

The great generality and simplicity of its rules makes arithmetic accessible to the dullest mind.

—Tobias Dantzig, *Numbers* (1930)

## Chapter 3

# In-place Truncated Fourier Transform

The fast Fourier transform (FFT) algorithm is a crucial operation in many areas of signal processing as well as computer science. Most relevant to our focus, all the asymptotically fastest methods for polynomial and long integer multiplication rely heavily on the FFT as a subroutine. Most commonly, particularly for multiplication algorithms, the radix-2 Cooley-Tukey FFT is used. In the typical case that the transform size is not exactly a power of two, the so-called truncated Fourier transform (TFT) can be used to avoid wasted computation. However, in using the TFT, a crucial property of the FFT is lost: namely, that it can be computed completely in-place, with the output overwriting the input. Here we present a new algorithm for the truncated Fourier transform which regains this property of the FFT and can be computed completely in-place and with the same asymptotic time cost as the original method.

The main results in this chapter represent joint work with David Harvey of New York University. We presented some of these results at ISSAC 2010 (Harvey and Roche, 2010). We also thank the helpful reviewer at that conference who pointed out the Devil’s convolution algorithm of Crandall (1996).

### 3.1 Background

#### 3.1.1 Primitive roots of unity and the discrete Fourier transform

Recall that an  $n$ th root of unity  $\omega \in \mathbb{R}$  is any element such that  $\omega^n = 1$ ; furthermore,  $\omega$  is said to be an  $n$ th *primitive* root of unity, or  $n$ -PRU, if  $n$  is the least integer such that  $\omega^n = 1$ . The only roots of unity in the real numbers  $\mathbb{R}$  are  $-1$  and  $1$ , whereas the complex numbers  $\mathbb{C}$  contain an  $n$ -PRU for any  $n$ , for instance,  $e^{2\pi i/n}$ . Finite fields are more interesting: the finite field with  $q$  elements  $\mathbb{F}_q$  contains an  $n$ -PRU if and only if  $n \mid (q - 1)$ . If we have a generator  $\alpha$  for the multiplicative group of the field, the ring element  $\alpha^{(q-1)/n}$  gives one  $n$ -PRU.

The cyclic nature of PRUs is what makes them useful for computation. Mathematically, a  $n$ -PRU  $\omega \in \mathbb{R}$  generates an order- $n$  subgroup of  $\mathbb{R}^*$ . In particular,  $\omega^k = \omega^{k \bmod n}$  for any integer exponent  $k$ .

The discrete Fourier transform (DFT) is a linear map that evaluates a given polynomial at powers of a root of unity. Given a polynomial  $f \in \mathbb{R}[x]$  and an  $n$ -PRU  $\omega \in \mathbb{R}$ , the discrete Fourier transform of  $f$  at  $\omega$ , written  $\text{DFT}_\omega(f)$ , is defined as

$$\text{DFT}_\omega(f) = \left( f(\omega^i) \right)_{0 \leq i < n}.$$

That is, the DFT at  $\omega$  of  $f$  is the polynomial  $f$  evaluated at every power of  $\omega$ . By writing the polynomial  $f = a_0 + a_1x + a_2x^2 + \dots$  in the dense representation as a list  $(a_0, a_1, a_2, \dots) \in \mathbb{R}^*$  of its coefficients, the function  $\text{DFT}_\omega$  is a map from  $\mathbb{R}^*$  to  $\mathbb{R}^n$ .

If we restrict to polynomials  $f$  with degree strictly less than  $n$ , then such polynomials can always be stored as an array of exactly  $n$  coefficients, and  $\text{DFT}_\omega$  is a map from  $\mathbb{R}^n$  to  $\mathbb{R}^n$ . In these cases the  $\text{DFT}_\omega$  map is also invertible, from the uniqueness of polynomial interpolation, using the fact that the values  $\omega^i$  for  $0 \leq i < n$  are all distinct.

In fact, the inverse DFT is simply a DFT at  $\omega^{-1}$ , scaled by  $n$ . To be precise,

$$\text{DFT}_{\omega^{-1}}(\text{DFT}_\omega(a_0, a_1, \dots, a_{n-1})) = (na_0, na_1, \dots, na_{n-1}).$$

This tight relationship between the forward and inverse DFT is crucial, especially for the application to multiplication.

### 3.1.2 Fast Fourier transform

Naïvely, evaluating  $\text{DFT}_\omega(f)$  requires evaluating  $f$  at  $n$  distinct points, for a total cost of  $O(n^2)$  ring operations. The fast Fourier transform (FFT) is a much more efficient algorithm to compute  $\text{DFT}_\omega(f)$ ; for certain values of  $n$ , it improves the cost to  $O(n \log n)$ .

The FFT algorithm has been hugely useful in computer science, and in particular in the area of signal processing. For instance, it was named one of the top ten algorithms of the twentieth century (Dongarra and Sullivan, 2000), where the authors cite it as “the most ubiquitous algorithm in use today to analyze and manipulate digital or discrete data”. The FFT has an interesting history: it was known to Gauss, but was later rediscovered in various forms by a number of authors in the twentieth century (Heideman, Johnson, and Burrus, 1984). The most famous and significant rediscovery was by Cooley and Tukey (1965), who were also the first to describe how to implement the algorithm in a digital computer.

We now briefly describe Cooley and Tukey’s algorithm. If  $n$  can be factored as  $n = \ell m$ , then the FFT is a divide-and-conquer strategy that computes a size- $n$  DFT by  $m$  size- $\ell$  DFTs, followed by  $\ell$  size- $m$  DFTs. Given  $f = a_0 + a_1x + \dots + a_{n-1}x^{n-1}$ , we define the polynomials  $g_i \in \mathbb{R}[x]$ , for  $0 \leq i < m$ , by

$$g_i = \sum_{0 \leq j < \ell} a_{i+mj} x^j = a_i + a_{i+m}x + a_{i+2m}x^2 + \dots + a_{i+(\ell-1)m}x^{\ell-1}.$$

Observe that every coefficient in  $f$  appears in exactly one  $g_i$ , and each  $\deg g_j < \ell$ . In fact, we can write  $f$  as

$$f = \sum_{0 \leq i < m} x^i g_i(x^m) = g_0(x^m) + x g_1(x^m) + \dots + x^{m-1} g_{m-1}(x^m).$$

Now let  $i, j \in \mathbb{N}$  such that  $0 \leq i < \ell$  and  $0 \leq j < m$ , and consider the  $(i + \ell j)$ th element of  $\text{DFT}_\omega(f)$ , which is  $f(\omega^{i+\ell j})$ . From above, we have

$$f(\omega^{i+\ell j}) = \sum_{0 \leq k < m} \omega^{(i+\ell j)k} g_k(\omega^{(i+\ell j)m}) = \sum_{0 \leq k < m} (\omega^\ell)^{jk} \omega^{ik} g_k((\omega^m)^i), \quad (3.1)$$

with the second equality following from the fact that  $\ell m = n$  and  $\omega^n = 1$ . Now observe that  $\omega^m$  is an  $\ell$ -PRU in  $\mathbb{R}$ , so each  $g_k(\omega^{im})$  is simply the  $i$ th element of  $\text{DFT}_{\omega^m}(g_k)$ . Once these DFTs are known, for each  $i \in \{0, 1, \dots, \ell - 1\}$ , we can compute the polynomials

$$h_i = \sum_{0 \leq k < m} \omega^{ik} g_k((\omega^m)^i) x^k.$$

Each  $h_i$  has degree less than  $m$ , and the coefficients of each  $h_i$  are computed by multiplying a power of  $\omega$  with an element of  $\text{DFT}_{\omega^m}(g_k)$  for some  $k$ . (The powers of  $\omega$  multiplied here, which have the form  $\omega^{ik}$ , are called “twiddle factors”.)

From (3.1), we can then write  $f(\omega^{i+\ell j}) = h_i((\omega^\ell)^j)$ . Since  $\omega^\ell$  is an  $m$ -PRU, these values are determined by a size- $m$  DFT of  $h_i$ . Specifically,  $(i + \ell j)$ th element of  $\text{DFT}_\omega(f)$  is equal to the  $j$ th element of  $\text{DFT}_{\omega^\ell}(h_i)$ .

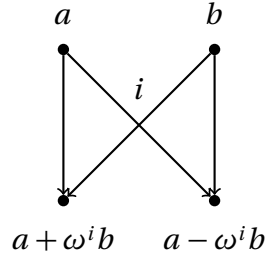
This completes the description of the FFT algorithm. In summary, we first write down the  $g_i$ s (which requires no computation) and compute  $m$  size- $\ell$  DFTs,  $\text{DFT}_{\omega^m}(g_i)$ . These values are then multiplied by the twiddle factors to compute the coefficients of every  $h_i$ . This step only uses  $O(n)$  ring operations, since we can first compute all powers of  $\omega$  and then perform all the multiplications with elements of  $\text{DFT}_{\omega^m}(g_i)$ . Finally, we compute  $\ell$  size- $m$  DFTs,  $\text{DFT}_{\omega^\ell}(h_i)$ , which give the elements in  $\text{DFT}_\omega(f)$ .

### 3.1.3 FFT variants

The radix-2 Cooley-Tukey FFT algorithm, which we will examine closely in this chapter, works when the size  $n$  of the DFT to be computed is a power of 2. This allows the FFT algorithm to be applied recursively. When  $\ell = n/2$  and  $m = 2$ , the resulting algorithm is called the *decimation-in-time* variant. Since  $\ell$  is also a power of 2, the 2 initial size- $\ell$  DFTs (on the  $g_i$ s) can be computed with recursive calls. The base case is when the size  $n = 2$ , in which case the operation is computed directly with constant cost. This recursive algorithm has a total cost of  $O(n \log n)$  ring operations.

The so-called decimation-in-frequency variant of the radix-2 FFT is just the opposite, setting  $\ell = 2$  and  $m = n/2$ . (Both of these terms come from the application of FFT to signal processing.) Halfway between these two versions would be choosing  $\ell$  to be the greatest power of two less than  $\sqrt{n}$ , and  $m$  the least power of two greater than  $\sqrt{n}$ . This type of FFT was carefully studied by Bailey (1990) and Harvey (2009) and shown to improve the I/O complexity of FFT computation. However, all these variants still have time cost  $O(n \log n)$ .

The idea of the radix-2 FFT can of course be generalised to radix-3 FFT or, more generally, radix- $k$  FFT for any  $k \geq 2$ . This will require that the input size  $n$  is a power of  $k$ , and the cost



**Figure 3.1:** Butterfly circuit for decimation-in-time FFT

of the radix- $k$  algorithm will be  $O(kn \log_k n)$  — the same cost as the radix-2 version if  $k$  is constant.

Even more generally, any factorization of  $n$  can be used to generate an FFT algorithm. This is the idea of the mixed-radix FFT algorithm, whose cost depends on the largest factor of  $n$  used in the algorithm, but is at least  $O(n \log n)$ .

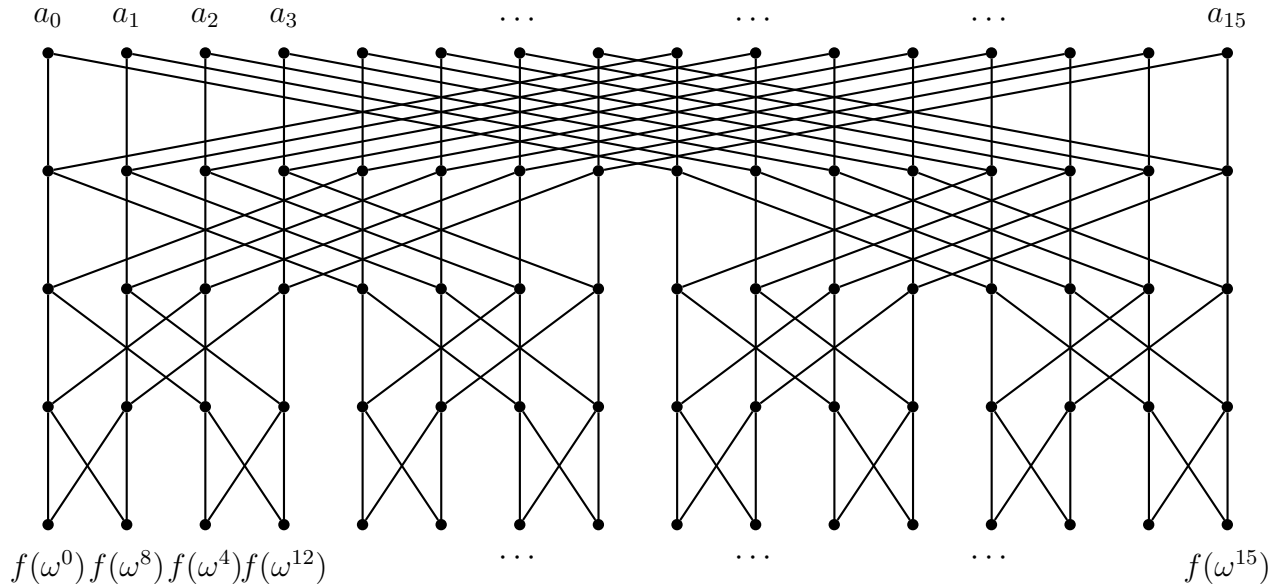
Though these methods are quite popular in signal processing algorithms, the radix-2 version (and sometimes radix 3) are used almost exclusively in symbolic computation. This is partly due to the fact that, at least for the application to integer multiplication, FFTs are most conveniently and commonly performed over fixed, chosen finite fields. Finding a prime  $p$  such that  $p - 1$  is divisible by a sufficiently high power of 2 is not difficult, and this then allows a radix-2 FFT of any smaller size to be performed in the field  $\mathbb{F}_p$  of integers modulo  $p$ . This is a fundamentally different situation than working over the complex numbers, where no special construction or pre-computation is needed to find PRUs.

### 3.1.4 Computing the FFT in place

An important property of the FFT algorithm is that it can be computed in-place. Recall in the IMM model that this means the input space is empty, and the output space is initialised with the data to be transformed. An in-place algorithm is one which uses only a constant amount of temporary storage to solve this sort of problem. Any formulation of the Cooley-Tukey FFT algorithm described above can be computed in-place, as long as the base case of the recursion has constant size. In particular, the radix-2 and radix-3 versions can be computed in-place.

Consider the radix-2 decimation-in-time FFT. Since the algorithm is recursive and the base case has size 2, the entire circuit for this computation is composed of smaller circuits for size-2 DFTs, with varying twiddle factors (powers of  $\omega$ ). Such a circuit is called a “butterfly circuit”, and is shown in Figure 3.1. The circuit shown takes ring elements  $a, b$  as input and computes  $a + \omega^i b$  and  $a - \omega^i b$ , where  $\omega^i$  is the twiddle factor for this butterfly.

Figure 3.2 shows a circuit for the entire computation of a size-16 FFT (radix-2, decimation-in-time variant), with input polynomial  $f = a_0 + a_1x + \dots + a_{15}x^{15}$ . Though not shown, each of the butterflies making up this circuit has an associated twiddle factor as well. Because the size-2 butterfly can clearly be computed using at most one extra unit of storage, the entire computation can be as well. At each of the  $\log_2 n$  steps through the algorithm, the elements



**Figure 3.2:** Circuit for radix-2 decimation-in-time FFT of size 16

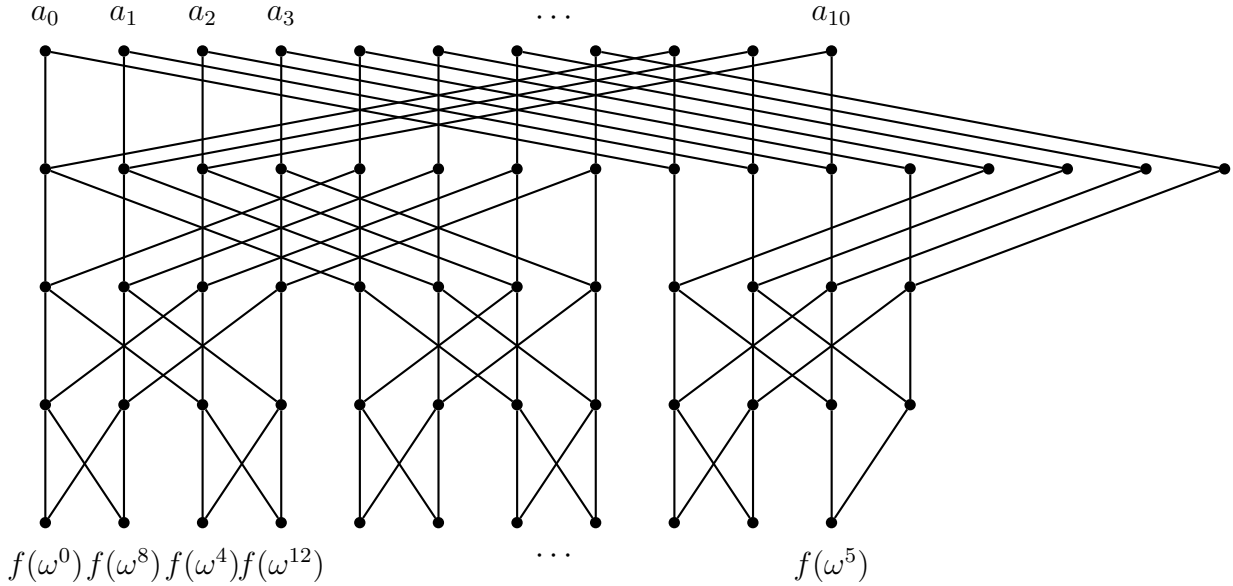
stored in the size- $n$  output space correspond to the nodes at one level of the circuit. Each step uses  $O(n)$  time and  $O(1)$  space to overwrite this array with the nodes at the next level of the circuit, until we reach the bottom and are done.

The only complication with this in-place computation is that the order of the elements is perturbed. In particular, if the  $i$ th position in the array holds the  $a_i$  at the beginning of the computation, the computed value in that position at the end of the algorithm will *not* be  $f(\omega^i)$  but rather  $f(\omega^{\text{rev}_k i})$ . Here  $k = \log_2 n$  and  $\text{rev}_k i$  is the  $k$ -bit reversal of the binary representation of  $i$ . For instance,  $\text{rev}_5(11) = 26$ , since 11 is written 01011 in base 2 and 26 is 11010. This permutation can easily be inverted in in-place to give the output in the correct order. However, for our application, this will actually not be necessary.

### 3.1.5 The truncated Fourier transform

As mentioned above, symbolic computation algorithms typically use radix-2 FFTs. However, the size of the input, for instance in the application to polynomial multiplication, will rarely be exactly a power of two. In this case, the traditional approach has been to pad the input with zeros so that the size is a power of two. However, this wastes not only memory space but also time in performing unnecessary computations.

Numerous approaches have been developed to address this problem and adapt radix-2 FFTs to arbitrary sized inputs and outputs. The “devil’s convolution” algorithm of [Crandall \(1996\)](#) tackles this issue in the higher-level context of multiplication by breaking the problem into a large power-of-two size, solved with radix-2 FFTs, and a smaller arbitrary size, solved recursively. However, there are still significant jumps in the time and space cost (though not exactly at powers of two).



**Figure 3.3:** Circuit for forward TFT of size 11

At the lower-level context of DFTs, it has been known for some time that if only a subset of the 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 (Markel, 1971; Sorensen and Burrus, 1993). More recently, van der Hoeven took the crucial step of showing how to invert this process by assuming a subset of input/output coefficients are zero, 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 (van der Hoeven, 2004, 2005).

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 (van der Hoeven, 2005; Li, Maza, and Schost, 2009; Harvey, 2009), with the most marked improvements in the multivariate case.

The structure of the TFT is shown in Figure 3.3, for an input of size 11. The structure is essentially the same as the size-16 decimation-in-time FFT, except that unnecessary computations — either those coming from inputs known to be zero, or those going to outputs that are not needed — are “pruned” away. Observe that, unlike with the FFT, inverting the TFT is completely non-trivial, and requires a more complicated algorithm, as shown by van der Hoeven (2004). We will not discuss the details of the inverse TFT algorithm presented there, but observe that it has roughly the same time cost as the forward TFT.

One drawback of van der Hoeven's algorithms is that while their time complexity varies smoothly with  $n$ , their space complexity does not. We can observe from the circuit for the size-11 TFT that 16 ring elements must be stored on the second step. In general, both the TFT and ITFT require  $2^{\lceil \log_2 n \rceil} - n$  units of extra storage space, which in the worst case is  $\Omega(n)$ . Therefore the TFT and ITFT are not in-place algorithms.

### 3.1.6 DFT notation

For the discussion of our new algorithms below, it will be convenient to have a simpler notation for power-of-two roots of unity and bit reversals. First, denote by  $\omega_{[k]}$  a primitive  $2^k$ -th root of unity, for some integer  $k$ . Our algorithm will require that such a root of unity exists for all  $k \leq \lceil \log_2 n \rceil$ .

We assume that these PRUs are chosen compatibly, so that  $\omega_{[k+1]}^2 = \omega_{[k]}$  for all  $k \geq 0$ . Define a sequence of roots  $\omega_0, \omega_1, \dots$  by  $\omega_s = \omega_{[k]}^{\text{rev}_k s}$ , where  $k \geq \lceil \lg(s+1) \rceil$  and  $\text{rev}_k s$  denotes the length- $k$  bit-reversal of  $s$ . This definition is consistent because the PRUs were chosen compatibly. This is because, for any  $i \in \mathbb{N}$ ,  $\text{rev}_{k+i} s = 2^i \cdot \text{rev}_k s$ , and therefore

$$\omega_{[k+i]}^{\text{rev}_{k+i} s} = \omega_{[k+i]}^{2^i \text{rev}_k s} = \omega_{[k]}^{\text{rev}_k s}.$$

Thus we have

$$\begin{array}{llll} \omega_0 = \omega_{[0]} (= 1) & \omega_2 = \omega_{[2]} & \omega_4 = \omega_{[3]} & \omega_6 = \omega_{[3]}^3 \\ \omega_1 = \omega_{[1]} (= -1) & \omega_3 = \omega_{[2]}^3 & \omega_5 = \omega_{[3]}^5 & \omega_7 = \omega_{[3]}^7 \end{array}$$

and so on. Note that

$$\omega_{2s+1} = -\omega_{2s} \quad \text{and} \quad \omega_{2s}^2 = \omega_{2s+1}^2 = \omega_s.$$

Let  $f \in \mathbb{R}[x]$  be a polynomial with  $\deg f < n$  as before. The notation just introduced has the convenient property that the  $i$ th element of  $\text{DFT}_\omega(f)$ , for *any* power-of-two PRU  $\omega$ , when written in the bit-reversed order as computed by the in-place FFT, is simply  $f(\omega_i)$ . If we again write  $a_i$  for the coefficient of  $x^i$  in  $f$ , then we will also write  $\hat{a}_i = f(\omega_i)$  for the  $i$ th term in the DFT of  $f$ .

Algorithms 3.1 and 3.2 below follow the outline of the decimation-in-time FFT, and decompose  $f$  as

$$f(x) = g(x^2) + x \cdot h(x^2),$$

where  $\deg g < \lfloor n/2 \rfloor$  and  $\deg h < \lfloor n/2 \rfloor$ . Write  $b_0, b_1, \dots, b_{\lfloor n/2 \rfloor - 1}$  for the coefficients of  $g$  and  $c_0, c_1, \dots, c_{\lfloor n/2 \rfloor}$  for the coefficients of  $h$ . Using the notation just introduced, the ‘‘butterfly relations’’ corresponding to the bottom level of the FFT circuit can be written

$$\begin{aligned} \hat{a}_{2s} &= \hat{b}_s + \omega_{2s} \hat{c}_s, \\ \hat{a}_{2s+1} &= \hat{b}_s - \omega_{2s} \hat{c}_s. \end{aligned} \tag{3.2}$$

## 3.2 Computing powers of $\omega$

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 pre-compute 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.

One solution, if using the ordinary FFT, is to use a decimation-in-frequency algorithm rather than decimation-in-time. In this version, at the  $k$ th level, the twiddle factors are accessed in normal order  $\omega_{[k]}^0, \omega_{[k]}^1, \dots$ . Hence this approach can be used to implement the radix-2 FFT algorithm in-place.

However, our algorithm critically requires the decimation-in-time formulation, and thus accessing the twiddle factors in reverted binary order  $\omega_0, \omega_1, \omega_2, \dots$ . To avoid this, 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. That is, we will access the array elements in reverted binary order, but the powers of  $\omega$  in the normal order.

Accessing array elements in reverted binary order simply requires incrementing an integer counter through  $\text{rev}_k 0, \text{rev}_k 1, \text{rev}_k 2, \dots$ . Of course such increments will no longer be single word operations, but fortunately the reverted binary increment operation can be performed in-place and in amortized constant time. The algorithm is simply the same as the usual bitwise binary increment operation, but performed in reverse.

## 3.3 Space-restricted TFT

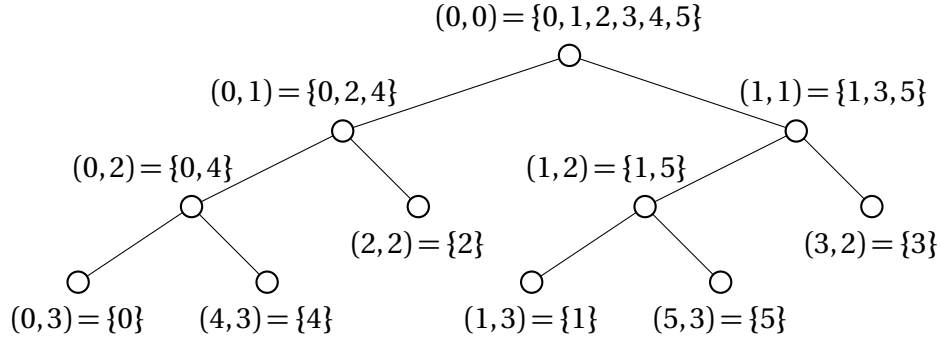
In this section we describe an in-place TFT algorithm that uses only  $O(1)$  temporary space (Algorithm 3.1). First we define the problem formally for the algebraic IMM model. There is no input or output in integers, so we denote an instance by the triple  $(I, O, O') \in (\mathbb{R}^*)^3$  containing ring elements on the algebraic side. This is an *in-output* problem, so every valid instance has  $I$  empty,  $O = (a_0, a_1, \dots, a_{n-1})$ , and  $O' = (\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1})$ , representing the computation of a size- $n$  DFT of the polynomial  $f \in \mathbb{R}[x]$  with coefficients  $a_0, \dots, a_{n-1}$ . To be completely precise, we should also specify that  $O$  contains a  $2^{\lceil \log_2 n \rceil}$ -PRU  $\omega_{\lceil \log_2 n \rceil}$ .

For the algorithm, we will use  $M_O[i]$  to denote the  $i$ th element of the (algebraic) output space, at any point in the computation. The in-place algorithm we will present will use a constant number of extra memory locations  $M_T[i]$ , but we will not explicitly refer to them in this way.

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 3.4. The node  $S = (q, r)$  represents a sub-array of the output space with offset  $q$  and stride  $2^r$ ; the  $i$ th element in this sub-array is  $S_i = M_O[q + i \cdot 2^r]$ , and the length of the sub-array is given by

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





**Figure 3.4:** Recursion tree for in-place TFFT of size  $n = 6$

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

To facilitate the path through the tree, we define a few functions on the nodes. First, for any non-leaf node  $(q, r)$ , define

$$\begin{aligned} \text{EvenChild}(q, r) &= (q, r + 1), \\ \text{OddChild}(q, r) &= (q + 2^r, r + 1). \end{aligned}$$

If  $(q, r)$  is not the root node, define

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

Finally, for any node  $S$  we define

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

So for example, in the recursion tree shown in Figure 3.4, let the node  $S = (1, 1)$ . Then  $\text{OddChild}(S) = (3, 2)$ , the right child in the tree,  $\text{Parent}(S) = (0, 0)$ , and  $\text{LeftmostLeaf}(S) = (1, 3)$ . We also see that  $\text{len}(S) = 3$  and  $S_i = M_O[1 + 2i]$ .

Algorithm 3.1 gives the details of our in-place algorithm to compute the TFFT, using these functions to move around in the recursion tree.

The structure of the algorithm on our running example of size 11 is shown in the circuit in Figure 3.5. The unfilled nodes in the diagram represent temporary elements that are computed on-the-fly and immediately discarded in the algorithm, and the dashed edges indicate the extra computations necessary for these temporary computations. Observe that the structure of the circuit is more tangled than before, and in particular it is not possible to perform the computation level-by-level as one can for the normal FFT and forward TFFT.

We begin with the following lemma.

---

**Algorithm 3.1:** In-place truncated Fourier transform
 

---

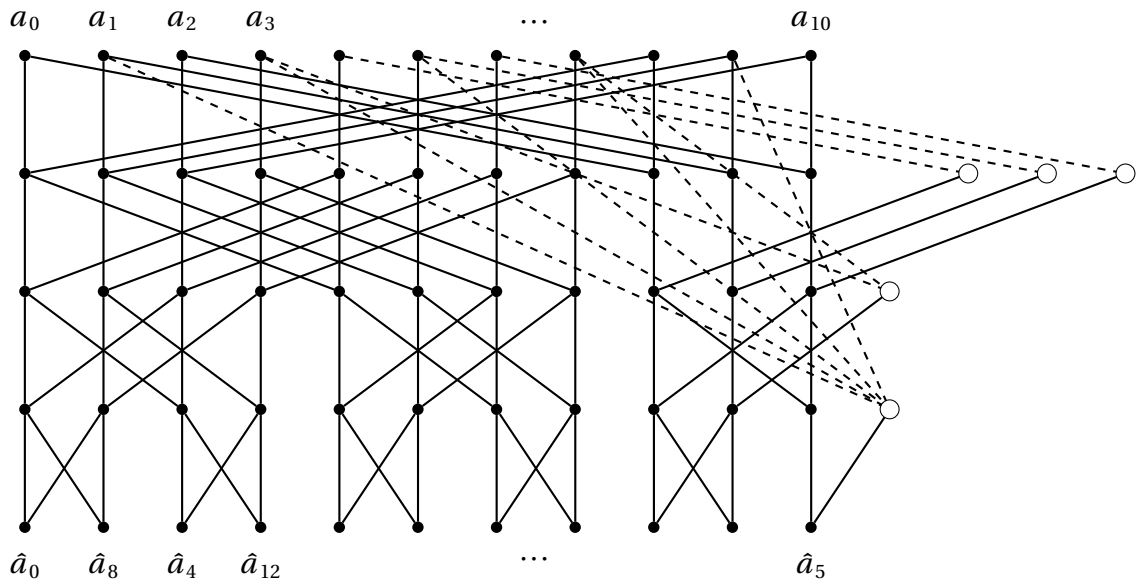
**Input:** Coefficients  $a_0, a_1, \dots, a_{n-1} \in \mathbb{R}$  and PRU  $\omega \in \mathbb{R}$ , stored in-order in the output space  $M_O$

**Output:**  $\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1}$ , stored in-order in the output space  $M_O$

```

1  $S \leftarrow \text{LeftmostLeaf}(0, 0)$ 
2  $prev \leftarrow \text{null}$ 
3 while true do
4    $m \leftarrow \text{len}(S)$ 
5   if IsLeaf(S) or prev = OddChild(S) then
6     for  $(i, \theta) \in \{(j, \omega_{2j}) : 0 \leq j < \lfloor m/2 \rfloor\}$  do
7        $\begin{bmatrix} S_{2i} \\ S_{2i+1} \end{bmatrix} \leftarrow \begin{bmatrix} S_{2i} + \theta S_{2i+1} \\ S_{2i} - \theta S_{2i+1} \end{bmatrix}$ 
8     if  $S = (0, 0)$  then halt
9      $prev \leftarrow S$ 
10     $S \leftarrow \text{Parent}(S)$ 
11  else if prev = EvenChild(S) then
12    if  $\text{len}(S) \equiv 1 \pmod 2$  then
13       $v \leftarrow \sum_{i=0}^{(m-3)/2} S_{2i+1} \cdot (\omega_{(m-1)/2})^i$ 
14       $S_{m-1} \leftarrow S_{m-1} + v \cdot \omega_{m-1}$ 
15     $prev \leftarrow S$ 
16     $S \leftarrow \text{LeftmostLeaf}(\text{OddChild}(S))$ 
    
```

---



**Figure 3.5:** Circuit for Algorithm 3.1 with size  $n = 11$

**Lemma 3.1.** *Let  $N$  be a node with  $\text{len}(N) = \ell$ , and for  $0 \leq i < \ell$ , define  $\alpha_i$  to be the value stored in position  $N_i$  before some iteration of line 3 in Algorithm 3.1. If also  $S = \text{LeftmostLeaf}(N)$  at this point, then after a finite number of steps, we will have  $S = N$  and  $N_i = \hat{\alpha}_i$  for  $0 \leq i < \ell$ , before the execution of line 8. No other array entries in  $M_O$  are affected.*

*Proof.* The proof is by induction on  $\ell$ . If  $\ell = 1$ , then  $\text{IsLeaf}(N)$  is true and  $\hat{\alpha}_0 = \alpha_0$  so we are done. Therefore assume  $\ell > 1$  and that the lemma holds for all shorter lengths.

Consider the ring elements  $\alpha_i$  as coefficients, and decompose the resulting polynomial into even and odd parts as

$$\alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_{\ell-1} x^{\ell-1} = g(x^2) + x \cdot h(x^2).$$

So if we write  $g = \sum_{0 \leq i < \lfloor \ell/2 \rfloor} b_i x^i$  and  $h = \sum_{0 \leq i < \lfloor \ell/2 \rfloor} c_i x^i$  as before, then each  $b_i = \alpha_{2i}$  and each  $c_i = \alpha_{2i+1}$ .

Since  $S = \text{LeftmostLeaf}(N)$  and  $N$  is not a leaf,  $S = \text{LeftmostLeaf}(\text{EvenChild}(N))$  as well, and the induction hypothesis guarantees that the even-indexed elements of  $N$ , corresponding to the coefficients of  $g$ , will be transformed into

$$\text{DFT}(g) = (\hat{b}_0, \hat{b}_1, \dots, \hat{b}_{\lfloor \ell/2 \rfloor}),$$

and we will have  $S = \text{EvenChild}(N)$  before line 8. The following lines set  $prev = \text{EvenChild}(N)$  and  $S = N$ , so that lines 12–16 are executed on the next iteration.

If  $\ell$  is odd, then  $(\ell - 1)/2 \geq \text{len}(\text{OddChild}(N))$ , so  $\hat{c}_{(\ell-1)/2}$  will not be computed in the odd subtree, and we will not be able to apply (3.2) to compute  $\hat{\alpha}_{\ell-1} = \hat{b}_{(\ell-1)/2} + \omega_{\ell-1} \hat{c}_{(\ell-1)/2}$ . This is why, in this case, we explicitly compute

$$v = h(\omega_{(\ell-1)/2}) = \hat{c}_{(\ell-1)/2}$$

on line 13, and then compute  $\hat{\alpha}_{\ell-1}$  directly on line 14, before descending into the odd subtree.

Another application of the induction hypothesis guarantees that we will return to line 8 with  $S = \text{OddChild}(N)$  after computing  $N_{2i+1} = \hat{c}_i$  for  $0 \leq i < \lfloor \ell/2 \rfloor$ . The following lines set  $prev = \text{OddChild}(N)$  and  $S = N$ , and we arrive at line 6 on the next iteration. The **for** loop thus properly applies the butterfly relations (3.2) to compute  $\hat{\alpha}_i$  for  $0 \leq i < 2\lfloor \ell/2 \rfloor$ , which completes the proof.  $\square$

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

**Theorem 3.2.** *Algorithm 3.1 correctly computes  $\text{DFT}(f)$ . It is an in-place algorithm and uses  $O(n \log n)$  algebraic and word operations.*

*Proof.* The correctness follows immediately from Lemma 3.1, since the algorithm begins by setting  $S = \text{LeftmostLeaf}(0,0)$ , 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  $prev = \text{EvenChild}(S)$  and once with  $prev = \text{OddChild}(S)$ ). 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  $O(n \log n)$ .  $\square$

### 3.4 Space-restricted ITFT

Next we describe an in-place inverse TFT algorithm (Algorithm 3.2). For some unknown polynomial  $f = a_0 + a_1x + \dots + a_{n-1}x^{n-1} \in R[x]$ , the algorithm takes as input  $\hat{a}_0, \dots, \hat{a}_{n-1}$  and overwrites these values with  $a_0, \dots, a_{n-1}$ .

The path of the algorithm is exactly the reverse of Algorithm 3.1, and we use the same notation as before to move through the tree. We only require one additional function:

$$\text{RightmostParent}(S) = \begin{cases} S, & \text{if } S = \text{OddChild}(\text{Parent}(S)), \\ \text{RightmostParent}(\text{Parent}(S)), & \text{otherwise.} \end{cases}$$

If  $\text{LeftmostLeaf}(\text{OddChild}(N_1)) = N_2$ , then  $\text{Parent}(\text{RightmostParent}(N_2)) = N_1$ . Therefore  $\text{RightmostParent}$  computes the inverse of the assignment on line 16 in Algorithm 3.1.

---

#### Algorithm 3.2: In-place inverse truncated Fourier transform

---

**Input:** Transformed coefficients  $\hat{a}_0, \hat{a}_1, \dots, \hat{a}_{n-1} \in R$  and PRU  $\omega \in R$ , stored in-order in the output space  $M_O$

**Output:** The original coefficients  $a_0, a_1, \dots, a_{n-1}$ , stored in-order in the output space  $M_O$

```

1  S ← (0, 0)
2  while S ≠ LeftmostLeaf(0, 0) do
3      if IsLeaf(S) then
4          S ← Parent(RightmostParent(S))
5          m ← len(S)
6          if len(S) ≡ 1 mod 2 then
7              v ← ∑i=0(m-3)/2 S2i+1 · ωi(m-1)/2
8              Sm-1 ← Sm-1 - v · ωm-1
9          S ← EvenChild(S)
10     else
11         m ← len(S)
12         for (i, θ) ∈ {(j, ω2j-1) : 0 ≤ j < ⌊m/2⌋} do
13             [ [ S2i ] ] ← [ (S2i + S2i+1)/2 ]
14             [ [ S2i+1 ] ] ← [ θ · (S2i - S2i+1)/2 ]
15         S ← OddChild(S)

```

---

We leave it to the reader to confirm that the structure of the recursion in Algorithm 3.2 is identical to that of Algorithm 3.1, but in reverse. From this, the following analogues of Lemma 3.1 and Theorem 3.2 follow immediately:

**Lemma 3.3.** *Let  $N$  be a node with  $\text{len}(N) = \ell$ , and  $a_0, a_1, \dots, a_{n-1} \in R$  be the coefficients of a polynomial in  $R[x]$ . If  $S = N$  and  $N_i = \hat{a}_i$  for  $0 \leq i < \ell$  before some iteration of line 2 in*

*Algorithm 3.2, then after a finite number of steps, we will have  $S = \text{LeftmostLeaf}(N)$  and  $N_i = a_i$  for  $0 \leq i < \ell$  before some iteration of line 2. No other array entries in  $M_O$  are affected.*

**Theorem 3.4.** *Algorithm 3.2 correctly computes the inverse TFT. It is an in-place algorithm that uses  $O(n \log n)$  algebraic and word operations.*

The fact that our in-place forward and inverse truncate Fourier transform algorithms are essentially reverses of each other is an interesting feature not shared by the original formulations in (van der Hoeven, 2004).

### 3.5 More detailed cost analysis

Our in-place TFT algorithm has the same asymptotic time complexity of previous approaches, but there will be a slight trade-off of extra computational cost against the savings in space. That is, our algorithms will perform more ring and integer operations than that of van der Hoeven (2004). Theorems 3.2 and 3.4 guarantee that this difference will only be a constant factor. By determining exactly what this constant factor is, we hope to gain insight into the potential practical utility of our in-place methods.

First we determine an upper bound on the number of multiplications in  $\mathbb{R}$  performed by Algorithm 3.1, for an input of length  $n$ . In all the FFT variants, multiplications in the underlying ring dominate the cost, since the number of additions in  $\mathbb{R}$  is at most twice the number of multiplications (and multiplications are more than twice as costly as additions), and the cost of integer (pointer) arithmetic is negligible.

**Theorem 3.5.** *Algorithm 3.1 uses at most*

$$\frac{5}{6}n \lceil \log_2 n \rceil + \frac{n-1}{3} \quad (3.3)$$

*multiplications in  $\mathbb{R}$  to compute a size- $n$  TFT.*

*Proof.* When  $n > 1$ , the number of multiplications performed depends on whether  $n$  is even or odd. If  $n$  is even, then the algorithm consists essentially of two recursive calls of size  $n/2$ , plus the cost at the top level (i.e., the root node in the recursion tree). Since  $n$  is even, the only multiplications performed at the top level are in lines 6–7, exactly  $n/2$  multiplications.

For the second case, if  $n$  is odd, the number of multiplications are those used by two recursive calls of size  $(n+1)/2$  and  $(n-1)/2$ , plus the number of multiplications performed at the top level. At the top level, there are again  $\lfloor n/2 \rfloor$  multiplications from lines 6–7, but since  $n$  is odd, we must also count the  $(n+1)/2$  multiplications performed on lines 13–14.

This leads to the following recurrence for  $T(n)$ , the number of multiplications performed by Algorithm 3.1 to compute a transform of length  $n$ :

$$T(n) = \begin{cases} 0, & n \leq 1 \\ 2T\left(\frac{n}{2}\right) + \frac{n}{2}, & n \geq 2 \text{ is even} \\ T\left(\frac{n-1}{2}\right) + T\left(\frac{n+1}{2}\right) + n, & n \geq 3 \text{ is odd} \end{cases} \quad (3.4)$$

When  $n$  is even, we claim an even tighter bound than (3.3):

$$T(n) \leq (5n/6) \lceil \log_2 n \rceil - 1/3, \quad n \text{ even.} \quad (3.5)$$

We now prove both bounds by induction on  $n$ .

For the base case, if  $n = 1$ , then the bound in (3.3) equals zero, and  $T(1) = 0$  also.

For the inductive case, assume  $n > 1$  and that (3.3) and (3.5) hold for all smaller values. We have two cases to consider: whether  $n$  is even or odd.

When  $n$  is even, from (3.4), the total number of multiplications is at most  $2T(n/2) + n/2$ . From the induction hypothesis, this is bounded above by

$$2 \left( \frac{5n}{12} \left\lceil \log_2 \frac{n}{2} \right\rceil + \frac{n-2}{6} \right) + \frac{n}{2} = \frac{5n}{6} \lceil \log_2 n \rceil - \frac{2}{3} < \frac{5n}{6} \lceil \log_2 n \rceil - \frac{1}{3},$$

as claimed.

When  $n$  is odd, (3.4) tells us that exactly  $T((n-1)/2) + T((n+1)/2) + n$  multiplications are performed. Now observe that at least one of the recursive-call sizes  $(n+1)/2$  and  $(n-1)/2$  must be even. Employing the induction hypothesis again, the total number of multiplications for odd  $n$  is bounded above by

$$\begin{aligned} & \frac{5(n-1)}{12} \left\lceil \log_2 \frac{n-1}{2} \right\rceil - \frac{1}{3} + \frac{5(n+1)}{12} \left\lceil \log_2 \frac{n+1}{2} \right\rceil + \frac{n-1}{6} + n \\ &= \frac{5n}{6} (\lceil \log_2 n \rceil - 1) + \frac{7n-3}{6} \\ &< \frac{5n}{6} \lceil \log_2 n \rceil + \frac{n-1}{3}. \end{aligned}$$

Therefore, by induction, the theorem holds for all  $n \geq 1$ .

□

Next we show that this upper bound is the best possible (up to lower order terms), by giving an exact analysis of the worst case.

**Theorem 3.6.** *Define the infinite sequence*

$$N_k = \frac{2^k - (-1)^k}{3}, \quad k = 1, 2, 3, \dots$$

*Then for all  $k \geq 1$ , we have*

$$T(N_k) = \frac{5}{6} k N_k - \frac{23}{18} N_k + \frac{(-1)^k}{18} k - \frac{(-1)^k}{2}. \quad (3.6)$$

*Proof.* First observe that, since  $4^i \equiv 1 \pmod{3}$  for any  $i \geq 0$ ,  $N_k$  is always an integer, and furthermore  $N_k$  is always odd. Additionally, for  $k \geq 3$ , we have  $\lceil \log_2 N_k \rceil = k - 1$ .

We now prove the theorem by induction on  $k$ . The two base cases are when  $k = 1$  and  $k = 2$ , for which  $T(N_1) = T(N_2) = T(1) = 0$ , and the formula is easily verified in these cases.

Now let  $k \geq 3$  and assume (3.6) holds for all smaller values of  $k$ . Because  $N_k$  is odd and greater than 1, from the recurrence in (3.4), we have  $T(N_k) = T((N_k - 1)/2) + T((N_k + 1)/2) + n$ . Rewriting  $-1$  and  $1$  as  $(-1)^k$  and  $-(-1)^k$  (*not necessarily respectively*), we have

$$\begin{aligned} T(N_k) &= T\left(\frac{2^k + 2(-1)^k}{6}\right) + T\left(\frac{2^k - 4(-1)^k}{6}\right) + \frac{2^k - (-1)^k}{3} \\ &= T\left(\frac{2^{k-1} - (-1)^{k-1}}{3}\right) + T\left(\frac{2^{k-1} - 2(-1)^{k-2}}{3}\right) + \frac{2^k - (-1)^k}{3} \\ &= T(N_{k-1}) + 2T(N_{k-2}) + \frac{5 \cdot 2^{k-2} - 2(-1)^k}{3}. \end{aligned}$$

Now observe that  $k - 1$  and  $k - 2$  are both at least 1, so we can use the induction hypothesis to write

$$\begin{aligned} T(N_k) &= \frac{5}{6}(k-1) \frac{N_k + (-1)^k}{2} - \frac{23}{18} \cdot \frac{N_k + (-1)^k}{2} - \frac{(-1)^k}{18}(k-1) + \frac{(-1)^k}{2} \\ &\quad + \frac{5}{3}(k-2) \frac{N_k - (-1)^k}{4} - \frac{23}{9} \cdot \frac{N_k - (-1)^k}{4} + \frac{(-1)^k}{9}(k-2) - (-1)^k \\ &\quad + \frac{5}{4}N_k - \frac{(-1)^k}{4} \\ &= \frac{5}{6}kN_k - \frac{23}{18}N_k + \frac{(-1)^k}{18}k - \frac{(-1)^k}{2}. \end{aligned}$$

Therefore, by induction, (3.6) holds for all  $k \geq 1$ . □

Let us compare this result to existing algorithms. First, the usual FFT algorithm, with padding to the next power of 2, uses exactly  $2^{\lceil \log_2 n \rceil - 1} \lceil \log_2 n \rceil$  multiplications for a size- $n$  transform, which in the worst case is  $n \lceil \log_2 n \rceil$ . The original TFT algorithm improves this to  $(n/2) \lceil \log_2 n \rceil + O(n)$  ring multiplications for any size- $n$  transform.

In conclusion, we see that our in-place TFT algorithm gains some improvement over the normal FFT for sizes that are not a power of two, but not as much improvement as the original TFT algorithm.

## 3.6 Implementation

Currently implemented in the MVP library are our new algorithms for the forward and inverse truncated Fourier transform, as well as our own highly optimized implementations of the normal radix-2 FFT algorithm both forward and reverse. We ran a number of tests on these algorithms, alternating between sizes that were exactly a power of two, one more than a power

Size	Forward FFT (seconds)	Inverse FFT (ratio)	Algorithm 3.1 (ratio)	Algorithm 3.2 (ratio)
1024	$7.50 \times 10^{-5}$ s	1.17	1.62	2.36
1025	$1.58 \times 10^{-4}$ s	1.23	.949	1.23
1536	$1.69 \times 10^{-4}$ s	1.09	1.21	1.70
2048	$1.62 \times 10^{-4}$ s	1.17	1.60	2.83
2049	$3.43 \times 10^{-4}$ s	1.13	.930	1.24
3072	$3.57 \times 10^{-4}$ s	1.12	1.21	1.77
4096	$3.61 \times 10^{-4}$ s	1.14	1.52	2.34
4097	$7.66 \times 10^{-4}$ s	1.10	.877	1.17
6144	$7.74 \times 10^{-4}$ s	1.12	1.23	1.78

**Table 3.1:** Benchmarking results for discrete Fourier transforms

of two, and halfway between two powers of two. Of course we expect the normal radix-2 algorithm to perform best when the size of the transform is exactly a power of two. It would be easy to handle sizes of the form  $2^k + 1$  specially, but we intentionally did not do this in order to see the maximum potential benefit of our new algorithms.

The results of our experiments are summarized in Table 3.1. The times for the forward FFT are reported in CPU seconds for a single operation, although as usual for the actual experiments there were several thousand iterations. The times for the other algorithms are reported only as ratios to the forward FFT time, and as usual a ratio less than one means that algorithm was faster than the ordinary forward FFT. Our new algorithms are competitive, but clearly not superior to the standard radix-2 algorithm, at least in the current implementation. While it might be possible to tweak this further, we submit a few justifications for why better run-time might not be feasible for these algorithms on modern systems.

First, observe that the structure of our in-place TFT algorithms is not level-by-level, as with the standard FFT and the *forward* TFT algorithm of [van der Hoeven \(2004\)](#). Both of our algorithms share more in common structurally with the original *inverse* TFT, which has an inherent recursive structure, similar to what we have seen here. In both these cases, the more complicated structure of the computation means in particular that memory will be accessed in a less regular way, and the consequence seems to be a slower algorithm.

Also regarding memory, it seems that a key potential benefit of savings in space is the ability — for certain sizes — to perform all the work at a lower level of the memory hierarchy. For instance, if two computations are identical, but one always uses twice as much space than the other, then for some input size the larger algorithm will require many accesses to main memory (or disk), whereas the smaller algorithm will stay entirely in cache (or main memory, respectively). Since the cost difference in accessing memory at different levels of the hierarchy is significant, we might expect some savings here.

In fact, we will see some evidence of such a phenomenon occurring with the multiplication algorithms of the next chapter. However, for the radix-2 DFT algorithms we have presented, this will *never* happen, precisely because the size of memory at any level in any mod-



ern architecture is always a power of two. This gives some justification for our findings here, but also lends motivation to studying the radix-3 case in the future, as we discuss below.

### 3.7 Future work

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. The next chapter will show the impact of this result on dense polynomial multiplication.

There is much work still to be done on truncated Fourier transforms. First, we see that our algorithm, while improving the memory usage, does not yet improve the time cost of other FFT algorithms. This is partly due to the cost of extra arithmetic operations, but as Theorem 3.5 demonstrates, our algorithm is actually superior to the usual FFT in this measure, at least in the worst case. A more likely cause for the inferior performance is the less regular structure of the algorithm. In particular, our in-place algorithms cannot be computed level-wise, as can the normal FFT and forward TFT algorithms. Developing improved in-place TFT algorithms whose actual complexity is closer to that of the original TFT would be a useful exercise.

To this end, it may be useful to consider an in-place truncated version of the radix-3 FFT algorithm. It should be relatively easy to adapt our algorithm to the radix-3 case, with the same asymptotic complexity. However, the radix-3 algorithm may perform better in practice, due to the nature of cache memory in modern computers.

Most of the cache in modern machines is least partly associative — for example, the L1 cache on our test machine is 8-way associative. This means that a given memory location can only be stored in a certain subset of the available cache lines. Since this associativity is always based on equivalence modulo powers of two, algorithms that access memory in power-of-two strides, such as in the radix-2 FFT, can suffer in cache performance. Radix-3 FFTs have been used in numerical libraries for many years to help mitigate this problem. Perhaps part of the reason they have not been used very much in symbolic computation software is that the blow-up in space when the size of the transform is not a power of 3 is more significant than in the radix-2 version. Developing an in-place variant for the radix-3 case would of course eliminate this concern, and so may give practical benefits.

Finally, it would be very useful to develop 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 (Li et al., 2009). 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.

# Bibliography

- David H. Bailey. FFTs in external or hierarchical memory. *The Journal of Supercomputing*, 4: 23–35, 1990. ISSN 0920-8542.  
doi: [10.1007/BF00162341](https://doi.org/10.1007/BF00162341). Referenced on page 37.
- James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.  
doi: [10.1090/S0025-5718-1965-0178586-1](https://doi.org/10.1090/S0025-5718-1965-0178586-1). Referenced on page 36.
- Richard E. Crandall. *Topics in advanced scientific computation*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 0-387-94473-7. Referenced on pages 35 and 39.
- Jack Dongarra and Francis Sullivan. Guest editors' introduction to the top 10 algorithms. *Computing in Science Engineering*, 2(1):22–23, January/February 2000. ISSN 1521-9615.  
doi: [10.1109/MCISE.2000.814652](https://doi.org/10.1109/MCISE.2000.814652). Referenced on page 36.
- David Harvey. A cache-friendly truncated FFT. *Theoretical Computer Science*, 410(27–29): 2649–2658, 2009. ISSN 0304-3975.  
doi: [10.1016/j.tcs.2009.03.014](https://doi.org/10.1016/j.tcs.2009.03.014). Referenced on pages 37 and 40.
- David Harvey and Daniel S. Roche. An in-place truncated Fourier transform and applications to polynomial multiplication. In *ISSAC '10: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, pages 325–329, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0150-3.  
doi: [10.1145/1837934.1837996](https://doi.org/10.1145/1837934.1837996). Referenced on page 35.
- Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *ASSP Magazine, IEEE*, 1(4):14–21, October 1984. ISSN 0740-7467.  
doi: [10.1109/MASPP.1984.1162257](https://doi.org/10.1109/MASPP.1984.1162257). Referenced on page 36.
- Joris van der Hoeven. The truncated Fourier transform and applications. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation, ISSAC '04*, pages 290–296, New York, NY, USA, 2004. ACM. ISBN 1-58113-827-X.  
doi: [10.1145/1005285.1005327](https://doi.org/10.1145/1005285.1005327). Referenced on pages 40, 47 and 50.
- Joris van der Hoeven. Notes on the truncated Fourier transform. Technical Report 2005-5, Université Paris-Sud, Orsay, France, 2005.  
URL <http://www.texmacs.org/joris/tft/tft-abs.html>. Referenced on page 40.

Xin Li, Marc Moreno Maza, and Éric Schost. Fast arithmetic for triangular sets: From theory to practice. *Journal of Symbolic Computation*, 44(7):891–907, 2009. ISSN 0747-7171. doi: [10.1016/j.jsc.2008.04.019](https://doi.org/10.1016/j.jsc.2008.04.019). International Symposium on Symbolic and Algebraic Computation. Referenced on pages 40 and 51.

John D. Markel. FFT pruning. *Audio and Electroacoustics, IEEE Transactions on*, 19(4):305–311, December 1971. ISSN 0018-9278. doi: [10.1109/TAU.1971.1162205](https://doi.org/10.1109/TAU.1971.1162205). Referenced on page 40.

Henrik V. Sorensen and C. Sidney Burrus. Efficient computation of the DFT with only a subset of input or output points. *Signal Processing, IEEE Transactions on*, 41(3):1184–1200, March 1993. ISSN 1053-587X. doi: [10.1109/78.205723](https://doi.org/10.1109/78.205723). Referenced on page 40.