WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

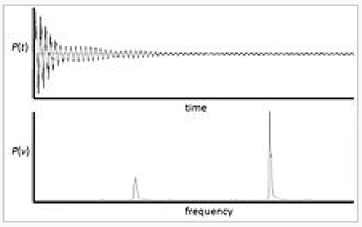
Languages
العربية
Català
Čeština
Dansk
Deutsch
Español
فارسی
Français
한국어
हिन्दी
Bahasa Indonesia
Italiano
עברית
Nederlands
日本語
Polski
Português
Русский
Српски / srpski
Srpskohrvatski / српскохрватски
Svenska
தமிழ்
Türkçe
Українська
Tiếng Việt
中文
Edit links

Fast Fourier transform

From Wikipedia, the free encyclopedia

"FFT" redirects here. For other uses, see FFT (disambiguation).

A **fast Fourier transform** (**FFT**) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors.[1] As a result, it manages to reduce the complexity of computing the DFT from $O(n^2)$, which arises if one simply applies the definition of DFT, to $O(n \log n)$, where n is the data size.



Time and frequency domain for the same signal

Fast Fourier transforms are widely used for many applications in engineering, science, and mathematics. The basic ideas were popularized in 1965, but some algorithms had been derived as early as 1805.[2] In 1994 Gilbert Strang described the FFT as "the most important numerical algorithm of our lifetime"[3] and it was included in Top 10 Algorithms of 20th Century by the IEEE journal Computing in Science & Engineering.[4]

Contents

Overview [edit]

This section **does not cite any references or sources**. Please help improve this section by adding citations to reliable sources. Unsourced material may be challenged and removed. *(June 2015)*

There are many different FFT algorithms involving a wide range of mathematics, from simple complex-number arithmetic to group theory and number theory; this article gives an overview of the available techniques and some of their general properties, while the specific algorithms are described in subsidiary articles linked below.

The DFT is obtained by decomposing a sequence of values into components of different frequencies. This operation is useful in many fields (see discrete Fourier transform for properties and applications of the transform) but computing it directly from the definition is often too slow to be practical. An FFT is a way to compute the same result more quickly: computing the DFT of *N* points in the naive way, using the definition, takes O(N^2) arithmetical operations, while an FFT can compute the same DFT in only O(*N* log *N*) operations. The difference in speed can be enormous, especially for long data sets where *N* may be in the thousands or millions. In practice, the computation time can be reduced by several orders of magnitude in such cases, and the improvement is roughly proportional to *N* / log(*N*). This huge improvement made the calculation of the DFT practical; FFTs are of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations to algorithms for quick multiplication of large integers.

The best-known FFT algorithms depend upon the factorization of *N*, but there are FFTs with O(*N* log *N*) complexity for all *N*, even for prime *N*. Many FFT algorithms only depend on the fact that $e^{-\frac{2\pi i}{N}}$ is an *N*-th primitive root of unity, and thus can be applied to analogous transforms over any finite field, such as number-theoretic transforms. Since the inverse DFT is the same as the DFT, but with the opposite sign in the exponent and a 1/*N* factor, any FFT algorithm can easily be adapted for it.

History [edit]

The development of fast algorithms for DFT can be traced to Gauss's unpublished work in 1805 when he needed it to interpolate the orbit of asteroids Pallas and Juno from sample observations.[5] His method was very similar to the one published in 1965 by Cooley and Tukey, who are generally credited for the invention of the modern generic FFT algorithm. While Gauss's work predated even Fourier's results in 1822, he did not analyze the computation time and eventually used other methods to achieve his goal.

Between 1805 and 1965, some versions of FFT were published by other authors. Yates in 1932 published his version called *interaction algorithm*, which provided efficient computation of Hadamard and Walsh transforms.[6] Yates' algorithm is still used in the field of statistical design and analysis of experiments. In 1942, Danielson and Lanczos published their version to compute DFT for x-ray crystallography, a field where calculation of Fourier transforms presented a formidable bottleneck.[7] While many methods in the past had focused on reducing the constant factor for $O(n^2)$ computation by taking advantage of *symmetries*, Danielson and Lanczos realized that one could use the *periodicity* and apply a "doubling trick" to get $O(n \log n)$ runtime.[8]

Cooley and Tukey published a more general version of FFT in 1965 that is applicable when N is composite and not necessarily a power of 2.[9] Tukey came up with the idea during a meeting of President Kennedy’s Science Advisory Committee where a discussion topic involved detecting nuclear tests by the Soviet Union by setting up sensors to surround the country from outside. To analyze the output of these sensors, a fast Fourier transform algorithm would be needed. Tukey's idea was taken by Richard Garwin and given to Cooley (both worked at IBM's Watson labs) for implementation while hiding the original purpose from him for security reasons. The pair published the paper in a relatively short six months.[10] As Tukey didn't work at IBM, the patentability of the idea was doubted and the algorithm went into the public domain, which, through the computing revolution of the next decade, made FFT one of the indispensable algorithms in digital signal processing.

Definition and speed [edit]

An FFT computes the DFT and produces exactly the same result as evaluating the DFT definition directly; the most important difference is that an FFT is much faster. (In the presence of round-off error, many FFT algorithms are also much more accurate than evaluating the DFT definition directly, as discussed below.)

Let x_0, ..., x_{N-1} be complex numbers. The DFT is defined by the formula

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}} \qquad k = 0, \dots, N-1.$$

Evaluating this definition directly requires O(N^2) operations: there are *N* outputs X_k, and each output requires a sum of *N* terms. An FFT is any method to compute the same results in O(*N* log *N*) operations. More precisely, all known FFT algorithms require Θ(*N* log *N*) operations (technically, O only denotes an upper bound), although there is no known proof that a lower complexity score is impossible.(Johnson and Frigo, 2007)

To illustrate the savings of an FFT, consider the count of complex multiplications and additions. Evaluating the DFT's sums directly involves N^2 complex multiplications and *N*(*N*−1) complex additions [of which O(*N*) operations can be saved by eliminating trivial operations such as multiplications by 1]. The well-known radix-2 Cooley–Tukey algorithm, for *N* a power of 2, can compute the same result with only (N/2)$\log_2(N)$ complex multiplications (again, ignoring simplifications of multiplications by 1 and similar) and $N\log_2(N)$ complex additions. In practice, actual performance on modern computers is usually dominated by factors other than the speed of arithmetic operations and the analysis is a complicated subject (see, e.g., Frigo & Johnson, 2005), but the overall improvement from O(N^2) to O(N log N) remains.

Algorithms [edit]

Cooley–Tukey algorithm [edit]

Main article: Cooley–Tukey FFT algorithm

By far the most commonly used FFT is the Cooley–Tukey algorithm. This is a divide and conquer algorithm that recursively breaks down a DFT of any composite size $N = N_1N_2$ into many smaller DFTs of sizes N_1 and N_2, along with O(N) multiplications by complex roots of unity traditionally called twiddle factors (after Gentleman and Sande, 1966[11]).

This method (and the general idea of an FFT) was popularized by a publication of J. W. Cooley and J. W. Tukey in 1965,[9] but it was later discovered[2] that those two authors had independently re-invented an algorithm known to Carl Friedrich Gauss around 1805[12] (and subsequently rediscovered several times in limited forms).

The best known use of the Cooley–Tukey algorithm is to divide the transform into two pieces of size $N/2$ at each step, and is therefore limited to power-of-two sizes, but any factorization can be used in general (as was known to both Gauss and Cooley/Tukey[2]). These are called the **radix-2** and **mixed-radix** cases, respectively (and other variants such as the split-radix FFT have their own names as well). Although the basic idea is recursive, most traditional implementations rearrange the algorithm to avoid explicit recursion. Also, because the Cooley–Tukey algorithm breaks the DFT into smaller DFTs, it can be combined arbitrarily with any other algorithm for the DFT, such as those described below.

Other FFT algorithms [edit]

Main articles: Prime-factor FFT algorithm, Bruun's FFT algorithm, Rader's FFT algorithm and Bluestein's FFT algorithm

There are other FFT algorithms distinct from Cooley–Tukey.

Cornelius Lanczos did pioneering work on the FFS and FFT with G.C. Danielson (1940).

For $N = N_1N_2$ with coprime N_1 and N_2, one can use the Prime-Factor (Good-Thomas) algorithm (PFA), based on the Chinese Remainder Theorem, to factorize the DFT similarly to Cooley–Tukey but without the twiddle factors. The Rader-Brenner algorithm (1976) is a Cooley–Tukey-like factorization but with purely imaginary twiddle factors, reducing multiplications at the cost of increased additions and reduced numerical stability; it was later superseded by the split-radix variant of Cooley–Tukey (which achieves the same multiplication count but with fewer additions and without sacrificing accuracy). Algorithms that recursively factorize the DFT into smaller operations other than DFTs include the Bruun and QFT algorithms. (The Rader-Brenner and QFT algorithms were proposed for power-of-two sizes, but it is possible that they could be adapted to general composite n. Bruun's algorithm applies to arbitrary even composite sizes.) Bruun's algorithm, in particular, is based on interpreting the FFT as a recursive factorization of the polynomial z^N-1, here into real-coefficient polynomials of the form z^M-1 and $z^{2M} + az^M + 1$.

Another polynomial viewpoint is exploited by the Winograd algorithm, which factorizes z^N-1 into cyclotomic polynomials—these often have coefficients of 1, 0, or −1, and therefore require few (if any) multiplications, so Winograd can be used to obtain minimal-multiplication FFTs and is often used to find efficient algorithms for small factors. Indeed, Winograd showed that the DFT can be computed with only O(N) irrational multiplications, leading to a proven achievable lower bound on the number of multiplications for power-of-two sizes; unfortunately, this comes at the cost of many more additions, a tradeoff no longer favorable on modern processors with hardware multipliers. In particular, Winograd also makes use of the PFA as well as an algorithm by Rader for FFTs of *prime* sizes.

Rader's algorithm, exploiting the existence of a generator for the multiplicative group modulo prime N, expresses a DFT of prime size n as a cyclic convolution of (composite) size $N-1$, which can then be computed by a pair of ordinary FFTs via the convolution theorem (although Winograd uses other convolution methods). Another prime-size FFT is due to L. I. Bluestein, and is sometimes called the chirp-z algorithm; it also re-expresses a DFT as a convolution, but this time of the *same* size (which can be zero-padded to a power of two and evaluated by radix-2 Cooley–Tukey FFTs, for example), via the identity

$$nk = -(k-n)^2/2 + n^2/2 + k^2/2.$$

FFT algorithms specialized for real and/or symmetric data [edit]

In many applications, the input data for the DFT are purely real, in which case the outputs satisfy the symmetry

$$X_{N-k} = X_k^*$$

and efficient FFT algorithms have been designed for this situation (see e.g. Sorensen, 1987). One approach consists of taking an ordinary algorithm (e.g. Cooley–Tukey) and removing the redundant parts of the computation, saving roughly a factor of two in time and memory. Alternatively, it is possible to express an *even*-length real-input DFT as a complex DFT of half the length (whose real and imaginary parts are the even/odd elements of the original real data), followed by O(*N*) post-processing operations.

It was once believed that real-input DFTs could be more efficiently computed by means of the discrete Hartley transform (DHT), but it was subsequently argued that a specialized real-input DFT algorithm (FFT) can typically be found that requires fewer operations than the corresponding DHT algorithm (FHT) for the same number of inputs. Bruun's algorithm (above) is another method that was initially proposed to take advantage of real inputs, but it has not proved popular.

There are further FFT specializations for the cases of real data that have even/odd symmetry, in which case one can gain another factor of (roughly) two in time and memory and the DFT becomes the discrete cosine/sine transform(s) (DCT/DST). Instead of directly modifying an FFT algorithm for these cases, DCTs/DSTs can also be computed via FFTs of real data combined with O(*N*) pre/post processing.

Computational issues [edit]

Bounds on complexity and operation counts [edit]

List of unsolved problems in computer science

What is the lower bound on the complexity of fast Fourier transform algorithms? Can they be faster than Θ(N log N)?

A fundamental question of longstanding theoretical interest is to prove lower bounds on the complexity and exact operation counts of fast Fourier transforms, and many open problems remain. It is not even rigorously proved whether DFTs truly require Ω(*N* log(*N*)) (i.e., order *N* log(*N*) or greater) operations, even for the simple case of power of two sizes, although no algorithms with lower complexity are known. In particular, the count of arithmetic operations is usually the focus of such questions, although actual performance on modern-day computers is determined by many other factors such as cache or CPU pipeline optimization.

Following pioneering work by Winograd (1978), a tight Θ(*N*) lower bound *is* known for the number of real multiplications required by an FFT. It can be shown that only $4N - 2\log_2^2 N - 2\log_2 N - 4$ irrational real multiplications are required to compute a DFT of power-of-two length $N = 2^m$. Moreover, explicit algorithms that achieve this count are known (Heideman & Burrus, 1986; Duhamel, 1990). Unfortunately, these algorithms require too many additions to be practical, at least on modern computers with hardware multipliers (Duhamel, 1990; Frigo & Johnson, 2005).

A tight lower bound is *not* known on the number of required additions, although lower bounds have been proved under some restrictive assumptions on the algorithms. In 1973, Morgenstern proved an Ω(*N* log(*N*)) lower bound on the addition count for algorithms where the multiplicative constants have bounded magnitudes (which is true for most but not all FFT algorithms). This result, however, applies only to the unnormalized Fourier transform (which is a scaling of a unitary matrix by a factor of $\sqrt{N}$), and does not explain why the Fourier matrix is harder to compute than any other unitary matrix (including the identity matrix) under the same scaling. Pan (1986) proved an Ω(*N* log(*N*)) lower bound assuming a bound on a measure of the FFT algorithm's "asynchronicity", but the generality of this assumption is unclear. For the case of power-of-two *N*, Papadimitriou (1979) argued that the number $N \log_2 N$ of complex-number additions achieved by Cooley–Tukey algorithms is *optimal* under certain assumptions on the graph of the algorithm (his assumptions imply, among other things, that no additive identities in the roots of unity are exploited). (This argument would imply that at least $2N \log_2 N$ real additions are required, although this is not a tight bound because extra additions are required as part of complex-number multiplications.) Thus far, no published FFT algorithm has achieved fewer than $N \log_2 N$ complex-number additions (or their equivalent) for power-of-two *N*.

A third problem is to minimize the *total* number of real multiplications and additions, sometimes called the "arithmetic complexity" (although in this context it is the exact count and not the asymptotic complexity that is being considered). Again, no tight lower bound has been proven. Since 1968, however, the lowest published count for power-of-two *N* was long achieved by the split-radix FFT algorithm, which requires $4N \log_2 N - 6N + 8$ real multiplications and additions for *N* > 1. This was recently reduced to $\sim \frac{34}{9} N \log_2 N$ (Johnson and Frigo, 2007; Lundy and Van Buskirk, 2007). A slightly larger count (but still better than split radix for *N*≥256) was shown to be provably optimal for *N*≤512 under additional restrictions on the possible algorithms (split-radix-like flowgraphs with unit-modulus multiplicative factors), by reduction to a Satisfiability Modulo Theories problem solvable by brute force (Haynal & Haynal, 2011).

Most of the attempts to lower or prove the complexity of FFT algorithms have focused on the ordinary complex-data case, because it is the simplest. However, complex-data FFTs are so closely related to algorithms for related problems such as real-data FFTs, discrete cosine transforms, discrete Hartley transforms, and so on, that any improvement in one of these would immediately lead to improvements in the others (Duhamel & Vetterli, 1990).

Approximations [edit]

All of the FFT algorithms discussed above compute the DFT exactly (in exact arithmetic, i.e. neglecting floating-point errors). A few "FFT" algorithms have been proposed, however, that compute the DFT *approximately*, with an error that can be made arbitrarily small at the expense of increased computations. Such algorithms trade the approximation error for increased speed or other properties. For example, an approximate FFT algorithm by Edelman et al. (1999) achieves lower communication requirements for parallel computing with the help of a fast multipole method. A wavelet-based approximate FFT by Guo and Burrus (1996) takes sparse inputs/outputs (time/frequency localization) into account more efficiently than is possible with an exact FFT. Another algorithm for approximate computation of a subset of the DFT outputs is due to Shentov et al. (1995). The Edelman algorithm works equally well for sparse and non-sparse data, since it is based on the compressibility (rank deficiency) of the Fourier matrix itself rather than the compressibility (sparsity) of the data. Conversely, if the data are sparse—that is, if only *K* out of *N* Fourier coefficients are nonzero—then the complexity can be reduced to O(*K*log(*N*)log(*N*/*K*)), and this has been demonstrated to lead to practical speedups compared to an ordinary FFT for N/K>32 in a large-N example (N=2^{22}) using a probabilistic approximate algorithm (which estimates the largest *K* coefficients to several decimal places).[13]

Accuracy [edit]

Even the "exact" FFT algorithms have errors when finite-precision floating-point arithmetic is used, but these errors are typically quite small; most FFT algorithms, e.g. Cooley–Tukey, have excellent numerical properties as a consequence of the pairwise summation structure of the algorithms. The upper bound on the relative error for the Cooley–Tukey algorithm is O(ε log *N*), compared to O(ε$N^{3/2}$) for the naïve DFT formula,[11] where ε is the machine floating-point relative precision. In fact, the root mean square (rms) errors are much better than these upper bounds, being only O(ε √log *N*) for Cooley–Tukey and O(ε √*N*) for the naïve DFT (Schatzman, 1996). These results, however, are very sensitive to the accuracy of the twiddle factors used in the FFT (i.e. the trigonometric function values), and it is not unusual for incautious FFT implementations to have much worse accuracy, e.g. if they use inaccurate trigonometric recurrence formulas. Some FFTs other than Cooley–Tukey, such as the Rader-Brenner algorithm, are intrinsically less stable.

In fixed-point arithmetic, the finite-precision errors accumulated by FFT algorithms are worse, with rms errors growing as O(√*N*) for the Cooley–Tukey algorithm (Welch, 1969). Moreover, even achieving this accuracy requires careful attention to scaling to minimize loss of precision, and fixed-point FFT algorithms involve rescaling at each intermediate stage of decompositions like Cooley–Tukey.

To verify the correctness of an FFT implementation, rigorous guarantees can be obtained in O(*N*log(*N*)) time by a simple procedure checking the linearity, impulse-response, and time-shift properties of the transform on random inputs (Ergün, 1995).

Multidimensional FFTs [edit]

As defined in the multidimensional DFT article, the multidimensional DFT

$$X_{\mathbf{k}} = \sum_{\mathbf{n}=0}^{\mathbf{N}-1} e^{-2\pi i \mathbf{k}\cdot(\mathbf{n}/\mathbf{N})} x_{\mathbf{n}}$$

transforms an array $x_{\mathbf{n}}$ with a *d*-dimensional vector of indices $\mathbf{n} = (n_1, \ldots, n_d)$ by a set of *d* nested summations (over $n_j = 0 \ldots N_j - 1$ for each *j*), where the division **n**/**N**, defined as $\mathbf{n}/\mathbf{N} = (n_1/N_1, \ldots, n_d/N_d)$, is performed element-wise. Equivalently, it is the composition of a sequence of *d* sets of one-dimensional DFTs, performed along one dimension at a time (in any order).

This compositional viewpoint immediately provides the simplest and most common multidimensional DFT algorithm, known as the **row-column** algorithm (after the two-dimensional case, below). That is, one simply performs a sequence of *d* one-dimensional FFTs (by any of the above algorithms): first you transform along the n_1 dimension, then along the n_2 dimension, and so on (or actually, any ordering works). This method is easily shown to have the usual O(*N*log(*N*)) complexity, where $N = N_1 \cdot N_2 \cdot \ldots \cdot N_d$ is the total number of data points transformed. In particular, there are *N*/*N*$_1$ transforms of size *N*$_1$, etcetera, so the complexity of the sequence of FFTs is:

$$\frac{N}{N_1} O(N_1 \log N_1) + \cdots + \frac{N}{N_d} O(N_d \log N_d)$$

$$= O\left(N \left[\log N_1 + \cdots + \log N_d\right]\right) = O(N \log N).$$

In two dimensions, the $x_{\mathbf{k}}$ can be viewed as an $n_1 \times n_2$ matrix, and this algorithm corresponds to first performing the FFT of all the rows (resp. columns), grouping the resulting transformed rows (resp. columns) together as another $n_1 \times n_2$ matrix, and then performing the FFT on each of the columns (resp. rows) of this second matrix, and similarly grouping the results into the final result matrix.

In more than two dimensions, it is often advantageous for cache locality to group the dimensions recursively. For example, a three-dimensional FFT might first perform two-dimensional FFTs of each planar "slice" for each fixed *n*$_1$, and then perform the one-dimensional FFTs along the *n*$_1$ direction. More generally, an asymptotically optimal cache-oblivious algorithm consists of recursively dividing the dimensions into two groups $(n_1, \ldots, n_{d/2})$ and $(n_{d/2+1}, \ldots, n_d)$ that are transformed recursively (rounding if *d* is not even) (see Frigo and Johnson, 2005). Still, this remains a straightforward variation of the row-column algorithm that ultimately requires only a one-dimensional FFT algorithm as the base case, and still has O(*N*log(*N*)) complexity. Yet another variation is to perform matrix transpositions in between transforming subsequent dimensions, so that the transforms operate on contiguous data; this is especially important for out-of-core and distributed memory situations where accessing non-contiguous data is extremely time-consuming.

There are other multidimensional FFT algorithms that are distinct from the row-column algorithm, although all of them have O(*N*log(*N*)) complexity. Perhaps the simplest non-row-column FFT is the vector-radix FFT algorithm, which is a generalization of the ordinary Cooley–Tukey algorithm where one divides the transform dimensions by a vector $\mathbf{r} = (r_1, r_2, \ldots, r_d)$ of radices at each step. (This may also have cache benefits.) The simplest case of vector-radix is where all of the radices are equal (e.g. vector-radix-2 divides *all* of the dimensions by two), but this is not necessary. Vector radix with only a single non-unit radix at a time, i.e. $\mathbf{r} = (1, \ldots, 1, r, 1, \ldots, 1)$, is essentially a row-column algorithm. Other, more complicated, methods include polynomial transform algorithms due to Nussbaumer (1977), which view the transform in terms of convolutions and polynomial products. See Duhamel and Vetterli (1990) for more information and references.

Other generalizations [edit]

An O(*N*$^{5/2}$log(*N*)) generalization to spherical harmonics on the sphere *S*^{2} with *N*^{2} nodes was described by Mohlenkamp,[14] along with an algorithm conjectured (but not proven) to have O(*N*^{2} log^2(*N*)) complexity; Mohlenkamp also provides an implementation in the libftsh library. A spherical-harmonic algorithm with O(*N*^{2}log(*N*)) complexity is described by Rokhlin and Tygert.[15]

The Fast Folding Algorithm is analogous to the FFT, except that it operates on a series of binned waveforms rather than a series of real or complex scalar values. Rotation (which in the FFT is multiplication by a complex phasor) is a circular shift of the component waveform.

Various groups have also published "FFT" algorithms for non-equispaced data, as reviewed in Potts *et al.* (2001). Such algorithms do not strictly compute the DFT (which is only defined for equispaced data), but rather some approximation thereof (a non-uniform discrete Fourier transform, or NDFT, which itself is often computed only approximately). More generally there are various other methods of spectral estimation.

Applications [edit]

FFT's importance derives from the fact that in signal processing and image processing it has made working in frequency domain equally computationally feasible as working in temporal or spatial domain. Some of the important applications of FFT includes,[10][16]

- Fast large integer and polynomial multiplication
- Efficient matrix-vector multiplication for Toeplitz, circulant and other structured matrices
- Filtering algorithms
- Fast algorithms for discrete cosine or sine transforms (example, Fast DCT used for JPEG, MP3/MPEG encoding)
- Fast Chebyshev approximation
- Fast Discrete Hartley Transform

- Solving Difference Equations

Time-Frequency Relationships [edit]

[Note: the terms “complex data” (= real and imaginary numbers) and “in-phase/quadrature data” (I/Q numbers) are used interchangeably].

Time Domain [edit]

The waveform is sampled at $\mathbf{N}$ equi-spaced points in time $n = 0, 1, ..., N-1$
(It is preferable for N to be a power of two).
When the sampled waveform is real, the data consists of N real numbers—but if the sampled waveform is complex, the data consists of N sample-pairs of I/Q data.
If the time interval between samples (the "sample interval") is $\mathbf{\Delta t}$ secs, then
the total length of the sample record is $\mathbf{T}$, where

$$\mathbf{T} = \mathbf{N}.\mathbf{\Delta t} \ \text{secs}$$

Frequency Domain [edit]

For real waveform data, the number of frequency points, ignoring the zero frequency bin, is N/2 +1. [i.e.there are N/2 +1 pairs of (I/Q) data].
For complex (I/Q) waveform data, the number of frequency points = N, with N/2 pairs of complex data at positive frequencies and N/2 pairs of complex data at negative frequencies.
If the frequency separation of the data points, (the “frequency resolution”) is $\mathbf{\Delta f}$ Hz, then
the maximum frequency of the display (the “bandwidth”) is $+\mathbf{F}_{max}$, for real numbers, and $\pm\mathbf{F}_{max}$ for complex data, where

$$\mathbf{F}_{max} = \frac{\mathbf{N}}{\mathbf{2}}.\mathbf{\Delta f} \ \text{Hz}$$

Interrelationships between time and frequency domains [edit]

$$\mathbf{\Delta t} = \frac{\mathbf{1}}{\mathbf{2}.\mathbf{F_{max}}} \ \text{secs}$$

$$\mathbf{\Delta f} = \frac{\mathbf{1}}{\mathbf{T}} \ \text{Hz}$$

Research Areas [edit]

- **Big FFTs**: With explosion of big data in fields such as astronomy, the need for 512k FFTs has arisen for certain interferometry calculations. The data collected by projects such as MAP and LIGO require FFTs of tens of billions of points. As this size does not fit in to main memory, so called out-of-core FFTs are an active area of research.[17]
- **Approximate FFTs**: For applications such as MRI, it is necessary to compute DFTs for nonuniformly spaced grid points and/or frequencies. Multipole based approaches can compute approximate quantities with factor of runtime increase.[18]
- **Group FFTs**: The FFT may also be explained and interpreted using group representation theory that allows for further generalization. A function on any compact group, including non cyclic, has an expansion in terms of a basis of irreducible matrix elements. It remains active area of research to find efficient algorithm for performing this change of basis. Applications including efficient spherical harmonic expansion, analyzing certain markov processes, robotics etc.[19]
- **Quantum FFTs**: Shor's fast algorithm for integer factorization on a quantum computer has a subroutine to compute DFT of a binary vector. This is implemented as sequence of 1- or 2-bit quantum gates now known as quantum FFT, which is effectively the Cooley-Tukey FFT realized as a particular factorization of the Fourier matrix. Extension to these ideas is currently being explored.

Implementation with C++ [edit]

Here is a C++ source code for a simple FFT implementation:[20]

```
const double TwoPi = 6.283185307179586;

void FFTAnalysis(double *AVal, double *FTvl, int Nvl, int Nft) {
  int i, j, n, m, Mmax, Istp;
  double Tmpr, Tmpi, Wtmp, Theta;
  double Wpr, Wpi, Wr, Wi;
  double *Tmvl;

  n = Nvl * 2; Tmvl = new double[n+1];

  for (i = 0; i <Nvl; i++) {
    j = i * 2; Tmvl[j] = 0; Tmvl[j+1] = AVal[i];
  }

  i = 1; j = 1;
  while (i <n) {
    if (j> i) {
      Tmpr = Tmvl[i]; Tmvl[i] = Tmvl[j]; Tmvl[j] = Tmpr;
      Tmpr = Tmvl[i+1]; Tmvl[i+1] = Tmvl[j+1]; Tmvl[j+1] = Tmpr;
    }
    i = i + 2; m = Nvl;
    while ((m>= 2) && (j> m)) {
      j = j - m; m = m>> 2;
    }
    j = j + m;
  }

  Mmax = 2;
  while (n> Mmax) {
    Theta = -TwoPi / Mmax; Wpi = Sin(Theta);
    Wtmp = Sin(Theta / 2); Wpr = Wtmp * Wtmp * 2;
    Istp = Mmax * 2; Wr = 1; Wi = 0; m = 1;

    while (m <Mmax) {
      i = m; m = m + 2; Tmpr = Wr; Tmpi = Wi;
      Wr = Wr - Tmpr * Wpr - Tmpi * Wpi;
      Wi = Wi + Tmpr * Wpi - Tmpi * Wpr;

      while (i <n) {
        j = i + Mmax;
        Tmpr = Wr * Tmvl[j] - Wi * Tmvl[j+1];
        Tmpi = Wi * Tmvl[j] + Wr * Tmvl[j+1];

        Tmvl[j] = Tmvl[i] - Tmpr; Tmvl[j+1] = Tmvl[i+1] - Tmpi;
        Tmvl[i] = Tmvl[i] + Tmpr; Tmvl[i+1] = Tmvl[i+1] + Tmpi;
        i = i + Istp;
      }
    }

    Mmax = Istp;
  }

  for (i = 1; i <Nft; i++) {
    j = i * 2; FTvl[i] = Sqrt(Sqr(Tmvl[j]) + Sqr(Tmvl[j+1]));
  }

  delete []Tmvl;
}
```

Popular Culture [edit]

Fourier transform has been occasionally mentioned in popular cultural through movies, TV series, cartoon series as a reference to someone geek.

In the movie Transformers (2007): Character played by Rachael Taylor informs that *"The signal pattern is learning, it's EVOLVING on its own, and you need to move past Fourier transforms and start thinking quantum* *mechanics..."*

In the movie No Way Out (1997): An image processing guru declares that "*Ya got the wrong eigenvalue!! Put a Fourier transform on it. PRONTO!!*" to help character played by Kevin Costner see the details in the photograph.

The webcomic xkcd has featured topic of Fourier transforms multiple times.[21]

See also [edit]

- Cooley–Tukey FFT algorithm
- Prime-factor FFT algorithm
- Bruun's FFT algorithm
- Rader's FFT algorithm
- Bluestein's FFT algorithm
- Butterfly diagram – a diagram used to describe FFTs.
- Odlyzko–Schönhage algorithm applies the FFT to finite Dirichlet series.
- Overlap add/Overlap save – efficient convolution methods using FFT for long signals
- Spectral music (involves application of FFT analysis to musical composition)
- Spectrum analyzers – Devices that perform an FFT
- FFTW "Fastest Fourier Transform in the West" – C library for the discrete Fourier transform (DFT) in one or more dimensions.
- FFTS – The Fastest Fourier Transform in the South.
- FFTPACK – another Fortran FFT library (public domain)
- Goertzel algorithm – Computes individual terms of discrete Fourier transform
- Time series
- Math Kernel Library
- Fast Walsh–Hadamard transform
- Generalized distributive law
- Multidimensional transform

References [edit]

1. ^ Charles Van Loan, *Computational Frameworks for the Fast Fourier Transform* (SIAM, 1992).
2. ^ *a b c* Heideman, M. T.; Johnson, D. H.; Burrus, C. S. (1984). "Gauss and the history of the fast Fourier transform". *IEEE ASSP Magazine* **1** (4): 14–21. doi:10.1109/MASSP.1984.1162257.
3. ^ Strang, Gilbert (May–June 1994). "Wavelets". *American Scientist* **82** (3): 253. JSTOR 29775194.
4. ^ Dongarra, J.; Sullivan, F. (January 2000). "Guest Editors Introduction to the top 10 algorithms". *Computing in Science Engineering* **2** (1): 22–23. doi:10.1109/MCISE.2000.814652. ISSN 1521-9615.
5. ^ Heideman, Michael T.; Johnson, Don H.; Burrus, C. Sidney (1985-09-01). "Gauss and the history of the fast Fourier transform". *Archive for History of Exact Sciences* **34** (3): 265–277. doi:10.1007/BF00348431. ISSN 0003-9519.
6. ^ Yates, Frank (1937). "The design and analysis of factorial experiments". *Technical Communication no. 35 of the Commonwealth Bureau of Soils*.
7. ^ Danielson and Lanczos (1942). "Some improvements in practical Fourier analysis and their application to x-ray scattering from liquids". *Journal of the Franklin Institute* **233** (4): 365–380. doi:10.1016/S0016-0032(42)90767-1.
8. ^ Cooley, James W.; Lewis, Peter A.W.; Welch, Peter D. (June 1967). "Historical notes on the fast Fourier transform". *IEEE Transactions on Audio and Electroacoustics* **15** (2): 76–79. doi:10.1109/TAU.1967.1161903. ISSN 0018-9278.
9. ^ *a b* Cooley, James W.; Tukey, John W. (1965). "An algorithm for the machine calculation of complex Fourier series". *Mathematics of Computation* **19** (90): 297–301. doi:10.1090/S0025-5718-1965-0178586-1. ISSN 0025-5718.
10. ^ *a b* Rockmore, D.N. (January 2000). "The FFT: an algorithm the whole family can use". *Computing in Science Engineering* **2** (1): 60–64. doi:10.1109/5992.814659. ISSN 1521-9615.
11. ^ *a b* Gentleman, W. M.; Sande, G. (1966). "Fast Fourier transforms—for fun and profit". *Proc. AFIPS* **29**: 563–578. doi:10.1145/1464291.1464352.
12. ^ Carl Friedrich Gauss, 1866. "Theoria interpolationis methodo nova tractata," *Werke* band **3**, 265–327. Göttingen: Königliche Gesellschaft der Wissenschaften.
13. ^ Haitham Hassanieh, Piotr Indyk, Dina Katabi, and Eric Price, "Simple and Practical Algorithm for Sparse Fourier Transform" (PDF), ACM-SIAM Symposium On Discrete Algorithms (SODA), Kyoto, January 2012. See also the sFFT Web Page.
14. ^ Mohlenkamp, Martin J. (1999). "A F ast T ransform for Spherical Harmonics" (PDF). *The Journal of F ourier A nalysis and Applic ations* **5** (2/3): 159–184. Retrieved 18 September 2014.

15. ^ ROKHLIN, VLADIMIR; TYGERT, MARK (2006). "FAST ALGORITHMS FOR SPHERICAL HARMONIC EXPANSIONS" (PDF). *SIAM Journal on Scientific Computing* **27** (6): 903–1928. Retrieved 18 September 2014.
16. ^ Chu and George. "16". *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press. pp. 153–168. ISBN 9781420049961.
17. ^ Cormen and Nicol (1998). "Performing out-of-core FFTs on parallel disk systems". *Parallel Computing* **24** (1): 5–20. doi:10.1016/S0167-8191(97)00114-2.
18. ^ Dutt, A.; Rokhlin, V. (November 1, 1993). "Fast Fourier Transforms for Nonequispaced Data". *SIAM Journal on Scientific Computing* **14** (6): 1368–1393. doi:10.1137/0914081. ISSN 1064-8275.
19. ^ Rockmore, Daniel N. (2004). Byrnes, Jim, ed. "Recent Progress and Applications in Group FFTs". *Computational Noncommutative Algebra and Applications*. NATO Science Series II: Mathematics, Physics and Chemistry (Springer Netherlands) **136**: 227–254. ISBN 978-1-4020-1982-1.
20. ^ fa:تبدیل سریع فوریه
21. ^ "xkcd: Fourier". *xkcd.com*. Retrieved 2015-05-24.

- Brenner, N.; Rader, C. (1976). "A New Principle for Fast Fourier Transformation". *IEEE Acoustics, Speech & Signal Processing* **24** (3): 264–266. doi:10.1109/TASSP.1976.1162805.
- Brigham, E. O. (2002). "The Fast Fourier Transform". New York: Prentice-Hall
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, 2001. *Introduction to Algorithms*, 2nd. ed. MIT Press and McGraw-Hill. ISBN 0-262-03293-7. Especially chapter 30, "Polynomials and the FFT."
- Duhamel, Pierre (1990). "Algorithms meeting the lower bounds on the multiplicative complexity of length-2^n DFTs and their connection with practical algorithms". *IEEE Trans. Acoust. Speech. Sig. Proc.* **38** (9): 1504–151. doi:10.1109/29.60070.
- P. Duhamel and M. Vetterli, 1990, Fast Fourier transforms: a tutorial review and a state of the art, *Signal Processing* **19**: 259–299.
- A. Edelman, P. McCorquodale, and S. Toledo, 1999, The Future Fast Fourier Transform?, *SIAM J. Sci. Computing* **20**: 1094–1114.
- D. F. Elliott, & K. R. Rao, 1982, *Fast transforms: Algorithms, analyses, applications*. New York: Academic Press.
- Funda Ergün, 1995, Testing multivariate linear functions: Overcoming the generator bottleneck, *Proc. 27th ACM Symposium on the Theory of Computing*: 407–416.
- Frigo, M.; Johnson, S. G. (2005). "The Design and Implementation of FFTW3" (PDF). *Proceedings of the IEEE* **93**: 216–231. doi:10.1109/jproc.2004.840301.
- H. Guo and C. S. Burrus, 1996, Fast approximate Fourier transform via wavelets transform, *Proc. SPIE Intl. Soc. Opt. Eng.* **2825**: 250–259.
- H. Guo, G. A. Sitton, C. S. Burrus, 1994, The Quick Discrete Fourier Transform, *Proc. IEEE Conf. Acoust. Speech and Sig. Processing (ICASSP)* **3**: 445–448.
- Steve Haynal and Heidi Haynal, "Generating and Searching Families of FFT Algorithms", *Journal on Satisfiability, Boolean Modeling and Computation* vol. 7, pp. 145–187 (2011).
- Heideman, Michael T.; Burrus, C. Sidney (1986). "On the number of multiplications necessary to compute a length-2^n DFT". *IEEE Trans. Acoust. Speech. Sig. Proc.* **34** (1): 91–95. doi:10.1109/TASSP.1986.1164785.
- Johnson, S. G.; Frigo, M. (2007). "A modified split-radix FFT with fewer arithmetic operations" (PDF). *IEEE Trans. Signal Processing* **55** (1): 111–119. doi:10.1109/tsp.2006.882087.
- T. Lundy and J. Van Buskirk, 2007. "A new matrix approach to real FFTs and convolutions of length 2^k," *Computing* **80** (1): 23–45.
- Kent, Ray D. and Read, Charles (2002). *Acoustic Analysis of Speech*. ISBN 0-7693-0112-6. Cites Strang, G. (1994)/May–June). Wavelets. *American Scientist, 82,* 250–255.
- Morgenstern, Jacques (1973). "Note on a lower bound of the linear complexity of the fast Fourier transform". *J. ACM* **20** (2): 305–306. doi:10.1145/321752.321761.
- Mohlenkamp, M. J. (1999). "A fast transform for spherical harmonics" (PDF). *J. Fourier Anal. Appl.* **5** (2–3): 159–184. doi:10.1007/BF01261607.
- Nussbaumer, H. J. (1977). "Digital filtering using polynomial transforms". *Electronics Lett.* **13** (13): 386–387. doi:10.1049/el:19770280.
- V. Pan, 1986, The trade-off between the additive complexity and the asyncronicity of linear and bilinear algorithms, *Information Proc. Lett.* **22**: 11–14.
- Christos H. Papadimitriou, 1979, Optimality of the fast Fourier transform, *J. ACM* **26**: 95–102.
- D. Potts, G. Steidl, and M. Tasche, 2001. "Fast Fourier transforms for nonequispaced data: A tutorial", in: J.J. Benedetto and P. Ferreira (Eds.), *Modern Sampling Theory: Mathematics and Applications* (Birkhauser).
- Press, WH; Teukolsky, SA; Vetterling, WT; Flannery, BP (2007), "Chapter 12. Fast Fourier Transform",

Numerical Recipes: The Art of Scientific Computing (3rd ed.), New York: Cambridge University Press, ISBN 978-0-521-88068-8

- Rokhlin, Vladimir; Tygert, Mark (2006). "Fast algorithms for spherical harmonic expansions". *SIAM J. Sci. Computing* **27** (6): 1903–1928. doi:10.1137/050623073.
- Schatzman, James C. (1996). "Accuracy of the discrete Fourier transform and the fast Fourier transform". *SIAM J. Sci. Comput* **17**: 1150–1166. doi:10.1137/s1064827593247023.
- Shentov, O. V.; Mitra, S. K.; Heute, U.; Hossen, A. N. (1995). "Subband DFT. I. Definition, interpretations and extensions". *Signal Processing* **41** (3): 261–277. doi:10.1016/0165-1684(94)00103-7.
- Sorensen, H. V.; Jones, D. L.; Heideman, M. T.; Burrus, C. S. (1987). "Real-valued fast Fourier transform algorithms". *IEEE Trans. Acoust. Speech Sig. Processing* **35** (35): 849–863. doi:10.1109/TASSP.1987.1165220. See also Sorensen, H.; Jones, D.; Heideman, M.; Burrus, C. (1987). "Corrections to "Real-valued fast Fourier transform algorithms"". *IEEE Transactions on Acoustics, Speech, and Signal Processing* **35** (9): 1353–1353. doi:10.1109/TASSP.1987.1165284.
- Welch, Peter D. (1969). "A fixed-point fast Fourier transform error analysis". *IEEE Trans. Audio Electroacoustics* **17** (2): 151–157. doi:10.1109/TAU.1969.1162035.
- Winograd, S. (1978). "On computing the discrete Fourier transform". *Math. Computation* **32** (141): 175–199. doi:10.1090/S0025-5718-1978-0468306-4. JSTOR 2006266.

External links [edit]

- Fast Fourier Algorithm
- *Fast Fourier Transforms*, Connexions online book edited by C. Sidney Burrus, with chapters by C. Sidney Burrus, Ivan Selesnick, Markus Pueschel, Matteo Frigo, and Steven G. Johnson (2008).
- Links to FFT code and information online.
- National Taiwan University – FFT
- FFT programming in C++ — Cooley–Tukey algorithm.
- Online documentation, links, book, and code.
- Using FFT to construct aggregate probability distributions
- Sri Welaratna, "Thirty years of FFT analyzers", *Sound and Vibration* (January 1997, 30th anniversary issue). A historical review of hardware FFT devices.
- FFT Basics and Case Study Using Multi-Instrument
- FFT Textbook notes, PPTs, Videos at Holistic Numerical Methods Institute.
- ALGLIB FFT Code GPL Licensed multilanguage (VBA, C++, Pascal, etc.) numerical analysis and data processing library.
- MIT's sFFT MIT Sparse FFT algorithm and implementation.
- VB6 FFT VB6 optimized library implementation with source code.
- Fast Fourier transform illustrated Demo examples and FFT calculator.

Categories: FFT algorithms | Digital signal processing | Discrete transforms