Truncated Fourier Transform

Mike Ghesquiere

December 15, 2012

Abstract

I outline the algorithm for the Truncated Fourier Transform from Joris van der Hoeven's paper: *The Truncated Fourier Transform and Application* making corrections and providing a more coherent alternative to the presented algorithm for its inverse.

1 Introduction

Let $\mathcal{R}$ be a ring. Additionally, we require that $\mathcal{R}$ be effective, that is, there are effective algorithms to determine $+$, $-$, and $\times$ and $1/2, 2 \in \mathcal{R}$.

Let ω be a primitive n-th root of unity with $n = 2^p$. If $\mathcal{R}$ does not admit such

2 Fast Fourier Transform

Given a $\mathcal{R}$ as above then, the Fast Fourier Transform with respect to ω is a morphism given by

$$\begin{aligned} FFT_\omega : \quad & \mathcal{R}^n \to \mathcal{R}^n \\ & (a_0, ..., a_{n-1}) \mapsto (\hat{a}_0, ..., \hat{a}_{n-1}) \end{aligned}$$

where $\hat{a}_i \in \mathcal{R}$ is given by $\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}$. We could also interpret $(a_0, ..., a_{n-1})$ as an element of $\mathcal{R}[X]$ by taking $A = a_0 + a_1 X + ... + a_{n-1} X^{n-1} \in \mathcal{R}[X]$. In this case $\hat{a_i} = A(\omega^i)$. Unlike $\mathcal{R}[X]$, multiplication in $\mathcal{R}^n$ can be done in linear time by componentwise multiplication. Provided an efficient algorithm for FFT and FFT^{-1}, this

To compute this efficiently, we use binary splitting $(a_0, a_1, ..., a_{n-1}) = (b_0, c_0, ..., b_{n/2-1}, c_{n/2-1})$

$$FFT_{\omega^2}(b_0, b_1, ..., b_{n/2-1}) = (\hat{b}_0, ..., \hat{b}_{n/2-1})$$
$$FFT_{\omega^2}(c_0, c_1, ..., c_{n/2-1}) = (\hat{c}_0, ..., \hat{c}_{n/2-1})$$

After splitting recursively, we recombine with:

$$\begin{aligned} FFT_\omega(a_0, ..., a_{n-1}) = \quad & \Big((\hat{b}_0 + \hat{c}_0), ..., (\hat{b}_{n/2-1} + \hat{c}_{n/2-1})\omega^{n/2-1} \\ & (\hat{b}_0 - \hat{c}_0), ..., (\hat{b}_{n/2-1} - \hat{c}_{n/2-1})\omega^{n/2-1}\Big) \end{aligned}$$

This requires $\mathcal{O}(n)$ multiplications and $\mathcal{O}(n)$ additions or subtractions per level with a total depth of $\mathcal{O}(p)$ giving a total cost of $\mathcal{O}(n \log n)$. FFT conceptually lends itself very well to a recursive algorithm (as covered in class) however in practice an in-place variant is more efficient.

We begin with $\vec{x}_0 = (x_{0,0}, ..., x_{0,n-1}) = (a_0, ..., a_{n-1})$ and define $m_s = 2^{p-s}$. Next, for all $s \in \{1, ..., p\}$, $i \in \{0, 2, 4, ..., n/m_s - 2\}$ and $j \in \{0, ..., m_s - 1\}$, we compute:

$$\begin{pmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{pmatrix} = \begin{pmatrix} 1 & \omega^{rev_s(i)m_s} \\ 1 & -\omega^{rev_s(i)m_s} \end{pmatrix} \begin{pmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{pmatrix} \tag{1}$$

Where $rev_p(i)$ is the bitwise reverse (van der Hoeven uses *mirror*) of i at length p. For example, $rev_6(22) = rev_6(010110_2) = 011010_2 = 26$.

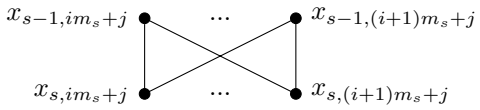

Figure 1: In a schematic representation for a FFT, each black dot represents some $x_{s,i}$ and each crossing or "butterfly" represents one matrix multiplication.

Using equation (1), we can push down $\vec{x}_i$ to $\vec{x}_{i+1}$, inductively obtaining $\vec{x}_p$ from $\vec{x}_0$. At each crossing we require one addition, one subtraction and one multiplication (the two products required differ only by sign). This construction is designed such that for $i \in \{0, ..., n/m - 1\}$ and $j \in \{0, ..., m - 1\}$:

$$x_{s,im_s+j} = (FFT_{\omega^{m_s}}(a_j, a_{m+j}, ..., a_{n-m+j}))_{rev_s(i)}$$

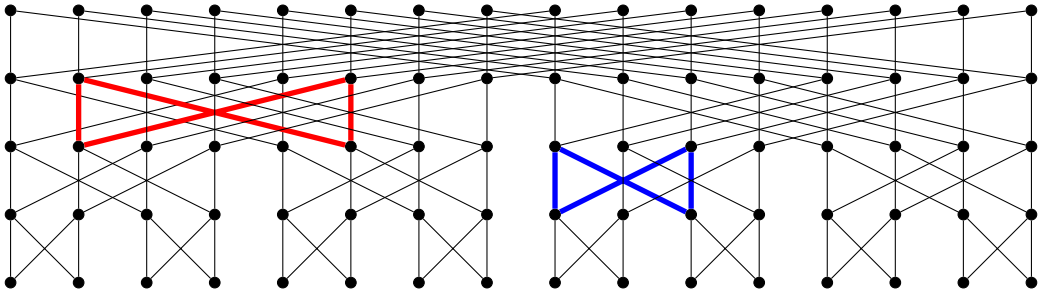

Figure 2: Schematic representation of FFT for $n = 16$. The top row being $(x_{0,0}, ..., x_{0,15}) = (a_0, ..., a_{15})$ and the bottom row being $(x_{4,0}, ...x_{4,15}) = (\hat{a}_{rev_5(0)}, \hat{a}_{rev_5(1)}, ..., \hat{a}_{rev_5(15)})$. Highlighted in red is the crossing for $s = 1, i = 0, j = 1$ and in blue, $s = 2, i = 4, j = 0$

.

In particular,

$$x_{p,i} = \hat{a}_{rev_p(i)}$$

Hence, computing $\vec{x}_p$ from $\vec{x}_0$ is equivalent to computing FFT_ω. The algorithm for the inverse FFT is identical since $(FFT_\omega)^{-1} = FFT_{\omega^{-1}}$.

3 Truncated Fourier Transform

The standard Fast Fourier Transform requires n to be a power of two making is clumsy for vectors of length $n + \delta$. Although one could simply carry out the FFT at precision $2n$, this causes "jumps" in complexity which we aim to eliminate with the *Truncated Fourier Transform* (TFT).

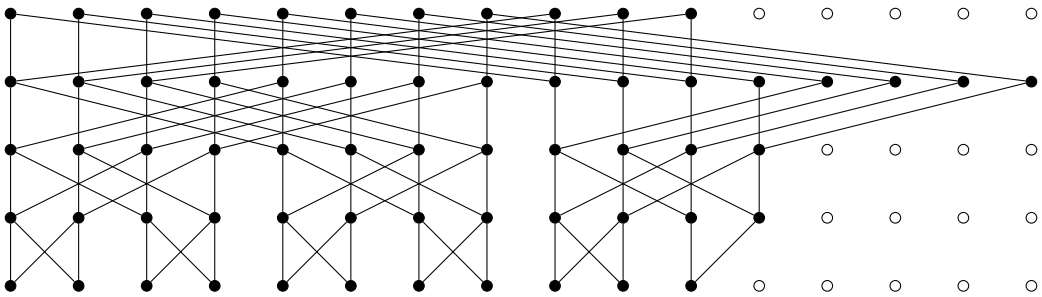

Figure 3: Schematic representation of FFT for $n = 16$. The top row being $(x_{0,0}, ..., x_{0,15}) = (a_0, ..., a_{15})$ and the bottom row being $(x_{4,0}, ...x_{4,15}) = (\hat{a}_{rev_5(0)}, \hat{a}_{rev_5(1)}, ..., \hat{a}_{rev_5(15)})$. Highlighted in red is the crossing for $s = 1, i = 0, j = 1$ and in blue, $s = 2, i = 4, j = 0$

Let $n = 2^p$, $n/2 \geq l < n$ and let ω be a primitive n-th root of unity. Given $(a_0, ..., a_{l-1}) \in \mathcal{R}^l$. We consider FFT on the completion of $(a_0, ..., a_{l-1})$ into $(a_0, ..., a_{l-1}, 0, ...0) \in \mathcal{R}^n$. We claim that many $x_{s,i}$ can be skipped and that it is sufficient to calculate $(x_{s,0}, ..., x_{s,(\lceil l/m_s \rceil m_s)-1})$.

Conceptually, this equates to rounding up to the next multiple of m_s for each $\vec{x}_s$. Since every term in the vector past $(\lceil l/m_s \rceil m_s) - 1$ is in a seperate 'branch' it will have no effect on $(x_{p,0}, ..., x_{p,l-1})$. With the exception of this truncation of $\vec{x}_s$, the algorithm for the TFT is identical to that of the in-place FFT.

4 Inverse Truncated Fourier Transform

Recall the matrix used to compute the TFT was of the form:

$$\begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} 1 & \alpha \\ 1 & -\alpha \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

However since this is invertible, we can of course compute (c, d) from (a, b):

$$\begin{pmatrix} a \\ b \end{pmatrix} = 2^{-1} \begin{pmatrix} 1 & 1 \\ \alpha^{-1} & -\alpha^{-1} \end{pmatrix} \begin{pmatrix} c \\ d \end{pmatrix} \tag{2}$$

Furthermore, we also have:

$$\begin{pmatrix} d \\ a \end{pmatrix} = \begin{pmatrix} 1 & -2\alpha \\ 1 & -\alpha \end{pmatrix} \begin{pmatrix} c \\ b \end{pmatrix} \tag{3}$$

$$\begin{pmatrix} c \\ b \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ -\alpha^{-1} & \alpha^{-1} \end{pmatrix} \begin{pmatrix} d \\ a \end{pmatrix} \tag{4}$$

Similar equations also exist for computing (a, c) from (b, d) and vice versa, although these are not required for our algorithm. Hence, given any two values in a butterfly crossing, we can compute the other two. Moreover, all of these equations involve only additions, subtractions, multiplications and divisions by two (shifts in practice), and multiplications by roots of unity. This suggests that not only can all of the computations required for the TFT be easily reversed

but it should be possible in similar time to the forward TFT. Our algorithm for the *Inverted Truncated Fourier Transform* ($ITFT$) will rely on attempting to reconstruct the graph used for computing the original TFT.

Given an input vector $(a_0, ..., a_{l-1})$ we select select a minimal n such that $n \geq l$ and $n = 2^p$ for some p. We then construct a graph of width n and height $p+1$. We will aim to maintain this 'aspect ratio' at all points in our algorithm. Finally we initialize $(x_{p,0}, ..., x_{p,l-1})$ to $(a_0, ..., a_{l-1})$ and $(x_{0,l}, x_{0,n-1})$ to $(0, ...0)$. We then call $ITFT$ with these two vectors as input.

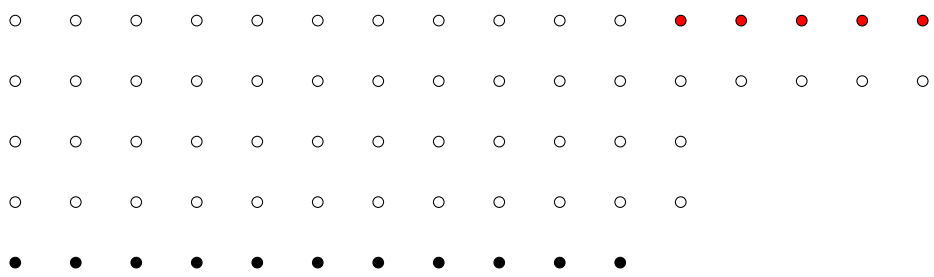

Figure 4: Initial state of the algorithm with input of size 11. Unknown values are given in white while our input vector is given in black and $\vec{b}$ given in red.

$ITFT$ will expect two vectors: $\vec{a} = (x_{p,k}, ..., x_{p,l-1})$ and $\vec{b} = (x_{s,l}, ..., x_{s,(\lceil l/m_s \rceil m_s)-1})$ and will return $(x_{s,k}, ..., x_{s,l-1})$. The points $x_{p,k}$ and $x_{s,(\lceil l/m_s \rceil m_s)-1})$ define the bounds of our current cell of operation. The algorithm then decomposes into two cases based on the relative lengths of $\vec{a}$ and $\vec{b}$.

Case 1: length$(\vec{a}) \geq$ length$(\vec{b})$

Select i such that $2^i \leq l - k$ is maximal. That is, the largest power of two which is less than the length of $\vec{a}$. We then "push up" the vector $(x_{p,k}, ..., x_{p,k+2^i-1})$ using Equation (2), we obtain $\vec{c} = (x_{(s+1),k}, ..., x_{(s+1),k+2^i-1})$. Given that our aspect ratio is preserved, it will always be possible to push up to $s+1$ since length$(\vec{a}) \geq$ length$(\vec{b})$ implies that length$(\vec{a})$ is atleast half the width of our current cell.

If the length of $\vec{b}$ is the same as the length of $\vec{c}$, then we apply Equation (3) to obtain $(x_{s,k}, ..., x_{s,l-1})$ and return. If not then we "push down" $\vec{b}$ using Equation (3) and the end of $\vec{c}$. We recursively call $ITFT$ on $(x_{p,k+2^i}, ..., xp, l-1)$ and $(x_{(s+1),l}, ..., x_{(s+1),(\lceil l/m_s \rceil m_s)-1})$ to obtain $(x_{(s+1),k+2^i}, ..., x_{(s+1),l-1})$. Note that this recursive call creates a cell with half

the width and depth one less that of the current cell, thus preserving our cell aspect ratio. Finally, we use this result and $\vec{c}$ with Equation (2) to compute $(x_{s,k}, ..., x_{s,l-1})$.

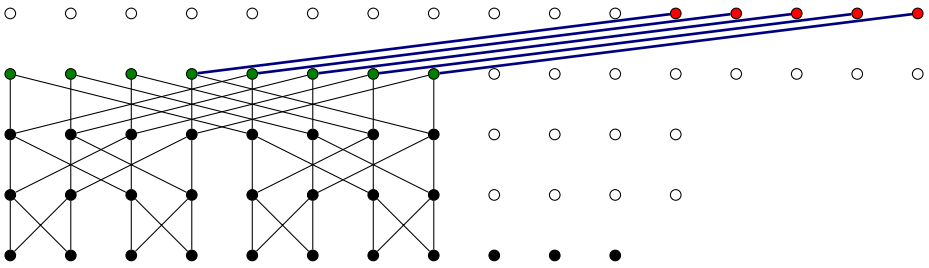

Figure 5: The result of pushing $(x_{4,0}, ..., x_{4,7})$ up to $\vec{c} = (x_{1,0}, ..., x_{1,7})$ (highlighted in green). Blue edges show where $\vec{c}\,\vec{b}$ are paired to compute (in this case), the vector $(x_{1,11}, ..., x_{1,15})$.

Case 2: length$(\vec{a})$ < length$(\vec{b})$

If the length of $\vec{a}$ is less than the length of $\vec{b}$, then there must exist at least one pair of points in $\vec{b}$ that can be pushed down with Equation (1). Hence, we will push down $\vec{b}$ to $\vec{c} = (x_{(s+1),l}, ..., x_{(s+1),(\lceil l/m_{(s+1)} \rceil m_{(s+1)})-1})$. We know that length$(vecc)$ = length$(\vec{b} - m_{s+1}$ since by moving down a step, we have increased our precision by a factor of two. Given that $\vec{a}$ is less than half the width of our cell, the right hand bound of our cell must also decrease.

Next we recurse with inputs $\vec{a}$ and $\vec{c}$ to calculate $(x_{(s+1),k}, ..., x_{(s+1),l-1})$. Again with a cell of half the width and depth minus one of our current cell; thus preserving aspect ratio. We then use this result in combination with Equation (5) and $\vec{b}$ to compute $(x_{s,k}, ..., x_{s,l-1})$.

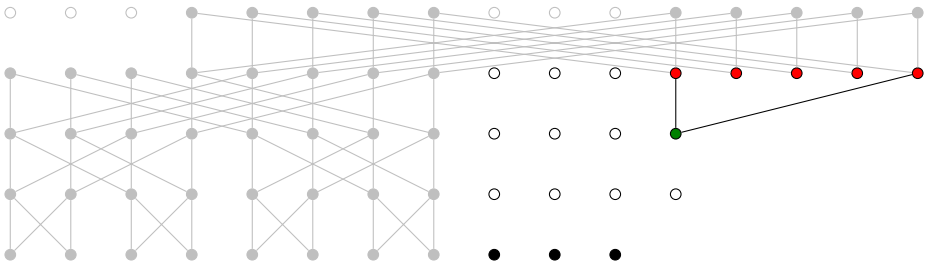

Figure 6: After pushing down $(x_{0,11}, ..., x_{0,15})$, the length of $\vec{b} = (x_{1,11}, ..., x_{1,15})$ is greater than the length of $\vec{a} = (x_{4,8}, ..., x_{4,11})$. Hence we use "push down" $\vec{b}$ to $(x_{2,11})$ and call $ITFT$ on $\vec{a}$ and $(x_{2,11})$.

The algorithm for $ITFT$ requires an equal number of comparable matrix multiplications to

the forward TFT. Although there is some marginal extra work to determine which $x_{s,i}$ should be computed next, the complexity of TFT compared to $ITFT$ are indistinguishable.

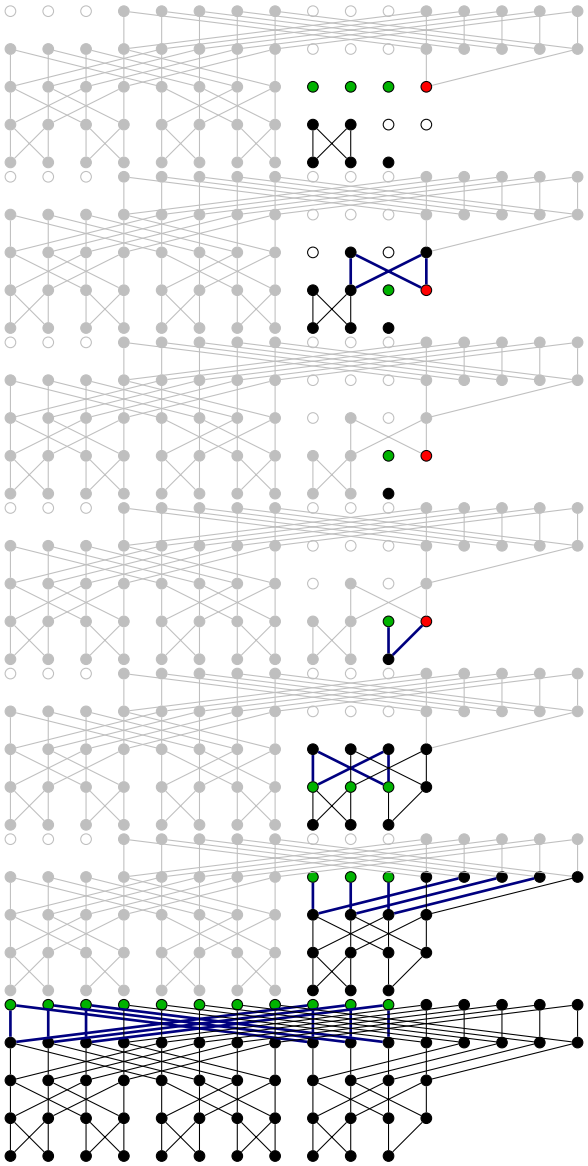

Figure 7: The conclusion of the example from Figure 4 through 6.