Università della Svizzera italiana
Faculty of Informatics
FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
FACULTY OF ENGINEERING

Technical Report - Efficient Computational Algorithms, Fall 2020

The fast Fourier transform
Perhaps the most ingenious algorithm of our lifetime

Glejdis Shkembi[†], Franciskus Xaverius Erick[†]

[†] Faculty of Engineering, Friedrich-Alexander Universität Erlangen-Nürnberg, Germany

Abstract

The fast Fourier transform is one of the most relevant computational algorithms developed for a wide range of applications. In this paper, we briefly discuss the mathematical background behind Fourier transforms such as Hilbert spaces, orthogonal bases, Fourier series, and discrete Fourier transforms. We explore in detail the decomposition methods of the computationally expensive discrete Fourier transform algorithm with $\mathcal{O}(N^2)$ operations into the most commonly used radix-2 Cooley –Tukey FFT algorithm with $\mathcal{O}(n\log(n))$ operations. We will also explore a widely used FFT implementation called FFTW with its ingenious use of planners and memory hierarchy. Finally we discuss parallel computing possibilities, comparison of different FFT algorithms, and limitations and outlooks of the FFT algorithm.

Report Info

Published
June 2021

Course
Efficient Computational Algorithms
Fall 2020

Institutions
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
&
Faculty of Engineering
Friedrich-Alexander Universität
Erlangen-Nürnberg, Germany

1 Motivation

The fast Fourier transform is a prime example of how computationally expensive mathematical operations can be beautifully optimized through various mathematical tricks. While the mathematical concepts behind Fourier transforms have existed for 200 years, the continuous research developments and hardware advancements have made it possible to optimize FFT algorithms. These optimizations lead to a myriad of FFT applications such as digital signal processing, spectral analysis, image and audio compression, filtering and denoising, solving of PDEs, and so on. A thorough study into the mathematical background and developments of FFT algorithms would thus offer an interesting insight into how computational algorithms are constantly developed to achieve better efficiency.

2 Mathematical Background

In its core the FFT is an algorithm that allows us to solve a mathematical problem with less computational effort than needed at first glance. It calculates the discrete Fourier transform (DFT), which can be described as a matrix vector multiplication in only $\mathcal{O}(n\log(n))$ operations[**?**].

In this section we will provide the most important mathematical basics of the FFT. We will cover the notion of Hilbert spaces, which gives a framework for function spaces and the understanding of functions as bases. Furthermore, we will introduce the main aspects of the Fourier series and transform in their analytical and discrete form.

2.1 Hilbert Spaces

Central to Hilbert spaces are inner products of functions as they allow te extension the concept of orthogonality from vectors and geometry to functions. For two functions $f(x)$ and $g(x)$ on a domain $x \in [a,b]$ the Hermitian inner product is defined by

$$< f(x), g(x) >= \int_a^b f(x)\overline{g}(x)dx, \tag{1}$$

where $\overline{g}(x)$ denotes the complex conjugate [**?**]. Note that other inner products can be defined as long as linearity, positive definiteness, and conjugate symmetry hold. If an inner product is defined for a vector space we can also introduce the following norm:

$$\|x\|_H = \sqrt{< x, x >}. \tag{2}$$

A vector space H is called a Hilbert space, if it is complete with respect to the norm in (2)[**?**]. On an analogy to geometry where two vectors are orthogonal if their inner product is zero, two functions are also called orthogonal if their inner product equals zero. Based on the definitions made above we can create an infinitely dimensionally Hilbert space H. We need to construct an infinite series of functions $(b_n)_{n\in\mathbb{N}}$, which are referred to as bases. Each of these functions has to be orthogonal to all other functions such that each basis spans a new dimension. However, the inner product with itself renders one. We can formalize this using the Kronecker delta

$$< b_n, b_m >= \delta_{n,m}. \tag{3}$$

Again we can use an analogy to geometry where each vector of a certain vector space can be described as a linear combination of the basis vectors. Also in Hilbert spaces of infinite dimensions each function $x \in H$ can be described with the bases and an appropriate set of parameters $(\lambda_i)_{i\in\mathbb{N}}, \lambda_i \in \mathbb{K}$:

$$x = \sum_{i=1}^{\infty} \lambda_i b_i. \tag{4}$$

2.2 Analytic Fourier Series and Fourier Transform

The Fourier series allows us to decompose a function $f(x)$ from $L^2[0,L]$ into sines and cosines in the following way:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty}\left(a_n \cos\left(\frac{2\pi n x}{L}\right) + b_n \sin\left(\frac{2\pi n x}{L}\right)\right). \tag{5}$$

It can be expressed in terms of the exponential function just as well:

$$f(x) = \sum_{n=1}^{\infty}(c_n)\exp\left(\frac{2i\pi n x}{L}\right). \tag{6}$$

Note that it is a implicitly assumed that $f(x)$ is periodic with period L. The decomposition uses the combination of a constant term $\frac{a_0}{2}$ and sines and cosines of increasing frequency $a_k \cos(kx) + b_k \sin(kx)$ as bases. With appropriate normalization of the terms it can be shown that they form an orthonormal basis, as defined in (3).

By comparison with (4) it can also be concluded that the Fourier series forms an infinite Hilbert basis[**?**]. This transform can be utilized to analyse physical phenomena: First, we get information about frequencies n and their amplitudes in the prefactors a_n and b_n. This information is essential for every analysis of oscillation phenomena like sound, mechanical vibrations or light [**?**]. Second, sines and cosines are of infinite smoothness, i.e., they are infinitely often differentiable. Hence, the Fourier transform can be very useful to solve differential equations of any phenomenon, for instance, the heat equation which was the actual motivation of Fourier to develop this change of bases [**?**]. The factors a_0, a_n, and b_n can be understood as projections of the function $f(x)$ on the chosen basis. As in geometrical projections can be computed in terms of inner products:

$$a_n = \frac{2}{L}\left\langle f(x), \cos\left(\frac{2\pi n x}{L}\right)\right\rangle \quad = \quad \frac{2}{L}\int_0^L f(x)\cos\left(\frac{2\pi n x}{L}\right)dx, \quad n = 0,1,2,3..., \tag{7}$$

$$b_n = \frac{2}{L}\left\langle f(x), \sin\left(\frac{2\pi n x}{L}\right)\right\rangle = \frac{2}{L}\int_0^L f(x)\sin\left(\frac{2\pi n x}{L}\right)dx, \quad n = 1,2,3..., \tag{8}$$

$$c_n = \frac{1}{L}\left\langle f(x), \exp\left(\frac{2i\pi n x}{L}\right)\right\rangle = \frac{1}{L}\int_0^L f(x)\exp\left(\frac{2i\pi n x}{L}\right)dx, \quad n = 1,2,3.... \tag{9}$$

We have already restricted $f(x)$ to the twice Lebeque integrable functions on a finite real interval. Further limitations apply, when analyzing the convergence of the series, which we want to omit here[**?**].

The Fourier transform extends the decomposition into sines and cosines to functions on infinite domains by considering a function $f(x)$, $x \in [-0.5L, 0.5L]$ and letting $L \to \infty$. For this purpose it is convenient to analyze the Fourier series in the form of the exponential function form (6) and its factor (9). In letting L go to infinity the following changes can be observed [**?**]:

- f is aperiodic.
- The discrete spectral lines approach each other.
- The line spacing $\Delta\nu = \frac{1}{L}$ becomes an infinitesimal $\mathrm{d}\nu$.
- The spectral line frequency $\nu_n = \frac{n}{L}$ becomes continuous.
- The summation in (6) becomes a Riemann integral
- The sequence of Fourier coefficients c_n becomes a function $F(\nu)$:

$$F(\nu) = \int_{-\infty}^{\infty} f(\xi)\exp(-i2\pi\nu\xi)d\xi. \tag{10}$$

Now we can state the forward Fourier transform, which transforms a function $f(x)$ into the frequency domain. It is often abbreviated by $\mathscr{F}$:

$$F(\nu) = \int_{-\infty}^{\infty} f(x)\exp(-i2\pi\nu x)dx. \tag{11}$$

The inverse Fourier transform $\mathscr{F}^{-1}$ is

$$f(x) = \int_{-\infty}^{\infty} F(\nu)\exp(i2\pi\nu x)d\nu. \tag{12}$$

Figure 1 depicts the Fourier transform of a unit pulse. It can be observed that the asymmetric pulse introduces a phase, which is represented by the imaginary part of its Fourier transform.

2.3 Discrete Fourier Transform

So far we have considered continuous functions. In most of engineering or data analysis tasks, however, observations are available from measurements in discrete data points forming a vector $\mathbf{x} = [x[1], x[2], ..., x[N]]^T$. Just as we did for the continuous case we would like to transform $\mathbf{x}$ into the frequency domain $\mathbf{X} = [X[1], X[2], ..., X[N]]^T, \mathbf{X} \in \mathbb{C}^N$. This is done by truncating the Fourier series (6) to a finite number of summands. The discrete Fourier transform (DFT) is then given by

$$X[k] = \sum_{j=0}^{N-1} x[j]\exp\left(\frac{-i2\pi jk}{N}\right), \tag{13}$$

and the inverse (iDFT) is defined by

$$x[k] = \frac{1}{N}\sum_{j=0}^{N-1} X[j]\exp\left(\frac{i2\pi jk}{N}\right). \tag{14}$$

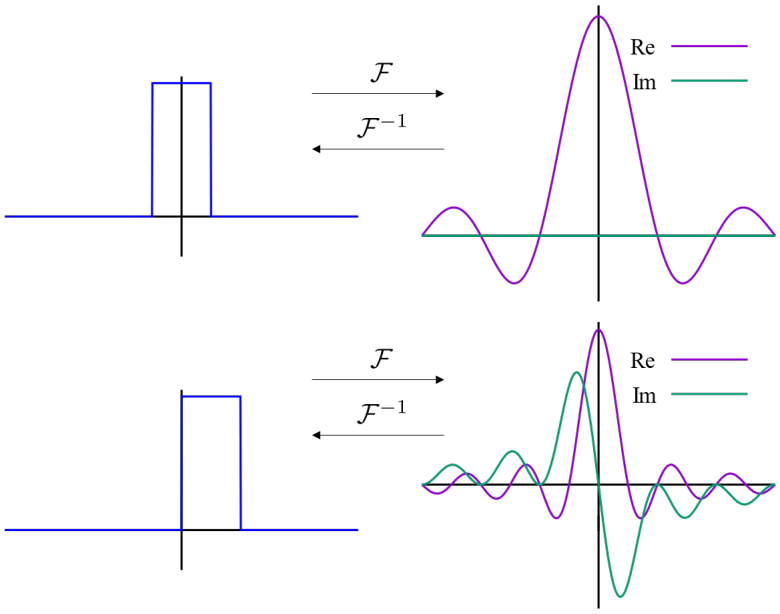

Figure 1: Fourier transform of a symmetric unit pulse is purely real. An offset induces a phase and therefore an imaginary part in the frequency domain[**?**].

We can observe that the DFT is a linear operator (i.e., a matrix) that maps data points from the time domain into the frequency domain:

$$\mathbf{x} \quad \underrightarrow{DFT} \quad \mathbf{X}. \tag{15}$$

By introducing the fundamental frequencies $\omega_N = \exp(\frac{2i\pi}{N})$ the DFT can be expressed as matrix vector multiplication:

$$\begin{bmatrix} X[1] \\ X[2] \\ X[3] \\ . \\ . \\ X[N] \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & . & . & 1 \\ 1 & \omega_N & \omega_N^2 & . & . & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & . & . & \omega_N^{2(n-1)} \\ . & . & . & . & & . \\ . & . & . & & . & . \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & . & . & \omega_N^{(N-1)^2} \end{bmatrix} \begin{bmatrix} x[1] \\ x[2] \\ x[3] \\ . \\ . \\ x[N] \end{bmatrix}. \tag{16}$$

A central property of the DFT is its periodicity which is inherited from its basis vectors. For any DFT pair $\mathbf{x}, \mathbf{X} \in \mathbb{C}^N$, $\mathbf{x}$ and $\mathbf{X}$ are one period of an infinite sequence with period N. Furthermore the transformed vector $\mathbf{X}$ is Hermitian if $\mathbf{x}$ is real:

$$X[N-m] = \overline{X[m]}. \tag{17}$$

This symmetry property is closely related to the Nyquist sampling theorem, as the highest frequency in $\mathbf{X}$ is half of the sampling rate [**?**].

A DFT is precisely a truncated or finite Fourier series. Approximations of function by DFTs suffer from high oscillations around discontinuous points as shown in Figure 2, because all of the basis functions's derivatives exist in a strong sense. In the next chapter we will discuss how the FFT works and why it scales only with $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$ for solving the linear system of equations in (13).

3 The fast Fourier transform

The fast Fourier transform allows us to compute the DFT of a data vector with fewer operations than with a straightforward implementation of the sum given in (13). With fewer computations the FFT also reduces round-off errors significantly [**?**].

As motivation, we will first break down the direct computation of (13) for $\mathbf{x} = [x[1], x[2], x[3], ..., x[N]]^T$. In general N multiplications and $N-1$ additions for each of the N elements in $\mathbf{x}$ are needed. If $\mathbf{x}$ is complex valued, each multiplication will require four real multiplies and two real adds. For complex additions, two

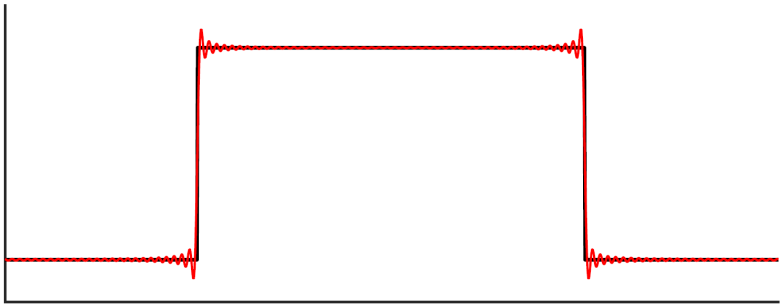

Figure 2: Gibb's phenomenon describes oscillations around discontinuities of function approximation by finite Fourier series[**?**].

real adds are needed. The most general case therefore adds up to $4N^2$ real multiplies and $4N^2-2N$ real adds. The count can be decreased for hardware, where a multiply-add is one operation. Also a real vector $\mathbf{x}$ decreases the necessary operations for the DFT to only N^2 operations, due to simpler calculations and the symmetry of $\mathbf{X}$. There are ways to further decrease operations for special cases of $\mathbf{x}$, yet the calculations will always scale with $\mathcal{O}(N^2)$[**?**],[**?**].

3.1 The Cooley–Tukey Algorithm

The Cooley–Tukey algorithm is the most popular method for calculating the DFT and was proposed by Cooley and Tukey in 1965 [**?**]. The basic idea was already known to Gauss, which is particularly interesting as his notes date even before Fourier postulated the Fourier series [**?**]. The algorithm was rediscovered several times and with the general availability of computers it became extremely useful.The underlying mechanism of the algorithm is to recursively split the initial DFT into smaller ones. Due to symmetry properties, the initial DFT can be computed by combining the smaller DFTs by twiddle factors.

3.1.1 Decimation in Time (DIT)

Starting from (13) and we can derive the following equalities:

$$\begin{aligned}
X[k] &= \sum_{j=0}^{N-1} x[j]\exp\left(-\frac{i2\pi jk}{N}\right) \quad (1)\\
&= \sum_{j=0}^{\frac{N}{2}-1} x[2j]\exp\left(-\frac{i2\pi(2j)k}{N}\right) + \sum_{j=0}^{\frac{N}{2}-1} x[2j+1]\exp\left(-\frac{i2\pi(2j+1)k}{N}\right) \quad (2)\\
&= \sum_{j=0}^{\frac{N}{2}-1} x[2j]\exp\left(-\frac{i2\pi(j)k}{\frac{N}{2}}\right) + \exp\left(-\frac{i2\pi j}{N}\right)\sum_{j=0}^{\frac{N}{2}-1} x[2j+1]\exp\left(-\frac{i2\pi(2j)k}{\frac{N}{2}}\right) \quad (3)\\
&= DFT_{\frac{N}{2}}[[x[0],x[2],..x[n-2]] + W_n^j DFT_{\frac{n}{2}}[[x[1],x[3],...,x[n-1]] \quad .(4)
\end{aligned} \tag{18}$$

First, we split up the initial vector $\mathbf{f}$ into even numbers $j = [0,2,4,...,N-2]$ and odd numbers $j = [1,3,...,N-1]$ (2). Second, we use the rules of the exponential function to pull a factor out of the sum which was hidden in $(2j+1)$. Furthermore, we move the factor 2 from the numerator into the denominator (3). As a result, we express a DFT of length N in two DFTs of length $\frac{N}{2}$ which are connected by the twiddle factor W_N^j[**?**].

The procedure is called decimation in time because the splitting was applied to the data vector $\mathbf{x}$ in the time domain. The presented derivation divides the initial DFT into two subsets. That is why it is called "radix-2". Figure 3 illustrates the splitting into smaller DFTs graphically. $G(k)$ contains the frequency outputs of the even-indexed time samples and $H(k)$ the odd-indexed accordingly. Due to the periodicity of the DFT with $\frac{N}{2}$, $G(k)$ and $H(k)$ can be used to compute two of the length-N frequencies: $X(k)$ and $X(k+\frac{N}{2})$. Only the twiddle factor in-between makes them different.

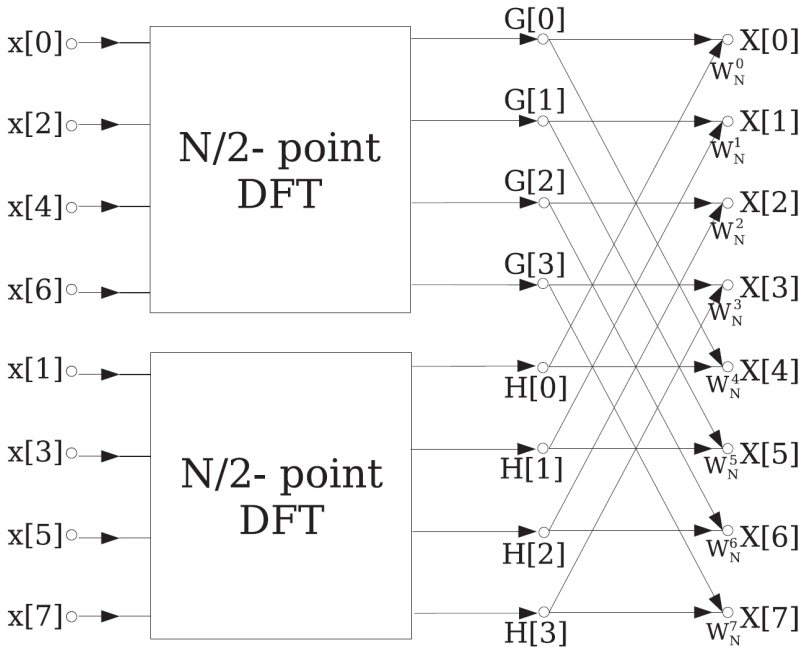

Figure 3: Decimation in time of DFT of length N into two DFTs of length $\frac{N}{2}$[**?**].

This reuse of computations is key to FFT's computational savings. The necessary operations have been reduced to $2(\frac{N}{2})^2+N=\frac{N^2}{2}+N$ complex multiplies and $2\frac{N}{2}(\frac{N}{2}-1)+N=\frac{N^2}{2}$ complex additions[**?**].

If the same splitting is applied to the new DFTs with half of the length of the previous one until only DFTs of length 2 remain, the radix-2 decimation in time FFT arises. The entire signal flow for an input of length 8 is depicted in Figure 4. The marked region in the diagram is called the butterfly operation. In the diagram we can observe that only $\frac{N}{2}$ twiddle multiplies are performed per stage. However, there is a minus sign applied on $\frac{N}{2}$ outputs per stage. This can be explained be the following property of the twiddle factors:

$$W_N^{j+\frac{N}{2}}=-W_N^j \tag{19}$$

[**?**],[**?**].

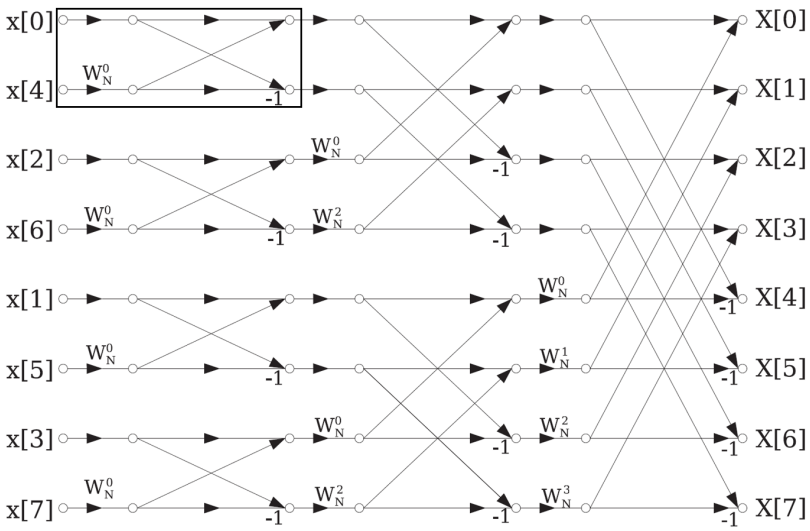

Figure 4: The signal-flow diagram for radix-2 DIT FFT of length 8 shows how bit-reversed input data are combined by twiddle factors in each stage until the DFT is computed[**?**].

The algorithm requires $M = \log_2 N$ stages with $\frac{N}{2}$ twiddle factors. After all, we end up with $\frac{N}{2}\log_2(N)$ complex multiplies and $N\log_2(N)$ complex adds. The radix-2 DIT FFT is therefore a $\mathcal{O}(N\log_2(N))$ algorithm. The DIT radix-2 algorithm will only run down to basis DFTs of length two for all sub-DFTs if the initial data input is of length 2^N. This is the reason why many basic implementations of the FFT only allow input lengths which are of powers of two. Another common strategy is to pad the input with zeros until an appropriate length is reached[**?**].

A comparison of the idealized scaling behavior of the DFT and the FFT is given in Figure 5. For a vector length $N = 512$ which is commonly used in signal analysis the difference in operations is already remarkable: approximately 4600 versus 256000.

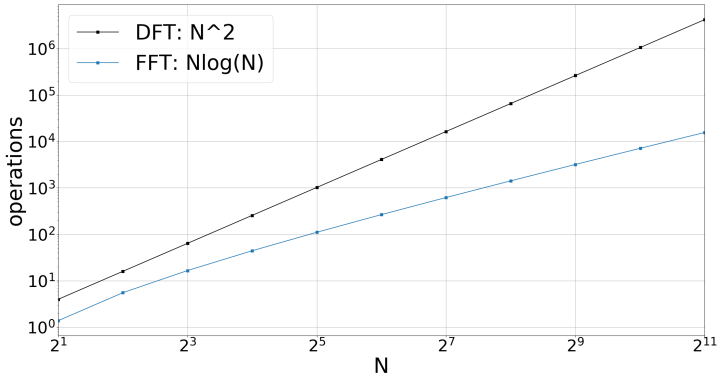

Figure 5: The computational savings from FFT $\mathcal{O}(N\log(N))$ to DFT $\mathcal{O}(N^2)$ are significant especially for large vectors.

The reordering of the input into odd and even numbers is also called bit-reversal as depicted in Table 1. If the input data are placed in bit-reversed order before launching the FFT algorithm, the result of each butterfly operation will be placed in the same memory from which the inputs were fetched and the resulting frequency values will appear in natural order. In this way, the FFT can be implemented in-place without requiring new memory. In contrast, natural input order does not allow in-place computation and leads to bit-reversed output in the frequency domain, which has to be untangled after the computation. In the more general case, for non-tworadix algorithms, the procedure is called digital reversal[**?**],[**?**] ,[**?**].

Table 1: Bit-reversal of input data for radix-2 DIT FFT algorithm for N = 4.

In-order index	In-order index binary	Bit-reversed binary	Bit-reversed index
0	00	00	0
1	01	10	2
2	10	01	1
3	11	11	3

3.1.2 Decimation in Frequency (DIF)

For a DIF algorithm, DFTs are again recursively split into smaller DFTs. Instead of rearranging the inputs in the time domain, the discrete frequency indices $X(k)$ are reordered into even and odd. Similar to radix-2 DIT, we can derive two DFTs of length $\frac{N}{2}$ with twiddle factors $W_N^k = \exp(-\frac{i2\pi k}{N})$:

$$
\begin{aligned}
X[2r] &= \sum_{j=0}^{N-1} x[j] W_N^{2rj} \\
&= \sum_{j=0}^{\frac{N}{2}-1} x[j] W_N^{2rj} + \sum_{j=0}^{\frac{N}{2}-1} x\left[j+\frac{N}{2}\right] W_N^{2r(j+\frac{N}{2})} \\
&= \sum_{j=0}^{\frac{N}{2}-1} \left(x[j] + x\left[j+\frac{N}{2}\right]\right) W_N^{2rj} \\
&= DFT_{\frac{N}{2}}\left[x[j] + x\left[j+\frac{n}{2}\right]\right], \\
X[2r+1] &= \sum_{j=0}^{N-1} x[j] W_N^{(2r+1)j} \\
&= \cdots \\
&= DFT_{\frac{N}{2}}\left[\left(x[j] + x\left[j+\frac{n}{2}\right]\right) W_N^{j}\right].
\end{aligned}
\tag{20}
$$

Terminology applies just as for DIT. Continuing this procedure down to base cases of length two leads to a radix-2 DIF FFT algorithm. DIF can also be computed in place if the input is provided bit-reversed and the computational complexity remains $\mathcal{O}(N\log(N))$ with the same amount of operations as for DIT[**?**],[**?**].

In this section, we have presented the radix-2 algorithms DIT and DIF in detail because they are the most basic algorithms and they are essential to understanding further variants of the FFT which may use other regrouping strategies and other group sizes such as radix-4 algorithms. Further algorithms are explained in section 3.3.

3.2 High Performance and Parallel Computing

FFT implementation can be accelerated with the help of multiple processors and parallel computing methods. In the context of 1D FFT, the input data can be split up and calculated in different processors. This can be done in either a block layout or cyclic layout [**?**]. For an m point FFT with p processors, the communication is significant in this first $\log(p)$ steps for block layout configuration, while no communication occurs in the last $\log(m/p)$ stages, while for cyclic layout configurations, communication between the different processors is significant in the last $\log(m/p)$ stages. The block and cyclic data layouts are illustrated, respectively, in Figures 6 and 7. In Figure 6, it can be seen that for p = 4 processors and for m = 16 data points, 4 data points of neighboring indices are grouped together into one processor. In the first $\log(p)=2$ stages, the butterfly operations are done with neighboring data points of bit-reversed indices. Therefore, this requires communication between different processors. In the last $\log(m/p)=2$ stages, the butterfly operations are done with data points within the same processors which then require no communication. Meanwhile in Figure 7, 4 data points with neighboring data points of bit-reversed indices are grouped together in the same processors, which consecutively results in no communication in the first $\log(p)=2$ stages and communication in the last $\log(m/p)=2$ stages.

Additionally, transpose operations can also be used such that cyclic layout is used in the first $\log(p)$ steps before transposing the vector for the last $\log(m/p)$ steps such that the data are then reverted back to block layout [**?**]. Therefore, communication between different data points always happens within the same processor. Similarly to the example parameters used in Figures 6 and 7, the transpose operation is also represented in Figure 8. Due to transpose operation of the data sets after the first $\log(p)=2$ steps, the butterfly operations always happen within the same processor. The parallelization of FFT operations is especially utilized in the context of 2D and 3D FFT operations, where the computational domains can be too large for just one processor to handle. The multidimensional operations are then distributed among different processors, whereby the FFT operations involve computations and communications between the different dimensions of the vectors. An overlap between communication and computation is thus of utmost importance considering the bandwidth and latency issues that may significantly hinder performance and efficiency. This can be achived by optimizing the transpose operations through grid decomposition techniques such as slab decomposition and pencil decomposition within the message passing interface (MPI)

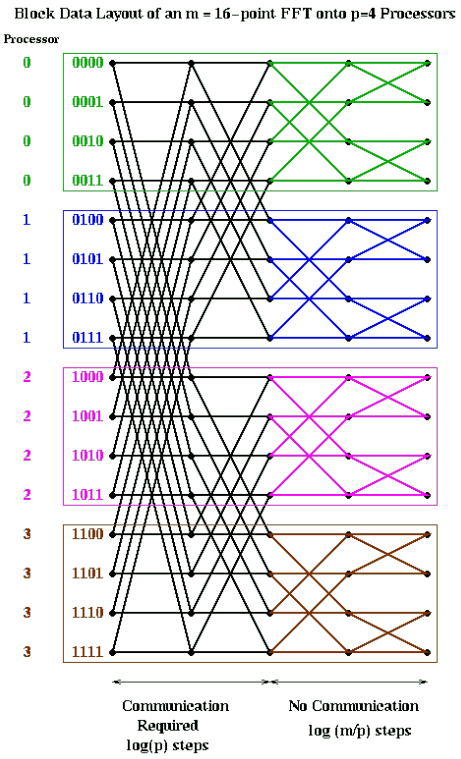

Figure 6: Block data layout for FFT [**?**].

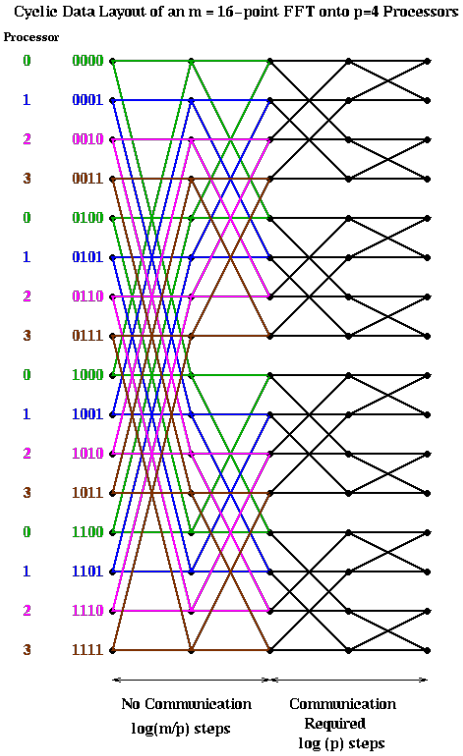

Figure 7: Cyclic data layout for FFT [**?**].

programs used for parallel computing [**?**]. These techniques are used in well known parallel computing FFT libraries such as MPI-FFTW, P3DFFT, and 2DECOMP&FFT.

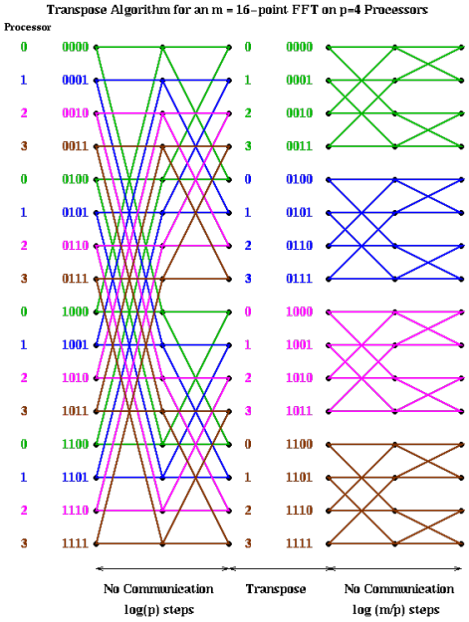

Figure 8: FFT with transpose[**?**].

3.3 Comparison of different FFT Algorithms

In the previous sections, our focus lies mostly on the Cooley–Tukey algorithm of FFT. While this is the most commonly used algorithm, there are also various other FFT algorithms that were developed separately over time. These different algorithms may offer significant benefits in terms of efficiency and computational costs for various specific cases compared to the more commonly used Cooley–Tukey algorithm.

One of the most commonly used alternate FFT algorithms is the radix-4 algorithm. It is similar to the radix-2 Cooley–Tukey algorithm, with the exception being that the butterfly operation is now carried out with 4 data points instead of with 2 data points [**?**]. The larger butterfly operations result in smaller numbers of multiplication operations and DFT decomposition stages. However, the larger butterfly operations lead to a higher code and memory management complexity. Similar behavior is also observed with larger radix algorithms such as radix-8. In the current context of computer hardware advancements, the number of multiplication operations is no longer a significant factor with hardwares that facilitate multiplication operations efficiently.

Another commonly featured FFT algorithm is the prime-factor algorithm and the Winograd algorithm [**?**]. In contrast to the Cooley–Tukey and radix algorithms, these two algorithms utilized concepts from number theory to divide DFT operations into smaller DFT operations. In the prime-factor algorithm, a DFT of size N is split into multidimensional DFTs of sizes N1 and N2, whereby N1 and N2 are relatively prime factors of the size $N = N1 * N2$. The prime factors are determined through reindexing of the input size N using concepts from Chinese remainder theorem. Afterwards, the DFT is then carried out as multidimensional DFTs of sizes N1 and N2. Since there are no longer twiddle factors involved in the computation, this algorithm also reduces the amount of multiplication operations as compared to the Cooley–Tukey algorithm. However, this algorithm only works for size N with strictly relatively prime factors. Additionally, the reindexing operation is more computationally complex compared to the odd and even split operation in the Cooley–Tukey algorithm.

The Winograd algorithm extends the idea of multidimensional representation of DFTs behind prime-factor algorithm. The Winograd algorithm utilizes multidimensional cyclic convolution concepts obtained from reordering and expanding the factors obtained from the prime-factor algorithm. This results in nested multiplication operations in the center of the Fourier transform operation, which then subsequently

results in a significantly lower number of multiplication operators during runtime. In return, the number of required addition operations increases significantly. While the number of multiplication operations is reduced, the Winograd algorithm is also significantly more complex to program and run than the Cooley–Tukey algorithm and prime factor algorithm.

While the Cooley–Tukey algorithm is still the most commonly used algorithm due to its relative simplicity, various algorithms can sometimes be combined to achieve more efficient computations. Different radix algorithms can be combined with the prime factor algorithm and the Winograd algorithms in various stages of DFT transforms. The question of which algorithms and the possible algorithm combinations perform the best is subject to a great deal of research, especially with regards to the specificites of the Fourier transform problem and hardware architecture in hand. There are also various other emerging algorithms such as the hexagonal FFT which utilizes the newly researched hexagonal grid sampling scheme. This is further explained in detail in the following reference [**?**].

3.4 FFTW

FFTW is a library for computing the DFT for inputs of arbitrary size and dimension for real or complex values. It was developed at MIT by Matteo Frigo and Steven G. Johnson mainly from 1998 through 2008[**?**]. The implementation of the FFT still proves to be competitive with more recent implementations due to its automatic code generation. Our goal, in this section is to present the structure of FFTW and its key features. In particular we want to address how general challenges of implementing the FFT are solved in this library instead of giving an in-depth description of technical details, which can be found in [**?**].

3.4.1 Philosophy and Key Features

The authors Frigo and Johnson stress that the most important aims, while developing FFTW, were flexibility and generality, which for them is the key to a widely used library. The highest possible performance was then attempted, while sticking to the desired generality. According to the authors, FFTW is probably the most flexible DFT library due to the following features[**?**]

- FFTW runs on many architectures and operating systems.
- FFTW computes DFTs in $\mathcal{O}(N\log(N))$ for any input length N.
- FFTW is not restricted to specific dimensions of transform.
- FFTW supports multiple and/or strided DFTs (i.e., non contiguous input data).
- FFTW supports DFTs of real and real symmetric or antisymmetric data.

3.4.2 Structure

FFTW has introduced its specific data structure to express DFT problems, so called I/O tensors. I/O tensors can be broken down into I/O dimensions $d = (n, \iota, o)$, where n is a non negative integer representing the length, ι is an integer describing the input stride, and o is also an integer for the output stride. An I/O tensor of rank $\rho = |t|$ can be defined as $t = [d_1, d_2, ..., d_\rho]$.

A specific DFT-problem is described as

$$dft(\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O}), \tag{21}$$

where $\mathbf{N}, \mathbf{V}$ are two I/O tensors and $\mathbf{I}, \mathbf{O}$ pointers to the memory locations of the input and the output, respectively[**?**].

FFTW gains its generality and performance from code generation. For each DFT problem a specific plan is created. In the planning stage FFTW searches for an optimal algorithm in terms of execution time for the specific problem and hardware. The "planner" explores a space of plans from which the fastest plan is chosen. The size of this space highly influences the contradictory goals of short planning time and fast execution. Hence, the choice of plans to consider during the planning phase is critical. FFTW uses the following categories of plans[**?**]

- No-op plans: no work has to be done.

- Rank-0 plans: a permutation from the input into the output array.
- Rank-1 plans: for ordinary 1D DFTs.
- Higher rank plans: for multidimensional DFTs.
- Higher vector rank plans: for multiple DFTs.
- Indirect plans: for inputs that require data shuffling.
- Other plans.

Rank-1 plans are of special interest for us as they handle 1D DFTs which were discussed in detail in section 3.1. FFTW breaks them down into two categories. Some of the DFTs can be computed directly by highly optimized straight-line C code, so called codelets. These codelets are generated by the special purpose compiler genfft, which was created by the authors dedicated to FFTW. For bigger problems FFTW mainly uses variants of the Cooley–Tukey algorithm to break the problem down until it can be solved by a single codelet leading to a direct plan. DIT as well as DIF algorithms are implemented.

The compiler genfft is able to generate highly optimized C code of different DFTs from an abstract mathematical description. It does optimizations like unrolling to exploit long processor pipelines, pre computing of factors or breaking down complex-number representation into real and imaginary components. genfft operates in the four phases creation, simplification, scheduling, and unparsing.

IA representation of the codelet is created in the form of a directed acyclic graph, by using one of the following algorithms: Cooley–Tukey, prime factor, split radix, and Radar. In the simplification phase algebraic transformations take place, such as common subexpression elimination. During scheduling, gennft searches a schedule such that a C compiler can perform a good register allocation. Finally, the schedule is unparsed to C[**?**].

```
fftw_plan plan;
fftw_complex in[n], out[n];

// plan a 1d forward DFT:
plan = fftw_plan_dft(n, in, out, FFTW_FORWARD, FFTW_PATIENT);

//Initialize [] with some data..

fftw_execute(plan); // compute DFT

//Write new data in []...

fftw_execute(plan); // reuse plan
```

Listing 1: Example of FFTW use. First a plan has to be defined, which can be reused[**?**].

The user of FFTW can specify the problem over an interface without dealing with the internal complexity of FFTW. The library provides different interfaces for different degrees of generality. An example of how to use FFTW is given in Listing 1. After initializing arrays for input and output, a plan has to be created, which needs the pointers to the input and output arrays as well as their lengths. The user also specifies whether a forward (-1) or backward (1) transform should be performed. Furthermore, a planner flag ("FFTW_PATIENT" in Listing 1) is defined which allows a trade-off between planning speed versus optimality. The effect is depicted in Figure 9. The general shape of the curve can be explained as follows. For small problem sizes the overhead created by the call of a FFTW routine prevents good performance. For large problems the performance of all routines drops as well reflecting the cache hierachy of the machine[**?**]. The three main operating modes are

- patient ("FFTW_PATIENT"): The planner tries all combinations of possible plans for optimal execution performance,
- impatient ("FFTW_MEASURE"): The planner searches a smaller space of plans and reduces the planning time,

- estimate ("FFTW_ESTIMATE"): No measurements are performed. Instead the planner minimizes a cost function: number of flops + "extraneous" loads/stores for minimal planning time.

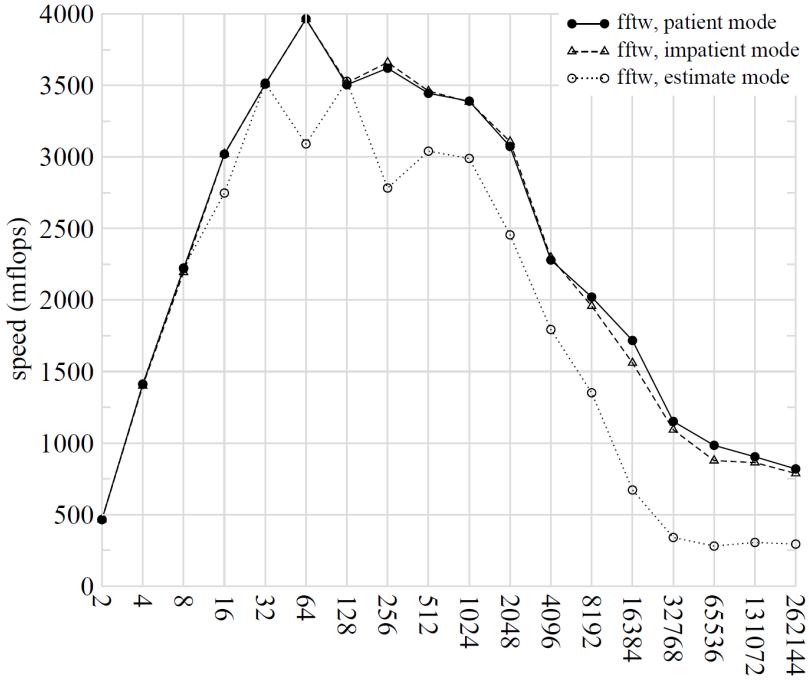

Figure 9: Effects of planner flags in FFTW on the execution performance for double-precision 1D complex DFTs, power-of-two sizes, on a 2 GHz PowerPC 970 (G5)[**?**].

3.4.3 Memory Hierachy: Breadth-First versus Depth-First

A major challenge when implementing the Cooley–Tukey algorithm is that the input **x** is discontiguous for a contiguous output **X** or vice versa. As a consequence discontiguous memory access and data reordering hinders efficient use of hierarchical memory architectures. The execution order is therefore a non obvious aspect of an efficient FFT implementation.

One order distinction among others can be made between recursion and iteration. As shown in Figure 10 the Cooley–Tukey algorithm can be represented as a tree of smaller and smaller DFTs. The tree can be traversed either breadth or depth first. The traditional Cooley–Tukey algorithm as presented in section 3.1 uses the breadth-first approach. When dividing the initial DFT of length 8, the traditional approach comes up with both representations of length-4 DFTs. For large inputs this can mean that parts of generated data for the sub-DFT is already out of cache, leading to cache misses and therefore additional time for reloading data. In contrast the recursive depth-first approach goes directly down until a base case is reached. Hence, loaded data in the cache are directly used, leading to fewer cache misses.

In this context the choice of radix is also relevant. A radix-2 algorithm only halves the length of the input vector. Hence, the way down to a base case incorporates significantly more steps than for an algorithm of higher radix. Due to the low rate of reducing necessary data for the computation algorithms of small radices can also suffer from cache misses. FFTW encompasses a breadth-first as well as depth-first approaches. However, it favors the recursive approach. Even though there is no fixed radix-length, 32 is typical [**?**].

3.4.4 Performance and Comparison with Intel MKL

FFTW is a library for scientific computing, performance is therefore an important aspect. The authors benchmarked their implementation against most other implementations of the FFT [**?**]. The results are published on the FFTW homepage [**?**]. However, these benchmark date back to 2003 and are not considered here. The most competitive implementation today is Intel's Math Kernel Library (Intel MKL) [**?**],[**?**].

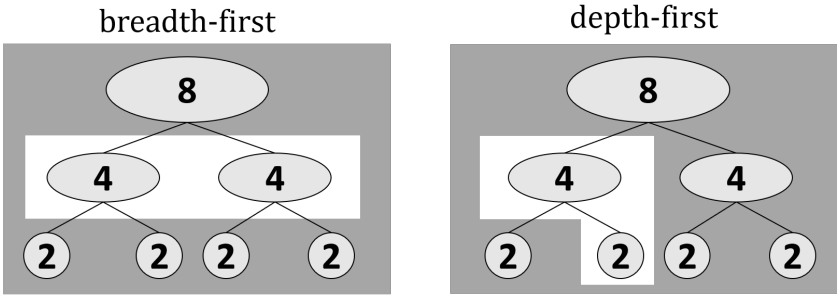

Figure 10: Traditional iterative breadth-first versus recursive depth-first orderung for radix-2 FFT of size 8[**?**].

In [**?**] the authors compare FFTW-2.15, FFTW-3.3.7, and Intel MKL FFT for 2D DFTs of complex power-of-two inputs using 36 threads. In this setting FFTW-2.1.5 showed better average speed than FFTW-3.3.7. The authors explain it by a less architecture-specific approach in FFTW-2.1.5 compared to FFTW-3.3.7. Figure 11 shows the comparison between FFTW-2.1.5 and Intel MKL FFT3. Intel's library shows higher peak performance as well as higher average speed. However, it has significantly higher variation in speed, because the library is optimized to specific DFT-lengths. FFTW-2.1.5. shows steadier performance and is faster in 162 out of 1000 problem sizes[**?**]. For non-power-of-two, FFTW still outperforms Intel MKL according to Steven G. Johnson in a talk at the JuliaCon 2019 [**?**].

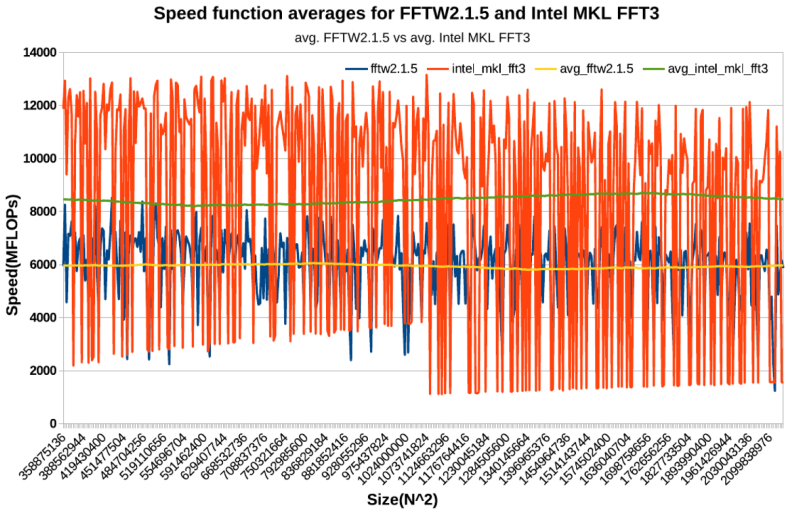

Figure 11: Speed functions of FFTW-2.1.5 and Intel MKL FFT3 and average speeds[**?**].

The results of the performance benchmark are consistent with the initial goals of FFTW, as the authors prioritized generality, which also applies to the length of the DFTs over specific optimization.

3.5 Limitations and Outlook

Although the FFT provides detailed information about the frequency contents of a given signal by calculating the DFT, it cannot detect when a certain frequency occurs. There are many signals of interest that are not stationary in frequency. A melody played by a single instrument could be (in a simplified way) described as the evolution in time of the frequency characterizing the note. The classical FFT cannot provide this information[**?**], [**?**].

In order to gain information of the time resolution of frequency, the DFT is performed on a specific window, which iterates over the time domain. Hence, a DFT of each window is created, giving frequency information in the specific time frame. This is a so called short-time Fourier transform (STFT). Gabor proposed to use Gaussian window functions. Figure 12 shows how a Gaussian function centered at τ translated

over the time signal to take a DFT in the domain $[\tau-a, \tau+a]$. The choice of an appropriate window function is essential for the properties of the STFT and the subject of deep mathematical discussions[**?**].

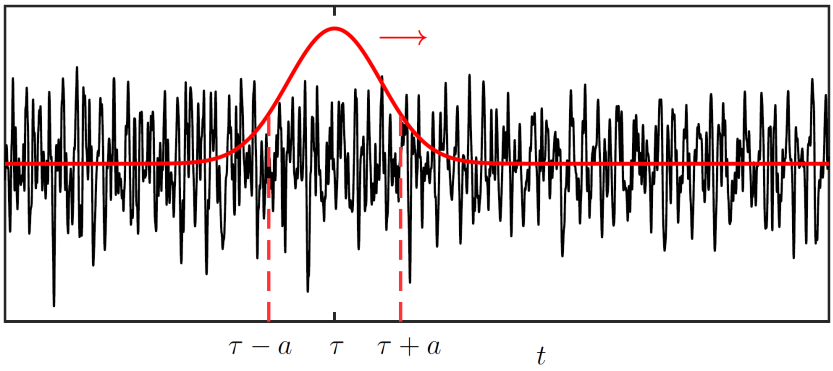

Figure 12: Illustration of the Gabor transform: A Gaussian window function translates over a time signal to compute the STFT [**?**].

The visualization of the STFT is the spectrogram. It is often plotted in 2D and the colors indicate the intensities of the frequencies at a specific time. An example for an audio signal is given in Figure 13.

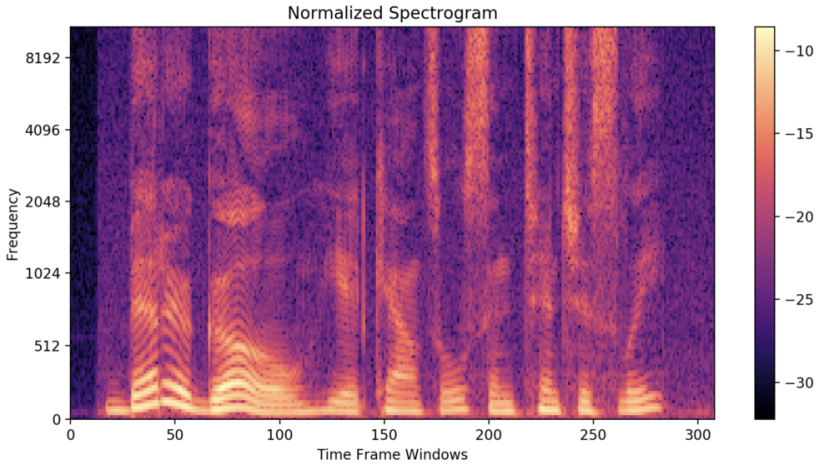

Figure 13: Spectrogram of an audio signal where the colors represent certain frequency intensities in time[**?**].

The size of the interval $[\tau-a, \tau+a]$ in Figure 12 determines the time resolution of the STFT. However, the smaller the interval chosen, the lesser information it contains about the frequencies. This is the fundamental uncertainty principle: It is impossible to attain both high time and frequency resolution. The Fourier transform is perfectly resolved in frequency without any information about time. The time series in contrast is perfectly resolved in time without any frequency information. The spectrogram resolves both but with lower resolution in each domain. Basically, the product of information in frequency and information in time is bounded by a constant[**?**].

The most important approach to get a better trade-off between time and frequency resolution are wavelet transforms. Wavelets transforms allow multiresolution analysis of signals. Different time and frequency fidelities can be used for different frequency bands as depicted in Figure 14(d). Typically higher time resolutions are also used for higher frequencies. Wavelet transforms are, in particular, useful for multiscale processes, which can be found in epidemiology, seismology, or turbulence. Wavelets are furthermore the leading approach for image compression[**?**],[**?**].

4 Conclusion

The FFT is considered to be one of the most important algorithms of the 20th century [**?**],[**?**]. Its significance can be grasped, by the vast variety of its applications: signal processing, data analysis, communication, data

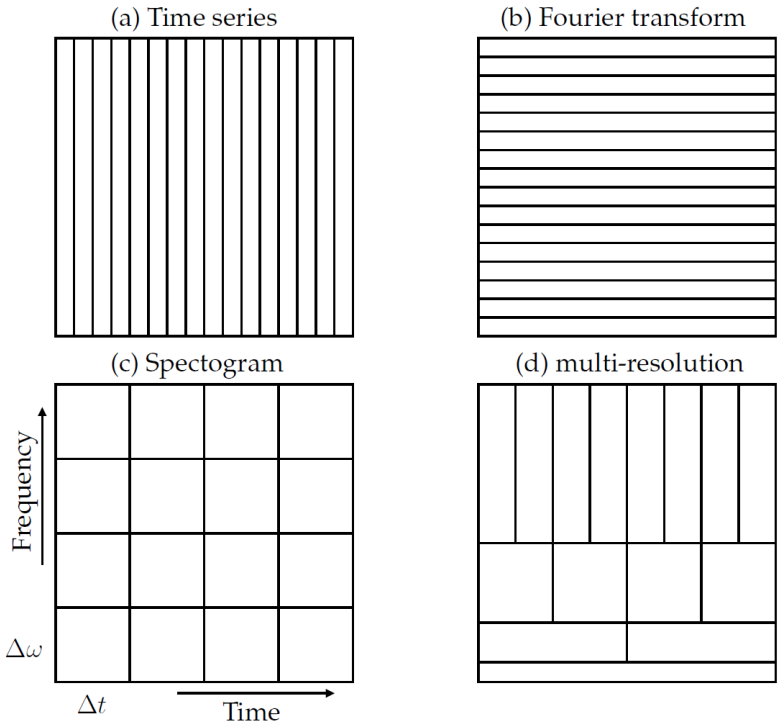

Figure 14: According to the uncertainty principle resolution in time limits resolution in frequency and vice versa. Multiresolution approaches resolve high frequency bands also high in time[**?**].

compression, and many more.

In its core the FFT allows us to compute the DFT very efficiently. Thus, it enables a fast transition from the time into the frequency domain and vice versa. Necessary computations can be reduced from $\mathcal{O}(N^2)$ to $\mathcal{O}(N\log(N))$. The most important and widely used FFT algorithm is the Cooley–Tukey algorithm. It splits the initial problem into smaller DFTs, which are reconnected by twiddle factors. The computational savings in this approach come from intrinsic symmetries in the DFT that allow the reuse of computations. Important classifications of the Cooley–Tukey algorithm are whether the initial data are split in the time or frequency domain (DIT versus DIF) and in how many sub-DFTs the intial problem is split "radix-X". Furthermore the problem can be recursively solved depth-first or iteratively breadth-first.

Beside the Cooley–Tukey algorithm, there are other FFT-algorithms of importance. Many are tailored for special problems. The fast Hatley transform specializes in real input data and leads to more efficient computations than the Cooley–Tukey algorithm. Rader's algorithm allows the computation of DFTs of prime size, by using mathematical convolutions.

In addition to a mathematically efficient algorithm, good implementations are needed. FFTW is one of the leading libraries implementing the FFT. It is designed to be very flexible in regard to the DFT that is to be solved. As it used code generation and performance measurements to find an optimal "plan" for each architecture, FFTW is also competitive in terms of performance.

The classical FFT does not give information about the change of frequency in time. However, splitting the initial time domain and performing separate FFTs leads to the STFT, which allows us to render frequencies dependent on time. The uncertainty principle limits the amount of information that can be gained from a signal. Hence better resolution in time leads directly to worse resolution in frequency and vice versa.