Swiss Federal Institute of Technology Zurich

Fast Fourier Transform

Numerical Analysis Seminar

Stefan Wörner

Contents

1 Historical Introduction

The history of the Fast Fourier Transform (FFT) is quite interesting. It starts in 1805, when Carl Friedrich Gauss tried to determine the orbit of certain asteroids from sample locations ([3]). Thereby he developed the Discrete Fourier Transform (DFT, see Definition 2.2), even before Fourier published his results in 1822. To calculate the DFT he invented an algorithm which is equivalent to the one of Cooley and Tukey ([3], [2]). However, Gauss never published his approach or algorithm in his lifetime. It appeared that other methods seemed to be more useful to solve this problem. Probably, that is why nobody realized this manuscript when Gauss' collected works were published in 1866. It took another 160 years until Cooley and Tukey reinvented the FFT. In that time the US-military was interested in a method to detect Soviet nuclear tests. One approach was to analyze seismological time-series data and Tukey was in President Kennedy's Science Advisory Committee ([6]) that handled the problem. Between 1805 and 1965, several scientists developed efficient methods to calculate the DFT, but none of them was as general as Gauss' or the one of Cooley and Tukey. Figure 1 gives an overview.

Researcher(s)	Date	Lengths of Sequence	Number of DFT Values	Application
C. F. Gauss [10]	1805	Any composite integer	All	Interpolation of orbits of celestial bodies
F. Carlini [28]	1828	12	7	Harmonic analysis of barometric pressure variations
A. Smith [25]	1846	4, 8, 16, 32	5 or 9	Correcting deviations in compasses on ships
J. D. Everett [23]	1860	12	5	Modeling underground temperature deviations
C. Runge [7]	1903	$2^n K$	All	Harmonic analysis of functions
K. Stumpff [16]	1939	$2^n K$, $3^n K$	All	Harmonic analysis of functions
Danielson & Lanczos [5]	1942	2^n	All	X-ray diffraction in crystals
L. H. Thomas [13]	1948	Any integer with relatively prime factors	All	Harmonic analysis of functions
I. J. Good [3]	1958	Any integer with relatively prime factors	All	Harmonic analysis of functions
Cooley & Tukey [1]	1965	Any composite integer	All	Harmonic analysis of functions
S. Winograd [14]	1976	Any integer with relatively prime factors	All	Use of complexity theory for harmonic analysis

Figure 1.1: Principal Discoveries of Efficient Methods of Computing the DFT ([3])

2 Continuous and Discrete Fourier Transform

This chapter provides the definitions of the Continuous Fourier Transform (CFT) and the Discrete Fourier Transform (DFT) and motivates the DFT-formula with the trigonometric interpolation problem.

2.1 Continuous Fourier Transform

Definition 2.1 (Continuous Fourier Transform)**.** *Let $f : [0, L] \to \mathbb{C}$ be a Riemann integrable function with $f(0) = f(L)$. The k-th complex Fourier-coefficient of f is defined as*

$$\hat{f}_k = \frac{1}{L} \int_0^L f(x) e^{-2\pi i \frac{k}{L} x} dx,\ k \in \mathbb{Z} \tag{2.1}$$

and it follows the complex Fourier series of f:

$$f(x) = \sum_{k=-\infty}^{\infty} \hat{f}_k e^{2\pi i \frac{k}{L} x} \tag{2.2}$$

2.2 Discrete Fourier Transform

Definition 2.2 (Discrete Fourier Transform)**.** *Let $x = (x_0, ..., x_{N-1}) \in \mathbb{C}^N$, then the DFT of x is defined as*

$$\hat{x}_k = \frac{1}{N} \sum_{j=0}^{N-1} x_k e^{-2\pi i \frac{jk}{N}},\ k = 0, ..., N-1 \tag{2.3}$$

and $\hat{x} = (\hat{x}_0, ..., \hat{x}_{N-1})$.

This can be written as a matrix-vector-multiplication $\hat{x} = W_N x$ with

$$(W_N)_{ij} = \frac{1}{N} e^{-2\pi i \frac{ij}{N}} \tag{2.4}$$

where W_N is called the Fourier-matrix. This implies that the number of needed complex multiplications is equal to N^2 and the number of needed additions is $N \cdot (N-1)$, i.e. an arithmetic complexity of $O(N^2)$.

2.2.1 Trigonometric Interpolation

This section motivates the DFT with the trigonometric interpolation of a function. Let $F = (f_0, ..., f_{N-1}) \in \mathbb{C}^N$, be equidistant samples of a function $f : [0, L] \to \mathbb{C}$. The CFT is a method to calculate the coordinates of f relative to the infinite basis $e^{2\pi i \frac{k}{L} x}, k \in \mathbb{Z}$. Now, we want to approximate f by the N functions $e^{2\pi i \frac{k}{L} x}, k = 0, ..., N-1$, i.e.

$$\tilde{f}(x) = \sum_{k=0}^{N-1} c_k e^{2\pi i \frac{k}{L} x} \tag{2.5}$$

where the c_k should be chosen such that $\tilde{f}(x_j) = f_j, x_j = j \cdot \frac{L}{N}, j = 0, ..., N-1$.

Theorem 2.1. *The c_k in (2.5) are exactly the DFT-values of F.*

Proof.

$$\begin{aligned}
\tilde{f}(x_j) &= \sum_{k=0}^{N-1} c_k e^{2\pi i \frac{k}{L} \frac{jL}{N}} && (2.6)\\
&= \sum_{k=0}^{N-1} c_k e^{2\pi i \frac{kj}{N}} && (2.7)\\
&= \sum_{k=0}^{N-1} \left(\frac{1}{N} \sum_{l=0}^{N-1} f_l e^{-2\pi i \frac{kl}{N}} \right) e^{2\pi i \frac{kj}{N}} && (2.8)\\
&= \sum_{l=0}^{N-1} \underbrace{\left(\frac{1}{N} \sum_{k=0}^{N-1} e^{2\pi i \frac{k(j-l)}{N}} \right)}_{=:D(j,l)} f_l && (2.9)
\end{aligned}$$

This yields the result if $D(j, l)$ is equal to δ_{jl}, i.e. the Kronecker-Delta.

1. if $j = l$:

$$\begin{aligned}
D(j,j) &= \frac{1}{N} \sum_{k=0}^{N-1} e^{-2\pi i \frac{k(j-j)}{N}} && (2.10)\\
&= \frac{1}{N} \sum_{k=0}^{N-1} e^{0} && (2.11)\\
&= \frac{1}{N} \sum_{k=0}^{N-1} 1 = \frac{1}{N} N = 1 && (2.12)
\end{aligned}$$

2. if $j \neq l$:

$$D(j,j) = \frac{1}{N}\sum_{k=0}^{N-1} e^{-2\pi i \frac{k(j-l)}{N}} \tag{2.13}$$

$$= \frac{1}{N}\sum_{k=0}^{N-1} \left(e^{-2\pi i \frac{(j-l)}{N}}\right)^k \tag{2.14}$$

$$\underset{\text{series}}{\overset{\text{geom.}}{=}} \frac{1}{N}\frac{1-\left(e^{-2\pi i \frac{j-l}{N}}\right)^N}{1-e^{-2\pi i \frac{j-l}{N}}} \tag{2.15}$$

$$= \frac{1-1}{1-e^{-2\pi i \frac{j-l}{N}}} = 0 \tag{2.16}$$

The denominator of the resulting fraction is not zero because $j \neq l$.

Therefore it is $f(x_j) = \tilde{f}(x_j), \forall j = 0, ..., N-1$. □

Theorem 2.1 implies, that the inverse DFT is defined as in (2.5). Since the calculation of the DFT and the inverse DFT are almost equal, it follows, that a efficient method to calculate the DFT, as the FFT algorithm, can be used to calculate the inverse DFT, as well.

3 Derivation of FFT

This chapter provides the theoretical background for the FFT algorithm and discusses some special but widely-used cases.

3.1 FFT for composite of two integers

Let $N = N_1 \cdot N_2, N_1, N_2 \in \mathbb{N}$. The DFT of an vector $(x_0, ..., x_{N-1}) \in \mathbb{C}^N$ is given by

$$\hat{x}_k = \sum_{j=0}^{N-1} x_j e^{-2\pi i \frac{jk}{N}} \tag{3.1}$$

The prefactor $\frac{1}{N}$ is omitted, since this is the usual done. We can turn the one dimensional formulation of the DFT into a two dimensional one with the following change of variables

$$j = j(a,b) = aN_1 + b; \qquad 0 \leq a < N_2,\ 0 \leq b < N_1 \tag{3.2}$$
$$k = k(c,d) = cN_2 + d; \qquad 0 \leq c < N_1,\ 0 \leq d < N_2 \tag{3.3}$$

It follows for $x_j = x(a,b)$, $\hat{x}_k = \hat{x}(c,d)$ and $W_N = e^{-\frac{2\pi i}{N}}$:

$$\hat{x}(c,d) = \sum_{a=0}^{N_2-1} \sum_{b=0}^{N_1-1} x(a,b) W_N^{(aN_1+b)(cN_2+d)} \tag{3.4}$$
$$= \sum_{b=0}^{N_1-1} W_N^{b(cN_2+d)} \underbrace{\sum_{a=0} x(a,b) W_{N_2}^{ad}}_{=:\tilde{x}(b,d)} \tag{3.5}$$

since $W_N^{acN_1N_2} = W_N^{acN} = 1$ and $W_N^{adN_1} = W_{N_2}^{ad}$. This can be considered as calculating first N DFT-values with length N_2 (i.e. $\tilde{x}(b,d)$) and then calculating N DFT-values with length N_1 (i.e. $\hat{x}(c,d)$ with new data $\tilde{x}(b,d)$. This leads to a arithmetic complexity of $O(N \cdot N_1 + N \cdot N_2)$, which is much better than the direct approach with complexity $O(N^2)$. As an result of this effort, we state the number of complex multiplications needed for a DFT of one million sample points (i.e. $N = 10^6$) with the direct and FFT approach.

The table shows, that the FFT approach needs 500 times less multiplications than the direct approach. It can be applied to $\tilde{x}(b,d)$, as well, if N_2 is again not prime. In

approach	# multiplications
direct:	$\left(10^6\right)^2 = 10^{12}$
2 step fft:	$10^6 \cdot \left(10^3 + 10^3\right) = 2 \cdot 10^9$

Table 3.1: Needed Multiplications

general, this approach can be applied for all prime factors of $N = \prod_{i=1}^{P_N} p_i^{r_i}$ which would lead to a arithmetic complexity of

$$O\left(N\left(\sum_{i=1}^{P_N} r_i \cdot p_i\right)\right) \tag{3.6}$$

A widely-spread situation is given if $N = 2^n$. In this case formula (3.6) implies the well-known result for the arithmetic complexity of the FFT:

$$O\left(N\left(\sum_{i=1}^{n} 2\right)\right) = O\left(N2n\right) = O(Nlog_2(N)) \tag{3.7}$$

An interesting result is given in ([6]) and ([7]). It states, that this arithmetic complexity is a lower bound, i.e. there can not be an algorithm with a better arithmetic complexity.

3.2 FFT for prime numbers

The approach of the last section, does not work if N is a prime number. However, ([5]) provides two methods to calculate the DFT efficiently. The first one can be applied if $N-1$ is highly composite. In this case the problem can be solved by running three FFT algorithms. The second case creates a larger data array with $\tilde{N} = 2^n > N$ data points. A usual FFT algorithm can be applied then. I will not provide more details because this algorithm use methods to compute circular correlation functions via FFT algorithms and this is not topic of this paper, for more details see ([5]). To conclude, these approaches achieve even for N prime, very fast algorithms to compute the DFT, i.e. the second one has arithmetic complexity $O(\tilde{N} \cdot log_2(\tilde{N}))$.

4 Implementation

This chapter gives two implementations of the FFT for $N = 2^n$ (this is called the radix-2-algorithm) and discusses their advantages and disadvantages. To see how the two methods can be derived, the following formula is provided, which is a special case of 3.4 & 3.5. The prefactor $\frac{1}{N}$ is again omitted.

$$\begin{aligned}
\hat{x}_k &= \sum_{j=0}^{2^n-1} x_j e^{-2\pi i \frac{jk}{2^n}} && (4.1)\\
&= \sum_{j=0}^{2^{n-1}-1} x_{2j} e^{-2\pi i \frac{(2j)k}{2^n}} + \sum_{j=0}^{2^{n-1}-1} x_{2j+1} e^{-2\pi i \frac{(2j+1)k}{2^n}} && (4.2)\\
&= \underbrace{\sum_{j=0}^{2^{n-1}-1} x_{2j} e^{-2\pi i \frac{jk}{2^{n-1}}}}_{=:\hat{x}_k^1} + \underbrace{e^{2\pi i \frac{k}{2^n}} \sum_{j=0}^{2^{n-1}-1} x_{2j+1} e^{-2\pi i \frac{jk}{2^{n-1}}}}_{=:\hat{x}_k^2} && (4.3)
\end{aligned}$$

This result means (as before) that the DFT can be computed by computing two DFTs with half the length and with all even indices, respectively all odd indices, as new data vectors. This result is used to derive the implementations.

4.1 Recursive Implementation

The recursive FFT algorithm is a classical divide and conquer algorithm. It is easy to verify that $\hat{x}_k^1 = \hat{x}_{k+N/2}^1$ and $\hat{x}_k^2 = -\hat{x}_{k+N/2}^2, k = 0, ..., N/2 - 1$. Therefore Formula 4.1 leads immediately to the Matlab code provided in Algorithm 1. The implementation is quite short and easy and the arithmetic complexity is $O(N \cdot log_2(N))$. However, the space complexity of this implementation is not very good. The function is called twice in every recursion and in total an array of length 2^n has to be stored n times (Stack problem). This leads to a space complexity of $O\left(2^n \cdot n\right)$. To store one complex number, two double variables are needed, i.e. 64 Bit = 8 Byte. For about one million samples (e.g. $2^{20} = 1048576$) the algorithm needs at least $20 \cdot 1048576 \cdot 8 = 160$ Megabyte. As we will see in the next section, the iterative implementation will just need 8 Megabyte for the data, i.e. an in-place algorithm.

Algorithm 1 Recursive FFT

```
function y = fft_rec(x)
n = length(x);
if n == 1
   y = x;
else
   m = n/2;
   y_top = fft_rec(x(1:2:(n-1)));
   y_bottom = fft_rec(x(2:2:n));
   d = exp(-2 * pi * i / n) .^ (0:m-1);
   z = d .* y_bottom;
   y = [ y_top + z , y_top - z ];
end
```

4.2 Iterative Implementation

The iterative implementation of the FFT algorithm is not as straight forward as the recursive. In Formula 4.1 the data is divided in two arrays. The first one contains all even and the second one all odd indices, respectively. If the Formula is applied again to the subproblems, there are four arrays and so on. Figure 4.1 shows for $N = 8$ how the indices are permuted if this approach is applied $log_2(N) = 3$ times.

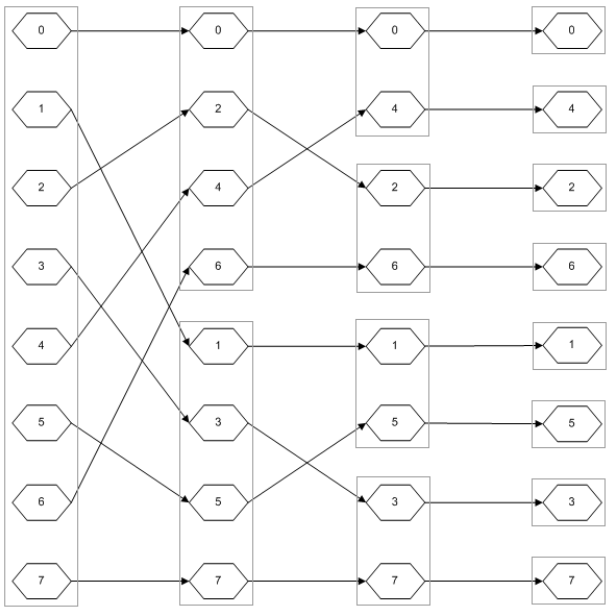

Figure 4.1: Data permutation for iterative FFT ([1])

This method is constructive for an arbitrary $N = 2^n$. It shows, that if the data is

rearranged in this manner, the DFT can be calculated as shown in Figure 4.2, where two Elements are merged with the same method as in Algorithm 1. $F(2,0)$ is the DFT of $(x_0, ..., x_7)$.

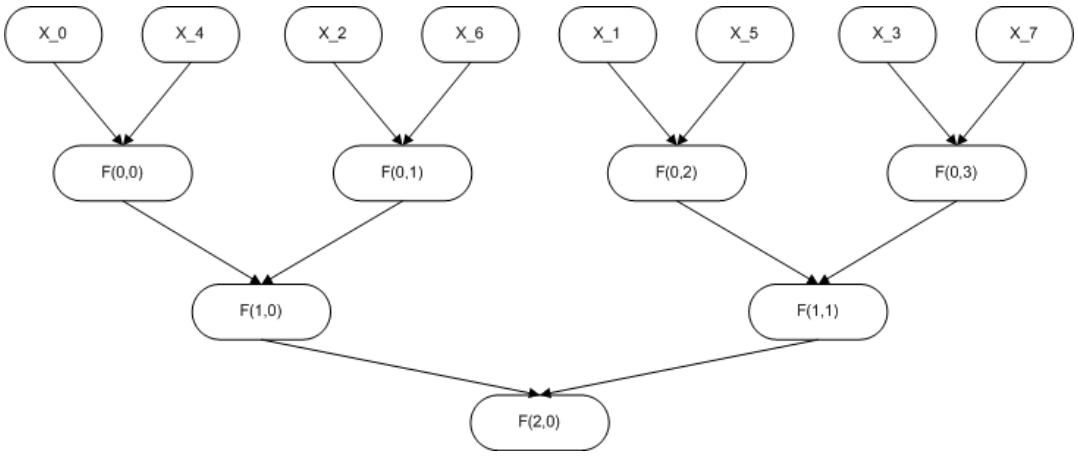

Figure 4.2: FFT - Scheme for $N = 8$ ([4])

The problem which is still to be solved is the rearrangement of the data. This can easily be done with a so called bit inversion. This means that the index k is written in binary representation and then read backwards. The new binary value is the permuted index. This is illustrated in Table 4.1

Index (dec)	Index (bin)	Inv. Index (bin)	Inv. Index (dec)
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 4.1: Bit Inversion ([4])

Theorem 4.1. *The permutation described through Figure 4.1 is equivalent to the bit inversion.*

Proof. The proof covers just the case where $N = 8$. But it is instructive and it is easily seen, that it holds for every $N = 2^n$. The binary representation of an index k and its

bit inversion, denoted by $\sigma(k)$, can be written as

$$k = \sum_{j=0}^{2} b_j 2^j \tag{4.4}$$

$$\sigma(k) = \sum_{j=0}^{2} b_{2-j} 2^j \tag{4.5}$$

The steps in the graph can be written as

$$k_0 = k \tag{4.6}$$

$$k_1 = (k \text{ div } 2) + b_0 \cdot 2^2 \tag{4.7}$$

$$k_2 = ((k \text{ div } 2) \text{ div } 2) + b_1 \cdot 2^1 + b_0 \cdot 2^2 \tag{4.8}$$

$$= b_2 \cdot 2^0 + b_1 \cdot 2^1 + b_0 \cdot 2^2 \tag{4.9}$$

$$= \sum_{j=0}^{2} b_{2-j} 2^j \tag{4.10}$$

which is equal to the bit inversion of k. □

Matlab provides a command for the bit inversion: "bitrevorder(x)". This leads to the Matlab code provided in Algorithm 2.

Algorithm 2 Iterative FFT ([4])

```
function y = fft_it(x)
n = length(x);
x = x(bitrevorder(1:n));
q = round(log(n)/log(2));
for j = 1:q
    m = 2^(j-1);
    d = exp(-2 *pi * i /m).^(0:m-1);
    for k = 1:2^(q-j)
        s = (k-1)*2*m+1;    % start-index
        e = k*2*m;          % end-index
        r = s + (e-s+1)/2;  % middle-index
        y_top = x(s:(r-1));
        y_bottom = x(r:e);
        z = d .* y_bottom;
        y = [y_top + z, y_top - z];
        x(s:e) = y;
    end
end
```

Bibliography

[1] E. Oran Brigham. *The Fast Fourier Transform And Its Applications*. Signal Processing Series. Prentice-Hall, 1988.

[2] James W. Cooley and John W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. 1965.

[3] Michael T. Heideman, Don H. Johnson, and C. Sideny Burrus. Gauss and the History of the Fast Fourier Transform. *Archive for History of Exact Sciences (Springer)*, 34(3):265–277, September 1985.

[4] Robert Plato. *Numerische Mathematik kompakt*. Numerische Mathematik. Vieweg, 2000.

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

[6] Daniel N. Rockmore. The FFT: An Algorithm the whole Family can use. *Computing in Science And Engineering*, 2(1):60–64, January 2000.

[7] Shmuel Winograd. *Arithmetic Complexity of Computations*, volume 33 of *Regional Conference Series in Applied Mathematics*. CBMS-NSF, 1980.