The Truncated Fourier Transform for mixed radices

Robin Larrieu
Laboratoire d'informatique de l'École polytechnique
91128 Palaiseau Cedex, France

Dec. 2016

Abstract

The standard version of the Fast Fourier Transform (FFT) is applied to problems of size $n = 2^k$. For this reason, FFT-based evaluation/interpolation schemes often reduce a problem of size l to a problem of size n, where n is the smallest power of two with $l \leqslant n$. However, this method presents "jumps" in the complexity at powers of two; and on the other hand, $n - l$ values are computed that are actually unnecessary for the interpolation. To mitigate this problem, a truncated variant of the FFT was designed to avoid the computation of these unnecessary values. In the initial formulation [14], it is assumed that n is a power of two, but some use cases (for example in finite fields) may require more general values of n. This paper presents a generalization of the Truncated Fourier Transform (TFT) for arbitrary orders. This allows to benefit from the advantages of the TFT in the general case.

1 Introduction

Many basic arithmetic operations on polynomials can be performed efficiently using evaluation/interpolation techniques. A typical example is the multiplication of polynomials: let A, B be polynomials in $\mathbb{K}[X]$ for a field $\mathbb{K}$, such that $\deg(AB) < n$. The product AB can be computed by choosing n different values $(x_1, \dots, x_n) \in \mathbb{K}^n$, and evaluating the $A(x_i)$ and $B(x_i)$ for all i. Then, a term-by-term multiplication leads to the values $(AB)(x_i)$ for all i, and an interpolation on these values allows to retrieve the polynomial AB.

The *Discrete Fourier Transform (DFT)* is a way to perform these evaluations and interpolations on specifically chosen values: let ω be a primitive n-th root of unity and $P \in \mathbb{K}[X]$ of degree less than n; then the DFT of P is the n-tuple $(P(1), P(\omega), \dots, P(\omega^{n-1}))$. Symmetrically, the inverse DFT computes the coefficients of P from the values $(P(\omega^i))_{0 \leqslant i < n}$. When n is highly composite, the DFT and its inverse can be computed efficiently using the *Fast Fourier Transform (FFT)* algorithm. This method was known to Gauss around 1805 [6], but it received little attention until after its rediscovery by Cooley and Tukey [3]. For this reason, and because they were the first to use this method as a systematic computation algorithm, the modern formulation of the FFT is usually attributed to Cooley and Tukey.

1.1 Jump phenomenon

The standard version of FFT-based multiplication algorithms relies on a primitive n-th root of unity $\omega \in \mathbb{K}$, where n is a power of two. This requirement causes the following

drawback: when a polynomial of degree less than d is considered (or when d evaluation points are needed) with d slightly larger than 2^k, one must perform a FFT of order 2^{k+1}. This causes a significant overhead since up to twice as many values are computed as what is actually needed.

This jump phenomenon can be mitigated by allowing a more precise choice of $n \geqslant d$. For example, instead of requiring $n = 2^k$, one can allow more general products of small primes such as $n = 2^k 3^l 5^m$. FFTs of such sizes reduce to DFTs of sizes 2, 3 and 5, for which efficient codelets are implemented e.g. in the FFTW3 library [5]. Alternatively, optimized radix-2 methods may be preferred because of their simplicity (fewer base cases to handle). For example, the *FFT pruning* [10] aims to reduce the overhead for a zero-padded sequence. Another example is Crandall and Fagin's *Discrete Weighted Transform* [4], which reduces a problem of size $d < 2^k$ to two problems of size 2^l and 2^m with $d < 2^l + 2^m < 2^k$.

Another elegant solution to this jump phenomenon is to use the *Truncated Fourier Transform* [14]. The TFT behaves as a usual FFT of order 2^k, but it performs a multipoint evaluation with exactly the desired length, while avoiding the computation of all intermediate values that are not needed for obtaining the output. Moreover, the interpolation can be performed with the same complexity using the inverse TFT. Improvements to this algorithm were made to reduce memory usage [9], and improve cache friendliness [8]. Mateer [11, Chapter 6] also proposed a different formulation of the TFT inspired by Crandall and Fagin's reduction. He mentions that this alternative formulation can be used with a few adaptations when $n = 3^k$, or another prime power.

1.2 Goal of this paper

The methods discussed above rely on a choice of n with a very specific form. This requires the base field $\mathbb{K}$ to contain primitive n-th roots of unity for all such n. This is true for $\mathbb{K} = \mathbb{C}$, but in general, the choice of roots of unity is restricted. It is always possible to add virtual roots of unity (with certain restrictions on their order if the field has non-zero characteristic) as in the Schönhage-Strassen algorithm [13], but this extension causes computational overhead.

Assume that the choice of roots of unity is restricted by both our base field $\mathbb{K}$ and practical considerations. Let $S \subset \mathbb{N}$ denote the set of orders n for the roots of unity that can be used in FFTs. For example, the use cases mentioned in previous section assume $\mathbb{K} = \mathbb{C}$, and only practical considerations are taken into account. This leads to sets of the form $S = \{2^k | k \in \mathbb{N}\}$, or $S = \{2^k 3^l 5^m | k, l, m \in \mathbb{N}\}$. In a finite field like $\mathbb{F}_{2^l}$, there are roots of unity only for specific orders, so we would have $S = \{n | n \text{ divides } 2^l - 1\}$. A remarkable example is $\mathbb{F}_{2^{60}}$ because there are efficient ways of computing in this field, and many roots of unity (with large, highly-composite order) are known [7].

As discussed previously, there is a jump phenomenon at elements of S (more or less important depending on their distribution). This paper aims to reduce this jump phenomenon through a generalization of the Truncated Fourier Transform to an arbitrary n.

At first, we give a brief reminder of the Fast Fourier Transform in the general case, and some useful notations are introduced. In section 3 and 4, we present algorithms to compute the Truncated Fourier Transform and its inverse for any n. Finally, the complexity of the TFT is discussed and compared with the ordinary FFT.

2 The Cooley-Tukey FFT

Let $A = (A_i)_{0 \leqslant i < n}$ be a vector in $\mathbb{K}^n$, and let ω be a primitive n-th root of unity. The Discrete Fourier Transform (DFT) of A with respect to ω is the vector $\hat{A} = (\hat{A}_i)_{0 \leqslant i < n}$ where

$$\hat{A}_i = \sum_{j=0}^{n-1} A_j \cdot \omega^{ij} \,. \tag{2.1}$$

Equivalently, if P_A is the polynomial $A_0 + A_1 X + \cdots + A_{n-1} X^{n-1}$, then $\hat{A}_i = P_A(\omega^i)$. The Fast Fourier Transform (FFT) describes an algorithm to compute the DFT efficiently.

2.1 Fundamentals of the FFT algorithm

The Cooley-Tukey algorithm [3] relies on the following remark: assuming that $n = n_1 n_2$ is composite, the following holds for all $k_1 < n_1$ and $k_2 < n_2$

$$\hat{A}_{k_1 + n_1 k_2} = \sum_{i=0}^{n_2-1} \sum_{j=0}^{n_1-1} A_{i+n_2 j} \cdot \omega^{(i+n_2 j)(k_1 + n_1 k_2)} \,,$$

that is:

$$\hat{A}_{k_1 + n_1 k_2} = \sum_{i=0}^{n_2-1} \omega^{i k_1} \cdot \left(\sum_{j=0}^{n_1-1} A_{i+n_2 j} \cdot (\omega^{n_2})^{j k_1} \right) \cdot (\omega^{n_1})^{i k_2} \,. \tag{2.2}$$

Using formula (2.2), a Discrete Fourier Transform of length $n = n_1 n_2$ can be decomposed into n_2 DFTs of length n_1 (*inner DFTs*) followed by n_1 DFTs of length n_2 (*outer DFTs*). These smaller DFTs can be computed recursively using the same method as long as their size not prime. In the special case where $n = 2^k$, this leads to the well-known complexity bound $O(n \lg n)$.

Remark 2.1. Each inner DFT is applied to a subvector of the input (with offset $i < n_1$ and stride n_2). Instead, it can be preferable to reindex the working array before the inner DFTs, and once again before the outer DFTs. The main purpose of this reindexing is to perform the recursive calls on contiguous memory blocks. This approach may lead to more efficient implementation because of cache effects: sparing frequent data exchanges with the RAM during the inner DFT compensates for the cost of the reindexing.

2.2 Generalized bitwise mirror

In an in-place implementation of the Cooley-Tukey FFT, it is convenient to order the output differently (see for example [7, Section 2.1]). The purpose of this different order is to ensure that the result of the full FFT is simply the concatenation of the outputs from the outer DFTs. As a consequence, it spares a matrix transposition at the end of the algorithm, and therefore also at each recursion step. The v-mirror noted $[i]_v$ defined in this section is a notation for this new indexation: $\mathrm{FFT}(A)_i = \hat{A}_{[i]_v}$

Let the vector $v = (p_0, \ldots, p_{d-1})$ be a decomposition of n ($n = \prod_{j=0}^{d-1} p_j$); with not necessarily prime p_j's. Then, any index $i < n$ can be uniquely written under the form:

$$i = i_0 \cdot p_1 \cdots p_{d-1} + i_1 \cdot p_2 \cdots p_{d-1} + \cdots + i_{d-2} \cdot p_{d-1} + i_{d-1} \,, \quad \text{where for all } j, i_j < p_j.$$

With these notations, the v-mirror of i is defined as

$$[i]_v = i_0 + i_1 \cdot p_0 + i_2 \cdot p_0 p_1 + \cdots + i_{d-1} \cdot p_0 \cdots p_{d-2} \,.$$

When all p_j's are 2, this definition coincides with the bitwise mirror introduced for the radix-2 TFT from [14]. Let $h \leqslant d$, $w_1 = (p_0, \ldots, p_{h-1})$, $w_2 = (p_h, \ldots, p_{d-1})$, $n_1 = \prod_{j=0}^{h-1} p_j$ and $n_2 = \prod_{j=h}^{d-1} p_j$ (so that $n = n_1 n_2$). It is easy to show the following basic properties of the v-mirror:

$$[[i]_v]_{\bar{v}} = i \qquad \text{with } \bar{v} = (p_{d-1}, \ldots, p_0), \tag{2.3}$$

$$[i]_v = i \qquad \text{if } d = 1 \text{ i.e. } v = (n), \tag{2.4}$$

$$[I + n_2\, J]_v = [J]_{w_1} + n_1\, [I]_{w_2} \qquad \text{for } I < n_2,\ J < n_1. \tag{2.5}$$

Remark 2.2. As claimed at the beginning of this subsection, if $\mathrm{FFT}(A)_i = \hat{A}_{[i]_v}$, then the output of the top-level FFT is indeed the concatenation of the outputs from the outer FFTs. This comes from properties (2.5) and (2.2) combined, and it will be more formally discussed in the next subsection, once additional notations have been introduced.

Remark 2.3. As shown in [14] for $n = 2^k$, it is crucial for the functioning of the TFT algorithm to order the output according to the v-mirror. Actually, using the natural ordering would require a transposition after the outer FFTs (and also at each recursion depth). In the case of the TFT, such a transposition would mix the known and unknown values, while the TFT relies on a contiguous block of known values.

2.3 Further notations for the steps of the FFT algorithm

Now consider an execution of the FFT algorithm as in section 2.1. The recursive calls of the algorithm define a decomposition $n = p_0 \cdots p_{d-1}$ of n, with $n_1 = p_0 \cdots p_{h-1}$ and $n_2 = p_h \cdots p_{d-1}$. We reuse the notations from the previous section: $v = (p_0, \ldots, p_{d-1})$, $w_1 = (p_0, \ldots, p_{h-1})$ and $w_2 = (p_h, \ldots, p_{d-1})$. As in [7], we assume the output verifies $\mathrm{FFT}(A)_i = \hat{A}_{[i]_v}$ (that is, the Discrete Fourier Transform that is returned is reordered according to the v-mirror).

Remark 2.4. Between the inner and outer DFTs, intermediates values are multiplied by a power of ω, more precisely $\omega^{i \cdot k_1}$ as in formula (2.2). This factor is usually called the *twiddle factor*. With an indexation of the output according to the v-mirror, the *twiddle factor* must be $\omega^{i[k_1]_{w_1}}$

For clarity, we will introduce different input and output vectors. These vectors are actually just names to represent specific parts of the working vector at different steps of the algorithm, but no duplication of data should occur. Let A and B (size n) be the input and output vectors of the algorithm. Let α_i and β_i ($i < n_2$, each vector has size n_1) be the input and output vectors of the i-th inner DFT (recursive call of the FFT algorithm). Let γ_j and δ_j ($j < n_1$, each vector has size n_2) be the input and output vectors of the j-th outer DFT. These notations are illustrated in Figure 1. By definition, we have the following properties for all i, j:

$$(\alpha_i)_j = A_{i+n_2 j}\,, \tag{2.6}$$

$$(\beta_i)_j = \mathrm{FFT}(\alpha_i)_j = (\widehat{\alpha_i})_{[j]_{w_1}}\,, \tag{2.7}$$

$$(\gamma_j)_i = \omega^{i[j]_{w_1}} \cdot (\beta_i)_j\,, \tag{2.8}$$

$$(\delta_j)_i = \mathrm{FFT}(\gamma_j)_i = (\widehat{\gamma_j})_{[i]_{w_2}}\,, \tag{2.9}$$

$$(\delta_j)_i = B_{i+n_2 j} = \hat{A}_{[i+n_2 j]_v}\,. \tag{2.10}$$

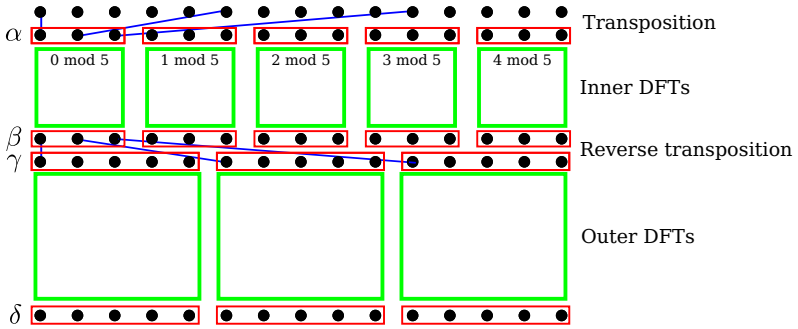

Figure 1: Notations and steps of the FFT

Remark 2.5. These properties can be seen as an implementation of the Cooley-Tukey FFT algorithm. As a proof, we can rewrite these equations with different indexes for consistency with section 2.1. Let $K_1, k_1 < n_1$ and $K_2, k_2 < N_2$ such that $k_1 = [K_1]_{w_1}$ and $k_2 = [K_2]_{w_2}$. Then, using the property (2.5) of the v-mirror in relation (2.10), we get: $(\alpha_i)_j = A_{i+n_2 j}$, $(\beta_i)_{K_1} = (\widehat{\alpha_i})_{k_1}$, $(\gamma_{K_1})_i = \omega^{ik_1} \cdot (\beta_i)_{K_1}$ $(\delta_{K_1})_{K_2} = (\widehat{\gamma_{K_1}})_{k_2}$ and $(\delta_{K_1})_{K_2} = \hat{A}_{k_1+n_1 k_2}$. This proves the correctness of the algorithm using formula (2.2).

2.4 Specification of the FFT algorithm and its inverse

This section formalizes FFT and inverse FFT as blackboxes to be used in TFT algorithms. In particular, assumptions on the input and the output are precised.

Because of Remark 2.3, we assume the output is ordered according to the v-mirror. Hence, the FFT algorithm can be described as follows:

Algorithm 2.1. FFT

- **INPUT:** an integer $n \in \mathbb{N}$, a vector $A = (a_i)_{i<n}$, a vector $v = (p_0, \ldots, p_{d-1})$ such that $n = \prod_{i=0}^{d-1} p_i$, and a primitive n-th root of unity ω
- **OUTPUT:** the vector $T = (\hat{A}_{[i]_v})_{i<n}$

Symmetrically, the inverse transformation is computed by the IFFT (inverse FFT) algorithm:

Algorithm 2.2. IFFT (inverse FFT)

- **INPUT:** an integer $n \in \mathbb{N}$, a vector $B = (b_i)_{i<n}$, a vector $v = (p_0, \ldots, p_{d-1})$ such that $n = \prod_{i=0}^{d-1} p_i$, and a primitive n-th root of unity ω
- **OUTPUT:** the vector $A = (a_i)_{i<n}$ such that $\forall i < n, b_i = \hat{A}_{[i]_v}$

Remark 2.6. The inverse FFT can be computed by reversing the FFT algorithm. Alternatively, using FFT as a blackbox, the following formula (and reordering of the input/output) can be used:

$$\mathrm{DFT}_{\omega^{-1}}\left(\mathrm{DFT}_{\omega}(A)\right)_k = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} a_i \cdot \omega^{(i-k)j} = n a_k$$

then,

$$(\mathrm{DFT}_{\omega})^{-1} = \frac{1}{n}\mathrm{DFT}_{\omega^{-1}}\,. \tag{2.11}$$

3 The Truncated Fourier Transform for arbitrary orders

This section generalizes the Truncated Fourier Transform (TFT) [14] for an arbitrary order $n = p_0 p_1 \cdots p_{d-1}$. Given a vector A of length n, the TFT computes $l \leqslant n$ well chosen values of the Discrete Fourier Transform of A; that is the vector $T = (T_0, \ldots, T_{l-1}) = (\hat{A}_{i_0}, \ldots, \hat{A}_{i_{l-1}})$. Note that these are not necessarily the first l values of $\hat{A}$. The algorithm presented in this section aims to perform less computation than by simply computing the DFT of A and discarding the unused values.

3.1 Atomic transforms

At first we consider the following base case: given $(a_0, \ldots, a_{p-1})$ for p prime or reasonably small, we want to compute directly the TFT $(\hat{A}_0, \ldots, \hat{A}_{l-1})$. To do so, one can naively apply Horner's rule for each value as in formula (2.1), which is especially efficient for small p (principle of a specialized *codelet*). For larger p, it becomes more interesting to compute the full DFT, then discard unused values. A full DFT of a such size can be computed using efficient transformations such as Rader's algorithm [12] and Bluestein's transform [1].

We assume that these considerations translate into the following algorithm:

Algorithm 3.1. atomicTFT

- **INPUT:** integers $n \in \mathbb{N}$ and $l < n$, a vector $A = (a_0, \ldots, a_{n-1})$ and and a primitive n-th root of unity ω
- **OUTPUT:** the vector $T = (\hat{A}_0, \ldots, \hat{A}_{l-1})$

Remark 3.1. In the base case, we do not use the indexation according to the v-mirror because of property (2.3).

3.2 General idea

Assume only $l \leqslant n$ values of the output are actually needed. The plain FFT algorithm can be modified to avoid computation of irrelevant intermediate values. As stated in Remark 2.3, the output of the DFT must be ordered according to the v-mirror.

If we want to return the tuple $(\hat{A}_{[i]_v})_{0 \leqslant i < l}$, then according to relation (2.10), the vectors δ_j need to be computed only for $j < m = \lceil l/n_2 \rceil$. This means by definition (2.9) that only the γ_j with $j < m$ are needed. From formula (2.8), we conclude that for every $i < n_2$, only the first m values of β_i need to be computed.

Moreover, if $q = (l \text{ quo } n_2) < m$, only the $r = (l \text{ rem } n_2)$ first values of δ_q are needed (where quo and rem represent the quotient and remainder operations in the euclidean division). Figure 2 gives a visual representation of which values are actually needed.

3.3 Presentation of the algorithm

The previous discussion suggest that a TFT of order $n = n_1 n_2$ can be decomposed into n_2 TFTs of order n_1 followed by m TFTs of order n_2 (as for the usual FFT). If the top-level TFT has length l, then the inner TFTs have length $m = \lceil l/n_2 \rceil$. Most of the outer TFTs

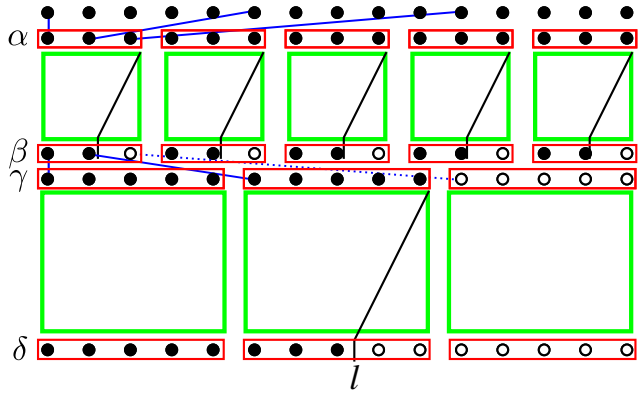

Figure 2: Example of Truncated Fourier Transform (TFT)

are actually usual FFTs; only the last one may be a TFT of length $r = (l \text{ rem } n_2)$ (unless $r = 0$). This leads to the following recursive algorithm:

Algorithm 3.2. TFT

- **INPUT:** integers $n \in \mathbb{N}$, $l \in \{1, \ldots, n\}$, a vector $A = (a_i)_{i<n}$, a vector $v = (p_0, \ldots, p_{d-1})$ such that $n = \prod_{i=0}^{d-1} p_i$, and a primitive n-th root of unity ω
- **OUTPUT:** the vector $T = (\hat{A}_{[i]_v})_{i<l}$

if $d = 1$ **then return** atomicTFT(n, l, A, ω) ▷ Algorithm 3.1
else if $l = n$ **then return** FFT(n, A, v, ω) ▷ Ordinary FFT (Algorithm 2.1)
else
  choose $h \in \{1, \ldots, d-1\}$; define n_1, n_2, w_1, w_2 as in section 2.3
  $m \leftarrow \lceil l/n_2 \rceil$; $q \leftarrow l$ quo n_2; $r \leftarrow l$ rem n_2
  for $0 \leqslant i < n_2$ **do** ▷ n_2 TFTs of size n_1
    $(\alpha_i)_j \leftarrow A_{i+n_2 j}$ for all $j < n_1$
    $\beta_i \leftarrow \text{TFT}(n_1, m, \alpha_i, w_1, \omega^{n_2})$
    $(\gamma_j)_i \leftarrow \omega^{i[j]_{w_1}} \cdot (\beta_i)_j$ for all $j < m$
  end for
  $T \leftarrow \emptyset$
  for $0 \leqslant j < m$ **do** ▷ n_1 TFTs of size n_2
    if $j < q$ **then**
      $\delta_j \leftarrow \text{FFT}(n_2, \gamma_j, w_2, \omega^{n_1})$ ▷ Ordinary FFT (Algorithm 2.1)
    else
      $\delta_j \leftarrow \text{TFT}(n_2, r, \gamma_j, w_2, \omega^{n_1})$
    end if
    $T \leftarrow (T, \delta_j)$
  end for
  return T
end if

Theorem 3.1. *Assuming correct implementations of Algorithms 2.1 and 3.1, the Algorithm 3.2 is correct.*

Proof. Case $l = n$ corresponds to a full FFT, then Algorithm 2.1 returns the expected result. If $d = 1$ (typically when n is prime), the Discrete Fourier Transform is computed

directly. Then, unnecessary outputs are discarded. The results are ordered as expected because of property (2.4). In the other cases, the algorithm is called recursively, and its correctness results by induction from the discussion in section 3.2. □

Remark 3.2. As for the in-place Cooley-Tukey FFT presented in [7], the result depends on the vector v, but not on the choice of h.

4 The inverse TFT for arbitrary orders

The formula (2.11) for the usual FFT cannot be used in the case of the TFT because not all values of the transform are known. As for the standard TFT ($n = 2^k$) [14], we revert the algorithm computing the TFT instead.

The inversion of a TFT is to be understood as follows: assume that the values $(B_i)_{0 \leqslant i < l}$ of the output, and $(A_i)_{l \leqslant i < n}$ of the input are known. Then, the goal is to retrieve the missing values $(A_i)_{0 \leqslant i < l}$ of the input. Typically, the values A_i (for $i \geqslant l$) are known to be 0 because of a simple analysis regarding the degree, but the coefficients of highest degree of a polynomial can also be deduced from a limit analysis.

At first, we provide a method to solve the base case of size p by direct computation (here p is not necessarily prime, but it should be reasonably small, so the problem can be solved without further decomposition). Then, we present a recursive algorithm that reduces the TFT inversion to such a base case.

4.1 Atomic inverse transforms

In this section, we consider the following *skew butterfly* problem: given $\hat{A}_0, \ldots, \hat{A}_{l-1}$ and $a_l, \ldots, a_{p-1}$, how can we compute the missing values $\hat{A}_l, \ldots, \hat{A}_{p-1}$ and $a_0, \ldots, a_{l-1}$?

The Discrete Fourier Transform is given by the following matrix-vector product:

$$\begin{pmatrix} \hat{A}_0 \\ \hat{A}_1 \\ \vdots \\ \hat{A}_{p-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 1 & \omega & \cdots & \omega^{p-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{p-1} & \cdots & \omega^{(p-1)(p-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{p-1} \end{pmatrix},$$

or, in a more compact form,

$$\hat{A} = V_\omega \cdot A$$

For any $m \leqslant p$, we define the submatrices

$$V_{\omega,m} = (\omega^{ij})_{0 \leqslant i,j < m},$$
$$\tilde{V}_{\omega,m} = (\omega^{(i+m)(j+m)})_{0 \leqslant i,j < p-m},$$
and
$$W_{\omega,m} = (\omega^{i(j+m)})_{i < m\,;\, j < p-m}.$$

Note that $V_{\omega,m}$ has size $m \times m$, $\tilde{V}_{\omega,m}$ has size $(p-m) \times (p-m)$ and $W_{\omega,m}$ has size $m \times (p-m)$. In other words:

$$V_\omega = \begin{pmatrix} V_{\omega,m} & W_{\omega,m} \\ W_{\omega,m}^\top & \tilde{V}_{\omega,m} \end{pmatrix} \tag{4.1}$$

The considered *skew butterfly* problem is equivalent to the resolution of the following matrix equation with parameters A_2, B_1 and unknowns A_1 and B_2:

$$\begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = \begin{pmatrix} V_{\omega,l} & W_{\omega,l} \\ W_{\omega,l}^\top & \tilde{V}_{\omega,l} \end{pmatrix} \cdot \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} \tag{4.2}$$

$V_{\omega,l}$ is a general Vandermonde matrix of determinant $\prod_{0\leqslant i<j<l}(\omega^i - \omega^j) \neq 0$, hence it is invertible. Therefore,

$$A_1 = (V_{\omega,l})^{-1}(B_1 - W_{\omega,l} \cdot A_2)\,. \tag{4.3}$$

Once A_1 is known, it is easy to compute B_2 as

$$B_2 = W_{\omega,l}^{\top} \cdot A_1 + \tilde{V}_{\omega,l} \cdot A_2\,. \tag{4.4}$$

Remark 4.1. In the case $l = 1, p = 2$, we get the equations $a_1 = b_1 - a_2$, then $b_2 = b_1 - 2a_2$, which corresponds to the inversion of the usual *butterfly* of size 2 as encountered when $n = 2^k$ [14].

Remark 4.2. Often, it is not necessary to compute B_2 entirely, but only specific values. In this case, equation (4.4) reduces to a much smaller computation. For example, when composing atomic transforms for the inverse TFT, returning the first value of B_2 is actually sufficient. For our usage, we assume that the results from this section translate into the following algorithm:

Algorithm 4.1. atomicITFT (atomic inverse TFT)

- **INPUT:** integers $n \in \mathbb{N}$ and $l < n$, vectors $A_2 = (a_i)_{l\leqslant i<n}$ and $B_1 = (b_i)_{i<l}$, and a primitive n-th root of unity ω
- **OUTPUT:** the vector $A_1 = (a_i)_{i<l}$ and the value b_l such that $\forall i \leqslant l, b_i = \hat{A}_i$ where
  $A = (A_1, A_2) = (a_i)_{i<n}$

4.2 Recursive algorithm

In a similar way as in the case $n = 2^k$ [14], intermediate results are not always computed in an order corresponding to the recursion depth. In a usual FFT, all outer FFTs are inverted, then the inner FFTs are inverted. On the contrary for the TFT, some of the inner TFTs must be inverted, and these inversions provide additional values that allow the outer TFT to be inverted. This behavior is illustrated in Figure 3). Therefore, the algorithm will return some of the missing output values $(b_i)_{l\leqslant i<n}$ in addition to the desired input values $(a_i)_{0\leqslant i<l}$. It turns out that the outer TFT needs only one value from each

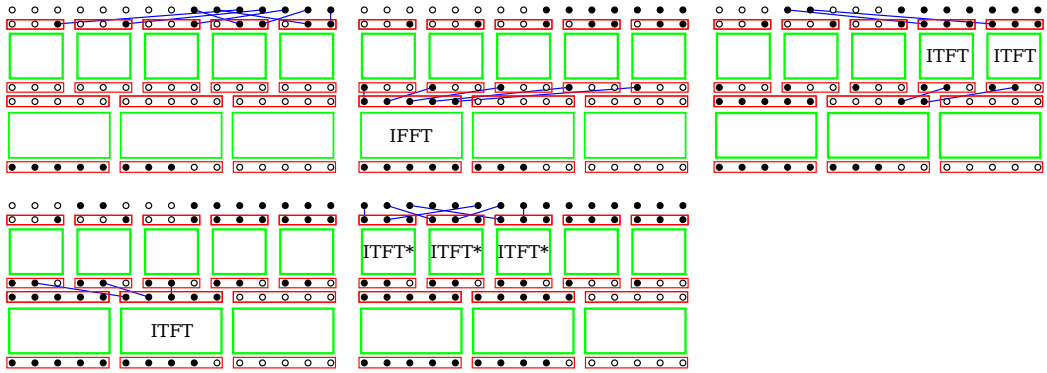

Figure 3: Example of inverse TFT. Black (resp. white) dots represent known (resp. unknown) values, ITFT* do not return additional output values

inner TFT before it can be inverted, so returning b_l is actually sufficient for the functioning of the recursive algorithm.

We assume that we have at our disposal an algorithm (Algorithm 2.2) to reverse a full FFT (case $l = n$). We also assume having another algorithm to reverse atomic TFTs (Algorithm 4.1). The design from previous paragraph translates into the following algorithm:

Algorithm 4.2. ITFT (inverse TFT)

- **INPUT:** integers $n \in \mathbb{N}$ and $l < n$, vectors $A_2 = (a_i)_{l \leqslant i < n}$ and $B_1 = (b_i)_{i<l}$, a vector $v = (p_1, \ldots, p_d)$ such that $n = \prod_{i=1}^{d} p_i$, and a primitive n-th root of unity ω
- **OUTPUT:** the vector $A_1 = (a_i)_{i<l}$ and the value b_l such that:

$$\forall i \leqslant l, b_i = \left(\hat{A}_{[i]_v}\right) \qquad \text{where } A = (A_1, A_2) = (a_i)_{i<n}$$

if $l = 0$ **then**
  return $\varnothing, \sum_{i=0}^{n-1} a_i$
else if $d = 1$ **then**
  return atomicITFT(n, l, A_2, B_1, ω) ▷ Algorithm 4.1
else
  choose $h \in \{1, \ldots, d-1\}$; define n_1, n_2, w_1, w_2 as in section 2.3
  $q \leftarrow l$ quo n_2, $r \leftarrow l$ rem n_2
  for $j < q$ **do** ▷ **"Step 1"**
    $(\delta_j)_i \leftarrow (B_1)_{i+n_2 j}$ for all $i < n_1$
    $\gamma_j \leftarrow \text{IFFT}(n_2, \delta_j, w_2, \omega^{n_1})$ ▷ Algorithm 2.2
    $(\beta_{i,1})_j \leftarrow \omega^{-i[j]_{w_1}} \cdot (\gamma_j)_i$ for all $i < n_1$
  end for
  for $r \leqslant i < n_2$ **do** ▷ **"Step 2"**
    $(\alpha_{i,2})_j \leftarrow a_{i+n_2 j}$ for all $j \geqslant q$ ▷ $i + n_2 j \geqslant l$
    $(\alpha_{i,1}), (\beta_i)_q \leftarrow \text{ITFT}(n_1, q, \alpha_{i,2}, \beta_{i,1}, w_1, \omega^{n_2})$ ▷ $q < n_1$ since $l < n$
    $(\gamma_q)_i \leftarrow \omega^{i[q]_{w_1}} \cdot (\beta_i)_q$
    $(A_1)_{i+n_2 j} \leftarrow (\alpha_{i,1})_j$ for all $j < q$
  end for
  $\Delta_1 \leftarrow ((B_1)_{i+n_2 q})_{i<r}$ ▷ **"Step 3"**
  $\Gamma_2 \leftarrow ((\gamma_q)_i)_{r \leqslant i < n_2}$
  $\Gamma_1, (\delta_q)_r \leftarrow \text{ITFT}(n_2, r, \Delta_1, \Gamma_2, w_2, \omega^{n_1})$
  for $i < r$ **do** ▷ **"Step 4"**
    $(\beta_{i,1})_q \leftarrow \omega^{-i[q]_{w_1}} \cdot (\Gamma_1)_i$
    if $q + 1 < n_1$ **then**
      $(\alpha_{i,2})_j \leftarrow a_{i+n_2 j}$ for all $j > q$ ▷ $i + n_2 j \geqslant l$
      $(\alpha_{i,1}), \text{dummy} \leftarrow \text{ITFT}(n_1, q+1, \alpha_{i,2}, \beta_{i,1}, w_1, \omega^{n_2})$
    else
      $(\alpha_{i,1}) \leftarrow \text{IFFT}(n_1, \beta_{i,1}, w_1, \omega^{n_2})$ ▷ Algorithm 2.2
    end if
    $(A_1)_{i+n_2 j} \leftarrow (\alpha_{i,1})_j$ for all $j \leqslant q$
  end for
  return $A_1, (\delta_q)_r$
end if

Theorem 4.1. *Assuming correct implementations of Algorithms 2.2 and 4.1, the Algorithm 4.2 is correct.*

Proof. We proceed by induction over d. The case $l = 0$ is clearly correct. For $d = 1$, the result is computed directly using Algorithm 4.1, which is supposed to be correct. As for Algorithm 3.2, the result is ordered as expected because of property (2.4) of the v-mirror.

In *Step 1*, the γ_j are computed for all $j < q$ using a full reverse FFT. This means γ_j and δ_j verify equation (2.9) for $j < q$. Then, the first part of every vector β_i is computed according to equation (2.8).

At this point, the vectors α_i and β_i are partially known. More precisely, we know the values with $j > q$ of α_i and the values $j < q$ of β_i. Moreover, $(\alpha_i)_q$ is known for $i \geqslant r$ (by definition, $l = n_2\, q + r$). In *Step 2*, a recursive call computes the missing part of α_i for these i, as well as the value $(\beta_i)_q$. This means α_i and $((\beta_i)_j)_{j \leqslant q}$ verify equation (2.7) for all $i \geqslant r$ (the recursive call is correct by the induction hypothesis).

At the end of *Step 2*, the second part of $(\gamma_q)_i$ (for $i \geqslant r$) is computed according to equation (2.8). It is noted Γ_2 in the algorithm. The first part Δ_1 of $(\delta_q)_i$ (for $i < r$) is also known from the input. In *Step 3*, the missing part Γ_1 of γ_q (that is the $(\gamma_q)_i$ for $i < r$) is computed as well as $(\delta_q)_r$ through a recursive call. Then, γ_q and $((\delta_q)_i)_{i \leqslant r}$ verify equation (2.9).

Finally in *Step 4*, the $(\beta_i)_q$ are computed for $i < r$. The $(\alpha_i)_j$ being given on input for $j > q$, and the $(\beta_i)_j$ being known from *Step 1*, the missing $(\alpha_i)_j$ ($j \leqslant q$) can be computed using a recursive call. Since this call is correct by the induction hypothesis, α_i and $((\beta_i)_j)_{j \leqslant q}$ verify equation (2.7) for $i < r$ (hence for all i because of *Step 2*.

All in all, the vectors (some truncated) α_i, $((\beta_i)_j)_{j \leqslant q}$, γ_j ($j \leqslant q$), δ_j ($j < q$) and $((\delta_q)_i)_{i \leqslant r}$ verify equations (2.6) to (2.10). This is sufficient to prove correctness as seen in section 3 (where l is replaced by $l' = l + 1$) □

4.3 Practical remarks

Remark 4.3. Algorithm 4.2 can be used to compute the unique polynomial P of degree less than n such that l evaluation points are given by the vector B_1 and the coefficients of degree at least l are given by A_2: $\forall i < l, P(\omega^{[i]_v}) = b_i$ and $\forall i \geqslant l, P_i = a_i$. In particular, it can be used to interpolate a polynomial of degree less than l by setting $A_2 = (0, \ldots, 0)$.

Remark 4.4. In section 4.1, the order p may be composite (which can happen if an element of the vector v from Algorithm 4.2 is composite), but the resolution of equation (4.3) is not efficient if p is large. Algorithm 4.2 shows that the problem can be reduced to smaller sizes as long as its order is composite. However, it seems difficult to solve a skew butterfly problem if its size is a large prime.

For example, in $\mathbb{F}_{2^{60}}$, we have primitive roots of unity of order $2^{60} - 1 = 3^2 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 31 \cdot 41 \cdot 61 \cdot 151 \cdot 331 \cdot 1321$. It is feasible to perform the inversion using linear algebra for size up to 13 efficiently, but direct computation may become too costly for $p = 331$ or 1321 for example.

Remark 4.5. An inverse TFT of length 0 is actually very simple: it reduces to the computation of b_0. For this reason, if (l rem n_2) $= 0$, then the recursive call in Algorithm 4.2 at Step 3 becomes trivial. This partially solves the problem mentioned in the previous remark: assume the prime factors of n are sorted in the vector v (in increasing order). If for example an inversion by direct computation is possible for $p_0, \ldots, p_{k-1}$ but not for $p_k, \ldots, p_{d-1}$ (because these primes are too large), then a TFT of length l can still be reverted if $\tilde{n} = p_k \times \cdots \times p_{d-1}$ divides l.

Remark 4.6. It is important that the recursive calls in *Step 2* return the additional output value $(\beta_i)_q$. However, it is not necessary that Algorithm 4.2 always returns the additional output value b_l. For example, this value is simply discarded in the recursive calls from *Step 4*. Another typical case where this value is not needed is for the interpolation a polynomial of degree less than l from the l values $(\hat{A}_{[i]_v})_{i<l}$. It is possible to adapt Algorithm 4.2 to avoid this unnecessary computation when the value b_l is not needed.

4.4 A remarkable duality for atomic inverse transforms

Direct resolution of the *skew butterfly* problem from section 4.1 requires the inversion of the matrix $V_{\omega,l}$ of size $l \times l$, which becomes expensive if l is large. In this section, we present a dual problem that can be solved through the inversion of matrix $V_{\omega^{-1},p-l}$, that has size $(p-l)\times(p-l)$. This duality ensures that the *skew butterfly* problem can always be solved through the inversion of a matrix $V_{\phi,m}$ (and a few matrix-vector products), where $m \leqslant p/2$ and ϕ is either ω or ω^{-1}.

Property (2.11) gives the inverse matrix $(V_\omega)^{-1} = (1/p)\cdot V_{\omega^{-1}}$. Then, with a decomposition of $V_{\omega^{-1}}$ as in formula (4.1), equation (4.2) can be rewritten as follows:

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} = \frac{1}{p}\cdot \begin{pmatrix} V_{\omega^{-1},l} & W_{\omega^{-1},l} \\ W_{\omega^{-1},l}^\top & \tilde{V}_{\omega^{-1},l} \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}.$$

We want to solve the equation above with parameters A_2, B_1 and unknowns A_1, B_2. We have

$$B_2 = (\tilde{V}_{\omega^{-1},l})^{-1}\cdot(pA_2 - W_{\omega^{-1},l}^\top B_1)\,. \tag{4.5}$$

Once B_2 is known, it is easy to compute A_1 since

$$A_1 = \frac{1}{p}(V_{\omega^{-1},l}B_1 + W_{\omega^{-1},l}B_2)\,. \tag{4.6}$$

Except for the factor $1/p$, these formulas are symmetric to the formulas (4.3) and (4.4). This shows the duality between these two problems.

The initial claim was that the problem could be solved through the inversion of $V_{\omega^{-1},p-l}$, but equation (4.5) involves the matrix $\tilde{V}_{\omega^{-1},l}$. Actually, it appears that $\tilde{V}_{\omega^{-1},l}$ deduces from $V_{\omega^{-1},p-l}$ by multiplication with diagonal matrices. To reduce notations, let $\phi = \omega^{-1}$. We introduce the diagonal matrices $D_{\phi,l} = \mathrm{Diag}\left((\phi^{l(l+i)})_{i<p-l}\right)$ and $\tilde{D}_{\phi,l} = \mathrm{Diag}\left((\phi^{li})_{i<p-l}\right)$ of size $(p-l)\times(p-l)$. We have the following property:

$$(\tilde{V}_{\phi,l})_{i,j} = \phi^{(l+i)(l+j)} = \phi^{l(l+j)}\cdot\phi^{ij}\cdot\phi^{li} \quad \text{for all } i,j\,,$$

hence,

$$\tilde{V}_{\phi,l} = D_{\phi,l}\cdot V_{\phi,p-l}\cdot\tilde{D}_{\phi,l}\,. \tag{4.7}$$

Remark 4.7. The matrix $V_{\omega^{-1}}$ deduces from V_ω by a simple row (or column) permutation. For this reason, the dual problem presented here is equivalent (in terms of field operations) to the direct problem from section 4.1 up to scalar-vector multiplications and multiplication by diagonal matrices.

Remark 4.8. Assume that we have an efficient method to compute the function $X \to V_\phi\cdot X$. For large p, this can be done using Rader's [12] or Bluestein's [1] algorithm. Then, the required matrix-vector products (for example $W_{\omega^{-1},l}^\top\cdot B_1$) can be computed efficiently using the relation

$$\begin{pmatrix} \text{dummy} \\ W_{\omega^{-1},l}^\top\cdot B_1 \end{pmatrix} = V_{\omega^{-1}}\cdot\begin{pmatrix} B_1 \\ 0 \end{pmatrix}.$$

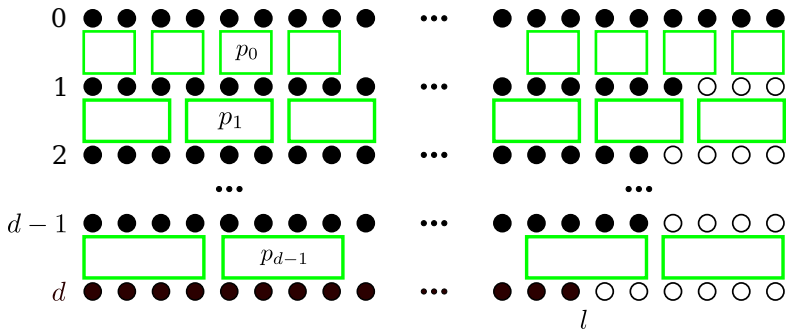

Figure 4: Successive transforms in the FFT/TFT algorithm after complete development of the recursive calls

5 Complexity analysis

This section aims to evaluate the field operation count for a TFT of size n and length $l \leqslant n$, and to compare it with the cost for a full FFT of size n. In the following, we note these costs $\mathsf{T}(l, n)$ and $\mathsf{F}(n)$ respectively. Asymptotic bounds involve the field operation count for common arithmetic operations on polynomials of degree n: $\mathsf{M}(n)$ for the multiplication and $\mathsf{C}(n)$ for the cyclic convolution (multiplication modulo $X^n - 1$).

Let $n = p_0 \cdots p_{d-1}$ be the size of the Discrete Fourier Transform (FFT or TFT) that is considered. If we develop completely the recursive calls of the FFT algorithm as in section 2.1, the execution decomposes into d successive transformations of a vector of length n (see Figure 4). At each row, the working vector is transformed using n/p_i independent DFTs of size p_i, which are computed directly.

5.1 Complexity of a full FFT

In a full FFT, all n/p_i atomic DFTs are computed at each row. If we note $\mathsf{f}(p) = \mathsf{F}(p)/p$ the normalized operation count (per intermediate value), then each of the atomic DFTs has an operation count of $p_i \mathsf{f}(p_i)$. Summing for all i, we get the following result:

Theorem 5.1 (Complexity of the usual Cooley-Tukey FFT)**.** *We have*

$$\mathsf{F}(n) = n \cdot (\mathsf{f}(p_0) + \mathsf{f}(p_1) + \cdots + \mathsf{f}(p_{d-1})) + O(nd)\,. \tag{5.1}$$

Remark 5.1. The term $O(nd)$ is the cost for the operations between rows of atomic DFTs, that is the multiplications by twiddle factors. The multiplicative constant is actually small; in fact, with a precomputed table of twiddle factors, this represents $n(d-1)$ multiplications, and even less if we take into account that some of the twiddle factors are equal to 1.

Remark 5.2. For small p, it is most efficient to compute the atomic DFTs using specialized codelets that perform naive matrix-vector products. This yields $\mathsf{F}(p) \sim p^2$, or $\mathsf{f}(p) \sim p$. For larger p, methods like Rader's or Bluestein's algorithms are more efficient. In this case, $\mathsf{F}(p) \sim p \log p$, or $\mathsf{f}(p) \sim \log p$ (with a larger constant factor than for the naive method).

5.2 Complexity of atomic TFTs

As discussed in section 3.1, there are two simple methods to compute an atomic TFT:

- for very small l, one can naively compute the l first values of the Fourier transform using Horner's rule (p additions and p multiplications for each evaluation point). This method has a cost of $\mathsf{T}(l,p) = 2lp$.
- for l near p, it is interesting to compute the full atomic DFT and discard the last $p-l$ values, at a cost of $\mathsf{T}(l,p) = \mathsf{F}(p)$.

Efficient methods for intermediate values of l are more complex. As an example, we sketch briefly here a method that is adapted for $p = O(l)$. This method is based on the following result from Bostan and Schost's work [2]:

Lemma 5.2. *Let $(1, \omega, \ldots, \omega^{l-1})$ be a geometric progression such that the points ω^i are pairwise distinct. Then, a polynomial of degree less than l can be evaluated on these points in $\mathsf{M}(l) + O(l)$ field operations.*

In our case, the points ω^i are indeed pairwise distinct because $l \leqslant p$ where p is the order of the root ω. Then, to evaluate the polynomial P of degree p, we write

$$P(X) = P_0(X) + X^l P_1(X) + \cdots + X^{ql} P_q(X) \quad \text{with } \deg P_i < l \text{ for all } i, \text{ and } q = p \text{ quo } l\,.$$

Then, the multipoint evaluation of P can be done by evaluating (p quo l) polynomials of degree less than l, followed by operations on vectors of size l (addition and multiplication by diagonal matrices) This method to compute atomic TFTs has an operation cost of

$$\mathsf{T}(l,p) = \frac{p}{l}\mathsf{M}(l) + O(p)\,.$$

As in the previous subsection, we normalize the operation count per intermediate value: $\mathsf{t}(l,p) = \mathsf{T}(l,p)/l$. We introduce the overhead for the atomic TFT as $\mathsf{k}(l,p) = \mathsf{t}(l,p)/\mathsf{f}(p)$. Since $\mathsf{k}(l,p)$ decreases with l, it is meaningful to also introduce

$$\mathsf{K}(p) = \sup_{l \leqslant p} l \cdot \mathsf{k}(l,p)\,.$$

By definition, we have:

$$\begin{aligned} \mathsf{t}(l,p) &= \mathsf{k}(l,p)\mathsf{f}(p)\,, \\ \mathsf{T}(l,p) = l \cdot \mathsf{t}(l,p) &\leqslant \mathsf{K}(p)\mathsf{f}(p)\,. \end{aligned} \tag{5.2}$$

Remark 5.3. Because $\mathsf{t}(l,p)$ and $\mathsf{f}(p)$ are normalized costs per evaluation point, it is clear that $\mathsf{t}(l,p) \geqslant \mathsf{f}(p)$, which means $\mathsf{k}(l,p) \geqslant 1$. The simple methods discussed before give upper bounds $\mathsf{k}(l,p) \leqslant 2p/\mathsf{f}(p)$ (using the naive method) and $\mathsf{k}(l,p) \leqslant p/l$ (compute the full DFT and discard unused values). This gives

$$\mathsf{k}(l,p) \leqslant \frac{p}{\max(\mathsf{f}(p)/2, l)}\,.$$

In particular, $l\,\mathsf{k}(l,p) \leqslant p$, which gives $\mathsf{K}(p) \leqslant p$.

Remark 5.4. Actually, the case $l = p$ corresponds to a full atomic DFT, so $\mathsf{T}(p,p) = \mathsf{F}(p) = p\,\mathsf{f}(p)$. Combined with the upper bound from the previous remark, we get $\mathsf{K}(p) = p$. However, we keep the notation $\mathsf{K}(p)$ to respect the symmetry with the inverse TFT.

5.3 Complexity of atomic inverse TFTs

Similarly, we introduce the corresponding costs for the inverse TFT: let $\mathsf{T}^*(l,p)$ be the cost for the inverse TFT, and $\mathsf{t}^*(l,p) = \mathsf{T}^*(l,p)/l$ is the normalized cost. The corresponding overhead is $\mathsf{k}^*(l,p) = \mathsf{t}^*(l,p)/\mathsf{f}(p)$ and $\mathsf{K}^*(p) = \sup_{l \leqslant p} l \cdot \mathsf{k}^*(l,p)$.

We know from sections 4.1 and 4.4 that the resolution of the *skew butterfly* problem requires a few matrix-vector products and the inversion of a matrix $V_{\phi,m}$, where $m \leqslant p/2$ and ϕ is either ω or ω^{-1}. Without loss of generality, we assume that $\phi = \omega$ and $m = l \leqslant p/2$ and we consider the direct method from section 4.1. (If $l > p/2$, then we reduce to the dual problem, which causes only $O(p)$ additional operations because of Remark 4.7.)

Actually, it is not necessary to compute $(V_{\omega,l})^{-1}$; it is sufficient to compute the function $Y \to (V_{\omega,l})^{-1}Y$, which is a polynomial interpolation on the points $1, \omega, \ldots, \omega^{l-1}$. The following result from Bostan and Schost [2, Section 5] gives an upper bound for the cost of this operation:

Lemma 5.3. *Let $(1, \omega, \ldots, \omega^{l-1})$ be a geometric progression such that the points ω^i are pairwise distinct. Then, the interpolation of a polynomial of degree less than l on these points can be performed in $2\mathsf{M}(l) + O(l)$ operations.*

Since $l \leqslant p/2$, a multiplication of size l can be seen as a cyclic convolution of size p, that is $\mathsf{M}(l) \leqslant \mathsf{C}(p)$. Moreover, Bluestein's transform is an efficient method to compute the DFT for a large p, which gives $\mathsf{F}(p) = \mathsf{C}(p) + O(p)$ asymptotically. We are now able to bound the cost of atomic inverse TFTs:

Lemma 5.4. *We have*

$$\mathsf{T}^*(l,p) \leqslant 4\mathsf{C}(p) + O(p) \quad \textit{for large } p\,.$$

Then,

$$\mathsf{K}^*(p) \leqslant 4p(1 + o(1))\,.$$

Proof. At first, we compute $Y = B_1 - W_{\omega,l}A_2$. The transformation $X \to V_\omega X$ is a DFT that can be performed in $\mathsf{C}(p) + O(p)$ operations (using for example Bluestein's transform). Then, using Remark 4.5, the matrix-vector product $W_{\omega,l}A_2$ can be computed using $\mathsf{C}(p) + O(p)$.

Then, we compute $A_1 = (V_{\omega,l})^{-1}Y$, which can be done using $2\mathsf{M}(l) + O(l)$ base field operations from Lemma 5.3; and we have $\mathsf{M}(l) \leqslant \mathsf{C}(p)$ because $l \leqslant p/2$.

Finally, the relation $B = V_\omega A$ allows to retrieve B_2 in $\mathsf{C}(p) + O(p)$ operations. □

5.4 Complexity of the TFT

When computing row $i+1$ from row i in the TFT, some of the atomic DFTs are not performed, most of the remaining are full DFTs and only a few are TFTs. More precisely, we can isolate the $i \to i+1$ transform as follows: using the notations from section 3, we split the TFT at $h = i+1$, then each of the n_2 inner TFTs are split at $h' = i$. This shows that the $i \to i+1$ transform consists in $\pi_i \lfloor l'/p_i \rfloor$ full DFTs and π_i atomic TFTs, where $\pi_i = p_{i+1} \cdots p_{d-1} (= n_2)$ and $l' = \lceil l/\pi_i \rceil$. The atomic TFTs have length $r_i = (l' \text{ rem } p_i)$.

As a consequence, the complexity of the complete TFT is given by

$$\mathsf{T}(l,n) = \sum_{i=0}^{d-1} \left(\pi_i \left\lfloor \frac{\lceil l/\pi_i \rceil}{p_i} \right\rfloor p_i \mathsf{f}(p_i) + \pi_i r_i \mathsf{t}(r_i, p_i) + O(\pi_i \lceil l/\pi_i \rceil) \right)\,.$$

Since $\lfloor l'/p_i \rfloor p_i + r_i = l'$ (euclidean division), this rewrites

$$\mathsf{T}(l,n) \leqslant \sum_{i=0}^{d-1} \left(\pi_i \left\lceil \frac{l}{\pi_i} \right\rceil \mathsf{f}(p_i) + O(\pi_i \lceil l/\pi_i \rceil) \right) + \sum_{i=0}^{d-1} \pi_i r_i (\mathsf{k}(r_i,p_i)-1)\mathsf{f}(p_i)\,.$$

We have $\lceil l/\pi_i \rceil \leqslant (l/\pi_i)+1$. Moreover, the term $O(\pi_i \lceil l/\pi_i \rceil) = O(l) + O(\pi_i)$ corresponds to the multiplication by twiddle factors at each step, as in Theorem 5.1. We then have the (otherwise abusive) simplification

$$\sum_{i=0}^{d-1} \big(l\,\mathsf{f}(p_i) + O(l) \big) = \frac{l}{n}\mathsf{F}(n)\,.$$

This yields

$$\mathsf{T}(l,n) \leqslant \frac{l}{n}\mathsf{F}(n) + \sum_{i=0}^{d-1} (\pi_i \mathsf{f}(p_i) + O(\pi_i)) + \sum_{i=0}^{d-1} \pi_i r_i (\mathsf{k}(r_i,p_i)-1)\mathsf{f}(p_i)$$

By definition (5.3), we have $r_i \mathsf{k}(r_i,p_i) \leqslant \mathsf{K}(p_i)$, and it is clear that $\mathsf{K}(p_i) \geqslant \mathsf{k}(1,p_i) \geqslant 1$. Then, handling the cases $r_i = 0$ and $1 \leqslant r_i \leqslant p_i - 1$ separately shows easily that $r_i(\mathsf{k}(r_i,p_i)-1)+1 \leqslant \mathsf{K}(p_i)$. Finally, using equation (5.1), we get the bound:

Theorem 5.5 (Complexity of the Truncated Fourier Transform (TFT))**.** *The Truncated Fourier Transform can be performed using*

$$\mathsf{T}(l,n) \leqslant \frac{l}{n}\mathsf{F}(n) + \sum_{i=0}^{d-1} (p_{i+1} \cdots p_{d-1}) \Big(\mathsf{K}(p_i)\mathsf{f}(p_i) + O(1) \Big) \tag{5.3}$$

field operations.

For the inverse TFT, all atomic inverse TFT (at a given row of atomic transform) do not necessarily have the same length. However the above reasoning still applies, and we get the bound:

Theorem 5.6 (Complexity of the Inverse Truncated Fourier Transform (ITFT))**.** *The Inverse Truncated Fourier Transform can be performed using*

$$\mathsf{T}^*(l,n) \leqslant \frac{l}{n}\mathsf{F}(n) + \sum_{i=0}^{d-1} (p_{i+1} \cdots p_{d-1}) \Big(\mathsf{K}^*(p_i)\mathsf{f}(p_i) + O(1) \Big) \tag{5.4}$$

field operations.

Remark 5.5. The overhead for atomic inverse transforms $\mathsf{K}^*(p)$ is larger than for atomic direct transforms $\mathsf{K}(p)$.

Remark 5.6. The bound (5.3) rewrites

$$\mathsf{T}(l,n) \leqslant \frac{l}{n}\mathsf{F}(n) + n \cdot \left(\sum_{i=0}^{d-1} \frac{\mathsf{K}(p_i)\mathsf{f}(p_i)}{p_0 \cdots p_i} \right) + O(n)\,.$$

Since $\mathsf{K}(p)\mathsf{f}(p)$ increases with p, this shows that it is best to sort the prime factors of n in increasing order ($p_0 \leqslant p_1 \leqslant \cdots \leqslant p_{d-1}$) to minimize the overhead. Moreover, assuming n is highly composite, $p_0 \cdots p_i$ grows much faster than $\mathsf{K}(p_i)\mathsf{f}(p_i)$, so that the predominant term in the linear factor is $\mathsf{K}(p_0)\mathsf{f}(p_0)/p_0$.

References

[1] L. Bluestein. A linear filtering approach to the computation of discrete fourier transform. *IEEE Transactions on Audio and Electroacoustics*, 18(4):451–455, Dec 1970.

[2] Alin Bostan and Éric Schost. Polynomial evaluation and interpolation on special sets of points. *Journal of Complexity*, 21(4):420 – 446, 2005.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. 19(90):297–301, April 1965.

[4] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Math. Comput.*, 62(205):305–324, January 1994.

[5] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[6] C. F. Gauss. Nachlass: Theoria interpolationis methodo nova tractata. In *Werke*, volume 3, pages 265–330. Königliche Gesellschaft der Wissenschaften, Göttingem, 1866.

[7] D. Harvey, J. van der Hoeven, and G. Lecerf. Fast polynomial multiplication over $\mathbb{F}_{2^{60}}$. In *Proc. ISSAC '16*, pages 255–262, New York, NY, USA, 2016. ACM.

[8] David Harvey. A cache-friendly truncated FFT. *Theoretical Computer Science*, 410(27):2649 – 2658, 2009.

[9] David Harvey and Daniel 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.

[10] J. Markel. FFT pruning. *IEEE Transactions on Audio and Electroacoustics*, 19(4):305–311, Dec 1971.

[11] Todd Mateer. *Fast Fourier Transform Algorithms with Applications*. PhD thesis, Clemson, SC, USA, 2008. AAI3316358.

[12] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proceedings of the IEEE*, 56(6):1107–1108, June 1968.

[13] A. Schönhage and V. Strassen. Fast multiplication of large numbers. *Computing*, 7(3):281–292, 1971.

[14] 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.