An in-place truncated Fourier transform and applications to polynomial multiplication

David Harvey
Courant Institute of Mathematical Sciences
New York University
New York, New York, U.S.A.
dmh Harvey@cims.nyu.edu
www.cims.nyu.edu/~harvey/

Daniel S. Roche
Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
droche@cs.uwaterloo.ca
www.cs.uwaterloo.ca/~droche/

ABSTRACT
The truncated Fourier transform (TFT) was introduced by van der Hoeven in 2004 as a means of smoothing the “jumps” in running time of the ordinary FFT algorithm that occur at power-of-two input sizes. However, the TFT still introduces these jumps in memory usage. We describe in-place variants of the forward and inverse TFT algorithms, achieving time complexity \(O(n \log n)\) with only \(O(1)\) auxiliary space.

As an application, we extend the second author’s results on space-restricted FFT-based polynomial multiplication to polynomials of arbitrary degree.

Categories and Subject Descriptors
F.2.1 [Analysis of Algorithms and Problem Complexity]: Numerical Algorithms and Problems—Computations on polynomials; G.4 [Mathematical Software]: Algorithm design and analysis, Efficiency; I.1.2 [Symbolic and Algebraic Manipulation]: Algorithms—Algebraic algorithms, Analysis of algorithms

General Terms
Algorithms, Performance, Theory

Keywords
Truncated Fourier transform, fast Fourier transform, polynomial multiplication, in-place algorithms

1. INTRODUCTION

1.1 Background
The discrete Fourier transform (DFT) is a linear map that evaluates a given polynomial at powers of a root of unity. Cooley and Tukey [3] were the first to develop an efficient method to compute this transform on a digital computer, known as the Fast Fourier Transform (FFT). This algorithm has since become one of the most important and useful tools in computer science, especially in the area of signal processing.

The FFT algorithm is also important in computer algebra, most notably in asymptotically fast methods for integer and polynomial multiplication. The first integer multiplication algorithm to run in softly linear time relies on the FFT [10], as do the recent theoretical improvement [4] and the best result for polynomial multiplication over arbitrary algebras [2]. Moreover, numerous other operations on polynomials — including division, evaluation/interpolation, and GCD computation — have been reduced to multiplication, so more efficient multiplication methods have an indirect effect on many areas in computer algebra [5, §8–11].

The simplest FFT to implement, and the fastest in practice, is the radix-2 Cooley-Tukey FFT. Because the radix-2 FFT requires the size to be a power of two, the simplest solution for all other sizes is to pad the input polynomials with zeros, resulting in large unwanted “jumps” in the complexity at powers of two.

1.2 The truncated Fourier transform
It has been known for some time that if only a subset of the DFT output is needed, then the FFT can be truncated or “pruned” to reduce the complexity, essentially by disregarding those parts of the computation tree not contributing to the desired outputs [8, 11]. More recently, van der Hoeven took the crucial step of showing how to invert this process, describing a truncated Fourier transform (TFT) and an inverse truncated Fourier transform (ITFT), and showing that this leads to a polynomial multiplication algorithm whose running time varies relatively smoothly in the input size [12, 13].

Specifically, given an input vector of length \(n \leq 2^k\), the TFT computes the first \(n\) coefficients of the ordinary Fourier transform of length \(2^k\), and the ITFT computes the inverse of this map. The running time of these algorithms smoothly interpolates the \(O(n \log n)\) complexity of the standard radix-2 Cooley-Tukey FFT algorithm. One can therefore deduce an asymptotically fast polynomial multiplication algorithm that avoids the characteristic “jumps” in running time exhibited by traditional FFT-based polynomial multiplication algorithms when the output degree crosses a power-of-two boundary. This observation has been confirmed with practical implementations [13, 7, 6], with the most marked improvements in the multivariate case.

One drawback of van der Hoeven’s algorithms is that while
their time complexity varies smoothly with \( n \), their space complexity does not. Both the TFT and ITFT operate in a buffer of length \( 2^{\lceil \lg n \rceil} \); that is, for inputs of length \( n \), they require auxiliary storage of \( 2^{\lceil \lg n \rceil} - n + O(1) \) cells to store intermediate results, which can be \( \Omega(n) \) in the worst case.

### 1.3 Summary of results

The main results of this paper are TFT and ITFT algorithms that require only \( O(1) \) auxiliary space, while respecting the \( O(n \log n) \) time bound.

The new algorithms have their origin in a cache-friendly variant of the TFT and ITFT given by the first author [6], that builds on Bailey’s cache-friendly adaptation of the ordinary FFT [1]. If the transform takes place in a buffer of length \( L = 2^k \), these algorithms decompose the transform into \( L_1 = 2^\ell_1 \) row transforms of length \( L_2 = 2^\ell_2 \) and \( L_2 \) column transforms of length \( L_1 \), where \( \ell_1 + \ell_2 = \ell \). Van der Hoeven’s algorithms correspond to the case \( L_1 = 2 \) and \( L_2 = L/2 \). To achieve optimal locality, [6] suggests taking \( L_1 \approx \sqrt{L} \) (\( \ell \approx \ell/2 \)). In fact, in this case one already obtains TFT and ITFT algorithms needing only \( O(\sqrt{n}) \) auxiliary space. At the other extreme we may take \( L_1 = L/2 \) and \( L_2 = 2 \), obtaining TFT and ITFT algorithms that use only \( O(1) \) space at each recursion level, or \( O(\log n) \) auxiliary space altogether. In signal processing language, these may be regarded as decimation-in-time variants of van der Hoeven’s decimation-in-frequency algorithms.

Due to data dependencies in the \( O(\log n) \)-space algorithms sketched above, the space usage cannot be reduced further by simply reordering the arithmetic operations. In this paper, we show that with a little extra work, increasing the implied constant in the \( O(\log n) \) auxiliary space to only \( O(1) \) is possible in-place and in amortized constant time. Our in-place truncated Fourier transform algorithms (Algorithms 1 and 2) fall under this model.

An out-of-place algorithm uses separate memory locations for input and output. Here, any element from the input array may be read from in constant time (but not overwritten), and any element in the output array may be read from or written to in constant time as well. This will be the situation in our multiplication algorithm (Algorithm 3).

The algorithms also need to store some number of pointers and ring elements not in the input or output arrays, which we define to be the auxiliary storage used by the algorithm. All the algorithms we present will use only \( O(1) \) auxiliary storage space.

This model should correspond well with practice, at least when the computations are performed in main memory and the ring \( R \) is finite.

### 2.2 DFT notation

We denote by \( \omega_{[k]} \) a primitive \( 2^k \)-th root unity, and we assume that these are chosen compatibly, so that \( \omega_{[k+1]} = \omega_{[k]} \) for all \( k \geq 0 \). Define a sequence of roots \( \omega_0, \omega_1, \ldots, \omega_s = \omega_{[k]}^{-rev_k s} \), where \( k \geq \lceil \lg(s + 1) \rceil \) and \( rev_k s \) denotes the length-\( k \) bit-reversal of \( s \). Thus we have

\[
\begin{align*}
\omega_0 &= \omega_{[0]} = (1) & \omega_2 &= \omega_{[2]} & \omega_4 &= \omega_{[4]} & \omega_8 &= \omega_{[8]} \\
\omega_1 &= \omega_{[1]} = (1) & \omega_3 &= \omega_{[2]} & \omega_5 &= \omega_{[4]} & \omega_7 &= \omega_{[8]} \\
\omega_2 &= \omega_{[3]} = (1) & \omega_3 &= \omega_{[2]} & \omega_5 &= \omega_{[4]} & \omega_7 &= \omega_{[8]}
\end{align*}
\]

and so on. Note that

\[
\omega_{2s+1} = -\omega_2s \text{ and } \omega_{2s}^2 = \omega_{2s+1} = \omega_s.
\]

If \( F \in R[x] \) is a polynomial with \( \deg F < n \), we write \( F_s \) for the coefficient of \( x^s \) in \( F \), and we define the Fourier transform \( \hat{F} \) by

\[
\hat{F}_s = F(\omega_s).
\]

In Algorithms 1 and 2 below, we decompose \( F \) as

\[
F(x) = G(x^2) + xH(x^2),
\]

where \( \deg G < \lceil n/2 \rceil \) and \( \deg H < \lceil n/2 \rceil \). Using the properties of \( \omega_s \) mentioned above, we obtain the “butterfly” relations

\[
\begin{align*}
\hat{F}_{2s} &= \hat{G}_s + \omega_{2s} \hat{H}_s, \\
\hat{F}_{2s+1} &= \hat{G}_s - \omega_{2s} \hat{H}_s.
\end{align*}
\]

Both the TFT and ITFT algorithm require, at each recursive level, iterating through a set of index-root pairs such as \( \{(i, \omega_i), 0 \leq i < n\} \). A traditional, time-efficient approach would be to precompute all powers of \( \omega_{[k]} \), store them in reverted-binary order, and then pass through this array with a single pointer. However, this is impossible under the restriction that no auxiliary storage space be used. Instead, we will compute the roots on-the-fly by iterating through the powers of \( \omega_{[k]} \) in order, and through the indices \( i \) in bit-reversed order. Observe that incrementing an integer counter through \( rev_k 0, rev_k 1, rev_k 2, \ldots \) can be done in exactly the same way as incrementing through \( 0, 1, 2, \ldots \), which is possible in-place and in amortized constant time.

### 2. PRELIMINARIES

#### 2.1 Computational model

We work over a ring \( R \) containing \( 2^k \)-th roots of unity for all (suitably large) \( k \), and in which 2 is not a zero-divisor.

Our memory model is similar to that used in the study of in-place algorithms for sorting and geometric problems, combined with the well-studied notion of algebraic complexity. Specifically, we allow two primitive types in memory: ring elements and pointers. A ring element is any single element of \( R \), and the input to any algorithm will consist of \( n \) such elements stored in an array. A pointer can hold a single integer \( a \in \mathbb{Z} \) in the range \(-cn \leq a \leq cn \) for some fixed constant \( c \in \mathbb{N} \). (In our algorithms, we could take \( c = 2 \).)

We say an algorithm is in-place if it overwrites its input buffer with the output. In this case, any element in this single input/output array may be read from or written to in constant time. Our in-place truncated Fourier transform algorithms (Algorithms 1 and 2) fall under this model.

An out-of-place algorithm uses separate memory locations for input and output. Here, any element from the input array may be read from in constant time (but not overwritten), and any element in the output array may be read from or written to in constant time as well. This will be the situation in our multiplication algorithm (Algorithm 3).
3. SPACE-RESTRICTED TFT

In this section we describe an in-place TFT algorithm that uses only $O(1)$ auxiliary space (Algorithm 1). The routine operates on a buffer $X_0, \ldots, X_{n-1}$ containing elements of $R$. It takes as input a root of unity of sufficiently high order and the coefficients $F_0, \ldots, F_{n-1}$ of a polynomial $F \in R[x]$, and overwrites these with $\hat{F}_0, \ldots, \hat{F}_{n-1}$.

The pattern of the algorithm is recursive, but we avoid recursion by explicitly moving through the recursion tree, avoiding unnecessary space usage. An example tree for $n = 6$ is shown in Figure 1. The node $S = (q, r)$ represents the subarray with offset $q$ and stride $2^r$; the $i$th element in this subarray is $S_i = x_{q+i2^r}$, and the length of the subarray is given by

$$\text{len}(S) = \left\lceil \frac{n - q}{2^r} \right\rceil.$$ 

The root is $(0, 0)$, corresponding to the entire input array of length $n$. Each subarray of length 1 corresponds to a leaf node, and we define the predicate $\text{IsLeaf}(S)$ to be true iff $\text{len}(S) = 1$. Each non-leaf node splits into even and odd child nodes. To facilitate the path through the tree, we define

\[
\begin{align*}
\text{Even}(q, r) &= (q, r + 1), \\
\text{Odd}(q, r) &= (q + 2^r, r + 1)
\end{align*}
\]

if $(q, r)$ is not a leaf,

$$\text{Parent}(q, r) = \begin{cases} (q, r - 1), & q < 2^r - 1, \\
(q - 2^r - 1, r - 1), & q \geq 2^r - 1
\end{cases}.$$ 

if $(q, r)$ is not the root, and for any node we define

$$\text{LeftmostLeaf}(S) = \begin{cases} S, & \text{IsLeaf}(S), \\
\text{LeftmostLeaf} (\text{Even}(S)), & \text{otherwise}.
\end{cases}$$

We begin with the following lemma.

**Lemma 3.1.** Let $N$ be a node with $\text{len}(N) = \ell$, and let

$$A(x) = \sum_{0 \leq i < \ell} A_i x^i \in R[x].$$

If $S = \text{LeftmostLeaf}(N)$ and $N_i = A_i$ for $0 \leq i < \ell$ before some iteration of line 8 in Algorithm 1 then after a finite number of steps, we will have $S = N$ and $N_i = \hat{A}_i$ for $0 \leq i < \ell$, before the execution of line 8. No other array entries in $X$ are affected.

---

**Algorithm 1:** InplaceTFT([X0, \ldots, X_{n-1}])

**Input:** $X_i = F_i$ for $0 \leq i < n$, where $F \in R[x]$, $\deg F < n$

**Output:** $X_i = \hat{F}_i$ for $0 \leq i < n$

1. $S \leftarrow \text{LeftmostLeaf}(0, 0)$
2. $\text{prev} \leftarrow \text{null}$
3. while true do
   4. $m \leftarrow \text{len}(S)$
   5. if $\text{IsLeaf}(S)$ or $\text{prev} = \text{Odd}(S)$ then
      6. for $(i, \theta) \in \{(j, \omega_2) : 0 \leq j < [m/2]\}$ do
         7. $S_{2i} \leftarrow S_{2i} + \theta S_{2i+1}$
         8. if $S = (0, 0)$ then halt
         9. $\text{prev} \leftarrow S$
     10. $S \leftarrow \text{Parent}(S)$
   else if $\text{prev} = \text{Even}(S)$ then
      11. if $\text{len}(S) \equiv 1 \mod 2$ then
         12. $v \leftarrow \sum_{i=0}^{(m-3)/2} S_{2i+1} \cdot (\omega^{(m-1)/2})^i$
         13. $S_{m-1} \leftarrow S_{m-1} + v \cdot \omega^{(m-1)/2}$
         14. $\text{prev} \leftarrow S$
     15. $S \leftarrow \text{LeftmostLeaf}(\text{Odd}(S))$

**Proof.** The proof is by induction on $\ell$. If $\ell = 1$, then $\text{IsLeaf}(N)$ is true and $A_0 = A_0$ so we are done. So assume $\ell > 1$ and that the lemma holds for all shorter lengths.

Decompose $A$ as $A(x) = G(x^2) + x H(x^2)$. Since $S = \text{LeftmostLeaf}(\text{Even}(N))$ as well, the induction hypothesis guarantees that the even-indexed elements of $N$, corresponding to the coefficients of $G$, will be transformed into $G$, and we will have $S = \text{Even}(N)$ before line 8. The following lines set $\text{prev} = \text{Even}(N)$ and $S = N$, so that lines 12-15 are executed on the next iteration.

If $\ell$ is odd, then $(\ell - 1)/2 \geq \text{len}(\text{Odd}(N))$, so $\hat{H}_{(\ell-1)/2}$ will not be computed in the odd subtree, and we will not be able to apply (1) to compute $\hat{A}_{\ell-1} = \hat{G}_{(\ell-1)/2} + \omega^{(\ell-1)/2} \hat{H}_{(\ell-1)/2}$. This is why, in this case, we explicitly compute

$$v = H(\omega^{(\ell-1)/2}) = \hat{H}_{(\ell-1)/2}$$
on line 13 and then compute $\hat{A}_{\ell-1}$ directly on line 14 before descending into the odd subtree.

Another application of the induction hypothesis guaran-
tees that we will return to line [8] with $S = \text{Odd}(N)$ after computing $N_{2i+1} = \hat{H}_i$ for $0 \leq i < \lceil \ell / 2 \rceil$. The following lines set $\text{prev} = \text{Odd}(N)$ and $S = N$, and we arrive at line [9] on the next iteration. The loop thus properly applies the butterfly relations [1] to compute $\hat{A}_i$ for $0 \leq i < 2\lceil \ell / 2 \rceil$, which completes the proof. \hfill \square

Now we are ready for the main result of this section.

**Proposition 3.2.** Algorithm [2] correctly computes $\hat{F}_i$ for $0 \leq i < n$. It performs $O(n \log n)$ ring and pointer operations, and uses $O(1)$ auxiliary space.

**Proof.** The correctness follows immediately from Lemma [3.1] since we start with $S = \text{LeftmostLeaf}(0, 0)$, which is the first leaf of the whole tree. The space bound is immediate since each variable has constant size.

To verify the time bound, notice that the while loop visits each leaf node once and each non-leaf node twice (once with $\text{prev} = \text{Even}(S)$ and once with $\text{prev} = \text{Odd}(S)$). Since always $q < 2^r < 2n$, there are $O(n)$ iterations through the while loop, each of which has cost $O(\text{len}(S) + \log n)$. This gives the total cost of $O(n \log n)$. \hfill \square

## 4. SPACE-RESTRICTED ITFT

Next we describe an in-place inverse TFT algorithm that uses $O(1)$ auxiliary space (Algorithm [2]). It takes as input $F_0, \ldots, F_{n-1}$ for some polynomial $F \in \mathcal{R}[x]$, deg $F < n$, and overwrites the buffer with $F_0, \ldots, F_{n-1}$.

The path of the algorithm is exactly the reverse of Algorithm [1] and we use the same notation as before to move through the tree. We only require one additional function:

- $\text{RightmostParent}(S) = \begin{cases} S, & \text{if } \text{LeftmostLeaf}(\text{Odd}(N_i)) = N_2, \\ \text{RightmostParent}(\text{Parent}(S)), & \text{otherwise.} \end{cases}$

If $\text{LeftmostLeaf}(\text{Odd}(N_i)) = N_2$, then $\text{Parent}(\text{RightmostParent}(N_2)) = N_1$, so $\text{RightmostParent}$ computes the inverse of the assignment on line [16] in Algorithm [1].

We leave it to the reader to confirm that the structure of the recursion is identical to that of Algorithm [1] but in reverse, from which the following analogues of Lemma [3.1] and Proposition [3.2] follow immediately:

**Lemma 4.1.** Let $N$ be a node with $\text{len}(N) = \ell$, and $A(x) = \sum_{0 \leq i < \ell} A_i x^i \in \mathcal{R}[x]$.

If $S = N$ and $N_i = \hat{A}_i$, for $0 \leq i < \ell$ before some iteration of line [2] in Algorithm [2], then after a finite number of steps, we will have $S = \text{LeftmostLeaf}(N)$ and $N_i = \hat{A}_i$, for $0 \leq i < \ell$ before some iteration of line [2] in Algorithm [2]. No other array entries in $X$ are affected.

**Proposition 4.2.** Algorithm [2] correctly computes $F_i$ for $0 \leq i < n$. It performs $O(n \log n)$ ring and pointer operations, and uses $O(1)$ auxiliary space.

The fact that our InplaceTFT and InplaceITFT algorithms are essentially reverses of each other is an interesting feature not shared by the original formulations in [12].

## 5. POLYNOMIAL MULTIPLICATION

We now describe the multiplication algorithm alluded to in the introduction. The strategy is similar to that of [9], with a slightly more complicated “folding” step. The input consists of two polynomials $A, B \in \mathcal{R}[x]$ with deg $A < n$ and deg $B < m$. The routine is supplied an output buffer $X$ of length $r = n + m - 1$ in which to write the product $C = AB$.

The subroutine FFT has the same interface as InplaceTFT, but is only called for power-of-two length inputs.

**Algorithm 3: Space-restricted product**

**Input:** $A, B \in \mathcal{R}[x]$, deg $A < n$, deg $B < m$.

**Output:** $X = C_s$ for $0 \leq s < n + m - 1$, where $C = AB$.

1. $r \leftarrow n + m - 1$
2. $q \leftarrow 0$
3. while $q < r - 1$ do
4. $\ell \leftarrow \lfloor \log(r - q) \rfloor - 1$
5. $L \leftarrow 2^\ell$
6. $[X_q, X_{q+1}, \ldots, X_{q+L-1}] \leftarrow [0, 0, \ldots, 0]$
7. for $0 \leq i < m$ do
8. $[X_q+i \text{ (mod } L) \leftarrow X_q+i \text{ (mod } L) + \omega^i A_i$
9. FFT$([X_q, X_{q+1}, \ldots, X_{q+L-1}])$
10. for $0 \leq i < n$ do
11. $[X_q+L+i \text{ (mod } L) \leftarrow X_q+L+i \text{ (mod } L) + \omega^i B_i$
12. FFT$([X_q+L, X_{q+L+1}, \ldots, X_{q+L+L-1}])$
13. for $0 \leq i < L$ do
14. $X_{q+i} \leftarrow X_{q+i} X_{q+L+i}$
15. $q \leftarrow q + L$
16. $X_{r-1} \leftarrow A(\omega_{r-1}) B(\omega_{r-1})$
17. InplaceITFT$([X_0, \ldots, X_{r-1}])$

**Proposition 5.1.** Algorithm [2] correctly computes the product $C = AB$, in time $O((m+n) \log(m+n))$ and using $O(1)$ space.
auxiliary space.

Proof. The main loop terminates since \( q \) is strictly increasing. Let \( N \) be the number of iterations, and let \( q_0 > q_1 > \cdots > q_{N-1} \) and \( L_0 \geq L_1 \geq \cdots \geq L_{N-1} \) be the values of \( q \) and \( L \) on each iteration. By construction, the intervals \( (q_0, q_0 + L_0) \) form a partition of \([0, r-1]\), and \( L_i \) is the largest power of two such that \( q_i + 2L_i \leq r \). Therefore each \( L \) can appear at most twice (i.e. if \( L_i = L_i+1 \) then \( L_i+1 < L_i \)). \( N \leq 2Lr \), and we have \( L_i \leq q_i \) for each \( i \).

At each iteration, lines \( 9\) compute the coefficients of the polynomial \( A(q_i \omega_l) \) in \( X_{q_i} \ldots X_{q_i+L_i-1} \). Line \( 9 \) then computes \( X_{q_i+L_i} = A(q_i \omega_l) \) for \( 0 \leq i < L \). Since \( L \mid q \) we have \( \omega_{q_i} \omega_l = \omega_{q_i+L_i} \), and so we have actually computed \( X_{q_i+L_i} = A(q_i \omega_l) \) for \( 0 \leq i < L \). The next two lines similarly compute \( X_{q_i+L_i+1} = B_{q_i+L_i} \) for \( 0 \leq i < L \). (The point of the condition \( q+2L \leq r \) is to ensure that both of these transforms fit into the output buffer.) Lines \( 13 \) then compute \( X_{q_i+L_i+1} = A_{q_i+L_i}B_{q_i+L_i} = C_{q_i+L_i} \) for \( 0 \leq i < L \).

After line \( 16 \) we finally have \( X_s = \hat{C}_i \) for all \( 0 \leq i < r \). (The last product was handled separately since the output buffer does not have room for the two Fourier coefficients.) Line \( 17 \) then recovers \( C_0, \ldots, C_{r-1} \).

We now analyze the time and space complexity. The loops on lines \( 8 \), \( 9 \), \( 10 \) and \( 13 \) contribute \( O(r) \) operations per iteration, or \( O(r \log r) \) in total, since \( N = O(\log r) \). The FFT calls contribute \( O(L_i \log L_i) \) per iteration, for a total of \( O(\sum L_i \log L_i) = O(\sum L_i \log L) = O(r \log r) \). Line \( 16 \) contributes \( O(r \log r) \), and line \( 17 \) contributes \( O(r \log r) \) by Proposition \( 1.2 \). The space requirements are immediate also by Proposition \( 1.2 \) since the main loop requires only \( O(1) \) space. \( \square \)

6. CONCLUSION

We have demonstrated that forward and inverse radix-2 truncated Fourier transforms can be computed in-place using \( O(n \log n) \) time and \( O(1) \) auxiliary storage. As a result, polynomials with degrees less than \( n \) can be multiplied out-of-place within the same time and space bounds. These results apply to any size \( n \), whenever the underlying ring admits division by 2 and a primitive root of unity of order \( 2^{\lceil \log_2 n \rceil}, \)

Numerous questions remain open in this direction. First, our in-place TFT and ITFT algorithms avoid using auxiliary space at the cost of some extra arithmetic. So although the asymptotic complexity is still \( O(n \log n) \), the implied constant will be greater than for the usual TFT or FFT algorithms. It would be interesting to know whether this extra cost is unavoidable. In any case, the implied constant would need to be reduced as much as possible for the in-place TFT/ITFT to compete with the running time of the original algorithms.

We also have not yet demonstrated an in-place multi-dimensional TFT or ITFT algorithm. In one dimension, the ordinary TFT can hope to gain at most a factor of two over the FFT, but a \( d \)-dimensional TFT can be faster than the corresponding FFT by a factor of \( 2^d \), as demonstrated in [7]. An in-place variant along the lines of the algorithms presented in this paper could save a factor of \( 2^d \) in both time and memory, with practical consequences for multivariate polynomial arithmetic.

Finally, noticing that our multiplication algorithm, despite using only \( O(1) \) auxiliary storage, is still an out-of-place algorithm, we restate an open question of [9]: Is it possible, under any time restrictions, to perform multiplication in-place and using only \( O(1) \) auxiliary storage? The answer seems to be no, but a proof is as yet elusive.

7. REFERENCES

[1] David H. Bailey. FFTs in external or hierarchical memory. Journal of Supercomputing, 4:23–35, 1990.
[2] David G. Cantor and Erich Kaltofen. On fast multiplication of polynomials over arbitrary algebras. Acta Inform., 28(7):693–701, 1991.
[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. Math. Comp., 19:297–301, 1965.
[4] Martin Führer. Faster integer multiplication. In STOC ’07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing, pages 57–66, New York, NY, USA, 2007. ACM Press.
[5] Joachim von zur Gathen and Jürgen Gerhard. Modern computer algebra. Cambridge University Press, Cambridge, second edition, 2003.
[6] David Harvey. A cache-friendly truncated FFT. Theoret. Comput. Sci., 410(27-29):2649–2658, 2009.
[7] Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: from theory to practice. J. Symbolic Comput., 44(7):891–907, 2009.
[8] J. Markel. FFT pruning. Audio and Electroacoustics, IEEE Transactions on, 19(4):305–311, Dec 1971.
[9] Daniel S. Roche. Space- and time-efficient polynomial multiplication. In ISSAC ’09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation, pages 295–302, New York, NY, USA, 2009. ACM.
[10] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. Computing (Arch. Elektron. Rechnen), 7:281–292, 1971.
[11] H.V. Sorensen and C.S. Burrus. Efficient computation of the DFT with only a subset of input or output points. Signal Processing, IEEE Transactions on, 41(3):1184–1200, Mar 1993.
[12] Joris van der Hoeven. The truncated Fourier transform and applications. In ISSAC 2004, pages 290–296. ACM, New York, 2004.
[13] Joris van der Hoeven. Notes on the truncated Fourier transform. unpublished, available from http://www.math.u-psud.fr/~vdhoeven/, 2005.