The Truncated Fourier Transform

Course Project
CS 9566A

Paul Vrbik

250389673

December 11, 2009

Abstract

I summarize (and correct some mistakes from) Joris van der Hoeven's papers [2] and [3]. These papers introduce the Truncated Fourier Transform (TFT) which is a variation of the Fast Fourier Transform (FFT) that allows one to work with input vectors that are *not* a power of two.

I also expand upon the development of the inverse TFT in order to impose some clarity on van der Hoeven's descriptions.

1 Introduction

Let $\mathcal{R}$ be an *effective ring* of constants (i.e. there are effective procedures for computing $+$, $-$ and $\times$ of two elements in $\mathcal{R}[x]$). If $\mathcal{R}$ has a primitive nth root of unity ω with $n = 2^p$ (i.e. $\omega^{n/2} = -1$) then we can use the Fast Fourier Transform (FFT) to compute the product of two polynomials $P, Q \in \mathcal{R}[x]$ with $\deg(PQ) < n$ in $O(n \log n)$. However, when $\deg(PQ)$ is sufficiently far from a power of two we can show that *many* computations are done to establish evaluations points that we ultimately do not need.

I outline the FFT, including a method for its a non-recursive implementation, and then develop a variant called the Truncated Fourier Transform (TFT). Finally I show how the TFT can be inverted.

2 The Fast Fourier Transform

Let $\mathcal{R}$, n, and ω be given as in the introduction. The discrete Fast Fourier Transform (FFT) — with respect to ω — of an n-tuple $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \mathcal{R}^n$ is the n-tuple $\hat{\mathbf{a}} = (\hat{a}_0, \ldots, \hat{a}_{n-1}) \in \mathcal{R}^n$ with

$$\hat{a}_i = \sum_{j=0}^{n-1} a_j \omega^{ij}.$$

Alternatively we can interpret these n-tuples as coefficients of polynomials in $\mathcal{R}[x]$ and define the FFT as the mapping which takes $A = a_0 + a_1 x + \cdots + a_{n-1} x^{n-1}$ to the n-tuple $(A(\omega^0), \ldots, A(\omega^{n-1}))$.

The FFT can be computed efficiently using binary splitting. By doing this we only evaluate at ω^{2^i} for $i \in \{0, \ldots, p-1\}$, rather than at all $\omega^0, \ldots, \omega^{n-1}$. To compute the FFT of $\mathbf{a}$ with respect to ω we write

$$a_0, \ldots, a_{n-1} = (b_0, c_0, \ldots, b_{n/2-1}, c_{n/2-1})$$

and recursively compute the Fourier transform of $(b_0, \ldots, b_{n/2-1})$ and $(c_0, \ldots, c_{n/2-1})$ at ω^2:

$$\begin{aligned} \mathrm{FFT}_{\omega^2}(b_0, \ldots, b_{n/2-1}) &= (\hat{b}_0, \ldots, \hat{b}_{n/2-1}); \\ \mathrm{FFT}_{\omega^2}(c_0, \ldots, c_{n/2-1}) &= (\hat{c}_0, \ldots, \hat{c}_{n/2-1}). \end{aligned}$$

Finally we construct $\hat{\mathbf{a}}$ according to

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

The polynomial interpretation of this procedure instead splits A into its even and odd parts, evaluates these parts at ω^2 and then reconstructs to retrieve $\hat{A}$. Of course the two methods are equivalent.

Clearly this description has a natural implementation as a recursive algorithm; but, in practice it is more efficient to implement an in-place algorithm that eliminates the overhead of creating recursive stacks.

Definition 1. We denote by $[i]_p$ the bitwise reverse[1] of i at length p. Suppose $i = i_0 2^0 + \cdots + i_p 2^p$ and $j = j_0 2^0 + \cdots + j_p 2^p$ then

$$[i]_p = j \iff i_k = j_{p-k} \text{ for } k \in \{0, \ldots, p\}.$$

Example 1.

$[3]_5 = 24$ because $3 = 00011_2$ whose reverse is $11000_2 = 24$.

$[11]_5 = 26$ because $11 = 01011_2$ whose reverse is $11010_2 = 26$.

The non-recursive (in-place) algorithm only requires *one* vector of length n. Initially, at step zero we start with the vector

$$\mathbf{x_0} = (x_{0,0}, \ldots, x_{0,n-1}) = (a_0, \ldots, a_{n-1})$$

and update this vector at step $s \in \{1, \ldots, p\}$ by the rule

$$\begin{bmatrix} x_{s,im_s+j} \\ x_{s,(i+1)m_s+j} \end{bmatrix} = \begin{bmatrix} 1 & \omega^{[i]_s m_s} \\ 1 & -\omega^{[i]_s m_s} \end{bmatrix} \begin{bmatrix} x_{s-1,im_s+j} \\ x_{s-1,(i+1)m_s+j} \end{bmatrix} \tag{1}$$

for all $i \in \{0, \mathbf{2}, \ldots, n/m_s - 2\}$ and $j \in \{0, \ldots, m_s - 1\}$, where $m_s = 2^{p-s}$.

Equation (1), being a relation among four values at two steps s and $s-1$, can be illustrated as in Figure 2. We call this relation a "butterfly" after the shape it forms. We may say that m_s controls the width of this butterfly — the value of which decreases as s increases. Note that two

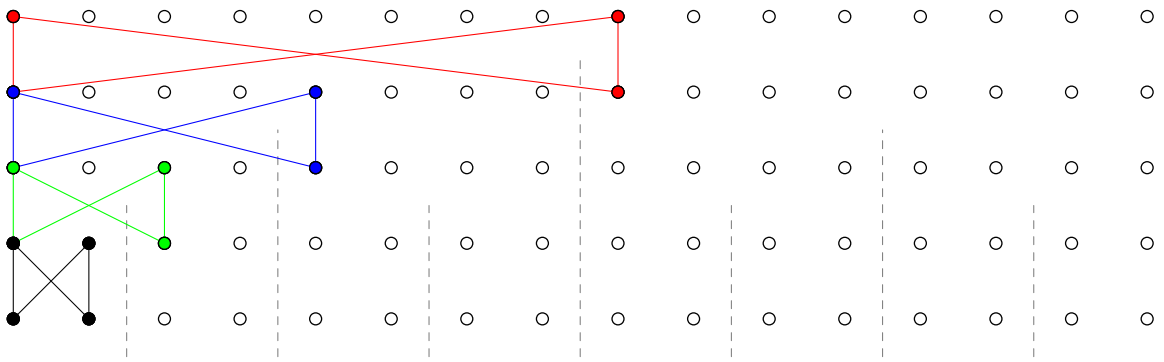


Figure 1: Butterflies. Schematic representation of Equation (1). The black dots correspond to the $x_{s,i}$. The top row corresponding to $s = 0$. In this case $n = 16 = 2^4$.

[1] In [2] the word "mirror" instead of reverse is used which I feel can lead to some ambiguity.

additions and *one* multiplication are done in Equation (1) as one product is merely the negation of the other.

Using induction over s, it can be easily shown [2] that

$$x_{s,im_s+j} = (\mathrm{FFT}_{\omega^{m_s}}(a_j, a_{m_s+j}, \ldots, a_{n-m_s+j}))_{[i]_s},$$

for all $i \in \{0, \ldots, n/m_s - 1\}$ and $j \in \{0, \ldots, m_s - 1\}$. In particular, when $s = p$ and $j = 0$ we have

$$x_{p,i} = \hat{a}_{[i]_p}$$
$$\hat{a}_i = x_{p,[i]_p}$$

for all $i \in \{0, \ldots, n-1\}$. That is, $\hat{\mathbf{a}}$ is a permutation of $\mathbf{x_p}$ as illustrated in Figure 2

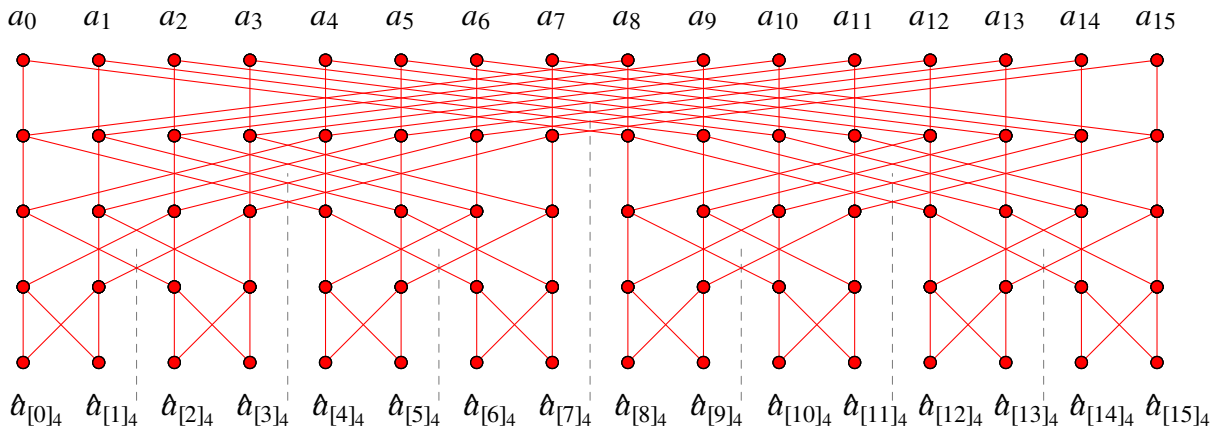


Figure 2: The Fast Fourier Transform for $n = 16$. The top row, corresponding to $s = 0$, represents the values of $\mathbf{x_0}$. The bottom row, corresponding to $s = 4$ is some permutation of $\hat{\mathbf{a}}$ (the result of the FFT on $\mathbf{a}$).

One nice feature of the FFT is that it is straightforward to recover $\mathbf{a}$ from $\hat{\mathbf{a}}$

$$\mathrm{FFT}_{\omega^{-1}}(\hat{\mathbf{a}})_i = \mathrm{FFT}_{\omega^{-1}}(\mathrm{FFT}_{\omega}(\mathbf{a}))_i = \sum_{k=0}^{n-1}\sum_{j=0}^{n-1} a_i \omega^{(i-k)j} = n a_i \tag{2}$$

since

$$\sum_{j=0}^{n-1} \omega^{(i-k)j} = 0$$

whenever $i \neq k$. This yields a polynomial multiplication algorithm of time complexity $O(n \log n)$ in $\mathcal{R}[x]$. For the sake of brevity I will refer the reader to [1] for the outline of this algorithm.

I give a simple MAPLE implementation of the in-place FFT and inverse FFT algorithms. First is sample output (the procedure listings follow).

```
[>F:=randpoly(x,coeffs=rand(1..1));
```

$$1 + x^5 + x^4 + x^3 + x^2 + x$$

```
[>as:=myFFT(F,3):
[>invFFT(as,3);
```

$$1 + x^5 + x^4 + x^3 + x^2 + x$$

In-place Fast Fourier Transform and Inverse Transform

```
h_myFFT::list(complex):=proc(xs::Array(complex),w::complex,p::nonnegint)
#intput      ::  xs[0..n-1]
#            ::  w - complex number such that w^(2^p) = 1
#output     ::  the FFT
local x,xp,s,indx,wn,n,ms,i,j;
global NumOps, SaveA;

    n:=ArrayNumElems(xs); #:=2^p
    if n <> 2^p then error ``something wrong"; end if;

    x:=Array(0..n-1,datatype=complex); xp:=Array(0..n-1,datatype=complex);
    x:=xs;

    for s from 1 to p do
        ms:=2^(p-s);
        for i from 0 to n/ms-2 by 2 do
            wn:=w^(BinFlip(i,s)*ms);
            for j from 0 to ms-1 do
                indx:=i*ms+j;
                xp[indx]    :=evalc(x[indx]+wn*x[indx+ms]);
                xp[indx+ms] :=evalc(x[indx]-wn*x[indx+ms]);
            end do;
        end do;
        (x,xp):=(xp,x); #this swaps pointers
    end do;

    for i from 0 to n-1 do
        xp[i]:=evalc( simplify( x[ BinFlip(i,p) ] ) );
    end do;

    return convert(xp,list);
end:

myFFT::list(complex):=proc(F,p::nonnegint)
local x,n,a;
    x:=indets(F)[1];
    n:=2^p;
    a:=Array(0..n-1, [ seq(coeff(F,x,i), i=0..degree(F)), 0$(n-degree(F))]);
    return h_myFFT(a, evalc(exp(2*I*Pi/n)) ,p);
end proc:

invFFT:=proc(xs::list(complex),p::nonnegint)
local n,w,as;

    n:=2^p;
    w:=exp(2*I*Pi/n);

    as:=h_myFFT( Array(0..n-1, xs),evalc(1/w), p);

    add( 1/n*as[i+1]*x^i, i=0..n-1 );
end proc:
```

3 The Truncated Fourier Transform

The motivation behind the Truncated Fourier Transform is the observation that many computations are wasted when the length of $\mathbf{a}$ (the input) is not a power of two. This is entirely the fault of the strategy where one "completes" the ℓ-tuple $\mathbf{a} = (a_0, \ldots, a_{\ell-1})$ by setting $a_i = 0$ when $i \geq \ell$ to artificially extend the length of $\mathbf{a}$ to the nearest power of two. Now the FFT can be executed as usual.

However, despite the fact that we may only want ℓ components of $\hat{\mathbf{a}}$, the FFT will calculate *all* of them. Thus computation is wasted. I illustrate this in Figures 3 and 4. This type of wasted calculation is relevant when using the FFT to multiply polynomials — their products are rarely of degree a power of two.

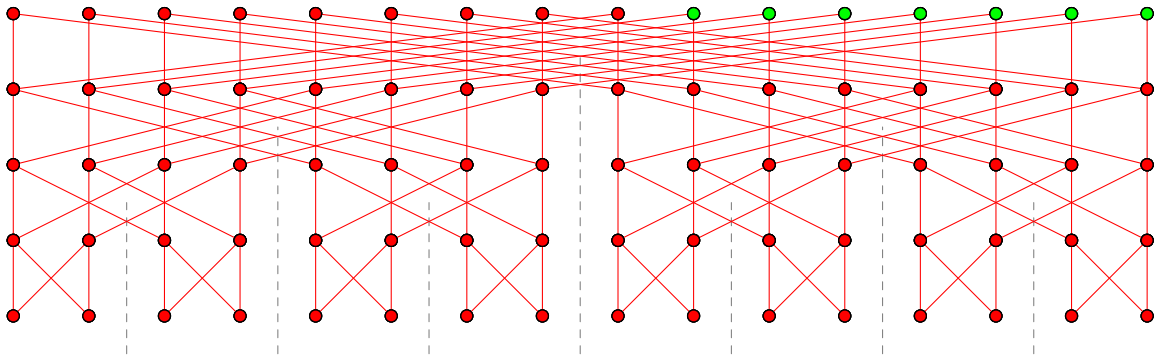


Figure 3: The FFT with "artificial" zero points (green).

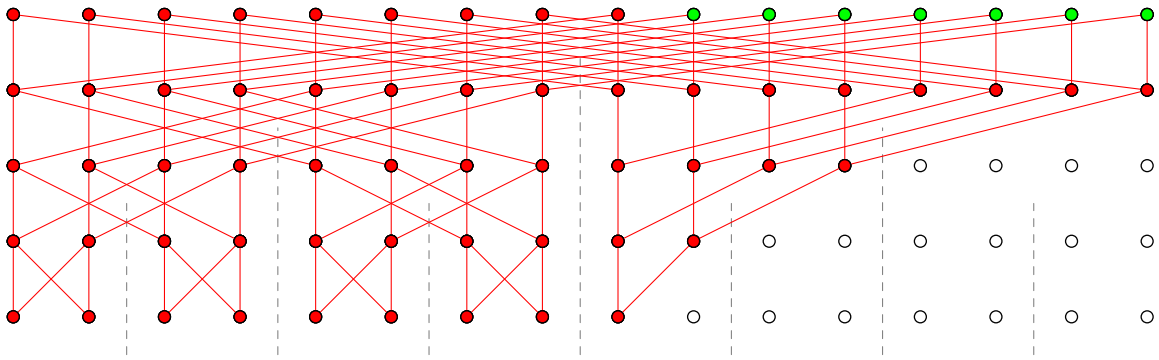


Figure 4: Removing all unnecessary computations from Figure 3 gives the schematic representation of the TFT.

The definition of the TFT is similar to that of the FFT with the exception that the input and output vector ($\mathbf{a}$ resp. $\hat{\mathbf{a}}$) are not necessarily of length some power of two. More precisely the TFT of an ℓ-tuple $(a_0, \ldots, a_{\ell-1}) \in \mathcal{R}^\ell$ is the ℓ-tuple

$$\left(A(\omega_p^{[0]},) \ldots, A(\omega^{[\ell-1]_p})\right) \in \mathcal{R}^\ell.$$

where $n = 2^p$, $\ell < n$ (usually $\ell \geq n/2$) and ω a n-th root of unity.

Remark 1. van der Hoeven gives a more general description of the TFT where one can chose an initial vector $(x_{0,i_0}, \ldots, x_{0,i_n})$ and target vector $(x_{p,j_0}, \ldots, x_{p,j_n})$. Provided that the i_k's are distinct one can (supposedly) carry out the TFT by considering the full FFT and removing all computations not required for the desired output. As I am ignorant to some sufficiently fast way of determining this subgraph, I restrict my discussion to that of the scenario in Figure 4 (where the input and output are the same initial segments).

It is actually straightforward to modify the in-place algorithm from the previous section to instead execute the TFT. At stage s it suffices to compute

$$(x_{s,0}, \ldots, x_{s,j}) \text{ with } j = (\lfloor (\ell - 1)/m_s \rfloor + 1)m_s - 1$$

where $m_s = 2^{p-s}$.[2]

Theorem 1. *Let $n = 2^p$, $\ell < n$ and $\omega \in \mathcal{R}$ be a primitive n-th root of unity in $\mathcal{R}$. Then the TFT of an ℓ-tuple $(a_0, \ldots, a_{\ell-1})$ w.r.t. ω can be computed using at most $\ell p + n$ additions and $\lfloor (\ell p + n)/2 \rfloor$ multiplications with powers of ω.*

Proof. Let $j = (\lfloor (\ell - 1)/m_s \rfloor + 1)m_s - 1$, at stage s we compute $(x_{s,0}, \ldots, x_{s,j})$. So, in addition to $x_{s,0}, \ldots, x_{s,\ell-1}$ we compute

$$(\lfloor (\ell - 1)/m_s \rfloor + 1)m_s - \ell \leq 1 + m_s - \ell \leq m_s$$

more values. Therefore, in total we compute at most

$$p\ell + 2^{p-1} + 2^{p-2} + \cdots + 1 < p\ell + n$$

values $x_{s,i}$. The result follows from this. □

We can make a trivial change to the FFT implementation to get another for the TFT. This is given on the following page.

4 Inverting The Truncated Fourier Transform

Unfortunately, the inverse TFT cannot be inverted by merely doing another TFT with $1/\omega$ and adjusting the output by some constant factor (like the FFT). Simply put: we are missing information and have to account for this.

Example 2. Let $\mathcal{R} = \mathbb{Z}/17\mathbb{Z}$, $n = 2^2 = 4$, with $\omega = 4$ a n-th primitive root of unity. The TFT of $\mathbf{a} = (a_0, a_1, a_2)$ is

$$\begin{aligned}\big(A(\omega^0), A(\omega^2), A(\omega^1)\big) &= (A(1), A(-1), A(3)) \\ &= (a_0 + a_1 + a_2, a_0 - a_1 + a_2, a_0 + 3a_1 + 9a_2).\end{aligned}$$

Now to show that the TFT of this w.r.t. $1/\omega$ is *not* $\mathbf{a}$ define

$$\mathbf{b} = (b_0, b_1, b_2) = (a_0 + a_1 + a_2, a_0 - a_1 + a_2, a_0 + 3a_1 + 9a_2).$$

[2] This is a correction to the bound given in [3].

In-place Trucnated Fast Fourier Transform

```
h_myTFT::list(complex):=proc(xs::Array(complex),w::complex,p::nonnegint,l::nonnegint)
#intput      ::  xs[0..n-1]
#            ::  w - complex number such that w^(2^p) = 1
#output     ::  the TFT (with indices unshuffled)
local x,xp,s,indx,wn,n,ms,i,j,uBound;
global NumOps;

    n:=2^p;

    x:=xs;  xp:=Array(0..n-1,datatype=complex);
    for s from 1 to p do
        ms:=2^(p-s);
        uBound:=( trunc( (l)/ms )+1 )*ms - 1;
        for i from 0 to n/ms-2 by 2 do
            wn:=w^(BinFlip(i,s)*ms);
            for j from 0 to ms-1 do
                indx:=i*ms+j;
                if indx > uBound then break; end if;
                xp[indx]     :=evalc(x[indx]+wn*x[indx+ms]);
            end do;

            for j from 0 to ms-1 do
                indx:=i*ms+j;
                if indx > uBound then break; end if;
                xp[indx+ms] :=evalc(x[indx]-wn*x[indx+ms]);
            end do;
        end do;
        (x,xp):=(xp,x); #this swaps pointers
    end do;

    return [ seq( evalc(simplify(x[i])), i=0..l-1) ];
end:

myTFT::list(complex):=proc(F,p::nonnegint)
local x,n,a;
    x:=indets(F)[1];
    n:=2^p;
    a:=Array(0..n-1, [ seq(coeff(F,x,i), i=0..degree(F)), 0$(n-degree(F))]);
    return h_myTFT(a, evalc(exp(2*I*Pi/n)), p, degree(F,x)+1 );
end proc:
```

The TFT of $\mathbf{b}$ w.r.t $1/\omega = -4$ is

$$\begin{aligned}\left(B\left(\omega^0\right), B\left(\omega^{-2}\right), B\left(\omega^{-1}\right)\right) &= (B(1), B(-1), B(-4)) \\ &= (b_0 + b_1 + b_2, b_0 - b_1 + b_2, b_0 - 4b_1 - b_2) \\ &= (3a_0 + 3a_1 + 11a_2, a_0 + 5a_1 + 9a_2, -4a_0 + 2a_1 + 5a_2)\end{aligned}$$

which is not some constant multiple of $\mathrm{TFT}_\omega(\mathbf{a})$.

This discrepancy is caused by the completion of $\mathbf{b}$ to $(b_0, b_1, b_2, 0)$. In fact we should instead be completing b to $(b_0, b_1, b_2, a_0 - 3a_1 + 9a_2)$ to correspond to the FFT of $\mathbf{a}$ w.r.t ω.

Essentially to invert the TFT we follow the paths from $\mathbf{x_p}$ back to $\mathbf{x_0}$. We will use the fact that whenever one value among

$$x_{s,im_s+j}, x_{s-1,im_s+j}$$

and one value among

$$x_{s,(i+1)m_s+j}, x_{s-1,(i+1)m_s+j}$$

are known then we can deduce the other values. That is, if two values of some butterfly are known then the other two values can be calculated using Equation (1) as the relevant matrix is invertible. Moreover, these relations only involve shifting (multiplication and division by two), additions, subtractions and multiplications by roots of unity — an ideal scenario for implmentation.

The key observation is: given $x_{p,0}, \ldots, x_{p,2^k-1}$ that $x_{p-k,0}, \ldots, x_{p-k,2^k-1}$ can be established. This is because all the butterfly relations necessary to move up like this never require $x_{s,2^k+j}$ for any $s \in \{p-k, \ldots, p\}$, $j > 0$. This is illustrated in Figure 5. More generally we have that

$$x_{p,2^j+2^k}, \ldots, x_{p,2^j+2^k-1}$$

is sufficient information to compute

$$x_{p-k,2^j}, \ldots, x_{p-k,2^j+2^k-1}$$

provided that $0 < k \leq j < p$.

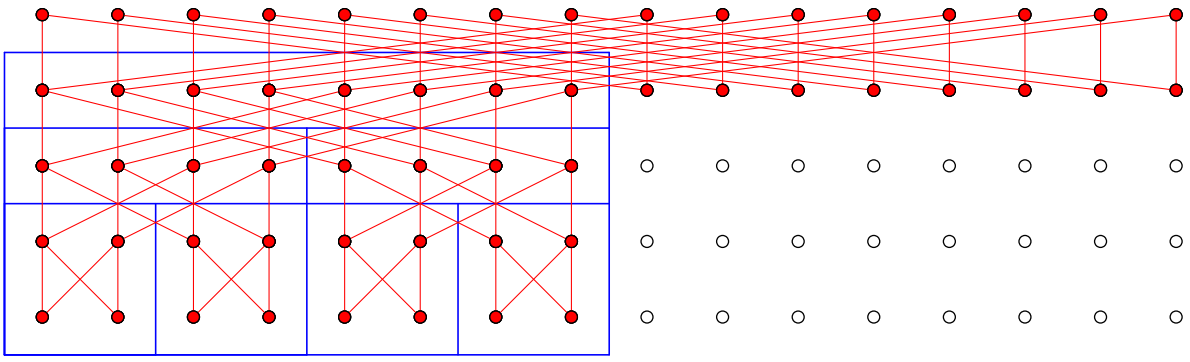


Figure 5: The computations in the boxes are self contained.

The algorithm for the inverse TFT will *initially* take as input the segment $x_{p,0}, \ldots, x_{p,m_s-1}$ with $m_s > n/2$. We will assume that this vector is the result of the TFT with $x_{0,m_s+j} = 0$ for $j > 0$. The output will be $x_{0,0}, \ldots, x_{0,m_s-1}$ (the original points $\mathbf{a}$).

More precisely, denote $k_s = \lfloor \ell/m_s \rfloor$ and $l_s = k_s + m_s$ at each stage s. The recursive algorithm will take the values

$$x_{p,k_s}, \ldots, x_{p,\ell-1} \text{ and (possibly empty) } x_{s,l}, \ldots, x_{s,\ell_s}$$

on input, and return $x_{s,k_s}, \ldots, x_{s,\ell-1}$. If $s = p$ then there is nothing to be done. Otherwise,

if $\ell_s = \ell_{s+1}$ then

1. compute $x_{s+1,k_s}, \ldots, x_{s+1,k_{s+1}-1}$ from $x_{p,k_s}, \ldots, x_{p,k_{s+1}-1}$ using repeated crossings (i.e. "push up").
2. compute $x_{s,i}$ and $x_{s+1,i+m_s/2}$ from $x_{s+1,i}$ and $x_{s,i+m_s/2}$ for $i \in \{\ell - m_s/2, \ldots, k_{s+1} - 1\}$.
3. by recursive call obtain $x_{s+1,k_{s+1}}, \ldots, x_{s+1,\ell-1}$.
4. compute $x_{s,i}$ and $x_{s,i+m_s/2}$ from $x_{s+1,i}$ and $x_{s+1,i+m_s/2}$ for $i \in \{k_s, \ldots, \ell - m_s/2 - 1\}$;

if $\ell_s > \ell_{s+1}$ then

1. compute $x_{s+1,i}$ from $x_{s,i}$ and $x_{s,i+m_s/2}$ for $i \in \{\ell, \ldots, \ell_{s+1} - 1\}$ (i.e. "push down").
2. by recursive call obtain $x_{s+1,k_s+1}, \ldots, x_{s+1,\ell-1}$.
3. compute $x_{s,i}$ from $x_{s+1,i}$ and $x_{s,i+m_s/2}$ for $i \in \{k_s, \ldots, \ell - 1\}$.

A visual depiction of this algorithm is given in Figure 4.

As the algorithm may be designed in such a way to remain in place it follows that the next theorem is proved [2].

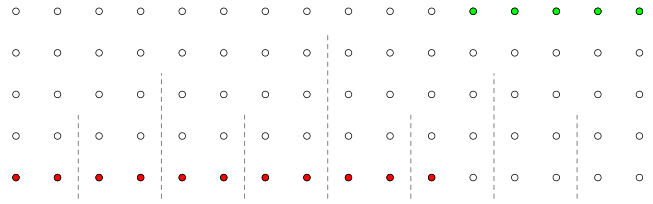
Theorem 2. *The ℓ-tuple $(a_0, \ldots, a_{\ell-1})$ can be recovered from its TFT with respect to ω using at most $\ell p + n$ shifted additions (or subtractions) and $\lfloor (\ell p + n)/2 \rfloor$ multiplications with powers of ω.*

That is, the cost of doing the inverse TFT is no more expensive than doing the TFT — in fact in most cases it's less expensive.

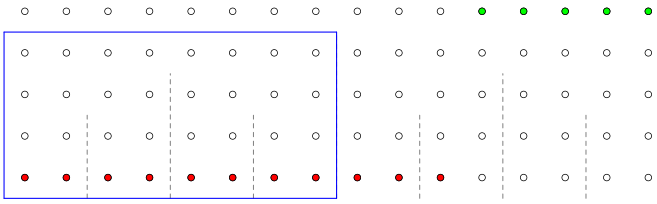
5 Conclusions

The Truncated Fourier Transform is a novel and elegant way to reduce the number of computations of an FFT-based computation by a possible factor of two (which may be significant). The hidden "cost" of working with the TFT is the increased difficulty of determining the inverse TFT. Although in most cases this is still less costly than the inverse FFT the algorithm is no doubt is much difficult to implement.

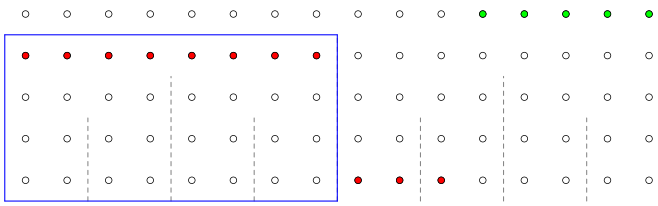
I feel that this implementation, done at very low level, by van der Hoeven is a significant contribution. I would certainly not like to be the one to duplicate this for MAPLE.



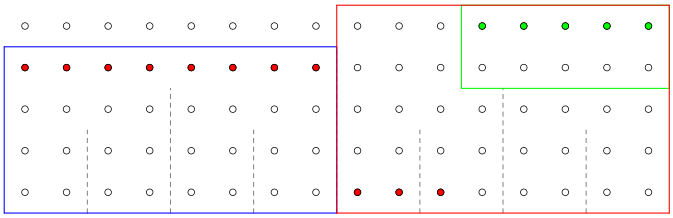
(a) Initial state of the algorithm. Red dots are the result of the forward TFT; green dots are zeros.



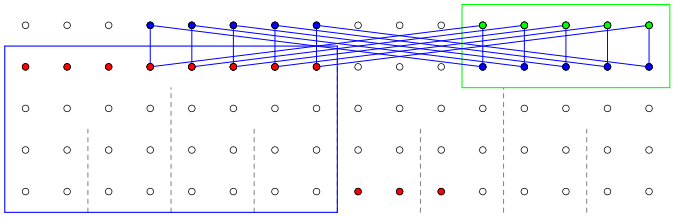
(b) Find "biggest" box where computations are self contained.



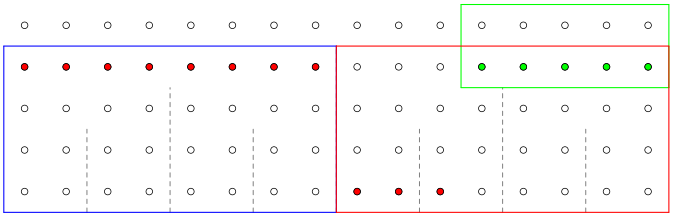
(c) "Push up"; calculate $x_{1,0}, \ldots, x_{1,7}$ from $x_{4,0}, \ldots, x_{4,7}$.



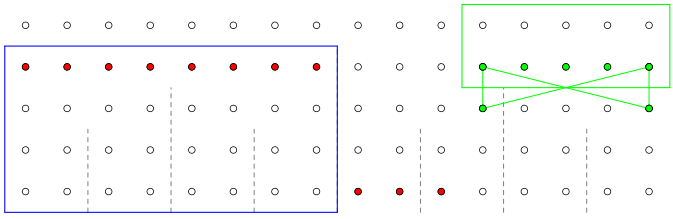
(d) Make a recursive call on the red box.



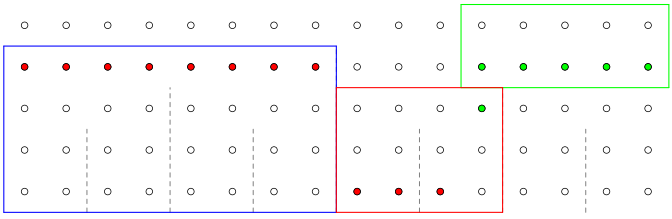
(e) "Push down" using cross relations.



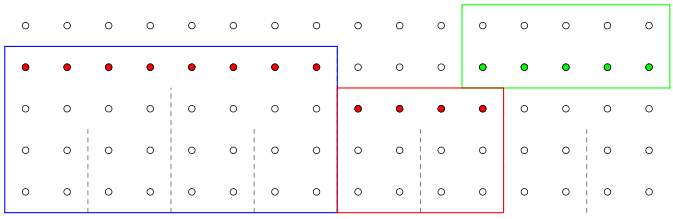
(f) Make a recursive call on the red box.



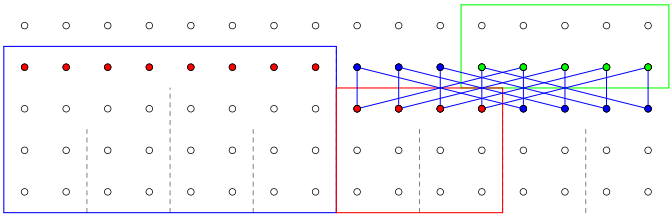
(g) "Push down" as much as possible.



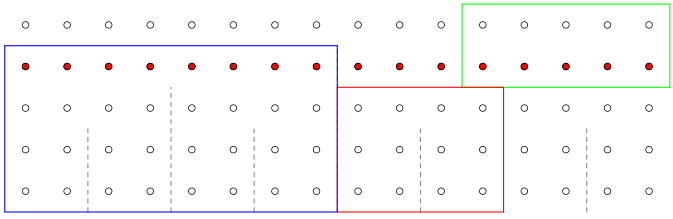
(h) Make a recursive call on the red box.



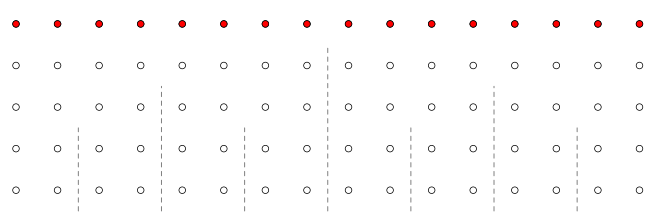
(i) The result of the previous recursive call.



(j) Use cross relations to "push up."



(k) "Push up" as much as possible.



(l) Done.

Figure 6: Schematic representation of the recursive computation of the inverse TFT for $n = 16$ and $\ell = 11$.

References

[1] K. O. Geddes, S. R. Czapor, and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[2] J. van der Hoeven. Notes on the Truncated Fourier Transform. Technical report, Université Paris-Sud, Orsay, France, 2008.

[3] Joris van der Hoeven. The truncated fourier transform and applications. In *ISSAC '04: Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, pages 290–296, New York, NY, USA, 2004. ACM.