A New Truncated Fourier Transform Algorithm

Andrew Arnold
Symbolic Computation Group
University of Waterloo
Waterloo, Ontario, Canada
a.arnold55@gmail.com
http://cs.uwaterloo.ca/~a4arnold

ABSTRACT

Truncated Fourier Transforms (TFTs), first introduced by van der Hoeven, refer to a family of algorithms that attempt to smooth "jumps" in complexity exhibited by FFT algorithms. We present an in-place TFT whose time complexity, measured in terms of ring operations, is asymptotically equivalent to existing not-in-place TFT methods. We also describe a transformation that maps between two families of TFT algorithms that use different sets of evaluation points.

Categories and Subject Descriptors

F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computation of transforms*; G.4 [**Mathematical Software**]: Algorithm design and analysis

Keywords

Truncated Fourier Transform; in-place algorithms

1. INTRODUCTION

Let $\mathcal{R}$ be a ring containing an N-th principal root of unity ω. Given two polynomials $f, g \in \mathcal{R}[z]$, $\deg(fg) < N$, we can compute fg by way of the Discrete Fourier Transform (DFT): a linear, invertible map which evaluates a given polynomial at the powers of ω.

Computing the DFT naively is quadratic-time. However, if N is strictly comprised of small prime factors, one can compute a DFT using $\mathcal{O}(N \log N)$ arithmetic operations by way of the Fast Fourier Transform (FFT). The most widely-known FFT, the *radix-2* FFT, requires that N is a power of two. To compute the DFT of an input of arbitrary size, one typically appends zeros to the input to give it power-of-two length, and then applies a radix-2 FFT. This method exhibits significant jumps in its time and memory costs.

Truncated Fourier Transforms (TFTs) flatten these jumps in complexity. A TFT takes an input of length $n \leq N$ and returns a size-n subset of its length-N DFT, with time complexity that grows comparatively smoothly with $n \log n$. Typically one chooses the first n entries of the DFT, with the DFT sorted in bit-reversed order. This is a natural choice as it comprises the first n entries of the output of an in-place FFT. We will call such a TFT the *bit-reversed* TFT.

Van der Hoeven [11] showed how one can obtain a polynomial $f(z)$ from its bit-reversed TFT, provided one knows the terms of $f(z)$ with degree at least n. This allows for faster FFT-based polynomial multiplication, particularly for products whose degree is a power of two or slightly larger. Harvey and Roche showed further in [5] how the bit-reversed TFT can be computed in-place, at the cost of a constant factor additional ring multiplications.

Mateer [7] devised a TFT algorithm based on a series of modular reductions, that acts as a preprocessor to the FFT. Mateer's TFT algorithm, which we discuss in section 3, reduces $f(z)$, $\deg(f) < n$, modulo cyclotomic polynomials of the form $z^k + 1$, k a power-of-two. We will call this TFT the *cyclotomic* TFT.

In [9], Sergeev shows how the cyclotomic TFT can be made in-place with cost asymptotically equivalent to not-in-place TFT algorithms. In section 4, we describe Sergeev's algorithm. In section 5, we give an in-place algorithm of similar cost, related to Sergeev's, for computing the cyclotomic TFT. One incremental improvement of this new TFT algorithm is that, for input sizes n of low Hamming weight, it requires fewer passes through our input array than Sergeev's algorithm in order to produce the polynomial images.

One caveat of the cyclotomic TFT is that different-sized inputs may use entirely different sets of evaluation points. This is problematic in applications to multivariate polynomial multiplication. In section 6, we show how an algorithm that computes the cyclotomic TFT can be modified to compute a bit-reversed TFT by way of an affine transformation.

As a proof of concept we implemented the algorithms introduced in this paper in Python. These implementations can be found at `http://cs.uwaterloo.ca/~a4arnold/tft`.

2. PRELIMINARIES

2.1 The Discrete Fourier Transform

The *Discrete Fourier Transform* (DFT) [4] of a polynomial $f(z)$ is its vector of evaluations at the distinct powers of a root of unity. Specifically, if $f(z) = \sum_{i=0}^{N-1} a_i z^i$ is a polynomial over a ring $\mathcal{R}$ containing an order-N root of unity ω, then we define the Discrete Fourier Transform of $f(z)$ as

$$\mathrm{DFT}_\omega(f) = \left(f(\omega^0), \ldots, f(\omega^{N-1})\right).$$

ISSAC'13, June 26–29, 2013, Boston, Massachusetts, USA.

We treat the polynomial f and its vector of coefficients $a = (a_0, a_1, \ldots, a_{N-1})$ as equivalent and use the notation $\mathrm{DFT}_\omega(a)$ and $\mathrm{DFT}_\omega(f)$ interchangeably. If we take addition and multiplication component-wise in $\mathcal{R}^N$, then the map $\mathrm{DFT}_\omega \colon \mathcal{R}[z]/(z^N - 1) \to \mathcal{R}^N$ forms a ring homomorphism. Throughout this paper we will assume ω is a *principal* root of unity, that is, for j not divisible by N, the order of ω, $\sum_{i=0}^{N-1} \omega^{ij} = 0$. In such case DFT_ω has an inverse map $\mathrm{IDFT}_\omega \colon \mathcal{R}^N \to \mathcal{R}[z]/(z^N - 1)$, defined by

$$\mathrm{IDFT}_\omega(\hat{a}) = \tfrac{1}{N}\mathrm{DFT}_{\omega^{-1}}(\hat{a}),$$

where $\hat{a} \in \mathcal{R}^N$ and we again treat a polynomial as equivalent to its vector of coefficients. This suggests a multiplication algorithm for f, $g \in \mathcal{R}[z]$.

THEOREM 1 (THE CONVOLUTION THEOREM). *Let f, g be polynomials over a ring $\mathcal{R}$ containing an N-th principal root of unity ω. Then*

$$fg \bmod (z^N - 1) = \mathrm{IDFT}_\omega\left(\mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)\right),$$

where "$\cdot$" is the vector component-wise product, and, given two polynomials $s(z), t(z) \in \mathcal{R}[z]$, $s(z) \bmod t(z)$ denotes the unique polynomial $r(z)$ such that $t(z)$ divides $r(z) - s(z)$ and $\deg(r) < \deg(t)$ throughout.

Thus to multiply f and g, we can choose $N > \deg(fg)$ and an N-th principal root of unity $\omega \in \mathcal{R}$, compute the length-N DFTs of f and g, take their component-wise product, and take the inverse DFT of that product.

2.2 The Fast Fourier Transform

We can compute the Discrete Fourier Transform of $f(z)$, f reduced modulo $(z^N - 1)$, by way of a *Fast Fourier Transform* (FFT). The FFT is believed to have been first discovered by Gauss, but did not become well known until it was famously rediscovered by Cooley and Tukey [1]. For a detailed history of the FFT we refer the reader to [6].

The simplest FFT, the *radix-2* FFT, assumes $N = 2^p$ for some $p \in \mathbb{Z}_{\geq 0}$. We describe the radix-2 FFT in terms of modular reductions. We break f into images modulo polynomials of decreasing degree until we have the images $f \bmod (z - \omega^i) = f(\omega^i)$, $0 \leq i < N$. At the start of the first iteration we have f reduced modulo $z^N - 1$. At the start of the i-th iteration, we will have the 2^{i-1} images

$$f \bmod (z^{2u} - \omega^{2uj}) \quad \text{for} \quad 0 \leq j < 2^{i-1},$$

where $u = N/2^i$. If $i = p + 1$ this gives us the DFT of f. Consider then the image $f' = f \bmod (z^{2u} - \omega^{2uj}) = \sum_{k=0}^{2u-1} b_k z^k$, for some j, $0 \leq j < 2^{i-1}$. We break this image into two images f_0 and f_1, where

$$f_0 = f' \bmod (z^u - \omega^{uj}), \text{ and}$$
$$f_1 = f' \bmod (z^u + \omega^{uj}) = f' \bmod (z^u - \omega^{uj+N/2}).$$

We can write f_0 and f_1 in terms of the coefficients b_k:

$$f_0 = \sum_{k=0}^{u-1} (b_k + \omega^{uj} b_{k+u}) z^k, \quad f_1 = \sum_{k=0}^{u-1} (b_k - \omega^{uj} b_{k+u}) z^k.$$

Thus, given an array containing the coefficients b_k of f', we can write f_0 and f_1 in place of f' by way of operations

$$\begin{bmatrix} b_k \\ b_{k+u} \end{bmatrix} \longleftarrow \begin{bmatrix} 1 & \omega^{uj} \\ 1 & -\omega^{uj} \end{bmatrix} \begin{bmatrix} b_k \\ b_{k+u} \end{bmatrix}, \quad 0 \leq i < u. \qquad (1)$$

The pair of assignments (1) are known as a *butterfly operation*, and can be performed with a ring multiplication by the *twiddle factor* ω^{uj}, and two additions. Note f_0 and f_1 are in a similar form as f', and if $u > 1$ we can break those images into smaller images in the same fashion. Starting this method with input $f \bmod (z^N - 1)$, will give us $f \bmod (z - \omega^j) = f(\omega^j)$, for $0 \leq j < N$.

If the butterfly operations are performed in place, the resulting evaluations $f(\omega^j)$ will be written in *bit-reversed* order. More precisely, if we let $[j]_p$ denote the integer resulting from reversing the p bits of j, $0 \leq j < 2^p$, we have that $f(\omega^j)$ will be written in place of a_k, where $k = [j]_p$. As an example,

$$[13]_5 = [01101_2]_5 = 10110_2 = 16 + 4 + 2 = 22.$$

We can make the FFT entirely in-place by computing the powers of ω^u sequentially at every iteration. This entails traversing the array in a non-sequential order. Procedure FFT describes such an implementation.

If we observe that

$$\begin{bmatrix} 1 & \omega^{uj} \\ 1 & -\omega^{uj} \end{bmatrix}^{-1} = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ \omega^{-uj} & -\omega^{-uj} \end{bmatrix},$$

then we can implement an inverse FFT by inverting the butterfly operations in reversed order. We can, moreover, delay multiplications by powers of $\frac{1}{2}$ until the end of the inverse FFT computation. This entails multiplying the result by $\frac{1}{N}$.

Procedure FFT(**a**, N), an in-place implementation of the radix-2 FFT

Input: **a**, a length $N = 2^p$ array containing $f \in \mathcal{R}[z]$.
Result: $\mathrm{DFT}_\omega(f)$ is written to **a** in bit-reversed order.

```
for i ⟵ 1 to p do
    u ⟵ N/2^i
    for j ⟵ 0 to N/(2u) − 1 do
        t ⟵ 2[j]_i u
        for k ⟵ 0 to u − 1 do
            // Butterfly operations (1)
            [a_{t+k}]     [1   ω^{uj}] [a_{t+k}]
            [a_{t+k+u}] ⟵ [1  −ω^{uj}] [a_{t+k+u}]
```

THEOREM 2. *Let N be a power of two and ω an N-th root of unity. Then* FFT *computes a length-N DFT in place using no more than $\frac{1}{2}N \log N + \mathcal{O}(N)$ ring multiplications and $N \log N + \mathcal{O}(N)$ ring additions, where* log *is taken to be base-2 throughout. The inverse FFT can be computed in-place with asymptotically equivalent cost.*

Using the radix-2 FFT, if $d = \deg(fg)$, we choose N to be the least power of 2 exceeding d. This entails appending zeros to the input arrays containing the coefficients of f and g respectively. By this method, computing a product of degree 2^p costs at least double that a product of degree less than 2^p. Crandall's "devil's convolution" algorithm [2] somewhat flattens these jumps in complexity, though not entirely. It works by reducing a discrete convolution of arbitrary length into more easily computable convolutions. More recently, Truncated Fourier Transform (TFT) algorithms, described hereafter, have addressed this issue.

2.3 Truncated Fourier Transforms

In many applications, it is useful to compute a *pruned* DFT, a subset of a length-N DFT, at a cost less than that to compute a complete DFT. In 2004, van der Hoeven [10] showed that, given some knowledge of the form of the input, one can invert some pruned DFTs. The inverse transform relies on the observation that, given any two of the inputs/outputs to a butterfly operation (1), one can compute the other two values. Suppose $n \in \mathbb{Z}_{>0}$ is arbitrary and N is the least power of two at least n. For ω, a primitive root of unity of order $N = 2^p$, van der Hoeven showed how to invert the length-n Truncated Fourier Transform,

$$\mathrm{TFT}_{\omega,n}(f) = \left(f(\omega^{[i]_p})\right)_{0 \le i < n},$$

when we know the terms of $f(z)$ of degree at least n (e.g., when $\deg(f) < n$). To distinguish this particular TFT we will call it the *bit-reversed* TFT.

THEOREM 3 (VAN DER HOEVEN, [11]). *Suppose f is a polynomial over $\mathcal{R}$ of degree less than n. $\mathrm{TFT}_{\omega,n}(f)$ can be computed using $n \log n + \mathcal{O}(n)$ ring additions and $\frac{1}{2} n \log n + \mathcal{O}(n)$ multiplications by powers of ω.*

$f(z)$ can be recovered from $\mathrm{TFT}_{\omega,n}(f)$ using $n \log n + \mathcal{O}(n)$ shifted ring additions and $\frac{1}{2} n \log n + \mathcal{O}(n)$ multiplications.

A shifted ring addition in this context merely means an addition plus a multiplication by $2^{\pm 1}$. Van der Hoeven's algorithm generalizes to allow us to compute arbitrary subsets of the DFT. Given subsets $S, T \subseteq \{0, 1, \ldots, N-1\}$, we define

$$\mathrm{TFT}_{\omega,S,T}(f) = \left(f(\omega^{[i]_p})\right)_{i \in T},$$

where we now assume f is of the form $f(z) = \sum_{i \in S} a_i z^i$. However, such a transform may have a greater complexity than stated in theorem 3, taking $n = \#S$. Moreover, such a map is not necessarily invertible, even in the case $S = T$.

EXAMPLE 4. *Let $N = 8$. Taking an 8-th principal root of unity ω with $S = T = \{0, 3, 4, 5\}$ gives an uninvertible map. To see that the map $\mathrm{TFT}_{\omega,S,T}$ is not invertible, one can check that the polynomial $f(z) = (1+\omega^2)z^5 - z^4 + (1-\omega^2)z^3 - 1$ evaluates to 0 for $z = \omega^k$, $k \in \{[0]_3, [3]_3, [4]_3, [5]_3\} = \{0, 6, 1, 5\}$.*

Van der Hoeven's method still exhibits significant jumps in space complexity, as it requires space for N ring elements regardless of n. In 2010, Harvey and Roche [5] introduced an in-place TFT algorithm, requiring $n + \mathcal{O}(1)$ ring elements plus an additional $\mathcal{O}(1)$ bounded-precision integers to compute $\mathrm{TFT}_{\omega,n}(f)$. Their method potentially requires evaluating polynomials using linear-time methods. This adds an additional constant factor to the algorithm's worst-case cost.

THEOREM 5 (ROCHE, THEOREM 3.5, [8]). *Let N, n, f and ω be as in theorem 3. Then $\mathrm{TFT}_{\omega,n}(f)$ can be computed in-place using at most $\frac{5}{6} n \log n + \mathcal{O}(n)$ ring multiplications and $\mathcal{O}(n \log n)$ ring additions.*

The inverse in-place transform entails similarly many ring multiplications and $\mathcal{O}(n \log n)$ shifted ring additions. As an application, Harvey and Roche used this transform towards asymptotically fast in-place polynomial multiplication.

3. THE CYCLOTOMIC TFT

3.1 Notation

We use the following notation throughout section 3 and thereafter. Suppose now that we have a polynomial $f(z) = \sum_{j=0}^{n-1} a_j z^j \in \mathcal{R}[z]$, where n is not necessarily a power of two. For the remainder of this paper, we write n as $n = \sum_{i=1}^{s} n_i$, where $n_i = 2^{n(i)}$, $n(i) \in \mathbb{Z}_{\ge 0}$ and $n_i > n_j$ for $1 \le i < j \le s$. Here we let N be the smallest power of two exceeding n. For $1 \le i \le s$, we let

$$\Phi_i = z^{n_i} + 1.$$

The TFT algorithms of sections 3 and thereafter will compute the evaluations of $f(z)$ at the roots of Φ_i. Namely, if we fix a canonical root $\omega = \omega_1$ of Φ_1, and then let, for $2 \le i \le s$, $\omega_i = \omega_1^{n_1/n_i}$, a root of Φ_i, these algorithms will compute

$$f(\omega_i^{2j+1}) \quad \text{for} \quad 0 \le j < n_i, \quad 1 \le i \le s, \tag{2}$$

the evaluation of $f(z)$ at the roots of the cyclotomic polynomials Φ_i. As such, we will call it here the *cyclotomic* TFT and denote it by $\mathrm{TFT}'_{\omega,n}(f)$. The choice of our order-N root ω only affects the ordering of the elements of the cyclotomic TFT. If we let

$$T(n) = \{k : n_i \le k < 2n_i \text{ for some } i,\ 1 \le i \le s\},$$

then we have that $\mathrm{TFT}'_{\omega,n}$ uses the same set of evaluation points as $\mathrm{TFT}_{\omega,S,T(n)}(f)$.

We define, for $1 \le i \le s$, the images

$$f_i = f \bmod \Phi_i.$$

The algorithms for computing a cyclotomic TFT all follow a similar template: we will produce the images f_i sequentially, and then evaluate f at the roots of Φ_i in place of each image f_i.

3.2 Discrete Weighted Transforms

Given the images $f_i = f \bmod \Phi_i$, one can evaluate f at the roots of Φ_i by way of a *Discrete Weighted Transform* (DWT), which comprises an affine transformation followed by an FFT [3].

In a more general setting, suppose we have an image $f^* = f \bmod (z^K - c)$, where K is a power-of-two. Assuming c has an K-th root over our ring $\mathcal{R}$, the roots of $z^K - c$ are all of the form $c^{1/K}\gamma^i$, $0 \le i < K$, where γ is an order-K root of unity. Thus to evaluate f at the roots of $(z^K - c)$ one can replace $f^*(z)$ with $f^*(c^{1/K} z)$, and then compute $\mathrm{DFT}_\gamma(f^*(c^{1/K} z))$. Replacing $f^*(z)$ with $f^*(c^{1/K} z)$ iteratively term-by-term entails fewer than $2K$ ring multiplications.

To evaluate f at the roots of $\Phi_i(z) = z^{n_i} - \omega_i^{n_i}$, one would write $f_i(\omega_i z)$ in place of $f_i(z)$, then compute $\mathrm{DFT}_{\omega_i^2}(f_i(\omega_i z))$ by way of the FFT. As both the FFT and the affine transformation are invertible, a Discrete Weighted Transform is easily invertible as well.

Procedure DWT($\mathbf{a}$, K, v), the Discrete Weighted Transform

Input: $\mathbf{a}$, a length-K array; $v \in \mathcal{R}$, a weight.
for $i \longleftarrow 0$ **to** $K - 1$ **do** $\mathbf{a}_i \longleftarrow v^i \mathbf{a}_i$
FFT($\mathbf{a}, K$)

3.3 Mateer's TFT algorithm

Procedure MateerTFT below gives a short description of Mateer's algorithm [7] for computing $\mathrm{TFT}'_{\omega,n}(f)$. Given array $\mathbf{a}$ and integer k we let the array $\mathbf{a}+k$ denote the array $\mathbf{b}$ such that $\mathbf{a}_{\ell+k}$ and $\mathbf{b}_\ell$ refer to the same element for all ℓ. MateerTFT will write f_i to $\mathbf{a}+n_i$.

Mateer's algorithm computes the images $f \bmod (z^K+1)$ for $K=2^\ell$, $n(s) \le \ell \le n(1)$, and the image $f \bmod (z^{n_s}-1)$. For the images $f \bmod (z^K+1)$ where $K=n_i$, $1 \le i \le s$, we perform a DWT to evaluate f at the roots of Φ_i. The remaining images we can simply discard.

Procedure MateerTFT($\mathbf{a}, n$)

Input: $\mathbf{a}$, a length $N = 2^{\lfloor \log n \rfloor + 1}$ array containing $f \in \mathcal{R}[z]$.
Result: $\mathrm{TFT}'_{\omega,n}(f)$ is written to $\mathbf{a}$.
$K \longleftarrow N/2$
while $K > n_s$ **do**
  for $e \longleftarrow 0$ **to** K **do**
    $(\mathbf{a}_e, \mathbf{a}_{e+K}) \longleftarrow (\mathbf{a}_e + \mathbf{a}_{e+K}, \mathbf{a}_e - \mathbf{a}_{e+K})$
  $K \longleftarrow K/2$
for $i \longleftarrow 1$ **to** s **do** DWT($\mathbf{a}+n_i, n_i, \omega_i$)

On input we are given $f(z)$ reduced modulo z^N-1. Given $f \bmod (z^{2K}-1) = \sum_{d=0}^{2K-1} b_d z^d$, Mateer's TFT breaks $f \bmod (z^{2K}-1)$ into the images

$$f \bmod (z^K-1) = \sum_{d=0}^{K-1} (b_d + b_{d+K}) z^d, \text{ and} \tag{3}$$

$$f \bmod (z^K+1) = \sum_{d=0}^{K-1} (b_d - b_{d+K}) z^d. \tag{4}$$

Performing the aforementioned modular reductions for $K = N/2, \ldots, n_s$ amounts to $\mathcal{O}(n)$ additions. The number of multiplications required to perform the DWTs is bounded by

$$\sum_{i=1}^{s} \tfrac{1}{2} n_i \log n_i + c n_i \le \tfrac{1}{2} n \log n + \mathcal{O}(n), \tag{5}$$

for some constant c. A similar analysis for the ring additions due to the DWTs gives us the following complexity:

LEMMA 6 (MATEER). *Procedure* MateerTFT *computes* $\mathrm{TFT}'_{\omega,n}(f)$ *from* f *using* $\frac{1}{2} n \log n + \mathcal{O}(n)$ *ring multiplications and* $n \log n + \mathcal{O}(n)$ *ring additions.*

Mateer gives a method of inverting the cyclotomic TFT with asymptotically equivalent cost, the details of which we omit here. MateerTFT is not in-place as the images $f \bmod (z^{N/2}-1)$ and $f \bmod (z^{N/2}+1)$ may have maximal degree.

4. COMPUTING THE CYCLOTOMIC TFT WITHOUT ADDITIONAL MEMORY

In order to compute the cyclotomic TFT in-place, it appears, unlike the Mateer TFT, that we need to use some of the information from the images $f_1, \ldots, f_i$ towards producing the image f_{i+1}. Both Sergeev's TFT and the new algorithm presented thereafter work in this manner.

For $1 \le i \le s$, let

$$\Gamma_i(z) = \prod_{j=1}^{i} \Phi_j(z) \quad \text{and} \quad C_i = f \bmod \Gamma_i.$$

We call C_i the *combined* image of f, as it is the result of Chinese remaindering on the images $f_1, \ldots, f_i$. We also define

$$q_i = \begin{cases} f & \text{if } i = 0, \\ f \text{ quo } \Gamma_i & \text{if } 1 \le i \le s, \end{cases}$$

the quotient produced dividing f by Γ_i, as well as

$$n_i^* = \begin{cases} n & \text{if } i = 0, \\ n \bmod n_i & \text{if } 1 \le i \le s. \end{cases}$$

Note that, as $\Phi_i \bmod \Phi_j = 2$ for $j > i$, we also have $\Gamma_i \bmod \Phi_j = 2^i$ for $j > i$. Similarly, $\Gamma_i \bmod (z^K-1) = 2^i$ for K, a power of two at most n_i.

For any choice of $1 \le i \le s$, we have

$$f(z) = C_i + \Gamma_i q_i.$$

It is straightforward to obtain q_i, given f. Note that the degrees of any two distinct terms of Γ_i differ by at least n_i, and that $\deg(q_i) < n_i$. Thus, as Γ_i is monic, we have that the coefficients of q_i merely comprise the coefficients of the higher-degree terms of f. More precisely,

$$q_i = \sum_{e=0}^{n_i^*-1} a_{n-n_i^*+e} z^e.$$

By a similar argument, we also have that, for $1 \le i \le s$,

$$q_i = q_{i-1} \text{ quo } \Phi_i.$$

We note that $q_s = 0$.

4.1 Computing images of C_i without explicitly computing C_i

We will express the combined image C_i, C_i reduced modulo $z^m \pm 1$, m a power of two, in terms of the coefficients of the images $f_1, \ldots, f_i$. To this end we introduce the following notation. Given an integer e, we will let $e[i]$ refer to the i-th bit of e, i.e.,

$$e = \sum_{i=0}^{\lfloor \log(e) \rfloor} e[i] 2^i, \qquad e[i] \in \{0, 1\}.$$

Sergeev's TFT relies on the following lemma, albeit stated differently here than in [9].

LEMMA 7. *Fix* i *and* j *such that* $1 \le j \le i \le s$. *Suppose that* $f_j = z^e$ *and that* $f_\ell = 0$ *for all* $\ell \ne j$, $1 \le \ell \le i$. *Let* m *be a power of two at most* n_i. *Then* $C_i \bmod (z^m-1)$ *is non-zero only if* $e[n(\ell)] = 1$ *for all* $\ell \in \{j+1, j+2, \ldots, i\}$, *in which case*

$$C_i \bmod (z^m-1) = 2^{i-j} z^e \bmod (z^m-1), \tag{6}$$
$$= 2^{i-j} z^{e \bmod m}.$$

Lemma 7 can be derived from lemma 1 in [9]. As Φ_k, $k > i$, divides $z^{n_i}-1$, lemma 7 gives us the following corollary.

COROLLARY 8. *Fix* i, j *and* k *such that* $1 \le j \le i < k \le s$. *Suppose that* $f_j = z^e$ *and that* $f_\ell = 0$ *for all* $\ell \ne j$,

$1 \le \ell \le i$. *Then* $C_i \bmod \Phi_k$ *is non-zero only if* $e[n(\ell)] = 1$ *for all* $\ell \in \{j+1, j+2, \ldots, i\}$, *in which case*

$$C_i \bmod \Phi_k = 2^{i-j} z^e \bmod \Phi_k.$$

Given that $z^{n_k} \bmod \Phi_k = -1$, we have that

$$2^{i-j} z^e \bmod \Phi_k = (-1)^{e[n(k)]} 2^{i-j} z^{(e \bmod n_k)},$$

where $e \bmod n_k$ is the integer e^* such that $n_k \mid (e - e^*)$ and $0 \le e^* < n_k$. The values $e[n(\ell)]$ can be determined from $n_\ell = 2^{n(\ell)}$ and e by way of a bitwise "and" operation.

EXAMPLE 9. *Suppose the input size is* $n = 86$ $(n = 64 + 16 + 4 + 2)$, *and suppose that*

$$f_1 = f \bmod (z^{64} + 1) = z^e, \qquad f_2 = 0, \qquad f_3 = 0.$$

In this example,

$$C_3 = f \bmod \left[(z^{64}+1)(z^{16}+1)(z^4+1)\right].$$

Let $g = C_3 \bmod (z^2 + 1)$. *Then by corollary 8,*

$$g = \begin{cases} 0 & \text{if } e \in [0,20) \cup [24,28) \cup [32,52) \cup [56,60), \\ 4 & \text{if } e = 20, 28, 52, \text{ or } 56, \\ 4z & \text{if } e = 21, 29, 53, \text{ or } 57, \\ -4 & \text{if } e = 22, 30, 54, \text{ or } 58, \\ -4z & \text{if } e = 23, 31, 55, \text{ or } 59. \end{cases}$$

REMARK 10. *Let* $1 \le j \le i \le s$. *A proportion of* 2^{j-i} *terms of* f_j *have an exponent satisfying the non-zero criterion of lemma 7 and corollary 8.*

PROOF OF LEMMA 7. We fix e and j and prove the lemma by induction on i.

Base case: Suppose $i = j$, in which case the non-zero criterion of the lemma always holds and we need only to show (6). We have that $C_i \bmod \Phi_i = z^e$ and $C_i \bmod \Phi_\ell = 0$ for $\ell < i$. Chinese remaindering gives us

$$C_i = C_{i-1} + \Gamma_{i-1}\left(\Gamma_{i-1}^{-1}(z^e - C_{i-1}) \bmod \Phi_i\right).$$

Furthermore, as $f_\ell = 0$ for $1 \le \ell < i$, we have $C_{i-1} = 0$, and $C_i = 2^{1-i}\Gamma_{i-1} z^e$. Reducing this modulo $z^m - 1$, we have $C_i = z^{e \bmod m}$ as desired.

Inductive step: Suppose now that the lemma holds for a fixed $i \ge j$, and consider $C_{i+1} \bmod (z^m - 1)$, m a power of two dividing n_{i+1}. We suppose that $f_\ell = 0$ for $1 \le \ell \ne j \le i+1$ and $f_j = z^e$. By Chinese remaindering,

$$C_{i+1} = C_i - \Gamma_i\left(\Gamma_i^{-1} C_i \bmod \Phi_{i+1}\right). \tag{7}$$

We prove the inductive step by cases:

Case 1: If $e[n(\ell)] = 0$ for some ℓ, $j < \ell \le i$, then by the induction hypothesis, $C_i \bmod (z^{n_i} - 1) = 0$. As Φ_{i+1} and $z^m - 1$ both divide $z^{n_i} - 1$, the images $C_i \bmod \Phi_{i+1}$ and $C_i \bmod (z^m - 1)$ are necessarily zero as well. It follows from (7) that $C_{i+1} \bmod (z^m - 1)$ is also zero.

Case 2: If $e[n(\ell)] = 1$ for all ℓ, $j < \ell \le i$, then by the induction hypothesis,

$$C_i \bmod (z^{n_i} - 1) = 2^{i-j} z^e \bmod (z^{n_i} - 1).$$

Reducing (7) modulo $z^m - 1$, and again using that $z^m - 1$ and Φ_{i+1} divide $z^{n_i} - 1$, we have

$$\begin{aligned} &C_{i+1} \bmod (z^m - 1), \\ &= 2^{i-j} z^e - \Gamma_i\left(\Gamma_i^{-1} 2^{i-j} z^e \bmod \Phi_{i+1}\right) \bmod (z^m - 1), \\ &= 2^{i-j} z^{e \bmod m} - 2^{i-j}\left(z^e \bmod \Phi_{i+1}\right) \bmod (z^m - 1), \\ &= 2^{i-j}\left(1 - (-1)^{e[n(i+1)]}\right) z^{e \bmod m}. \end{aligned}$$

Since $1 - (-1)^{e[n(i+1)]}$ evaluates to 2 if $e[n(i+1)] = 1$ and 0 otherwise, this completes the proof. □

4.2 Sergeev's in-place cyclotomic TFT

In [9], Sergeev describes an algorithm for computing a length-n cyclotomic TFT requiring space for $n + \mathcal{O}(1)$ ring elements, with cost asymptotically equivalent to van der Hoeven's algorithm for the bit-reversed TFT. This algorithm, like Mateer's, breaks f into the images f_i with linear cost, and then applies a DWT on each image. Procedure SergeevBreakIntoImages gives Sergeev's algorithm for computing the images f_i in place of f. Using our array notation, we let $\mathbf{a}^{(i)} = \mathbf{a} + n_1 + \cdots + n_{i-1}$ and write f_i to $\mathbf{a}_i$.

Procedure SergeevBreakIntoImages($\mathbf{a}, n$)

Input: $\mathbf{a}$, a length-n array containing $f \in \mathcal{R}[z]$.
Result: $f_1, \ldots, f_s$ is written in place of f.
$\mathbf{a}^{(1)} \longleftarrow \mathbf{a}$
for $i \longleftarrow 1$ **to** $s-1$ **do** $\mathbf{a}^{(i+1)} \longleftarrow \mathbf{a}^{(i)} + n_i$

$N \longleftarrow 2^{\lfloor \log n \rfloor + 1}$
$(K, i) \longleftarrow (N/2, 0)$
while $K \ge n_s$ **do**
  if $K > n_{i+1}$ **then**
    for $d \longleftarrow 0$ **to** $n_i^* - 1$ **do**
      $\mathbf{a}_d^{(i+1)}$ **+=** $\displaystyle\sum_{j=1}^{i} 2^{i-j} \sum_{\substack{e=0 \\ e=d+K \bmod 2K \\ e[n(\ell)]=1, j<\ell\le i}}^{n_j - 1} \mathbf{a}_e^{(j)}$
  else
    for $d \longleftarrow n_{i+1}^*$ **to** $n_{i+1} - 1$ **do**
      $\mathbf{a}_d^{(i+1)}$ **-=** $\displaystyle\sum_{j=1}^{i} 2^{i-j} \sum_{\substack{e=0 \\ e=d+K \bmod 2K \\ e[n(\ell)]=1, j<\ell\le i}}^{n_j - 1} \mathbf{a}_e^{(j)}$
    for $d \longleftarrow 0$ **to** $n_{i+1}^* - 1$ **do**
      $\begin{bmatrix} \mathbf{a}_d^{(i+1)} \\ \mathbf{a}_{d+K}^{(i+1)} \end{bmatrix} \longleftarrow \begin{bmatrix} \mathbf{a}_d^{(i+1)} - \mathbf{a}_{d+K}^{(i+1)} \\ \mathbf{a}_d^{(i+1)} + \mathbf{a}_{d+K}^{(i+1)} \end{bmatrix}$
    $i \longleftarrow i + 1$
  $K \longleftarrow K/2$

At the start of an iteration of the while loop of Sergeev's TFT, we have $\mathbf{a}^{(j)}$ containing f_j for $1 \le j \le i$, and $\mathbf{a}^{(i+1)}$ containing the first n_i^* coefficients of $f \bmod (z^{2K} - 1)$, for a power of two $K \in [n_{i+1}, n_i)$. If $K > n_{i+1}$, the algorithm writes the first n_i^* coefficients of $f \bmod (z^K - 1)$ to $\mathbf{a}^{(i+1)}$. If $K = n_{i+1}$, the algorithm writes f_{i+1} to $\mathbf{a}^{(i+1)}$ and the first n_{i+1}^* coefficients of $f \bmod (z^K - 1)$ to $\mathbf{a}^{(i+2)}$.

Consider then the case $K > n_i$. Write $f \bmod (z^{2K} - 1) = \sum_{d=0}^{2K-1} b_d z^d$ and $f_j = \sum_{e=0}^{n_j} a_e^{(j)} z^e$. The coefficients of

$f \bmod (z^K - 1)$ are given by (3), as is used in Mateer's algorithm. To write the length-n_i^* truncation of $f \bmod (z^K - 1)$ in place of $f \bmod (z^{2K} - 1)$ we merely add b_{d+K} to b_d for $0 \le d < n_i^*$. Since

$$f \bmod (z^{2K} - 1) = C_i + 2^i q_i \bmod (z^{2K} - 1)$$

and $\deg(q_i) < n_i^* < K$, it follows that b_{d+K} depends strictly on $C_i \bmod (z^{2K} - 1)$. Thus by lemma 7,

$$b_{d+K} = \sum_{j=1}^{i} 2^{i-j} \sum_{e \in S_{j,i}^d} a_e^{(j)}, \tag{8}$$

where

$$S_{j,i}^d = \left\{ e \in [0, n_j) : 2K \mid (e - d - K),\ \prod_{\ell=j+1}^{i} e[n(\ell)] = 1 \right\}.$$

We call the sum (8) the *contributions* of $f_1, \ldots, f_i$ towards b_{d+K}.

For the case $K = n_{i+1}$, the coefficients of $f_{i+1} = f \bmod (z^K + 1)$ are given by (4). We compute the first n_{i+1}^* coefficients of $f \bmod (z^K \pm 1)$, $b_d \mp b_{d+K}$, in place of the stored values b_d and b_{d+K}, for $0 \le d < n_{i+1}^*$. For the remaining coefficients of f_{i+1} we compute b_{d+K} using (8) as in the first case.

If we compute (8) in an intelligent order, then the cost of Sergeev's algorithm amounts to linearly many additions and multiplications by powers of 2, plus the cost of the Discrete Weighted Transforms. Sergeev measured the cost of his approach for computing the images f_i in terms of ring additions and multiplications by powers of 2:

THEOREM 11. *Let $f \in \mathcal{R}[z]$ have degree less than n. Then one can compute $f_1, \ldots, f_s$ in place of f using $4n - 6n_1$ ring multiplications by powers of 2 and $4n - 4n_1$ ring additions.*

This linear cost is absorbed into the $\mathcal{O}(n)$ factor appearing in the cost bound (5) of the Discrete Weighted Transforms. Thus Sergeev's algorithm is asymptotically equivalent to Mateer's and van der Hoeven's TFT algorithms.

In order to compute a new coefficient of an image of f in Sergeev's algorithm, one effectively has to make a pass through the array in order to sum the contributions (8), for every coefficient computed in this manner. This is because, in order to keep things in-place without increasing the cost, one has to compute the entire sum (8) before adding it back into the array. One improvement is to instead work with the *weighted* images

$$f_j^* = 2^{-j+1} f_j$$

instead of f_j, and $2^{-i-1} f \bmod (z^{2K} - 1)$ instead of $f \bmod (z^{2K} - 1)$, where $K \in (n_{i+1}, n_i]$. If we now let $a_d^{(j)}$ and b_d be the coefficients of these weighted images, our equation (8) for b_{d+K} becomes

$$b_{d+K} = \sum_{j=1}^{i} \sum_{e \in S_{j,i}^d} a_e^{(j)}.$$

This allows us to progressively add b_{d+K} to another value in our array without having to compute b_{d+K} in entirety first. We use such weighted images in our implementation of Sergeev's algorithm, as well as in the algorithm we describe in the next section, where for that method we describe the reweighting in greater detail.

5. A NEW IN-PLACE CYCLOTOMIC TFT ALGORITHM

We present an algorithm related to Sergeev's algorithm, also in-place, with asymptotically equivalent cost. Unlike Sergeev's algorithm, we will forego producing part of the images $f \bmod (z^K - 1)$, K a power of two. We will compute f_{i+1} immediately after producing f_i. An advantage of such an approach is, we require fewer passes through our array for input sizes n of low Hamming weight, i.e., n containing few non-zero bits.

We write $f_1, \ldots, f_s$ in place of f in three steps: we first compute the remainders produced by dividing q_{i-1} by Φ_i,

$$r_i = q_{i-1} \bmod \Phi_i, \quad 1 \le i \le s,$$

in place of f; we then iteratively write f_i^* in place of r_i for $i = 1, 2, \ldots, s$; lastly we reweight f_i^* to get f_i. After we have the images f_i we again compute a DWT of each image f_i separately to give us the weighted evaluation points.

5.1 Breaking f into the remainders r_i

We first break $f = q_0$ into its quotient and remainder dividing by $z^{n_1} + 1$,

$$r_1 = f \bmod (z^{n_1} + 1) = \sum_{i=0}^{n_1 - 1} (a_i - a_{i+n_1})\, z^i,$$

$$q_1 = f \text{ quo } (z^{n_1} + 1) = \sum_{i=0}^{n_1^* - 1} a_{i+n_1} z^i,$$

where $a_i = 0$ for $i \ge n$. This can be done in place with n_1^* subtractions in $\mathcal{R}$. We then similarly break q_1 into r_2 and q_2, then q_2 into r_3 and q_3, and continue until we have $r_1, \ldots, r_{s-1}$ and q_{s-1}. Since $\deg(q_{s-1}) < n_s = \deg(\Phi_s)$, r_s is exactly q_{s-1}.

For the purposes of the inverse transform, computing f from the remainders r_i is equally uncomplicated. Given q_{i+1} and r_{i+1}, we recompute q_i in place of q_{i+1} and r_{i+1} as $q_i = q_{i+1}(z^{n_{i+1}} + 1) + r_{i+1}$.

5.2 Computing f_i^* in place of r_i

We first note that f_1^* is precisely r_1. We will iteratively produce the remaining weighted images. Suppose, at the start of the i-th iteration, we have f_j^*, and r_k, for $1 \le j \le i < k \le s$. We want to write f_{i+1}^* in place of r_{i+1}. We have

$$\begin{aligned} f_{i+1}^* &= 2^{-i} f \bmod \Phi_{i+1}, \\ &= 2^{-i} (\Gamma_i q_i + C_i) \bmod \Phi_{i+1}, \\ &= \left(q_i + 2^{-i} C_i \right) \bmod \Phi_{i+1}, \\ &= r_{i+1} + \left(2^{-i} C_i \bmod \Phi_{i+1} \right). \end{aligned} \tag{9}$$

Unfortunately, we do not have the combined image C_i, but rather the weighted images f_j^*, $1 \le j \le i$, from which we can reconstruct C_i. We would like to be able to compute the sum (9) in place from the remainder r_{i+1} and the weighted images f_j^*.

Corollary 8 tells us the contribution of f_i^* towards subsequent images. If e satisfies the non-zero criterion of the corollary, then by (9), a term $c_{j,e} z^e$ of f_j^*, $j \le i$ will contribute $\frac{1}{2}(-1)^{e[n(i+1)]} z^{(e \bmod n_{i+1})}$ to f_{i+1}^*. In order to make the contributions have weight ± 1, we instead first reweight r_i by 2 and compute $2f_i^*$, and then divide by 2 thereafter.

We note that this cost of reweighting by $2^{\pm 1}$ requires fewer multiplications than instead introducing a factor $\frac{1}{2}$ into all the contributions. AddContributions describes how we add the contributions of $f_1^*, \ldots, f_i^*$ to f_{i+1}^*.

Procedure AddContributions($\mathbf{a}$, n, i)

Input: $\mathbf{a}$, a length-n array containing $f_1^*, \ldots, f_i^*$ and $2r_{i+1}$, in that order.
Result: The contributions of $f_1^*, \ldots, f_i^*$ towards $2f_{i+1}^*$ are added to $2r_{i+1}$. As a result we will have $2f_{i+1}^*$ in place of $2r_{i+1}$.

$\mathbf{a}^{(1)} \longleftarrow \mathbf{a}$
for $j \longleftarrow 1$ **to** i **do** $\mathbf{a}^{(j+1)} \longleftarrow \mathbf{a}^{(j)} + n_j$

for $j \longleftarrow 1$ **to** i **do**
  // Add contribution of f_j^* to f_i^*
  for $e \longleftarrow 0$ **to** $n_j - 1$ **do**
    if $e[n(\ell)] = 0$ *for all* ℓ, $j < \ell \leq i$ **then**
      $\mathbf{a}^{(i+1)}_{e \bmod n_{i+1}}$ += $(-1)^{e[n(i+1)]}\mathbf{a}_e^{(j)}$

According to corollary 8, only a proportion of 2^{j-i} of the terms of f_j^* will have a non-zero contribution to f_{i+1}^*. Thus the total cost of adding contributions of f_j^* towards f_ℓ^*, for all $\ell > j$, is less than $2\#f_j^* = 2n_j$. It follows that the total additions and subtractions in $\mathcal{R}$ required to add all these contributions are bounded by $2n$. Since AddContributions only scales array ring elements by ± 1, we have the following complexity:

LEMMA 12. *Calling* AddContributions($\mathbf{a}, n, i$) *for* $1 \leq i < s$ *entails no more than* $2n$ *ring additions and no ring multiplications.*

In the manner we have chosen to add these contributions, we will have to make $s-1$ passes through our array to add them all. One way we could avoid this is to instead add all the contributions from f_1^*, and then all the contributions from f_2^*, and so forth, adding up all the contributions from a single term at once. We could use that a term $c_{j,e}z^e$ of f_j^* that does not contribute towards f_{i+1}^* will not contribute to f_k^* for any $k > i$. This would reduce the number of passes we make through the larger portion of the array, though the cache performance of potentially writing to $s-1$ images at once raises questions.

When adding contributions to f_{i+1}^*, we need only inspect the non-zero criterion of one exponent e in a block of exponents $kn_i \leq e < (k+1)n_i$. Similarly, we need only inspect one exponent in a block of n_{i+1} consecutive exponents in order to determine their shared value of $(-1)^{e[n(i+1)]}$. There may appear long segments of consecutive exponents not satisfying the non-zero criterion. To avoiding traversing such segments unnecessarily, our implementation computes the next exponent that satisfies the non-zero criterion by way of bit operations.

We note here that, for the purposes of the inverse transform, we can as easily reobtain r_i from f_i^*. We merely multiply f_i^* by 2, instead subtract the contributions to get $2r_i$, and then multiply by $\frac{1}{2}$ to get r_i.

Procedure BreakIntoImages breaks f into the images f_i, after which we can again use Discrete Weighted Transforms to compute $\mathrm{TFT}'_{n,\omega}(f)$.

Procedure BreakIntoImages($\mathbf{a}$, n)

Input: $\mathbf{a}$, a length-n array containing $f \in \mathcal{R}[z]$.
Result: The images $f_1, \ldots, f_s$ are written in place of f.

$\mathbf{a}^{(1)} \longleftarrow \mathbf{a}$
for $i \longleftarrow 1$ **to** $s-1$ **do** $\mathbf{a}^{(i+1)} \longleftarrow \mathbf{a}^{(i)} + n_i$

// 1: Write $r_1, \ldots, r_s$ in place of f
for $i \longleftarrow 1$ **to** $s-1$ **do**
  for $e \longleftarrow 0$ **to** $n_i^* - 1$ **do**
    $\mathbf{a}_e^{(i)} \longleftarrow \mathbf{a}_e^{(i)} - \mathbf{a}_{e+n_i}^{(i)}$

// 2: Write f_{i+1}^* in place of r_{i+1}
for $i \longleftarrow 1$ **to** $s-1$ **do**
  for $e \longleftarrow 0$ **to** $n_{i+1} - 1$ **do** $\mathbf{a}_e^{(i+1)} \longleftarrow 2\mathbf{a}_e^{(i+1)}$
  AddContributions($\mathbf{a}, n, i$)
  for $e \longleftarrow 0$ **to** $n_{i+1} - 1$ **do** $\mathbf{a}_e^{(i+1)} \longleftarrow \frac{1}{2}\mathbf{a}_e^{(i+1)}$

// 3: Reweight f_i^* to get f_i
for $i \longleftarrow 1$ **to** $s-1$ **do**
  for $e \longleftarrow 0$ **to** $n_i^* - 1$ **do** $\mathbf{a}_e^{(i+1)} \longleftarrow 2\mathbf{a}_e^{(i+1)}$

5.3 Cost analysis

Procedure BreakIntoImages effectively has three parts. In the first part of the algorithm, we break f into the remainders r_i. Producing r_i entails $n_i^* < n_i$ additions, and so producing all the r_i entails less than $\sum_{i=1}^{s} n_i = n$ ring additions.

In the second part we write the weighted images f_i^* in place of the r_i. Adding all the contributions, per lemma 12, requires $2n$ additions. We reweight the last $n - n_1$ coefficients of f by 2, then by $\frac{1}{2}$. This constitutes less than n multiplications in total.

In the third part we reweight the weighted images f_i^* to get the images f_i. This entails less than n multiplications by 2. This gives us the following complexity:

LEMMA 13. *Procedure* BreakIntoImages($\boldsymbol{a}, n$) *requires at most* $3n$ *ring additions and* $2n$ *ring multiplications by* $2^{\pm 1}$.

Thus, like Mateer's and Sergeev's algorithms, the TFT algorithm suggested in this section requires a linear cost to separate f into the images f_i, and again has cost asymptotically equivalent to van der Hoeven's bit-reversed TFT.

We briefly mention that it is reasonably straightforward to invert BreakIntoImages. We have described in sections 5.1 and 5.2 how to reverse the first two parts of the algorithm. Reversing the third part is trivial. Without giving a detailed analysis of the inverse transform, we remark that computing f from $f_1, \ldots, f_s$ also has a linear cost.

6. A LINK BETWEEN BIT-REVERSED & CYCLOTOMIC TFT METHODS

The bit-reversed TFT has the property that, for $m < n$, $\mathrm{TFT}_{\omega,m}(f)$ is merely the first m entries of $\mathrm{TFT}_{\omega,n}(f)$, whereas the cyclotomic TFT does not, in general, have this property. Hence the bit-reversed TFT lends itself more readily to multivariate polynomial arithmetic. We show how an

algorithm for computing the cyclotomic TFT may be modified to compute a bit-reversed TFT.

As before, let $n = \sum_{i=1}^{s} n_i = \sum_{i=1}^{s} 2^{n(i)}$ and let $N = 2^p$ be the least power of two exceeding n. Let $\omega = \omega_1$ be a root of Φ_1 and $\omega_i = \omega^{n_1/n_i}$ be a root of Φ_i. Define

$$\Omega_i = \prod_{\ell=1}^{i} \omega_\ell \qquad \text{and} \qquad \Psi_j(z) = z^{n_j} - \Omega_{j-1}^{n_j},$$

for $0 \le i \le s$ and $1 \le j \le s$. $\mathrm{TFT}_{\omega,n}(f)$ is comprised of the evaluation of $f(z)$ at the roots of the polynomials $\Psi_j(z)$. If $n_1 + \cdots + n_{j-1} \le \ell < n_1 + \cdots + n_j$, we have that $\omega^{[\ell]_p}$ is a root of Ψ_j. To see this, write $\ell = n_1 + \cdots + n_{j-1} + \ell'$, $0 \le \ell' < n_j$, and observe that

$$\begin{aligned} \omega^{[\ell]_p n_j} &= \omega^{\left(N/(2n_1)+\cdots+N/(2n_{j-1})\right)n_j} \omega^{[\ell']_p n_j}, \\ &= (\omega_1 \cdots \omega_{j-1})^{n_j} \omega^{[\ell']_p n_j}, \\ &= \Omega_{j-1}^{n_j} \omega^{[\ell']_p n_j}. \end{aligned} \tag{10}$$

Every n_j-th root of unity in $\mathcal{R}$ is of the form $\omega^{[k]_p}$, where $0 \le k < n_j$. In particular, $\omega^{[\ell']_p}$ is an n_j-th root of unity. Thus (10) is precisely $(\Omega_{j-1})^{n_j}$ and $\omega^{[\ell]_p}$ is a root of Ψ_j.

Consider the affine transformation $z \mapsto \Omega_s z$. Then

$$\begin{aligned} \Psi_i(\Omega_s z) &= \Omega_s^{n_i} z^{n_i} - \Omega_{i-1}^{n_i}, \\ &= \Omega_{i-1}^{n_i} \left[\left(\prod_{j=i}^{s} \omega_j^{n_i} \right) z^{n_i} - 1 \right], \\ &= -\Omega_{i-1}^{n_i} (z^{n_i} + 1) = -\Omega_{i-1}^{n_i} \Phi_i. \end{aligned}$$

Thus, for a polynomial $f(z)$, $f(z) \bmod \Phi_i(z) = f(z) \bmod \Psi_i(\Omega_s z)$. We can break $f(z)$ into its images modulo the polynomials Φ_i as follows:

1. Replace $f(z)$ with $f(\Omega_s z)$.
2. Break $f(\Omega_s z)$ into its images modulo $\Phi_i(z)$ per a cyclotomic TFT method. Equivalently, this gives us the images $f(\Omega_s z) \bmod \Psi_i(\Omega_s z)$.
3. Apply transformation $z \mapsto \Omega_s^{-1} z$ to every image to give us $f(z) \bmod \Psi_i(z)$ in place of $f(\Omega_s z) \bmod \Psi_i(\Omega_s z)$.

Note that the affine transformations of steps 1 and 3 both have complexity $\mathcal{O}(n)$. We can get $\mathrm{TFT}_{\omega,n}(f)$ from the images $f \bmod \Psi_i$ by applying a DWT with weight Ω_{i-1} to each image $f \bmod \Psi_i$. As each step here is invertible, we can invert a bit-reversed TFT using an inverse cyclotomic TFT algorithm. We can similarly use a bit-reversed TFT algorithm to compute a cyclotomic TFT.

7. CONCLUSION

We have presented a method of computing a cyclotomic TFT in-place, with cost, in terms of ring multiplications, asymptotically equivalent to out-of-place TFT methods. We have also shown a means of using a cyclotomic TFT algorithm to compute a bit-reversed TFT.

As future work, we would like to make fine-tuned implementations of Sergeev's TFT algorithm and the TFT method presented here, to gauge how well they perform compared to existing competitive TFT implementations.

We also would like to better understand exactly which families of Truncated Fourier Transforms are equivalent by way of transformations as in section 6.

8. ACKNOWLEDGMENTS

Thanks to: Igor Sergeev for bringing [7] and [9] to my attention, and for his assistance in helping me understand his TFT algorithm; Joris van der Hoeven for posing the question of non-invertible TFTs; Dan Roche for providing his implementation of bit-reversed TFT algorithms; and Curtis Bright, Reinhold Burger, Mustafa Elsheikh, Mark Giesbrecht, and Colton Pauderis for their comments and feedback.

The author would also like to thank the referees for their careful reading of the original draft of this paper and for their many suggestions.

9. REFERENCES

[1] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[2] R. Crandall. *Advanced Topics in Scientific Computation*. Springer-Verlag, 1996. ISBN 0-387-94473-7.

[3] R. Crandall and B. Fagin. Discrete weighted transforms and large-integer arithmetic. *Math. Comp.*, 62(205):305–324, 1994.

[4] J. V. Z. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2-nd edition, 2003.

[5] D. Harvey and D. S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ISSAC '10, pages 325–329, New York, NY, USA, 2010. ACM.

[6] M. Heideman, D. Johnson, and C. Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, 1984.

[7] T. Mateer. Fast Fourier Transform Algorithms with Applications. Master's thesis, Clemson University, 2008.

[8] D. Roche. *Efficient Computation With Sparse and Dense Polynomials*. PhD thesis, University of Waterloo, 2011.

[9] I. S. Sergeev. Regular estimates for the complexity of polynomial multiplication and truncated Fourier transform. *Prikl. Diskr. Mat.*, pages 72–88, 2011. (Russian) `http://mi.mathnet.ru/eng/pdm347`.

[10] J. van der Hoeven. The Truncated Fourier Transform and Applications. In J. Gutierrez, editor, *Proc. ISSAC 2004*, pages 290–296, Univ. of Cantabria, Santander, Spain, July 4–7 2004.

[11] J. van der Hoeven. Notes on the Truncated Fourier Transform. Technical Report 2005-5, Université Paris-Sud, Orsay, France, 2005.