{B} \cdot n \tag{A.22}
$$

where $G(n)$ is a function that has the same properties as considered in Master Equation V and $F_6(1) = 0$. We will now compute an upper bound for $F_6(n)$ using iteration:

$$
\begin{aligned}
F_6(n) &= 2 \cdot F_6\left(\frac{n}{2}\right) + \mathcal{A} \cdot G(n) + \mathcal{B} \cdot n \qquad \text{(A.23)} \\
&= 2 \cdot \left(2 \cdot F_6\left(\frac{n}{4}\right) + \mathcal{A} \cdot G\left(\frac{n}{2}\right) + \mathcal{B} \cdot \frac{n}{2} \right) + \mathcal{A} \cdot G(n) + \mathcal{B} \cdot n \\
&= 2^2 \cdot F_6\left(\frac{n}{4}\right) + \mathcal{A} \cdot 2 \cdot G\left(\frac{n}{2}\right) + \mathcal{A} \cdot G(n) + 2 \cdot \mathcal{B} \cdot n \\
&\leq 2^2 \cdot F_6\left(\frac{n}{4}\right) + 2 \cdot \mathcal{A} \cdot G(n) + 2 \cdot \mathcal{B} \cdot n \\
&\leq 2^3 \cdot F_6\left(\frac{n}{8}\right) + 3 \cdot \mathcal{A} \cdot G(n) + 3 \cdot \mathcal{B} \cdot n \\
&\quad \cdots \\
&\leq n \cdot F_6(1) + \log_2(n) \cdot \mathcal{A} \cdot G(n) + \mathcal{B} \cdot n \cdot \log_2(n) \\
&= \mathcal{A} \cdot G(n) \cdot \log_2(n) + \mathcal{B} \cdot n \cdot \log_2(n).
\end{aligned}
$$

The final row of (A.23) is referred to as Master Equation VI in the manuscript and can be used to solve operation count formulas of the form (A.22).

Appendix B

Operation count: split-radix FFT

The total number of operations to compute the FFT of size n is

$$M(n) = M\left(\frac{n}{2}\right) + 2 \cdot M\left(\frac{n}{4}\right) + \frac{1}{2} \cdot n - 2, \tag{B.1}$$

$$A(n) = A\left(\frac{n}{2}\right) + 2 \cdot A\left(\frac{n}{4}\right) + \frac{3}{2} \cdot n \tag{B.2}$$

where $M(1) = 0$, $A(1) = 0$, $M(2) = 0$, and $A(2) = 2$.

We are going to use a combinatorics-based technique to solve for these operation counts. This technique is developed in [65] and [78] for the case of linear, homogeneous recurrence relations and limited nonhomogeneous cases and can be used as an alternative method of deriving a number of the Master Equations. The combinatorics text [55] covers additional nonhomogenous cases and can be used to solve the above recurrence relations.

First, let us solve for $M(n)$. Let $n = 2^k$ and rewrite the recurrence relation as $m_k = m_{k-1} + 2 \cdot m_{k-2} + 2^{k-1} - 2$ where $m_i = M(2^i)$. The characteristic polynomial of the homogeneous part of the recurrence relation, i.e. $m_k = m_{k-1} + 2 \cdot m_{k-2}$, is $x^2 - x - 2 = (x + 1) \cdot (x - 2)$. The general solution to this recurrence relation is $m_k = c_1' \cdot (2)^k + c_2' \cdot (-1)^k$ where c_1' and c_2' need to be determined using the initial conditions. According to the technique covered in [55], the solution to the given recurrence relation will have the form $m_k = c_1 \cdot k \cdot (2)^k + c_2 \cdot (2)^k + c_3 \cdot (-1)^k + c_4 \cdot k + c_5$. We are given that $m_0 = 0$ and $m_1 = 0$. We can use the recurrence relation to determine that $m_2 = 0$, $m_3 = 2$, and $m_4 = 8$. Now, solve the system of equations

$$\begin{array}{rcrcrcrcrcl}
0 \cdot c_1 & + & 1 \cdot c_2 & + & 1 \cdot c_3 & + & 0 \cdot c_4 & + & 1 \cdot c_5 & = 0, \\
2 \cdot c_1 & + & 2 \cdot c_2 & + & (-1) \cdot c_3 & + & 1 \cdot c_4 & + & 1 \cdot c_5 & = 0, \\
8 \cdot c_1 & + & 4 \cdot c_2 & + & 1 \cdot c_3 & + & 2 \cdot c_4 & + & 1 \cdot c_5 & = 0, \\
24 \cdot c_1 & + & 8 \cdot c_2 & + & (-1) \cdot c_3 & + & 3 \cdot c_4 & + & 1 \cdot c_5 & = 2, \\
64 \cdot c_1 & + & 16 \cdot c_2 & + & 1 \cdot c_3 & + & 4 \cdot c_4 & + & 1 \cdot c_5 & = 8
\end{array} \tag{B.3}$$

to obtain $c_1 = \frac{1}{3}$, $c_2 = -\frac{8}{9}$, $c_3 = -\frac{1}{9}$, $c_4 = 0$ and $c_5 = 1$. Thus, the closed-form formula for $M(n)$ is

$$M(n) = \begin{cases} \frac{1}{3} \cdot n \cdot \log_2(n) - \frac{8}{9} \cdot n + \frac{8}{9} & \text{if } \log_2(n) \text{ is even} \\ \frac{1}{3} \cdot n \cdot \log_2(n) - \frac{8}{9} \cdot n + \frac{10}{9} & \text{if } \log_2(n) \text{ is odd.} \end{cases} \tag{B.4}$$

Now, let us solve for $A(n)$. Let $n = 2^k$ and rewrite the recurrence relation as $a_k = a_{k-1} + 2 \cdot a_{k-2} + \frac{3}{2} \cdot 2^k$ where $a_i = A(2^i)$. The characteristic polynomial of the homogeneous part of the recurrence relation, i.e. $a_k = a_{k-1} + 2 \cdot a_{k-2}$, is $x^2 - x - 2 = (x + 1) \cdot (x - 2)$. The general solution to this recurrence relation is the same as m_k. The solution to a_k will have the form $a_k = (c_1 \cdot k + c_2) \cdot (2)^k + c_3 \cdot (-1)^k$. We are given that $a_0 = 0$ and $a_1 = 2$. We can use the recurrence relation to determine that $a_2 = 8$. Now, solve the system of equations

$$\begin{array}{rcrcrcl}
0 \cdot c_1 & + & 1 \cdot c_2 & + & 1 \cdot c_3 & = 0, \\
2 \cdot c_1 & + & 2 \cdot c_2 & + & (-1) \cdot c_3 & = 2, \\
8 \cdot c_1 & + & 4 \cdot c_2 & + & 1 \cdot c_3 & = 8
\end{array} \tag{B.5}$$

to obtain $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$. Thus, the closed-form formula for $A(n)$ is

$$A(n) \;=\; n \cdot \log_2(n). \tag{B.6}$$

The number of multiplications by primitive 8th roots of unity in the split-radix algorithm is given by the recurrence relation

$$M_8(n) \;=\; M_8\left(\frac{n}{2}\right) + 2 \cdot M_8\left(\frac{n}{4}\right) + 2 \tag{B.7}$$

for $n \geq 8$ where $M_8(1) = 0$, $M_8(2) = 0$ and $M_8(4) = 0$.

Let us solve for $M_8(n)$. If $n = 2^k$, we can rewrite the recurrence relation as $\psi_k = \psi_{k-1} + 2 \cdot \psi_{k-2} + 2$ where $\psi_i = M_8(2^i)$. The characteristic polynomial of the homogeneous part of the recurrence relation, i.e. $\psi_k = \psi_{k-1} + 2 \cdot \psi_{k-2}$, is again $x^2 - x - 2 = (x+1) \cdot (x-2)$. The general solution to this recurrence relation is the same as the general solution for m_k and a_k. The solution to the given recurrence relation will have the form $\psi_k = c_1 \cdot (2)^k + c_2 \cdot (-1)^k + c_3 \cdot k + c_4$. We are given that $\psi_1 = 0$ and $\psi_2 = 0$. We can use the recurrence relation to determine that $\psi_3 = 2$, $\psi_4 = 4$. Now, solve the system of equations

$$\begin{array}{rcrcrcrl}
2 \cdot c_1 & + & (-1) \cdot c_2 & + & 1 \cdot c_3 & + & 1 \cdot c_4 & = 0, \\
4 \cdot c_1 & + & 1 \cdot c_2 & + & 2 \cdot c_3 & + & 1 \cdot c_4 & = 0, \\
8 \cdot c_1 & + & (-1) \cdot c_2 & + & 3 \cdot c_3 & + & 1 \cdot c_4 & = 2, \\
16 \cdot c_1 & + & 1 \cdot c_2 & + & 4 \cdot c_3 & + & 1 \cdot c_4 & = 4
\end{array} \tag{B.8}$$

to obtain $c_1 = \frac{1}{3}$, $c_2 = -\frac{1}{3}$, $c_3 = 0$, and $c_4 = -1$. Thus, a closed-form formula for $M_8(n)$ is

$$M_8(n) = \begin{cases} \frac{1}{3} \cdot n - \frac{4}{3} & \text{if } \log_2(n) \text{ is even} \\ \frac{1}{3} \cdot n - \frac{2}{3} & \text{if } \log_2(n) \text{ is odd} \end{cases} \tag{B.9}$$

which holds for all $n > 1$. When $n = 1$, then $M_8(1) = 0$.

Appendix C

Additional details of the modified split-radix FFT

In this section of the appendix, we will first present the four different reduction steps of the new split-radix algorithm. In each version, we will use the notation $f^{\circ}_{4m,j}$ to indicate the polynomial where the degree d coefficient of $f(\omega^{\sigma'(j)/(4m)} \cdot x) \bmod (x^{4m}-1)$ has been scaled by $S_{4m,d}$ for all d in $0 \leq d < 4m$ where $S_{4m,d}$ was defined in Chapter 2.

Version A will be used for every reduction step where $j = 0$. It receives as input the unscaled polynomial $f^{\circ}_{4m,0}$. Its role is to introduce the scaling factor $S_{m,d}$ into each term of $f^{\circ}_{m,2}$ and $f^{\circ}_{m,3}$ so that the scaled twisted polynomials can be used on the reduction steps that receive these two polynomials as input. Pseudocode for this reduction step is given in Figure C.1.

Version B of the reduction step is the most favorable case that takes advantage of the scaled twisted polynomials. In order to use this reduction step, the input polynomials must be scaled by $S_{4m,d}$ for all $d < 4m$ and $S_{4m,d} = S_{4m,d+m} = S_{4m,d+2m} = S_{4m,d+3m}$ for all $d \leq m$. We will defer adjustment of $f^{\circ}_{2m,2j}$ until a lower level reduction step. The two results of size m will always be able to use version B for each of the four versions of the algorithm. Pseudocode for this reduction step is given in Figure C.2.

Versions C and D of the reduction step handle the two cases where $f^{\circ}_{2m,2j}$ has not been adjusted in the previous reduction step. In version C, the input will be scaled by $S_{8m,d}$ where $S_{8m,d} = S_{8m,d+2m}$ for all $d \leq 2m$, but $S_{8m,d} \neq S_{8m,d+m}$ for all $d \leq 3m$. In this case, the pairs $\{f_A, f_C\}$ and $\{f_B, f_D\}$ can be combined into f_α and f_β since the same scaling factor is used for each degree term in each pair. When f_α and f_β are combined into f_Y and f_Z, different scaling factors appear in common degree terms of these two expressions. We will scale each term of these two expressions in

Algorithm : Modified split-radix FFT (version A reduction step)

Input: The unscaled polynomial $f^{\circ}_{4m,0}$ where $4m$ is a power of two.

Output: $f(\omega^{\sigma'(0)}), f(\omega^{\sigma'(1)}), \ldots, f(\omega^{\sigma'(4m-1)})$.

0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(0)}) = f(\omega^{\sigma'(0)} \cdot x) \bmod (x-1)$.

0B. If $(4m) = 2$, then call a radix-2 algorithm to compute the FFT.

1. Split $f^{\circ}_{4m,0}$ into four blocks f_A, f_B, f_C, and f_D
   each of size m such that $f^{\circ}_{4m,0} = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
2. Compute $f^{\circ}_{2m,0} = f_W \cdot x^m + f_X = (f_A + f_C) \cdot x^m + (f_B + f_D)$.
3. Compute $f_\alpha = -f_B + f_D$.
4. Compute $f_\beta = \mathfrak{I} \cdot (-f_A + f_C)$.
5. Compute $f_Y = f_\alpha + f_\beta$.
6. Compute $f_Z = -f_\alpha + f_\beta$.
7. Compute $f^{\circ}_{m,2}$ by multiplying f_Y by $\omega^{\sigma'(2)/m} \cdot S_{m,d}$ for all $d < m$.
8. Compute $f^{\circ}_{m,3}$ by multiplying f_Z by $\omega^{-\sigma'(2)/m} \cdot S_{m,d}$ for all $d < m$.
9. Compute the FFT of $f^{\circ}_{2m,0}$ using the version A reduction step to obtain
   $f(\omega^{\sigma'(0)}), f(\omega^{\sigma'(1)}), \ldots, f(\omega^{\sigma'(2m-1)})$.
10. Compute the FFT of $f^{\circ}_{m,2}$ using the version B reduction step to obtain
    $f(\omega^{\sigma'(2m)}), f(\omega^{\sigma'(2m+1)}), \ldots, f(\omega^{\sigma'(3m-1)})$.
11. Compute the FFT of $f^{\circ}_{m,3}$ using the version B reduction step to obtain
    $f(\omega^{\sigma'(3m)}), f(\omega^{\sigma'(3m+1)}), \ldots, f(\omega^{\sigma'(4m-1)})$.
12. Return $f(\omega^{\sigma'(0)}), f(\omega^{\sigma'(1)}), \ldots, f(\omega^{\sigma'(4m-1)})$.

Figure C.1 Pseudocode for modified split-radix FFT (version A reduction step)

Algorithm : Modified split-radix FFT (version B reduction step)

Input: The polynomial $f^{\circ}_{4m,j}$ that has been scaled by $S_{4m,d}$ for all $d < 4m$.
Here, $S_{4m,d} = S_{4m,d+m} = S_{4m,d+2m} = S_{4m,d+3m}$ for all $d \leq m$.

Output: $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.

0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(j)}) = f(\omega^{\sigma'(j)} \cdot x) \bmod (x-1)$.
0B. If $(4m) = 2$, then call a radix-2 algorithm to compute the FFT.
1. Split $f^{\circ}_{4m,j}$ into four blocks f_A, f_B, f_C, and f_D
each of size m such that $f^{\circ}_{4m,j} = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
2. Compute $f^{\circ}_{2m,2j} = f_W \cdot x^m + f_X = (f_A + f_C) \cdot x^m + (f_B + f_D)$.
3. Compute $f_\alpha = -f_B + f_D$.
4. Compute $f_\beta = \mathtt{I} \cdot (-f_A + f_C)$.
5. Compute $f_Y = f_\alpha + f_\beta$.
6. Compute $f_Z = -f_\alpha + f_\beta$.
7. Compute $f^{\circ}_{m,4j+2}$ by multiplying f_Y by $T_{4m,d}$ for all $d < m$.
8. Compute $f^{\circ}_{m,4j+3}$ by multiplying f_Z by $\overline{T_{4m,d}}$ for all $d < m$.
Note: $\overline{T_{4m,d}}$ is the complex conjugate of $T_{4m,d}$.
9. Compute the FFT of $f^{\circ}_{2m,2j}$ using the version C reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+2m-1)})$.
10. Compute the FFT of $f^{\circ}_{m,4j+2}$ using the version B reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+2m)}), f(\omega^{\sigma'(j\cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+3m-1)})$.
11. Compute the FFT of $f^{\circ}_{m,4j+3}$ using the version B reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+3m)}), f(\omega^{\sigma'(j\cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.
12. Return $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.

Figure C.2 Pseudocode for modified split-radix FFT (version B reduction step)

Algorithm : Modified split-radix FFT (version C reduction step)

Input: The polynomial $f^{\circ}_{4m,j}$ that has been scaled by $S_{8m,d}$ for all $d < 4m$.
Here, $S_{8m,d} = S_{8m,d+2m}$ for all $d \leq 2m$, but $S_{8m,d} \neq S_{8m,d+m}$ for all $d \leq 3m$.

Output: $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.

0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(j)}) = f(\omega^{\sigma'(j)} \cdot x) \bmod (x-1)$.
0B. If $(4m) = 2$, then call a radix-2 algorithm to compute the FFT.
1. Split $f^{\circ}_{4m,j}$ into four blocks f_A, f_B, f_C, and f_D
each of size m such that $f^{\circ}_{4m,j} = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
2. Compute $f^{\circ}_{2m,2j} = f_W \cdot x^m + f_X = (f_A + f_C) \cdot x^m + (f_B + f_D)$.
3. Compute $f_\alpha = -f_B + f_D$.
3A. Compute $f_{\alpha'}$ by scaling term d of f_α by $(S_{4m,d}/S_{8m,d})$ for all $d < m$.
4. Compute $f_\beta = \mathtt{I} \cdot (-f_A + f_C)$.
4A. Compute $f_{\beta'}$ by scaling term d of f_β by $(S_{4m,d}/S_{8m,d+2m})$ for all $d < m$.
5. Compute $f_Y = f_{\alpha'} + f_{\beta'}$.
6. Compute $f_Z = -f_{\alpha'} + f_{\beta'}$.
7. Compute $f^{\circ}_{m,4j+2}$ by multiplying f_Y by $T_{4m,d}$ for all $d < m$.
8. Compute $f^{\circ}_{m,4j+3}$ by multiplying f_Z by $\overline{T_{4m,d}}$ for all $d < m$.
Note: $\overline{T_{4m,d}}$ is the complex conjugate of $T_{4m,d}$.
9. Compute the FFT of $f^{\circ}_{2m,2j}$ using the version D reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+2m-1)})$.
10. Compute the FFT of $f^{\circ}_{m,4j+2}$ using the version B reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+2m)}), f(\omega^{\sigma'(j\cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+3m-1)})$.
11. Compute the FFT of $f^{\circ}_{m,4j+3}$ using the version B reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+3m)}), f(\omega^{\sigma'(j\cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.
12. Return $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.

Figure C.3 Pseudocode for modified split-radix FFT (version C reduction step)

such a way as to set up $f^{\circ}_{2m,4j+2}$ and $f^{\circ}_{2m,4j+3}$ for the more favorable version B in the next reduction step as well as implement the adjustment on $f^{\circ}_{4m,j}$ that was put off in the previous reduction step. Pseudocode for the version C reduction step is given in Figure C.3.

Version D of the reduction step is the least favorable case. Here, the input has been subdivided into f_A, f_B, f_C, and f_D, where none of the corresponding degree terms among any pair of $\{f_A, f_B, f_C, f_D\}$ has been scaled by the same amount. At

Algorithm : Modified split-radix FFT (version D reduction step)

Input: The polynomial $f^{\circ}_{4m,j}$ that has been scaled by $S_{16m,d}$ for all $d < 4m$.
Here, $S_{16m,d} \neq S_{16m,d+m} \neq S_{16m,d+2m} \neq S_{16m,d+3m}$ for all $d \leq m$.

Output: $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.

0A. If $(4m) = 1$, then return $f(\omega^{\sigma'(j)}) = f(\omega^{\sigma'(j)} \cdot x) \bmod (x-1)$.
0B. If $(4m) = 2$, then call a radix-2 algorithm to compute the FFT.
1. Split $f^{\circ}_{4m,j}$ into four blocks f_A, f_B, f_C, and f_D
each of size m such that $f^{\circ}_{4m,j} = f_A \cdot x^{3m} + f_B \cdot x^{2m} + f_C \cdot x^m + f_D$.
1A. Compute $f_{A'}$ by scaling term d of f_A by $(S_{4m,d}/S_{16m,d})$ for all $d < m$.
Compute $f_{B'}$ by scaling term d of f_B by $(S_{4m,d}/S_{16m,d+m})$ for all $d < m$.
Compute $f_{C'}$ by scaling term d of f_C by $(S_{4m,d}/S_{16m,d+2m})$ for all $d < m$.
Compute $f_{D'}$ by scaling term d of f_D by $(S_{4m,d}/S_{16m,d+3m})$ for all $d < m$.
2. Compute $f^{\circ}_{2m,2j} = f_W \cdot x^m + f_X = (f_{A'} + f_{C'}) \cdot x^m + (f_{B'} + f_{D'})$.
3. Compute $f_\alpha = -f_{B'} + f_{D'}$.
4. Compute $f_\beta = \mathtt{I} \cdot (-f_{A'} + f_{C'})$.
5. Compute $f_Y = f_\alpha + f_\beta$.
6. Compute $f_Z = -f_\alpha + f_\beta$.
7. Compute $f^{\circ}_{m,4j+2}$ by multiplying f_Y by $T_{4m,d}$ for all $d < m$.
8. Compute $f^{\circ}_{m,4j+3}$ by multiplying f_Z by $\overline{T_{4m,d}}$ for all $d < m$.
Note: $\overline{T_{4m,d}}$ is the complex conjugate of $T_{4m,d}$.
9. Compute the FFT of $f^{\circ}_{2m,2j}$ using the version C reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+2m-1)})$.
10. Compute the FFT of $f^{\circ}_{m,4j+2}$ using the version B reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+2m)}), f(\omega^{\sigma'(j\cdot 4m+2m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+3m-1)})$.
11. Compute the FFT of $f^{\circ}_{m,4j+3}$ using the version B reduction step to obtain
$f(\omega^{\sigma'(j\cdot 4m+3m)}), f(\omega^{\sigma'(j\cdot 4m+3m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.
12. Return $f(\omega^{\sigma'(j\cdot 4m+0)}), f(\omega^{\sigma'(j\cdot 4m+1)}), \ldots, f(\omega^{\sigma'(j\cdot 4m+4m-1)})$.

Figure C.4 Pseudocode for modified split-radix FFT (version D reduction step)

this point, we have no choice but to scale each of these polynomials before proceeding with the rest of the reduction step. Since some type of scaling is necessary for each of the four subdivisions of the input polynomial, we should select a scaling that puts us in as favorable a position as possible. With the scaling shown in the version D pseudocode given in Figure C.4, we are now essentially in the version B case and copy the rest of that pseudocode to complete the remainder of the reduction step.

Bernstein presents the "Tangent FFT" [4] in terms of two reduction steps rather than four. One of these reduction steps is essentially version A given above. The other is a mixture of version B and version C. Note that the only difference between version B and version D is that version D scales the input polynomial at the beginning of the algorithm. Bernstein instead scales one of the outputs of his reduction step at the end of the reduction for the same effect. Thus, Bernstein's algorithm is essentially equivalent to that of Johnson and Frigo.

We will now compute the cost of an algorithm based on the new reduction steps following the analysis presented by Johnson and Frigo in their paper. Note that the number of additions required by this algorithm is the same as the conjugate pair split-radix FFT algorithm. Rather than explicitly computing the multiplication count, we will instead compute the number of multiplications saved in this algorithm compared to the split-radix algorithm. Let $M_A(n)$, $M_B(n)$, $M_C(n)$, and $M_D(n)$ be the number of multiplications in $\mathbb{R}$ saved by using each of the four versions of the reduction step compared to the split-radix algorithm with input size n. Following the arguments made by Johnson and Frigo, it can be shown that

$$\begin{aligned} M_A(n) &= 0 + M_A\left(\frac{n}{2}\right) + 2 \cdot M_B\left(\frac{n}{4}\right), && \text{(C.1)} \\ M_B(n) &= n - 4 + M_C\left(\frac{n}{2}\right) + 2 \cdot M_B\left(\frac{n}{4}\right), && \text{(C.2)} \\ M_C(n) &= -2 + M_D\left(\frac{n}{2}\right) + 2 \cdot M_B\left(\frac{n}{4}\right), && \text{(C.3)} \\ M_D(n) &= -n - 2 + M_C\left(\frac{n}{2}\right) + 2 \cdot M_B\left(\frac{n}{4}\right). && \text{(C.4)} \end{aligned}$$

The base cases of the recursion are $M_A(1) = M_A(2) = M_B(1) = M_B(2) = M_C(1) = M_C(2) = M_D(1) = 0$ and $M_D(2) = -2$.

Version B is a reduction step that will save us multiplications, versions A and C are reduction steps that are fairly neutral, while version D is a reduction step that will significantly cost us multiplications. As long as version B is called significantly more times than the other versions, an overall savings will be achieved for the new algorithm. As discussed in [44], this system of recurrence relations can be solved through the use of generating functions and the number of multiplications in $\mathbb{R}$ saved by using the new algorithm compared to the split-radix algorithm is given by

$$\frac{2}{9} \cdot n \cdot \log_2(n) - \frac{38}{27} \cdot n + 2 \cdot \log_2(n) + \frac{2}{9} \cdot (-1)^{\log_2(n)} \cdot \log_2(n) - \frac{16}{27} \cdot (-1)^{\log_2(n)} \tag{C.5}$$

which results in a reduced multiplication count of

$$\begin{aligned} M_{\mathbb{R}}(n) &= \frac{6}{9} \cdot n \cdot \log_2(n) - \frac{10}{27} \cdot n + 2 \cdot \log_2(n) \\ &\quad -\frac{2}{9} \cdot (-1)^{\log_2(n)} \cdot \log_2(n) + \frac{22}{27} \cdot (-1)^{\log_2(n)} + 2. \end{aligned} \tag{C.6}$$

Appendix D

Complex conjugate properties

The following theorems give several properties involving complex conjugates. One may assume that R is the complex numbers, but R can be any ring where the element $\varepsilon \in R$ can be expressed as $\varepsilon_r + \mathtt{I} \cdot \varepsilon_\iota$ where $\mathtt{I}^2 = -1$. These properties are used in the new radix-3 FFT and IFFT algorithms presented in Chapters 2 and 4.

Theorem 44 *Let a and b be elements of a ring R where each element ε of R is expressed as $\varepsilon_r + \mathtt{I} \cdot \varepsilon_\iota$. Furthermore, let $\overline{a}$ be the complex conjugate of a and $\overline{b}$ be the complex conjugate of b. Then $\overline{a} \cdot \overline{b} = \overline{a \cdot b}$.*

Proof: Let $a = a_r + \mathtt{I} \cdot a_\iota$ and $b = b_r + \mathtt{I} \cdot b_\iota$ be elements of R. Now $a \cdot b$ can be computed using $(a_r \cdot b_r - a_\iota \cdot b_\iota) + \mathtt{I} \cdot (a_\iota \cdot b_r + a_r \cdot b_\iota)$. Next, let $\overline{a}$ be the complex conjugate of a, i.e. $a = a_r + \mathtt{I} \cdot (-a_\iota)$ and let $\overline{b}$ be the complex conjugate of b, i.e. $b = b_r + \mathtt{I} \cdot (-b_\iota)$. Now, $\overline{a} \cdot \overline{b} = (a_r \cdot b_r + (-a_\iota) \cdot (-b_\iota)) + \mathtt{I} \cdot ((-a_\iota) \cdot b_r + a_r \cdot (-b_\iota)) = (a_r \cdot b_r + a_\iota \cdot b_\iota) - \mathtt{I} \cdot (a_\iota \cdot b_r + a_r \cdot b_\iota) = \overline{ab}$. □

Theorem 45 *Let a, b, c, and d be elements of a ring R where each element ε of R is expressed as $\varepsilon_r + \mathtt{I} \cdot \varepsilon_\iota$. Furthermore, let $\overline{a}$ be the complex conjugate of a and let $\overline{c}$ be the complex conjugate of c. If $a \cdot b + c \cdot d$ has been computed, then $\overline{a} \cdot b + \overline{c} \cdot d$ can be computed at the cost of one addition in R.*

Proof: First compute the expressions $(a_r \cdot b_r + c_r \cdot d_r) + \mathtt{I} \cdot (a_r \cdot b_\iota + c_\iota \cdot d_\iota)$ and $(-a_\iota \cdot b_\iota - c_\iota \cdot d_\iota) + \mathtt{I} \cdot (a_\iota \cdot b_r + c_\iota \cdot d_r)$. The two expressions can be added to form $a \cdot b + c \cdot d$. The total cost of this computation is equivalent to other methods of computing $a \cdot b + c \cdot d$.

Assuming that these two expressions used to form $a \cdot b + c \cdot d$ are saved, subtract $(-a_\iota \cdot b_\iota - c_\iota \cdot d_\iota) + \mathtt{I} \cdot (a_\iota \cdot b_r + c_\iota \cdot d_r)$ from $(a_r \cdot b_r + c_r \cdot d_r) + \mathtt{I} \cdot (a_r \cdot b_\iota + c_\iota \cdot d_\iota)$

at the cost of one addition in R. It can be verified that the resulting expression is equivalent to $\overline{a} \cdot b + \overline{c} \cdot d$. $\square$

Theorem 46 *Let a, b, and c be elements of a ring R where each element ε of R is expressed as $\varepsilon_r + \mathtt{I} \cdot \varepsilon_\iota$. Furthermore, let $\overline{a}$ be the complex conjugate of a. Then $ab + \overline{a}c$ can be computed at the cost of one multiplication and two additions in R.*

Proof: Let $a = a_r + \mathtt{I} \cdot a_\iota$, $b = b_r + \mathtt{I} \cdot b_\iota$, and $c = c_r + \mathtt{I} \cdot c_\iota$ be elements of a ring R. Since $\overline{a} = a_r - \mathtt{I} \cdot a_\iota$, then $ab + \overline{a}c = (a_r + \mathtt{I} \cdot a_\iota) \cdot b + (a_r - \mathtt{I} \cdot a_\iota) \cdot c = a_r \cdot (b + c) + \mathtt{I} \cdot a_\iota \cdot (b - c)$. So compute $b + c$ and $b - c$ at the cost of two additions in R. The two components of $b + c$ are multiplied by a_r, the two components of $b - c$ are multiplied by $\mathtt{I} \cdot a_\iota$ and the results are added together. The cost of this operation is equivalent to one multiplication in R. $\square$

Appendix E

Proof of the existence of the Cantor basis

In this section of the appendix, we will prove that the set of elements $\{\beta_1, \beta_2, \cdots, \beta_k\}$ which satisfy

$$\begin{aligned} \beta_1 &= 1, \\ \beta_i &= (\beta_{i+1})^2 + \beta_{i+1} \quad \text{for } 1 \leq i < k \end{aligned} \tag{E.1}$$

can be constructed and that they form a basis for $\mathbb{F}$, a finite field with 2^k elements where k is a power of two. The following theorem will be used to construct the set of elements. Proof of this theorem can be found in any standard textbook on finite fields ([52], [80]).

Lemma 47 *Let $\mathbb{F} = GF(2^k)$ where k is a power of two. Then for every $a \in \mathbb{F}$ such that the trace of a over $GF(2)$ is equal to zero, there exists an element $b \in \mathbb{F}$ such that $b^2 + b = a$.*

Consider the function $\varphi(x) = x^2 + x$ given in [32]. Here, each coefficient of φ is regarded as an element of $GF(2)$. Cantor defines $\varphi^{m+1}(x) = \varphi^1(\varphi^m(x))$ in [12] where $\varphi^1(x) = \varphi(x)$. By associativity of composition, it is also true that $\varphi^{m+1}(x) = \varphi^1(\varphi^m(x)) = \varphi^m(\varphi^1(x))$ for all $m \geq 1$. Cantor also states the following result for a nonrecursive definition of $\varphi^m(x)$. Here, we also provide a proof of this theorem which is not found in [12].

Theorem 48 *For $m \geq 1$, we have*

$$\varphi^m(x) = \sum_{i=0}^{m} \binom{m}{i} x^{2^i}, \tag{E.2}$$

where the notation $\binom{m}{i}$ *will be used throughout this section to represent the binomial coefficient* $C(m,i)$ *reduced modulo 2.*

Proof: We prove the theorem by induction. Note that

$$\varphi^1(x) \;=\; x^2 + x = \sum_{i=0}^{1} \binom{1}{i} x^{2^i}, \tag{E.3}$$

so the theorem holds for $m = 1$. Assume that the result is true for $m = \kappa$. Then

$$\begin{aligned}
\varphi^{\kappa+1}(x) &= \varphi(\varphi^{\kappa}(x)) \qquad\qquad (E.4)\\
&= \left(\sum_{i=0}^{\kappa} \binom{\kappa}{i} x^{2^i}\right)^2 + \left(\sum_{i=0}^{\kappa} \binom{\kappa}{i} x^{2^i}\right)\\
&= \left(\sum_{i=0}^{\kappa} \binom{\kappa}{i}^2 x^{2^{i+1}}\right) + \left(\sum_{i=0}^{\kappa} \binom{\kappa}{i} x^{2^i}\right)\\
&= \left(\sum_{i=1}^{\kappa+1} \binom{\kappa}{i-1} x^{2^i}\right) + \left(\sum_{i=0}^{\kappa} \binom{\kappa}{i} x^{2^i}\right)\\
&= \binom{\kappa+1}{0} x^{2^0} + \sum_{i=1}^{\kappa} \left(\binom{\kappa}{i} + \binom{\kappa}{i-1}\right) x^{2^i} + \binom{\kappa+1}{\kappa+1} x^{2^{\kappa+1}}\\
&= \binom{\kappa+1}{0} x^{2^0} + \sum_{i=1}^{\kappa} \binom{\kappa+1}{i} x^{2^i} + \binom{\kappa+1}{\kappa+1} x^{2^{\kappa+1}}\\
&= \sum_{i=0}^{\kappa+1} \binom{\kappa+1}{i} x^{2^i}.
\end{aligned}$$

So, the result is true for $m = \kappa + 1$. By induction, the theorem holds for all k. $\square$

The following special case of Lucas' Lemma will be helpful for determining which coefficients of $\varphi^m(x)$ are ones. The following proof is based on the method used in [26].

Lemma 49 *Suppose $i \leq m$ and let d be the smallest integer such that $m \leq 2^d$. Now represent m and i in binary form, i.e. $m = (m_d m_{d-1} \cdots m_1 m_0)$ and $i = (i_d i_{d-1} \cdots i_1 i_0)$ where $m_j, i_j \in \{0, 1\}$. Then*

$$\binom{m}{i} = \binom{m_0}{i_0}\binom{m_1}{i_1}\cdots\binom{m_d}{i_d}.$$

Proof: On the one hand, by the Binomial Theorem, we have

$$(x+1)^m = \sum_{i=0}^{m} \binom{m}{i} x^i. \tag{E.5}$$

On the other hand,

$$\begin{aligned}
(x+1)^m &= (x+1)^{m_0 + m_1 \cdot 2 + \cdots + m_d 2^d} \\
&= \prod_{j=0}^{d} \left(x^{2^j} + 1\right)^{m_j} \\
&= \prod_{j=0}^{d} \left(\sum_{i_j=0}^{m_j} \binom{m_j}{i_j} x^{i_j 2^j} \right) \\
&= \sum_{i=0}^{m} \left(\prod_{j=0}^{d} \binom{m_j}{i_j} \right) x^i.
\end{aligned} \tag{E.6}$$

The lemma follows by comparing the coefficients of (E.5) with the coefficients in the last expression of (E.6). □

Theorem 50 *Let d be the smallest integer such that $m \leq 2^d$ and let $(m_{d-1}m_{d-2}\cdots m_2 m_1 m_0)_2$ be the binary representation of m. If $c = m_d + m_{d-1} + \cdots + m_1 + m_0$, then 2^c of the coefficients of $\varphi^m(x)$ are 1.*

Proof: By Lemma 49, $\binom{m}{i} = 1$ iff $m_j = 1$ or $i_j = 0$ for all j in $0 \leq j \leq d$. Let us count the number of integers in the range $0 \leq i \leq m$ that satisfy these restrictions. For each j such that $m_j = 1$, then there are 2 possibilities for i_j while for each j such that $m_j = 0$, then there is only one possibility for i_j. Multiplying the number of possibilities for each j, we obtain $2^{m_d} \cdot 2^{m_{d-1}} \cdot 2^{m_{d-2}} \cdots 2^{m_1} \cdot 2^{m_0} = 2^{m_d + m_{d-1} + m_{d-2} + \cdots + m_1 + m_0} = 2^c$ terms of $\varphi^m(x)$ that have a coefficient of 1. □

Corollary 51 *If k is a power of two, then the trace function of an element in $GF(2^k)$ over $GF(2)$ given by $Tr(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{k-1}}$, is equivalent to $\varphi^{k-1}(x)$.*

Proof: Since k is a power of two, then $k = 2^d$ for some d. Then

$$\begin{aligned} k - 1 &= 2^d - 1 \\ &= \sum_{i=0}^{d-1} 2^i \end{aligned} \tag{E.7}$$

and the binary expansion of $k-1$ is all ones. Thus, by Theorem 50, all $2^d = k$ of the coefficients of $\varphi^{k-1}(x)$ are 1. In other words,

$$\begin{aligned} \varphi^{k-1}(x) &= \sum_{i=0}^{k-1} x^{2^i} \\ &= \mathrm{Tr}(x). \end{aligned} \tag{E.8}$$

□

Corollary 52 *If k is a power of two, then $\varphi^k(x) = x^{2^k} + x$.*

Proof: Let k be a power of two and let d be the integer that satisfies $k = 2^d$. Since the binary expansion of k is $(1000\ldots0)_2$, then by Theorem 50, $\binom{k}{i}$ is 1 if $i_j = 0$ for all $j < d$ in the binary expansion of i. The only integers that satisfy this restriction in $0 \leq i \leq k$ are 0 and k. Thus, $\varphi^k(x) = x^{2^k} + x$. □

We are now ready to construct the set of elements. Let $\beta_1 = 1$. If $k = 1$, then the basis has been determined. Otherwise, $\mathrm{Tr}(\beta_1) = \varphi^{k-1}(\beta_1)$ has k terms where $k > 1$ is a power of two. Thus, $\mathrm{Tr}(\beta_1) = 0$. By Theorem 47, there exists some β_2 such that $\beta_1 = {\beta_2}^2 + \beta_2 = \varphi(\beta_2)$. For each $2 \leq i < k$, we must first show that β_i has trace 0. Observe that $\mathrm{Tr}(\beta_i) = \varphi^{k-1}(\beta_i) = \varphi^{k-2}(\varphi(\beta_i)) = \varphi^{k-2}(\beta_{i-1}) = \cdots = \varphi^{k-i}(\beta_1) = \varphi^{k-i}(1)$. Let c be the number of nonzero entries in the binary expansion of $k - i$. Since $k - i > 0$, then $c \geq 1$ and 2^c must be even. So there must be an even number of 1's added together to compute the trace of β_i and thus $\mathrm{Tr}(\beta_i) = 0$. So by Theorem 47, there exists some β_{i+1} such that $\beta_i = {\beta_{i+1}}^2 + \beta_{i+1} = \varphi(\beta_{i+1})$. By repeating this procedure, it is possible to solve for each of $\{\beta_1, \beta_2, \ldots, \beta_k\}$.

The above analysis only says that it is possible to solve for each of the elements, but does not give a method for constructing the elements. In [80], the formula

$$\begin{aligned} \beta_{i+1} &= \beta_i \cdot \theta^2 + (\beta_i + {\beta_i}^2) \cdot \theta^{2^2} + \ldots \\ &\quad + (\beta_i + {\beta_i}^2 + \cdots + {\beta_i}^{2^{k-2}}) \cdot \theta^{2^{k-1}} \end{aligned} \tag{E.9}$$

is provided to construct the elements and the author states that the construction is based on Hilbert's Theorem 90. Here θ is any element of trace 1 in $\mathbb{F}$. Since $\mathbb{F}$ has characteristic 2, then half of the elements in $\mathbb{F}$ must have this property.

Now that we have shown that it is possible to construct the elements, we need to show that the elements form a basis for $\mathbb{F}$.

Theorem 53 *The elements* $\{\beta_1, \beta_2, \ldots, \beta_k\}$ *are linearly independent in* $\mathbb{F}$*, i.e. if* $a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_k \cdot \beta_k = 0$ *where* $\{a_1, a_2, \ldots, a_k\} \in GF(2)$*, then* $a_1 = a_2 = \cdots = a_k = 0$.

Proof: We will prove the theorem inductively. If $a_1 \cdot \beta_1 = 0$, then clearly $a_1 = 0$ since $\beta_1 = 1$. Thus, the result is true for $k = 1$.

Assume that $\{\beta_1, \beta_2, \ldots, \beta_\kappa\}$ are linearly independent where $\kappa < k$. We need to show that $\{\beta_1, \beta_2, \ldots, \beta_{\kappa+1}\}$ are linearly independent. Suppose instead that this set of elements is linearly dependent. So if $a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_{\kappa+1} \cdot \beta_{\kappa+1} = 0$ where $\{a_1, a_2, \ldots, a_{\kappa+1}\} \in GF(2)$, then $a_i \neq 0$ for some i. Since $\{\beta_1, \beta_2, \ldots, \beta_\kappa\}$ are linearly independent, then it must be the case that $a_{\kappa+1} = 1$. Then $\beta_{\kappa+1} = a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_\kappa$. By the equations used to construct the elements and the Freshman's Dream Theorem,

$$\begin{aligned}
\beta_\kappa &= (\beta_{\kappa+1})^2 + \beta_{\kappa+1} \qquad \text{(E.10)} \\
&= (a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_\kappa)^2 + (a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_\kappa) \\
&= {a_1}^2 \cdot {\beta_1}^2 + {a_2}^2 \cdot {\beta_2}^2 + \cdots + {a_\kappa}^2 \cdot {\beta_\kappa}^2 + a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_\kappa \\
&= a_1 \cdot {\beta_1}^2 + a_2 \cdot {\beta_2}^2 + \cdots + a_\kappa \cdot {\beta_\kappa}^2 + a_1 \cdot \beta_1 + a_2 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_\kappa \\
&= a_1 \cdot ({\beta_1}^2 + \beta_1) + a_2 \cdot ({\beta_2}^2 + \beta_2) + \cdots + a_\kappa \cdot ({\beta_\kappa}^2 + \beta_\kappa) \\
&= a_1 \cdot (0) + a_2 \cdot \beta_1 + a_3 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_{\kappa-1} \\
&= a_2 \cdot \beta_1 + a_3 \cdot \beta_2 + \cdots + a_\kappa \cdot \beta_{\kappa-1}.
\end{aligned}$$

So β_κ can be expressed as a linear combination of $\{\beta_1, \beta_2, \ldots, \beta_{\kappa-1}\}$, contradicting

the assumption that $\{\beta_1, \beta_2, \ldots, \beta_\kappa\}$ are linearly dependent. It must be the case that $\{\beta_1, \beta_2, \ldots, \beta_{\kappa+1}\}$ are linearly independent.

By induction, $\{\beta_1, \beta_2, \ldots, \beta_k\}$ are linearly independent. □

Theorem 54 *The set of elements $S = \{\beta_1, \beta_2, \ldots, \beta_k\}$ forms a basis for $\mathbb{F} = GF(2^k)$ over $GF(2)$.*

Proof: Since $\mathbb{F}$ is a vector space of dimension k over GF(2) and S is a linearly independent set of k vectors in $\mathbb{F}$, then S spans $\mathbb{F}$. Since S is a linearly independent set of vectors that spans $\mathbb{F}$, then S is a basis for $\mathbb{F}$. □

Thus, we have shown that the collection of elements introduced in [32] can be constructed and forms a basis for $\mathbb{F}$. [1]

[1] In the above iterative construction of the β's, there are two choices at each step. Hence we have a binary tree of height k starting at $\beta_1 = 1$. Each sequence of β's corresponds to a path of the tree. Cantor shows how to get a specific sequence. A topic for further exploration is whether there exists a simpler construction.

Appendix F

Taylor shift of a polynomial with finite field coefficients

Given a polynomial

$$f(x) = f_{2m-1} \cdot x^{2m-1} + f_{2m-2} \cdot x^{2m-2} + \cdots + f_1 \cdot x + f_0 \tag{F.1}$$

of degree less than $2m$ in $R[x]$, the problem of computing the Taylor shift of f at an element $\xi \in R$ is to find coefficients $g_0, g_1, g_2, \ldots, g_{2m-1} \in R$ such that

$$f(x) = g_{2m-1} \cdot (x+\xi)^{2m-1} + g_{2m-2} \cdot (x+\xi)^{2m-2} + \cdots + g_1 \cdot (x+\xi) + g_0. \tag{F.2}$$

In other words,

$$g(y) = f(x-\xi). \tag{F.3}$$

where $y = x - \xi$.

Since f has a derivative of order $n-1$, then the Taylor shift can be computed using techniques learned in a standard Calculus course (e.g. [73]) using $\Theta(n^2)$ operations. In the paper [33], von zur Gathen and Gerhard discuss some alternative techniques which may be faster than the Calculus method if efficient methods for performing polynomial multiplication such as those discussed in Chapter 5 are used

in the algorithms. In [30], Shuhong Gao presented one of these techniques in the case where R is a finite field, a ring where the Calculus-based techniques do not apply anyway. Gao's presentation exploits certain properties of finite fields to eliminate the need for the polynomial multiplications and substantially reduces the overall number of operations.

We will assume that R has characteristic 2 and m is a power of two to simplify the presentation of Gao's algorithm. Let us split f into two polynomials of degree less than m denoted by f_A and f_B such that

$$f = f_A \cdot x^m + f_B. \tag{F.4}$$

Since m is a power of two, then by the Freshman's Dream Theorem,

$$\begin{aligned} x^m &= x^m - \xi^m + \xi^m \\ &= (x - \xi)^m + \xi^m. \end{aligned} \tag{F.5}$$

Then (F.4) becomes

$$\begin{aligned} f &= f_A \cdot ((x + \xi)^m - \xi^m) + f_B \\ &= f_A \cdot (x + \xi)^m + (f_B - f_A \cdot \xi^m). \end{aligned} \tag{F.6}$$

So the problem of computing the Taylor shift of f at ξ has been simplified into the two subproblems of computing the Taylor shift of f_A at ξ and the Taylor shift of

Algorithm : Taylor expansion of a polynomial at ξ
Input: f, a polynomial of degree less than $2m$ in $\mathbb{F}$ where $\mathbb{F}$ is a finite field of characteristic 2. An element ξ in $\mathbb{F}$.
Output: The Taylor expansion of f at ξ, i.e. a polynomial $g(y)$ of degree less than $2m$ such that $g(y) = f(x - \xi)$.
0. If $(2m) = 1$, then return $g = f$ (f is a constant). 1. Split f into two blocks f_A and f_B given by $f = f_A \cdot x^m + f_B$. 2. Compute $f_C = f_B - f_A \cdot \xi^m$. 3. Compute the Taylor expansion of f_C at ξ to obtain $\{g_0, g_1, g_2, \ldots g_{m-1}\}$. 4. Compute the Taylor expansion of f_A at ξ to obtain $\{g_m, g_{m+1}, g_{m+2}, \ldots g_{2m-1}\}$. 5. Return $g(y) = g_{2m-1} \cdot y^{2m-1} + g_{2m-2} \cdot y^{2m-2} + \cdots + g_1 \cdot y + g_0$.

Figure F.1 Pseudocode for Taylor expansion of a polynomial at ξ

$f_C = f_B - f_A \cdot \xi^m$ at ξ. By recursively applying this reduction step, we can compute the Taylor shift of a polynomial f of degree less than $n = 2^k$ at ξ. The pseudocode in Figure F.1 can be used to perform the computation.

Assume that $\{\xi, \xi^2, \xi^4, \ldots, \xi^m\}$ have been precomputed prior to the first call to the algorithm. If these elements are not stored, then a total of $\log_2(m) - 1$ multiplications in $\mathbb{F}$ are required to generate an array of these elements.

Let us now compute the cost of this algorithm. Line 0 ends the recursion and costs no operations. Line 1 just involves logicially splitting f into two blocks and does not require any operations. In line 2, we must first multiply f_A by ξ^m and then add the result to f_B at a cost of m multiplications and m additions. Lines 3 and 4 each involve the computation of a Taylor expansion of a polynomial of degree less than m at a by recursively calling the algorithm. Line 5 requires no operations. The total number of operations to compute this Taylor expansion for a polynomial of degree less than n is

$$M(n) = 2 \cdot M\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n, \tag{F.7}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n \tag{F.8}$$

where $M(1) = 0$ and $A(1) = 0$. These recurrence relations can be solved using Master Equation I to obtain

$$M(n) = \frac{1}{2} \cdot n \cdot \log_2(n), \tag{F.9}$$

$$A(n) = \frac{1}{2} \cdot n \cdot \log_2(n). \tag{F.10}$$

Appendix G

Taylor expansion of a polynomial with finite field coefficients at x^τ

Suppose that we are given $f \in \mathbb{F}[x]$ of degree less than n where $\mathbb{F}$ is a field of characteristic 2, $n = 2^{2k}$ and k is a nonnegative power of two. Let $\tau = \sqrt{n} = 2^k$. In this section, we are going to compute the Taylor expansion of f at x^τ, i.e. $f(x - x^\tau) = f(x^\tau - x)$. In other words, we desire $g_{\tau-1}(x), g_{\tau-2}(x), \ldots, g_1(x), g_0 \in \mathbb{F}[x]$, each a polynomial of degree less than τ in x, such that

$$\begin{aligned} f &= g_{\tau-1} \cdot (x - x^\tau)^{\tau-1} + g_{\tau-2} \cdot (x - x^\tau)^{\tau-2} + \cdots + g_1 \cdot (x - x^\tau) + g_0 \quad \text{(G.1)} \\ &= g_{\tau-1} \cdot (x^\tau - x)^{\tau-1} + g_{\tau-2} \cdot (x^\tau - x)^{\tau-2} + \cdots + g_1 \cdot (x^\tau - x) + g_0. \end{aligned}$$

The polynomial

$$g(y) = g_{\tau-1} \cdot y^{\tau-1} + g_{\tau-2} \cdot y^{\tau-2} + \cdots + g_1 \cdot y + g_0 \quad \text{(G.2)}$$

is called the Taylor expansion of f at x^τ. Here, $y = x^\tau - x$.

First, we will show how to convert the problem of finding the Taylor expansion of f of degree less than $(2m) \geq 2\tau$ into two problems of finding the Taylor expansion of a polynomial of degree less than m where m is a power of two. Let $\delta = m/\tau$ and partition f into three blocks called f_A, f_B and f_C as follows:

$$f = f_A \cdot x^{2m-\delta} + f_B \cdot x^m + f_C. \quad \text{(G.3)}$$

Here f_A consists of at most δ coefficients of f, f_B consists of at most $m-\delta$ coefficients of f, and f_C consists of at most m coefficients of f.

By the Freshman's Dream Theorem,

$$\begin{aligned}
f &= f_A \cdot x^{2m-\delta} + f_B \cdot x^m + f_C \qquad \text{(G.4)} \\
&= \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot x^m + f_C \\
&= \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot x^m - \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot x^\delta + \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot x^\delta + f_C \\
&= \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot (x^m - x^\delta) + f_A \cdot x^m + f_B \cdot x^\delta + f_C \\
&= \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot (x^m - x^\delta) + f_A \cdot x^m - f_A \cdot x^\delta + f_A \cdot x^\delta + f_B \cdot x^\delta + f_C \\
&= \left(f_A \cdot x^{m-\delta} + f_B\right) \cdot (x^m - x^\delta) + f_A \cdot (x^m - x^\delta) + (f_A + f_B) \cdot x^\delta + f_C \\
&= \left(f_A \cdot x^{m-\delta} + (f_A + f_B)\right) \cdot (x^m - x^\delta) + (f_A + f_B) \cdot x^\delta + f_C \\
&= \left(f_A \cdot x^{m-\delta} + (f_A + f_B)\right) \cdot (x^{\tau\delta} - x^\delta) + (f_A + f_B) \cdot x^\delta + f_C \\
&= \left(f_A \cdot x^{m-\delta} + (f_A + f_B)\right) \cdot (x^\tau - x)^\delta + \left((f_A + f_B) \cdot x^\delta + f_C\right),
\end{aligned}$$

so the problem of finding the Taylor expansion of f at x^τ has been reduced to finding the Taylor expansions of $(f_A \cdot x^{m-\delta} + f_A + f_B)$ at x^τ and $(f_A + f_B) \cdot x^\delta + f_C$ at x^τ. Each of these is a polynomial of degree less than m. By recursively applying the reduction step, we can compute the Taylor expansion of f at x^τ given by (G.2). Pseudocode for an algorithm that can compute this Taylor expansion is given in Figure G.1.

Let us now compute the cost of this algorithm. Line 0 ends the recursion and costs no operations. Line 1 just involves logicially splitting f into three blocks and does not require any operations. In line 2, we must first add f_A to f_B at a cost of $\delta = m/\tau$ additions. Then in line 3, $m - \delta$ additions are needed to add $(f_A + f_B)$

Algorithm : Taylor expansion at x^τ
Input: f, a polynomial of degree less than $2m$ in $\mathbb{F}$ where $\mathbb{F}$ is a finite field of characteristic 2.
Output: The Taylor expansion of f at x^τ, i.e. a polynomial $g(y)$ such that $f(x - x^\tau) = g(y)$. Here, $\delta = m/\tau$.
0. If $(2m) \leq \tau$, then return $g = f$ (no expansion is possible). 1. Split f into three blocks f_A, f_B, and f_C given by $f = f_A \cdot x^{2m-\delta} + f_B \cdot x^m + f_C$ where $\delta = m/\tau$. 2. Compute $f_Y = f_A \cdot x^{m-\delta} + (f_A + f_B)$. 3. Compute $f_Z = (f_A + f_B) \cdot x^\delta + f_C$. 4. Compute the Taylor expansion of f_Y at x^τ to obtain $g_{2\delta-1}, g_{2\delta-2}, \ldots, g_{\delta+1}, g_\delta$. 5. Compute the Taylor expansion of f_Z at x^τ to obtain $g_{\delta-1}, g_{\delta-2}, \ldots, g_1, g_0$. 6. Return $g(y) = g_{2\delta-1} \cdot y^{2\delta-1} + g_{2\delta-2} \cdot y^{2\delta-2} + \cdots + g_1 \cdot y + g_0$.

Figure G.1 Pseudocode for Taylor expansion at x^τ

to f_C, starting at the coefficient of degree δ. Now, lines 4 and 5 each involve the computation of a Taylor expansion of a polynomial of degree less than m at x^τ by recursively calling the algorithm. Line 6 requires no operations. The total number of operations to compute this Taylor expansion for a polynomial of degree less than n is

$$M(n) = 0, \tag{G.5}$$

$$A(n) = 2 \cdot A\left(\frac{n}{2}\right) + \frac{1}{2} \cdot n \tag{G.6}$$

where $A(\sqrt{n}) = 0$.

Let $n = 2^k$ and let us solve for $A(2^k)$ using iteration with the initial condition that $A(2^{k/2}) = 0$. So,

$$
\begin{aligned}
A(2^k) &= 2 \cdot A(2^{k-1}) + 2^{k-1} \qquad \text{(G.7)} \\
&= 2 \cdot (2 \cdot A(2^{k-2}) + 2^{k-2}) + 2^{k-1} \\
&= 2^2 \cdot A(2^{k-2}) + 2 \cdot 2^{k-1} \\
&= 2^3 \cdot A(2^{k-3}) + 3 \cdot 2^{k-1} \\
&= \cdots \\
&= 2^{k/2} \cdot A(2^{k/2}) + (k/2) \cdot 2^{k-1} \\
&= 2^{k/2} \cdot 0 + \frac{1}{4} \cdot 2^k \cdot k \\
&= \frac{1}{4} \cdot 2^k \cdot k.
\end{aligned}
$$

Thus,

$$
A(n) = \frac{1}{4} \cdot n \cdot \log_2(n) \qquad \text{(G.8)}
$$

additions are required. An important feature of this Taylor expansion is that no multiplications are needed to compute it.

Appendix H

Additional recurrence relation solutions for additive FFT algorithms

We will first compute the number of copy operations needed in both Gao's algorithm and the new additive FFT algorithm if $n \geq 2^{32}$. Let us model one of these copies as the cost of an addition and let $A_c(n)$ denote the number of these copies required. If $n = 2^{2^I}$, then we will solve for $A_c(2^{2^I})$ using iteration where $A_c(2^{2^4}) = 0$.

$$\begin{aligned}
A_c\left(2^{2^I}\right) &= 2 \cdot 2^{2^I} + 2 \cdot 2^{2^{I-1}} \cdot A\left(2^{2^{I-1}}\right) \qquad \text{(H.1)} \\
&= 2 \cdot 2^{2^I} + 2 \cdot 2^{2^{I-1}} \cdot \left(2 \cdot 2^{2^{I-1}} + 2 \cdot 2^{2^{I-2}} \cdot A\left(2^{2^{I-2}}\right)\right) \\
&= 2 \cdot 2^{2^I} + 2^2 \cdot 2^{2^I} + 2^2 \cdot 2^{2^{I-1}+2^{I-2}} \cdot A\left(2^{2^{I-2}}\right) \\
&= 2 \cdot 2^{2^I} + 2^2 \cdot 2^{2^I} + 2^3 \cdot 2^{2^I} + 2^3 \cdot 2^{2^{I-1}+2^{I-2}+2^{I-3}} \cdot A\left(2^{2^{I-3}}\right) \\
&= \cdots \\
&= (2 + 2^2 + 2^3 + \cdots + 2^{I-4}) \cdot 2^{2^I} + 2^{I-4} \cdot 2^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots+2^4} \cdot A\left(2^{2^4}\right) \\
&= \left(\frac{1}{8} \cdot 2^I - 2\right) \cdot 2^{2^I} + 2^{I-4} \cdot 2^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots+2^4} \cdot (0) \\
&= \frac{1}{8} \cdot 2^{2^I} \cdot 2^I - 2 \cdot 2^{2^I}.
\end{aligned}$$

So the number of copies needed to compute a large FFT of size n is

$$A_c(n) = \frac{1}{8} \cdot n \cdot \log_2(n) - 2 \cdot n. \qquad \text{(H.2)}$$

Now, let us determine the number of multiplications saved in the new algorithm when $j = 0$. A recurrence relation which gives the number of these cases is given by

$M_s(n) = (\sqrt{n}+1) \cdot M_s(\sqrt{n})$ where $M_s(2) = 1$. Let $n = 2^{2^I}$ and let us solve $M_s(2^{2^I})$ using iteration. So,

$$\begin{aligned} M_s(2^{2^I}) &= (2^{2^{I-1}}+1) \cdot M_s(2^{2^{I-1}}) \\ &= (2^{2^{I-1}}+1) \cdot (2^{2^{I-2}}+1) \cdot M_s(2^{2^{I-2}}) \\ &= (2^{2^{I-1}}+1) \cdot (2^{2^{I-2}}+1) \cdot (2^{2^{I-3}}+1) \cdot M_s(2^{2^{I-3}}) \\ &= \cdots \\ &= \prod_{d=0}^{I-1} (2^{2^d}+1) \cdot M_s(2^{2^0}) \\ &= \prod_{d=0}^{I-1} (2^{2^d}+1) \cdot (1) \\ &= \prod_{d=0}^{I-1} (2^{2^d}+1) \\ &= 2^{2^I} - 1. \end{aligned} \tag{H.3}$$

The simplification used in the last step of this derivation can be proven using the following theorem.

Theorem 55 $\prod_{d=0}^{I-1}(2^{2^d}+1) = 2^{2^I} - 1$ *for any* $I > 0$.

Proof: We will prove the result by induction. The result holds for $I = 1$ since $2^1 + 1 = 2 + 1 = 3 = 4 - 1 = 2^{2^2} - 1$. Suppose the result holds for some $\kappa > 0$. We need to show that the result holds for $\kappa + 1$. Now,

$$\begin{aligned} \prod_{d=0}^{\kappa} \left(2^{2^d}+1\right) &= (2^{2^\kappa}+1) \cdot \prod_{d=0}^{\kappa-1} \left(2^{2^d}+1\right) \\ &= (2^{2^\kappa}+1) \cdot (2^{2^\kappa}-1) \\ &= 2^{2^{\kappa+1}} - 1. \end{aligned} \tag{H.4}$$

Therefore, the result holds by mathematical induction. □

The number of additions saved when $j = 0$ uses the same recurrence relation and initial condition. Thus, $A_s(n) = n - 1$ as well.

Appendix I

Operation count: Karatsuba's multiplication algorithm

The total number of operations to compute the product of two polynomials of size n using Karatsuba's algorithm is given by

$$M(2n-1) = 3 \cdot M\left(2 \cdot \frac{n}{2} - 1\right), \tag{I.1}$$

$$A(2n-1) = 3 \cdot A\left(2 \cdot \frac{n}{2} - 1\right) + 4 \cdot n - 4 \tag{I.2}$$

where $M(1) = 1$ and $A(1) = 0$. The algorithm will compute a product of size $2n-1$.

First, let us solve for $M(2n-1)$ using iteration.

$$\begin{aligned} M(2n-1) &= 3 \cdot M\left(2 \cdot \frac{n}{2} - 1\right) \\ &= 3 \cdot \left(3 \cdot M\left(2 \cdot \frac{n}{4} - 1\right)\right) \\ &= 3^2 \cdot M\left(2 \cdot \frac{n}{4} - 1\right) \\ &= 3^3 \cdot M\left(2 \cdot \frac{n}{8} - 1\right) \\ &= \ldots \\ &= 3^{\log_2(n)} \cdot M(1) \\ &= 3^{\log_2(n)} \cdot 1 \\ &= n^{\log_2(3)} \approx n^{1.59}. \end{aligned} \tag{I.3}$$

Next, let us solve for $A(2n-1)$ using iteration.

$$
\begin{aligned}
A(2n-1) &= 3 \cdot A\left(2 \cdot \frac{n}{2} - 1\right) + 4n - 4 \qquad \text{(I.4)} \\
&= 3 \cdot \left(3 \cdot A\left(2 \cdot \frac{n}{4} - 1\right) + 4 \cdot \frac{n}{2} - 4\right) + 4n - 4 \\
&= 3^2 \cdot A\left(2 \cdot \frac{n}{4} - 1\right) + 4n \cdot \left(1 + \frac{3}{2}\right) - 4 \cdot (1+3) \\
&= 3^3 \cdot A\left(2 \cdot \frac{n}{8} - 1\right) + 4n \cdot \left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2\right) - 4 \cdot \left(1 + 3 + 3^2\right) \\
&\quad \cdots \\
&= 3^{\log_2(n)} \cdot A(1) + 4n \cdot \left(1 + \frac{3}{2} + \left(\frac{3}{2}\right)^2 + \cdots \left(\frac{3}{2}\right)^{\log_2(n)-1}\right) \\
&\quad -4 \cdot (1 + 3 + 3^2 + \cdots + 3^{\log_2(n)-1}) \\
&= 0 + 4n \cdot \sum_{i=0}^{\log_2(n)-1} \left(\frac{3}{2}\right) - 4 \cdot \sum_{i=0}^{\log_2(n)-1} (3) \\
&= 4n \cdot \frac{\left(\frac{3}{2}\right)^{\log_2(n)} - 1}{\frac{3}{2} - 1} - 4 \cdot \frac{3^{\log_2(n)} - 1}{3 - 1} \\
&= 8n \cdot \left(\frac{3^{\log_2(n)}}{n} - 1\right) - 2 \cdot (3^{\log_2(n)} - 1) \\
&= 8 \cdot 3^{\log_2(n)} - 8n - 2 \cdot 3^{\log_2(n)} + 2 \\
&= 6 \cdot 3^{\log_2(n)} - 8n + 2 \\
&= 6 \cdot n^{\log_2(3)} - 8n + 2 \approx 6 \cdot n^{1.585} - 8n + 2.
\end{aligned}
$$

Appendix J

Operation count: Schönhage's algorithm

Let us now compute the cost of multiplying two polynomials using Schönhage's algorithm. These two polynomials can be of any size, provided that the sum of the sizes is no greater than $2n$ where $n = 3^{2^I}$. In this case, we will compute lower bounds of the operation counts to support the claim that FFT-based multiplication using the new additive FFT algorithm requires less effort than Schönhage's algorithm.

We will first find $M(2n)$ using iteration and the initial condition that $M(2 \cdot 3^{2^1}) = 102$.

$$
\begin{aligned}
M(2 \cdot 3^{2^I}) &= 2 \cdot 3^{2^{I-1}} \cdot M(2 \cdot 3^{2^{I-1}}) \qquad \text{(J.1)} \\
&= 2 \cdot 3^{2^{I-1}} \cdot 2 \cdot 3^{2^{I-2}} \cdot M(2 \cdot 3^{2^{I-2}}) \\
&= 2^2 \cdot 3^{2^{I-1}+2^{I-2}} \cdot M(2 \cdot 3^{2^{I-2}}) \\
&= 2^3 \cdot 3^{2^{I-1}+2^{I-2}+2^{I-3}} \cdot M(2 \cdot 3^{2^{I-3}}) \\
&= \cdots \\
&= 2^{I-1} \cdot 3^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots 2^1} \cdot M(2 \cdot 3^{2^1}) \\
&= 2^{I-1} \cdot 3^{2^I-2^1} \cdot (102) \\
&= \frac{102}{2 \cdot 9} \cdot 3^{2^I} \cdot 2^I \\
&= \frac{17}{3} \cdot 3^{2^I} \cdot 2^I.
\end{aligned}
$$

Now, let us solve $A(2 \cdot 3^{2^I})$ using iteration and the initial condition that $A(2 \cdot 3^{2^1}) = 519$.

$$
\begin{aligned}
A(2\cdot 3^{2^I}) &= 2\cdot 3^{2^{I-1}}\cdot A\left(2\cdot 3^{2^{I-1}}\right)+\frac{33}{2}\cdot 3^{2^I}\cdot 2^I+\frac{3}{2}\cdot 3^{2^I}+\frac{11}{2}\cdot 3^{2^{I-1}} \qquad \text{(J.2)}\\
&= 2\cdot 3^{2^{I-1}}\cdot\left(2\cdot 3^{2^{I-2}}\cdot A\left(2\cdot 3^{2^{I-2}}\right)+\frac{33}{2}\cdot 3^{2^{I-1}}\cdot 2^{I-1}\right.\\
&\quad \left.+\frac{3}{2}\cdot 3^{2^{I-1}}+\frac{11}{2}\cdot 3^{2^{I-2}}\right)+\frac{33}{2}\cdot 3^{2^I}\cdot 2^I+\frac{3}{2}\cdot 3^{2^I}+\frac{11}{2}\cdot 3^{2^{I-1}}\\
&= 2^2\cdot 3^{2^{I-1}+2^{I-2}}\cdot A\left(2\cdot 3^{2^{I-2}}\right)+2\cdot\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\\
&\quad +\frac{3}{2}\cdot 3^{2^I}\cdot(1+2)+\frac{11}{2}\cdot\left(3^{2^{I-1}}+2\cdot 3^{2^{I-1}+2^{I-2}}\right)\\
&= 2^3\cdot 3^{2^{I-1}+2^{I-2}+2^{I-3}}\cdot A\left(2\cdot 3^{2^{I-3}}\right)+3\cdot\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\\
&\quad +\frac{3}{2}\cdot 3^{2^I}\cdot(1+2+4)\\
&\quad +\frac{11}{2}\cdot\left(3^{2^{I-1}}+2\cdot 3^{2^{I-1}+2^{I-2}}+4\cdot 3^{2^{I-1}+2^{I-2}+2^{I-3}}\right)\\
&= \cdots\\
&= 2^{I-1}\cdot 3^{2^{I-1}+2^{I-2}+2^{I-3}+\cdots+2^1}\cdot A\left(2\cdot 3^{2^1}\right)+\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\cdot(I-1)\\
&\quad +\frac{3}{2}\cdot 3^{2^I}\cdot\left(1+2+4+\cdots+2^{I-2}\right)\\
&\quad +\frac{11}{2}\cdot\sum_{d=0}^{I-2}\left(2^d\cdot 3^{2^{I-1}+2^{I-2}+\cdots 2^{I-1-d}}\right)\\
&= 2^{I-1}\cdot 3^{2^I-2^1}\cdot 519+\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\cdot I-\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\\
&\quad +\frac{3}{2}\cdot 3^{2^I}\cdot(2^{I-1}-1)+\frac{11}{2}\cdot\sum_{d=0}^{I-2}\left(2^d\cdot 3^{2^I-2^{I-1-d}}\right)\\
&= \frac{519}{18}\cdot 2^I\cdot 3^{2^I}+\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\cdot I-\frac{33}{2}\cdot 3^{2^I}\cdot 2^I\\
&\quad +\frac{3}{4}\cdot 2^I\cdot 3^{2^I}-\frac{3}{2}\cdot 3^{2^I}+\frac{11}{2}\cdot\sum_{d=0}^{I-2}\left(2^d\cdot\frac{3^{2^I}}{3^{2^{I-1-d}}}\right)\\
&= \frac{33}{2}\cdot 3^{2^I}\cdot 2^I\cdot I+\frac{157}{12}\cdot 3^{2^I}\cdot 2^I-\frac{3}{2}\cdot 3^{2^I}+\frac{11}{2}\cdot\sum_{d=0}^{I-2}\left(2^d\cdot\frac{3^{2^I}}{3^{2^{I-1-d}}}\right)\\
&\geq \frac{33}{2}\cdot 3^{2^I}\cdot 2^I\cdot I+\frac{157}{12}\cdot 3^{2^I}\cdot 2^I-\frac{3}{2}\cdot 3^{2^I}+\frac{11}{2}\cdot 3^{2^{I-1}}.
\end{aligned}
$$

Appendix K

Karatsuba's algorithm in FFT-based multiplication using the new additive FFT

One may have noticed that Schönhage's algorithm uses Karatsuba multiplication to handle the polynomial products of small degree and may wonder if it is possible to include this technique with FFT-based multiplication using the new additive FFT algorithm. It turns out that this is indeed possible and there are two options for implementing this idea.

The first option is to use Karatsuba's algorithm to directly multiply polynomial reductions of degree less than four. The input for each multiplication in this process will be $f \bmod (x^4 - x - \varpi)$ and $g \bmod (x^4 - x - \varpi)$ for some $\varpi \in W_{k-2}$. The cost of using Karatsuba's algorithm to multiply two polynomials of degree less than four is 9 multiplications and 24 additions. The resulting polynomial of degree less than seven must be reduced modulo $x^4 - x - \varpi$ to obtain $h \bmod (x^4 - x - \varpi)$ where $h = f \cdot g$. The reduction costs 6 additions if $\varpi \neq 0$ and 3 additions if $\varpi = 0$. So a total of 9 multiplications and 24-30 additions are required using this technique. Alternatively, one could compute 2 FFTs of size 4, 4 pointwise products, and one inverse FFT of size 4 at a cost of 7 multiplications and 21 additions if $\varpi = 0$ and 16 multiplications and 30 additions if $\varpi \neq 0$. So if $\varpi = 0$, then it is better to stick with the original method. If $\varpi \neq 0$, then using Karatsuba's algorithm saves 7 multiplications. In the multiplication of two polynomials with product degree less than n, one can apply Karatsuba's algorithm $n/4 - 1$ times in the computation and obtain a total savings of $(n/4 - 1) \cdot 7 = 7/4 \cdot n - 7$ multiplications in $\mathbb{F}$.

The second option is to use Karatsuba's algorithm to directly multiply the results of degree less than 16. In this case, we are interested in multiplying $f \bmod (x^{16} - x - \varpi)$ and $g \bmod (x^{16} - x - \varpi)$ into $h \bmod (x^{16} - x - \varpi)$ for each $\varpi \in W_{k-4}$. If $\varpi = 0$, using Karatsuba's algorithm again requires more operations than

the original method. If $\varpi \neq 0$, then using Karatsuba's algorithm with modular reduction requires 81 multiplications and 392 additions while the original method requires 112 multiplications and 288 additions. For the moment, let us ignore the fact that more additions are required if Karatsuba's algorithm is used. Observe that 31 multiplications are saved in each polynomial product and a total of $n/16$ products involving Karatsuba's algorithm are required to determine $h = f \cdot g$. Since one of these products is the case where $\varpi = 0$, then a total of $(n/16-1)\cdot 31 = 31/16 \cdot n - 31$ multiplications in $\mathbb{F}$ are saved in the computation of h.

The decision of which of these two options to select will be determined by the implementation of multiplication in $\mathbb{F}$. Observe that the second option offers a greater savings in terms of multiplications in $\mathbb{F}$, but requires more additions in $\mathbb{F}$. It turns out that if a multiplication in $\mathbb{F}$ costs less than 35 times the cost of an addition, then the first option should be selected. For practical sizes of $\mathbb{F}$ in the early 21st century, the first option will always be used. [2] It is conjectured that as computers increase in capacity to allow computations over 2^{64} or 2^{128}, then it will also be possible to store larger lookup tables in the computer. If this conjecture turns out to be true, then the first option will always be more attractive. Otherwise, the second option should be more carefully considered. One can verify that application of Karatsuba's algorithm to any other size is less attractive than the two cases considered above.

[2] If $\mathbb{F}$ has size 2^{16} or less, von zur Gathen and Gerhard [32] give a method of implementing multiplication in $\mathbb{F}$ at a cost of about 3 times the cost of addition. If $\mathbb{F}$ has size 2^{32}, von zur Gathen and Gerhard give a second method of implementing multiplication in $\mathbb{F}$ at a cost of about 16 times the cost of addition.

Appendix L

Reischert's multiplication method

Suppose that we wish to multiply two polynomials over a finite field with coefficients in $GF(n)$, but the degree of the product polynomial is greater than n. In this case, FFT-based multiplication cannot be used directly because there are not enough points in $GF(n)$ to interpolate into the desired product polynomial. One may be forced to using Schönhage's algorithm to compute the desired product.

However, if the product degree is less than $\mu \cdot n$ for some "small" value of μ (say $\mu \leq 32$), we can adapt a technique first introduced by Reischert [64] to perform the multiplication. [3] Another description of the method is given in [32]. Here, we will allow R to be any ring that supports some type of FFT, but it will mainly be used for the case where R is a finite field of characteristic 2.

Suppose that we wish to compute a product of $f, g \in R[x]$ with degree less than $\mu \cdot n$ where μ is some "small" integer. In the paper [2], Bernstein discusses the technique he calls "clumping" whereby f and g can be mapped to the isomorphic ring $R[x][y]/(x^\mu - y)$ which can be represented by polynomials in $R[x][y]$. The mapping can be achieved by factoring out the largest power of x^μ out of each term of each polynomial in $R[x]$ and replacing $(x^\mu)^d$ with y^d. All terms with a common value of y^d should then be grouped together. The result will be polynomials f' and g' with y-degree less than n where each "coefficient" of y is a polynomial in x of degree less than μ.

[3] This technique was originally invented for computing products of size n where n is not a power of 2. However, the truncated FFT discussed in Chapter 6 works much better for this case.

Since f' and g' have y-degree less than n, we can view f' and g' as polynomials modulo $\mathcal{M}(y)$ where $\mathcal{M}$ is a degree n polynomial such that its roots $\{\varepsilon_0, \varepsilon_1, \cdots, \varepsilon_n\}$ have some special structure which can be exploited by an FFT algorithm.

Reischert's multiplication method first computes the FFT of f' and g' at each of the roots of $\mathcal{M}(y)$. This is equivalent to computing μ FFTs of polynomials of degree less than n in $R[x]$. The FFT of f' can be expressed as $f' \bmod (y - \varepsilon) = f'(y = \varepsilon)$ for each ε that is a root of $\mathcal{M}$. The FFT of g' can be expressed similarly. Each "evaluation" of f' and g' will be a polynomial in x of degree less than μ. Substituting x^μ for y, then each evaluation can also be viewed as a polynomial modulo $x^\mu - \varepsilon$.

The second step of Reischert's multiplication method is to pointwise multiply the evaluations of f' and g' for all values of y that are roots of $\mathcal{M}$. Because μ is assumed to be 32 or less, the fastest method to compute these pointwise products is to use Karatsuba's multiplication and then reduce the results modulo $x^\mu - \varepsilon$. It will usually be the case that each $x^\mu - \varepsilon$ cannot be factored in μ distinct factors in R, so FFT-multiplication could not be applied to these multiplications anyway. If $h' = f' \cdot g'$, then each pointwise product also represents $h' \bmod (y - \varepsilon)$ where ε is a root of $\mathcal{M}$.

The third step of Reischert's multiplication method interpolates the n pointwise evaluations $h' \bmod (y - \varepsilon)$ into $h' \bmod \mathcal{M}(y)$ using an inverse FFT algorithm. Since h' will have y-degree less than n, then this result is equivalent to h'.

To recover h, we need to "unchunk" h'. This is accomplished by substituting x^μ in place of every y in h'. The resulting polynomial in $R[x]$ is the desired product of f and g. Pseudocode for Reischert's multiplication method is given in Figure L.1.

Let us compute the cost of this algorithm. Assume that lines 1, 2, 7, and 8 do not require any operations as most of these instructions are simply a relabeling of the algorithm inputs and outputs. Lines 3 and 4 each require $\mu \cdot M_F(n)$ multiplications

Algorithm : Reischert multiplication

Input: f and g, polynomials in a ring R such that $\deg(f) + \deg(g) < \mu \cdot n$.
A polynomial $\mathcal{M}$ of degree n such that its n roots $\{\varepsilon_0, \varepsilon_1, \cdots, \varepsilon_n\}$
has some special structure that can be exploited by an FFT algorithm.

Output: The product polynomial $h = f \cdot g$.

1. Transform f to $f' \in R[x][y]$ by replacing x^μ with y.
2. Transform g to $g' \in R[x][y]$ by replacing x^μ with y.
3. Call FFT to evaluate $f'(y = \varepsilon_d) = f' \bmod (x^\mu - \varepsilon_d)$ for $0 \leq d < n$.
4. Call FFT to evaluate $g'(y = \varepsilon_d) = g' \bmod (x^\mu - \varepsilon_d)$ for $0 \leq d < n$.
5. For all $0 \leq d < n$, compute
   $h' \bmod (x^\mu - \varepsilon_d) = (f' \bmod (x^\mu - \varepsilon_d) \cdot g' \bmod (x^\mu - \varepsilon_d)) \bmod (x^\mu - \varepsilon_d)$
   using Karatsuba's algorithm and reducing modulo $x^\mu - \varepsilon_d$.
6. Call IFFT to interpolate $h' \bmod (y - \varepsilon_d)$ for $0 \leq d < n$
   into $h' \bmod \mathcal{M}$. This result is equivalant to h'.
7. Transform h' into $h \in R[x]$ by replacing y with x^μ.
8. Return h.

Figure L.1 Pseudocode for Reischert multiplication

and $\mu \cdot A_F(n)$ additions where $M_F(n)$ is the number of multiplications in R needed to compute an FFT of size n and $A_F(n)$ is the number of additions required. Similarly, line 6 requires $\mu \cdot M_F(n)$ multiplications and $\mu \cdot A_F(n)$ additions.[4] Note that each of these computations can also be viewed as μ FFTs or IFFTs of the original polynomials f and g at coefficients located μ positions apart.

The operation count for line 5 of the algrorithm is determined by multiplying n times the sum of (1) the number of operations needed to use Karatsuba's algorithm to multiply two polynomials of degree less than μ into a polynomial of degree less than $2\mu - 1$ and (2) the number of operations needed to reduce this polynomial modulo $x^\mu - \varepsilon$ for some ε. The paper by [82] gives operation counts for Karatsuba's algorithm

[4] For the purposes of the operation count, we will assume that R supports an additive FFT. An additional n multiplications are required if a multiplicative FFT algorithm is used.

Table L.1 Operation counts of "pointwise products" involved in Reischert's multiplication method

μ	$M_K(\mu)$	$A_K(\mu)$	μ	$M_K(\mu)$	$A_K(\mu)$	μ	$M_K(\mu)$	$A_K(\mu)$
1	1	0	12	65	232	23	184	777
2	4	5	13	96	361	24	185	778
3	8	15	14	97	362	25	240	1063
4	12	27	15	95	374	26	241	1064
5	19	50	16	95	375	27	242	1065
6	23	64	17	124	501	28	279	1182
7	33	105	18	125	502	29	271	1232
8	34	106	19	153	616	30	272	1233
9	44	147	20	154	617	31	273	1234
10	54	183	21	182	775	32	274	1235
11	64	231	22	183	776			

for arbitrary N up to 128. A total of $\mu - 1$ multiplications and $\mu - 1$ additions are needed in R to implement each modular reduction.[5] Table L.1 gives the total number of multiplications $M_K(\mu)$ and additions $A_K(\mu)$ for each pointwise product of size μ in the range $1 \leq \mu \leq 32$.

So the total number of operations needed to implement Reischert's polynomial multiplication technique is given by

$$M(\mu \cdot n) = 3 \cdot \mu \cdot M_F(n) + n \cdot M_K(\mu), \tag{L.1}$$

$$A(\mu \cdot n) = 3 \cdot \mu \cdot A_F(n) + n \cdot A_K(\mu). \tag{L.2}$$

[5] However, if $\varepsilon = 0$, then no multiplications and no additions are needed. If $\varepsilon = 1$ or $\varepsilon = -1$, then no multiplications are needed.

Appendix M

Two positions on future polynomial multiplication algorithm performance

A subject for debate is whether Schönhage's algorithm or FFT-based multiplication using the new additive FFT will perform better when we wish to multiply polynomials with coefficients in $GF(2)$. One view counts the number of bit operations, applying Schönhage's algorithm directly to the input polynomials. The view can also be modified to use the technique described in [32] to map the input polynomials to $\mathbb{F}[x]$ where $\mathbb{F}$ is an extension field of characteristic 2. In either case, this view argues that scalar multiplication in $\mathbb{F}$ is then a function of the input polynomial size and that Schönhage's algorithm is superior to FFT-based multiplication for very large input sizes.

A second view argues that scalar multiplication in $\mathbb{F}$ can be implemented in a constant number of operations, determined by the physical constraints of a computer.[6] As mentioned in a previous section of the appendix, von zur Gathen and Gerhard [32] give a method of implementing multiplication in $\mathbb{F}$ at a cost of about 3 times the cost of an addition in $\mathbb{F}$ if the size of the finite field is 2^{16} or less. If $\mathbb{F}$ is of size 2^{32}, then [32] gives a second method of implementing multiplication in $\mathbb{F}$ at a cost of about 16 times the cost of addition. Using the argument given in [32], this technique is sufficient for computing the product of two polynomials with product degree less than 2^{36}. The storage of just the input and output polynomials would require 2^{34} bytes of memory. In all of these cases, the FFT-based multiplication is expected to require fewer operations than Schönhage's algorithm based on the formulas derived in Chapter 5.

---

[6] Let us assume that our current computing environment consists of a typical 64-bit machine capable of storing several arrays of size $2 \cdot 2^{16}$ bytes in memory. No further assumptions about current computing capabilities will be made throughout this section and will likely change over the coming decades.

The debate of which algorithm is better begins when we wish to compute products of degree greater than 2^{36}. Reischert's Method discussed in the previous section of the appendix can be used to combine Karatsuba's algorithm with FFT-based multiplication and will likely outperform Schönhage's algorithm for product polynomial degrees of size 2^{40} or lower. At this point, a computer's memory should be large enough to store several tables of size $4 \cdot 2^{32}$. So a multiplication in $GF(2^{32})$ can now be implemented in about three times the cost of an addition and multiplication in $GF(2^{64})$ can be implemented in about 16 times the cost of an addition. In this case, FFT-based multiplication will definitely outperform Schönhage's method. Similarly, once a computer's memory becomes large enough to store several tables of size $8 \cdot 2^{64}$, then multiplication in $GF(2^{128})$ can be implemented in constant time. This requires a 128-bit computer to store this lookup table. As the available memory increases further on a computer, it is expected that extension fields of even greater size will be able to be computed in constant time.

In any event, the present author feels that the techniques discussed in this manuscript are sufficient to outperform Schönhage's algorithm up to the memory constraints of a computer. The second view argues that as the input sizes which can be stored in a computer increase in size, the sizes of the lookup tables which can be stored in a computer will also increase. In this case, multiplication in $\mathbb{F}$ should always be able to be completed in a constant number of operations and the new FFT-based multiplication should outperform Schönhage's algorithm for all input sizes. Of course, future changes in computer architecture may introduce new factors that will dominate any arguments made here.

Appendix N

Complexity of truncated algorithms

Let n be an integer with binary representation $n = (b_{K-1}b_{K-2}\ldots b_1b_0)_2$ and let $N = 2^K$ be the smallest power of 2 such that $n \leq N$. Observe that

$$
\begin{aligned}
F(n) &= \sum_{i=1}^{K-1} \left(\mathcal{A}\cdot 2^i + b_i \cdot (\mathcal{B}\cdot 2^i + \mathcal{C}\cdot 2^i \cdot i)\right) \qquad \text{(N.1)}\\
&= \mathcal{A}\cdot \sum_{i=1}^{K-1} 2^i + \sum_{i=1}^{K-1} b_i \cdot \mathcal{B}\cdot 2^i + \sum_{i=1}^{K-1} b_i \cdot \mathcal{C}\cdot 2^i \cdot i\\
&< \mathcal{A}\cdot \sum_{i=1}^{K-1} 2^i + \mathcal{B}\cdot \sum_{i=1}^{K-1} b_i \cdot 2^i + \mathcal{C}\cdot \sum_{i=1}^{K-1} b_i \cdot 2^i \cdot (K-1)\\
&< \mathcal{A}\cdot N + \mathcal{B}\cdot n + \mathcal{C}\cdot (K-1)\cdot n\\
&< \mathcal{A}\cdot 2n + \mathcal{B}\cdot n + \mathcal{C}\cdot \log_2(n)\cdot n\\
&< (2\mathcal{A}+\mathcal{B})\cdot n + \mathcal{C}\cdot n\cdot \log_2(n)\\
&< (2\mathcal{A}+\mathcal{B}+\mathcal{C})\cdot n\cdot \log_2(n).
\end{aligned}
$$

Substituting $\mathcal{A} = 0$, $\mathcal{B} = 1$, and $\mathcal{C} = 1$, we get an upper bound of $2\cdot n\cdot \log_2(n)$ for the multiplication count of the truncated FFT based on the multiplicative FFT. Similarly, substituting $\mathcal{A} = 1$, $\mathcal{B} = 1$, and $\mathcal{C} = 1/2$, we get an upper bound of $3.5\cdot n\cdot \log_2(n)$ for the addition count. A lower bound for each case can be obtained by observing that the total cost is greater than the cost of computing an FFT of size $1/2\cdot N$ and that $1/2\cdot n \leq 1/2\cdot N$. Thus, this algorithm is $\Theta(n\cdot \log_2(n))$.

The technique can be repeated with the additive FFT to yield a complexity of $\Theta(n\cdot \log_2(n))$ multiplications and $\Theta(n\cdot (\log_2(n))^{1.585})$ additions. The overall complexity of the algorithm is $\Theta(n\cdot (\log_2(n))^{1.585})$ in this case.

Since the inverse algorithms require the same number of operations as the forward algorithms plus $d \cdot n$ operations for some d, then it can be easily shown that the complexity of the inverse truncated FFT algorithm is the same as the results given above for the truncated FFT algorithm in each case.

Appendix O

Alternative derivation of Newton's Method

Theorem 56 *Suppose that $g_{(i-1)}$ is a polynomial of degree less than $x^{2^{i-1}}$ that satisfies $f \cdot g_{(i-1)} \bmod x^{2^{i-1}} = 1$ where f is some polynomial in $R[x]$ and $i \geq 1$. Then*

$$g_{(i)} = 2 \cdot g_{(i-1)} - f \cdot (g_{(i-1)})^2 \bmod x^{2^i} \tag{O.1}$$

is a polynomial of degree less than 2^i that satisfies $f \cdot g_{(i)} \bmod x^{2^i} = 1$.

Proof: Let $m = 2^{i-1}$ so that $2m = 2^i$. Now write $f = f_A \cdot x^{2m} + f_B \cdot x^m + f_C$ where f_A is a polynomial of degree $\deg(f) - 2m$, and f_B and f_C are polynomials of degree less than m. Furthermore, write

$$g_{(i)} = g_A \cdot x^m + g_B \tag{O.2}$$

where g_A and g_B are unknown polynomials of degree less than m such that $g_{(i)}$ satisfies the condition $f \cdot g_{(i)} \bmod x^{2m} = 1$. If g_A and g_B are of degree less than m, then $g_{(i)}$ will naturally satisfy the second condition that $g_{(i)}$ is of degree less than $2m$. If $f \cdot g_{(i)} \bmod x^{2m} = 1$, then

$$((f_B \cdot g_B + f_C \cdot g_A) \cdot x^m + f_C \cdot g_B) \bmod x^{2m} = 1, \tag{O.3}$$

$$(f_C \cdot g_B) \bmod x^m = 1. \tag{O.4}$$

By the hypothesis, $f \cdot g_{(i-1)} \bmod x^m = 1$ which implies that $f_C \cdot g_{(i-1)} \bmod x^m = 1$. Let q_A and q_B be the unique polynomials that satisfy

$$f_C \cdot g_{(i-1)} = q_A \cdot x^m + 1, \tag{O.5}$$

$$f_C \cdot g_B = q_B \cdot x^m + 1. \tag{O.6}$$

By substituting (O.5) into (O.6), we obtain

$$(q_A - q_B) \cdot x^m + f_C \cdot (g_B - g_{(i-1)}) = 0. \tag{O.7}$$

Since f_C, g_B and $g_{(i-1)}$ are polynomials of degree less than m, it must be the case that $g_B = g_{(i-1)}$.

Now, express $f_C \cdot g_B = f_C \cdot g_{(i-1)}$ as

$$f_C \cdot g_{(i-1)} = \mathcal{A} \cdot x^m + 1, \tag{O.8}$$

where $\mathcal{A}$ is some polynomial of degree less than m.

By (O.3),

$$\left(f_B \cdot g_{(i-1)} + f_C \cdot g_A + \mathcal{A}\right) \bmod x^m = 0, \tag{O.9}$$

or

$$\left(f_B \cdot g_{(i-1)} + f_C \cdot g_A + \mathcal{A}\right) \;=\; q_C \cdot x^m \tag{O.10}$$

for some $q_C \in R[x]$. Multiplying (O.10) by x^m and substituting (O.8) into the resulting equation for $\mathcal{A} \cdot x^m$, we obtain

$$(f_B \cdot g_{(i-1)} + f_C \cdot g_A) \cdot x^m + f_C \cdot g_{(i-1)} - 1 \;=\; q_C \cdot x^{2m}. \tag{O.11}$$

Solving this equation for $f_C \cdot g_A \cdot x^m$ and multiplying the result by $g_{(i-1)}$, we obtain

$$f_C \cdot g_{(i-1)} \cdot g_A \cdot x^m \;=\; -(f_B \cdot x^m + f_C) \cdot {g_{(i-1)}}^2 + g_{(i-1)} + q_C \cdot g_{(i-1)} \cdot x^{2m}. \tag{O.12}$$

The expressions in this equation are valid because R is a commutative ring. We can truncate the results of (O.12) to the terms of degree less than $2m$ and use the fact that $f_C \cdot g_{(i-1)} \bmod x^m = 1$ to obtain

$$\begin{aligned} g_A \cdot x^m &= -(f_B \cdot x^m + f_C) \cdot {g_{(i-1)}}^2 + g_{(i-1)} \bmod x^{2m} \\ &= -(f_A \cdot x^{2m} + f_B \cdot x^m + f_C) \cdot {g_{(i-1)}}^2 + g_{(i-1)} \bmod x^{2m} \\ &= -f \cdot {g_{(i-1)}}^2 + g_{(i-1)} \bmod x^{2m}. \end{aligned} \tag{O.13}$$

Substituting this result and $g_B = g_{(i-1)}$ into (O.2), we obtain

$$
\begin{aligned}
g_{(i)} &= 2 \cdot g_{(i-1)} - f \cdot {g_{(i-1)}}^2 \bmod x^{2m} \\
&= 2 \cdot g_{(i-1)} - f \cdot {g_{(i-1)}}^2 \bmod x^{2^i}
\end{aligned}
\tag{O.14}
$$

and the theorem is proven. □

Clearly, the sequence of polynomials $\{g_{(0)}, g_{(1)}, g_{(2)}, \cdots, g_{(k)}\}$ has the property that $g_{(i)}$ has twice as many coefficients of $g(x)$ computed compared to $g_{(i-1)}$. This is quadratic convergence property of Newton inversion mentioned earlier and was demonstrated without the use of derivatives.

BIBLIOGRAPHY

1. Bergland, Glenn D. A Radix-Eight Fast Fourier Transform Subroutine for Real-Valued Series. *IEEE Transactions on audio and electroacoustics*, 17(2):138-44, 1969.

2. Bernstein, D. Multidigit Multiplication for mathematicians. Preprint. Available at: <http://cr.yp.to/papers.html\#m3>.

3. Bernstein, D. Fast Multiplication and its applications. Preprint. Available at: <http://cr.yp.to/papers.html\#multapps>.

4. Bernstein, D. The Tangent FFT. Preprint. Available at: <http://cr.yp.to/papers.html\#tangentfft>.

5. Bittinger, Marvin L. *Intermediate Algebra, 9th Edition,* Pearson Education (2003).

6. Bouguezel, Saad, M. Omair Ahmad, and M.N.S. Swamy. An Improved Radix-16 FFT Algorithm, *Canadian Conference on Electrical and Computer Engineering*, 2: 1089-92, 2004.

7. Bouguezel, Saad, M. Omair Ahmad, and M.N.S. Swamy. Arithmetic Complexity of the Split-Radix FFT Algorithms, *International Conference on Acoustics, Speech, and Signal Processing*, 5: 137-40, 2005.

8. Brent, Richard P., Fred G. Gustavson, and David Y. Y. Yun. Fast Solution of Toeplitz Systems of Equations and Computation of Pade Approximants. *Journal of Algorithms*, 1: 259-295, 1980.

9. Brigham, E. Oran. *The Fast Fourier Transform and its Applications*, Prentice Hall (1988).

10. Buneman, Oscar. Inversion of the Helmholtz (or Laplace-Poisson) Operator for Slab Geometry, *Journal of Computational Physics*, 12: 124-30, 1973.

11. Burden, Richard L. and J. Douglas Faires. *Numerical Analysis, Fifth Edition*, PWS Publishing Company (1993).

12. Cantor, David G. On arithmetical algorithms over finite fields, *J. Combinatorial Theory, Series A*, 50(2): 285-300, 1989.

13. Cantor, David G. and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras, *Acta Informatica*, 28: 693-701, 1991.

14. Chu, Eleanor and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, CRC Press (2000).

15. Conway, John. *The Book of Numbers*, Springer (1996).

16. Cooley, J. and J. Tukey. An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation*, 19: 297-301, 1965.

17. Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, MIT Press (1997).

18. Crandall, Richard and Barry Fagin. Discrete weighted transforms and large-integer arithmetic, *Mathematics of Computation*, 62: 325-324, 1994.

19. Ding, C., D. Pei, and A. Salomaa. *Chinese Remainder Theorem: Applications in Computing, Coding, Cryptography*, World Scientific (1996).

20. Dubois, Eric and Anastasios N. Venetsanopoulos. A New Algorithm for the Radix-3 FFT, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 26(3): 222-5, 1978.

21. Duhamel, Pierre and H. Hollmann. Split-radix FFT algorithm, *Electronic Letters*, 20: 14-6, 1984.

22. Duhamel, P. and M. Vetterli. Fast Fourier Transforms: A tutorial review and a state of the art, *Signal Processing*, 19: 259-99, 1990.

23. Federenko, Sergei V. A Simple Algorithm for Decoding Reed-Solomon Codes and Its Relation to the Welch Berlekamp Algorithm, *IEEE Trans. Inf. Theory*, 51(3): 1196-98, 2005.

24. Federenko, Sergei V. Correction to "A Simple Algorithm for Decoding Reed-Solomon Codes and Its Relation to the Welch Berlekamp Algorithm", *IEEE Trans. Inf. Theory*, 52(3): 1278, 2006.

25. Fiduccia, Charles. Polynomial evaluation via the division algorithm: the fast Fourier transform revisited, *Proceedings of the fourth annual ACM symposium on theory of computing*, 88-93, 1972.

26. Fine, N. J. Binomial coefficients modulo a prime, *Mathematical Association of America Monthly*, 54(10): 589-92, 1947.

27. Fourier, Joseph. *The Analytical Theory of Heat*, 1822; English translation: Dover, 1955.

28. Frigo, Matteo and Steven Johnson. The Design and Implementation of FFTW3, *Proceedings of the IEEE*, 93(2): 216-231, 2005.

29. Gao, Shuhong. A new algorithm for decoding Reed-Solomon codes, *Communications, Information and Network Security*, 712: 55-68, 2003.

30. Gao, Shuhong. *Clemson University Mathematical Sciences 985 Course Notes*, Fall 2001.

31. Garg, Hari Krishna, *Digital Signal Processing Algorithms: Number Theory, Convolution, Fast Fourier Transforms, and Applications*, CRC Press (1998).

32. von zur Gathen, Joachim and Jürgen Gerhard. *Arithmetic and Factorization of Polynomials over F_2*. Technical report, University of Paderborn, 1996.

33. von zur Gathen, Joachim and Jürgen Gerhard. Fast Algorithms for Taylor Shifts and Certain Difference Equations. *Proceedings of the 1997 International Symposism on Symbolic and Algebraic Computation*, 40-7, 1997.

34. von zur Gathen, Joachim and Jürgen Gerhard. *Modern Computer Algebra*, Cambridge University Press (2003).

35. Gentleman, Morven and Gordon Sande. Fast Fourier Transforms – for fun and profit, *AFIPS 1966 Fall Joint Computer Conference*. Spartan Books, Washington, 1966.

36. Good, Irving J. Random motion on a finite abelian group, *Proceedings of the Cambridge Philosophical Society*, 47: 756-62, 1951.

37. Gopinath, R. A. Comment: Conjugate Pair Fast Fourier Transform, *Electronic Letters*, 25(16) 1084, 1989.

38. Gorenstein, D. and N. Zierler, A Class of error-correcting codes in p^m symbols, *J. Soc. Indust. Appl. Math*, 9: 207-14, 1961.

39. Heideman, M. T. and C. S. Burrus, *A Bibliography of Fast Transform and Convolution Algorithms II*, Technical Report Number 8402, Electrical Engineering Dept., Rice University, Houston, TX 77251-1892, 1984.

40. Heideman, M. T., Don H. Johnston, and C. S. Burrus, Gauss and the History of the Fast Fourier Transform, *IEEE ASSP Magazine*, 14-21, October 1984.

41. van der Hoeven, Joris. The Truncated Fourier Transform and Applications. *ISSAC '04 Proceedings*, 2004.

42. van der Hoeven, Joris. Notes on the Truncated Fourier Transform. Preprint., 2005.

43. Horowitz, Ellis. A fast method for interpolation using preconditioning, *Information Processing Letters*, 1: 157-63, 1972.

44. Johnson, Steven G. and Matteo Frigo. A modified split-radix FFT with fewer arithmetic operations, *IEEE Trans. Signal Processing*, 55 (1): 111-119, 2007.

45. Kamar, I. and Y. Elcherif. Conjugate Pair Fast Fourier Transform. *Electronic Letters*, 25 (5): 324-5, 1989.

46. Karatsuba, A. and Y. Ofman. Multiplication of Multidigit Numbers on Automata, *Soviet Physics - Doklady*, 7: 595-6, 1963.

47. Karatsuba, A. A. The Complexity of Computations, *Proceedings from the Steklov Institute of Mathematics*, 211: 169-183, 1995.

48. Knuth, D. E. *The Art of Computer Programming, Vol I: Fundamental Algorithms, 3rd edition.*, Addison Wesley (1997).

49. Knuth, D. E. *The Art of Computer Programming, Vol II: Seminumerical Algorithms, 3rd edition.*, Addison Wesley (1998).

50. Krot, A. M. and H. B. Minervina. Comment: Conjugate Pair Fast Fourier Transform, *Electronic Letters*, 28(12): 1143-4, 1992.

51. Lay, David C. *Linear Algebra and its Applications, Second Edition*, Addison-Wesley (2000).

52. Lidl, Rudolf, and Harald Neiderreiter. *Finite Fields. Encyclopedia of Mathematics and Its Applications, Volume 20*, Cambridge University Press (1987).

53. Van Loan, Charles. *Computational Frameworks for the Fast Fourier Transform*, Society of Industrial and Applied Mathematics (1992).

54. McEliece, R. J. *The Theory of Information and Coding*, Addison-Wesley (1977).

55. Merris, Russell. *Combinatorics (Second Edition)*, Wiley (2003).

56. Moenck, R. T. Fast Computation of GCD's, *Proceedings of the Fifth Annual ACM Symposium on the Theory of Computing*, 142-151, 1973.

57. Moenck, R. and A. Borodin. Fast modular transform via division, *Proceedings of the 13th Annual IEEE Symposium on Switching and Automata Theory*, 90-96, 1972.

58. Montgomery, Peter L. Five, Six, and Seven-Term Karatsuba-Like Formulae, *IEEE Transactions on Computers*, 54(3): 362-9, 2005.

59. Moon, Todd K. *Error Correction Coding: Mathematical Methods and Algorithms*, Wiley (2005).

60. Nicholson, W. Keith. *Introduction to Abstract Algebra, Third Edition*, Wiley (2007).

61. Nussbaumer, H.J. *Fast Fourier Transforms and Convolution Algorithms*, Springer (1990).

62. Proakis, John G. and Dimitris G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications, Third Edition*, Prentice Hall (1996).

63. Reed, I. S. and G. Solomon. Polynomial codes over certain finite fields, *SIAM J. Appl. Math*, 8: 300-4, 1960.

64. Reischert, D. *Schnelle Multiplikation von Polynomen über GF(2) und Anwendungen*. Diplomarbeit, University of Bonn, Germany, 1995.

65. Roberts, Fred S. *Applied Combinatorics*, Prentice Hall (1984).

66. Rosen, Kenneth H. *Discrete Mathematics and its applications, Sixth Edition*, McGraw Hill (2007).

67. Saff, E. B. and A. D. Snider. *Fundamentals of Complex Analysis with Applications to Engineering and Science*, Peason Education, Inc. (2003).

68. Schonhage, Arnold. Schnelle Multiplikation von Polynomen uber Korpern der Characteristik 2, *Acta Informatica*, 7: 395-8, 1977.

69. Schonhage, Arnold. Variations on Computing Reciprocals of Power Series, *Information Processing Letters*, 74: 41-6, 2000.

70. Schonhage, Arnold and V. Strassen. Schnelle Multiplikation grosser Zahlen, *Computing*, 7: 281-292, 1971.

71. Shiozaki, A. Decoding of redundant residue polynomial codes using Euclid's Algorithm, *IEEE Trans. Inf. Theory*, 34(5): 1351-1354, 1988.

72. Strassen, V. The Computational Complexity of continued fractions, *SIAM Journal on Computing*, 12(1): 1-27, 1983.

73. Stewart, James. *Calculus, Fourth edition*, Brooks/Cole Publishing Company (1999).

74. Suguyama, Y., M. Kasahara, S. Hirasawa, and Toshihiko Namekawa. A method for solving key equation for decoding Goppa codes, *Information and Control*, 27: 87-99, 1975.

75. Sunzi Suanjing (Sunzi's Computational Canon).

76. Suzuki, Yōiti, Toshio Sone, and Kenuti Kido. A New FFT algorithm of Radix 3, 6, and 12, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 34(2): 380-3, 1986.

77. Takahashi, Daisuke. An Extended Split-Radix FFT Algorithm, *IEEE Processing Letters*, 8 (5): 145-7, 2001.

78. Tucker, Alan. *Applied Combinatorics: Fourth Edition*, Wiley (2002).

79. Vetterli, Martin and Pierre Duhamel, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(1): 57-64, 1989.

80. Wan, Zhe-Xian. *Lectures on Finite Fields and Galois Rings*, World Scientific Publishing Co. (2003).

81. Wang, Yao and Xuelong Zhu. A Fast Algorithm for Fourier Transform Over Finite Fields and its VLSI Implementation, *IEEE Journal on Selected Areas in Communications*, 6 (3): 572-7, 1988.

82. Weimerskirch, Andre and Christof Paar. Generalizations of the Karatsuba Algorithm for Polynomial Multiplication. Preprint submitted to Design, Codes, and Cryptography, 2002.

83. Winograd, S. On Computing the DFT, *Math. Comp.*, 32(1): 175-99, 1978.

84. Yavne, R. An economical method for calculating the discrete Fourier transform, *Proc. Fall Joint Computing Conference*, 115-25, 1968.