From DFT to FFT — Why Complexity Matters

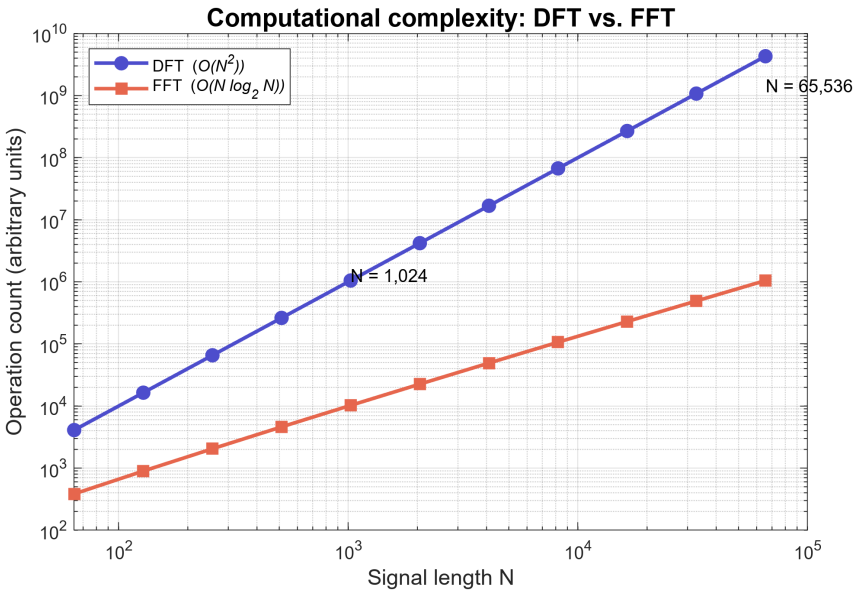
The goal of this lecture is to understand the computational cost of the Discrete Fourier Transform (DFT) and to motivate the need for a faster algorithm: the Fast Fourier Transform (FFT).

Computational cost of the DFT

The one-dimensional DFT of a discrete signal $x[n]$ of length N is defined as

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}kn}, \qquad k = 0, \ldots, N-1.$$

For each output coefficient $X[k]$, we perform N complex multiplications and $(N-1)$ complex additions. Since there are N outputs, the total computational cost scales as $\mathcal{O}(N^2)$. This quadratic complexity means that the computation time increases rapidly with signal length.
For example, for $N = 1{,}024$, the DFT requires on the order of 10^6 complex multiplications. For $N = 65{,}536$, the count grows to approximately 4.3×10^9. Even on modern hardware, such computation would be prohibitively expensive for large datasets.



Extension to 2D DFT

For a two-dimensional signal (image) of size $M \times N$, the DFT is written as

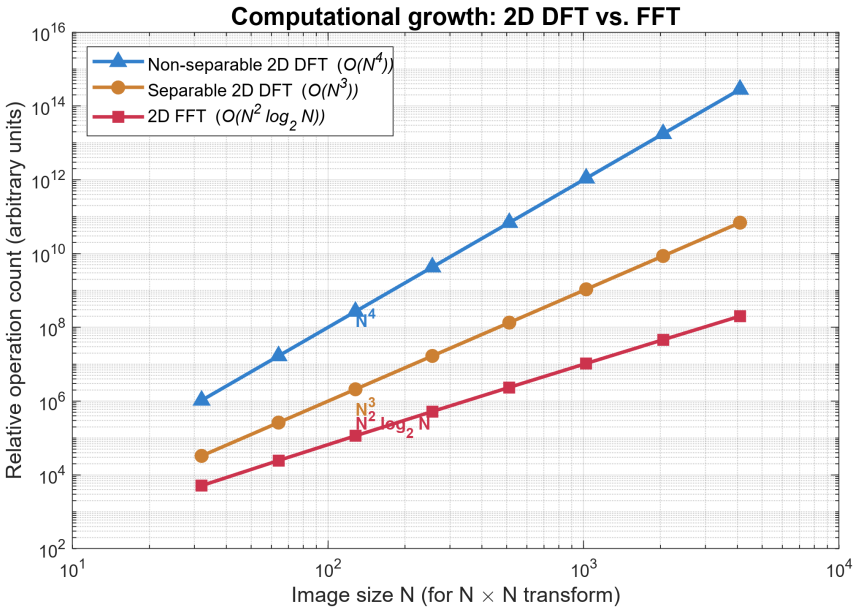
$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}.$$

The naive implementation performs a double sum for every pair (u, v), resulting in $\mathcal{O}(M^2N^2)$ operations.
Fortunately, the 2D DFT is ***separable***: a multidimensional transform can be computed as successive one-dimensional transforms, e.g. first along rows and then along columns.

This reduces the computational cost to

$$\mathcal{O}(MN(M+N)),$$

which for square images ($M = N$) simplifies to $\mathcal{O}(N^3)$. Although better than $\mathcal{O}(N^4)$, this cubic complexity is still high and becomes impractical for large images.



For an image of size $N \times N$, the orders of growth can be summarized as:

$$\text{Naive 2D DFT: } \mathcal{O}(N^4), \qquad \text{Separable 2D DFT: } \mathcal{O}(N^3), \qquad \text{2D FFT: } \mathcal{O}(N^2 \log_2 N).$$

Even with separability, the direct DFT becomes too slow as N increases, while the FFT provides orders of magnitude improvement.

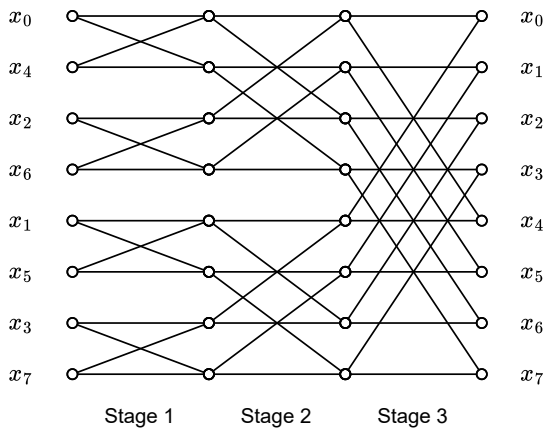
Practical considerations

Theoretical complexity only tells part of the story. In practice, runtime also depends on constant factors, memory access patterns, cache usage, and vectorization. The FFT algorithm is not only asymptotically faster but also ***cache-friendly***: it reorganizes computations into small "butterfly" structures that reuse data efficiently and minimize memory transfers. By contrast, the direct DFT involves poor locality and redundant recomputation.
Power-of-two lengths are typically fastest because they fit naturally into the radix-2 structure of the FFT. For other lengths, mixed-radix and Bluestein algorithms can be used. In higher dimensions (2D or 3D), the FFT is applied along each dimension sequentially, again leveraging separability and good data locality.

Reusing structure in the FFT

The key insight of the FFT lies in ***reusing structure*** within the DFT computation. The DFT involves many repeated exponential terms, called ***twiddle factors***. The FFT exploits their periodicity and symmetry to avoid redundant computations.

By recursively dividing the problem into even and odd indexed samples (the ***divide-and-conquer*** approach), large portions of the computation are shared. This reduces the overall complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log_2 N)$.
This idea is implemented in the ***radix-2 Cooley–Tukey FFT*** algorithm, which processes data in stages. Each stage performs small two-point "butterfly" operations, combining even and odd elements and multiplying by appropriate twiddle factors. The overall computation forms a highly structured flow graph.



Decimation in Time (DIT)

The most common FFT algorithm, known as ***decimation in time*** (DIT), begins by separating the DFT into contributions from the even and odd indexed samples. If we write $n = 2m$ for even indices and $n = 2m + 1$ for odd ones, we obtain

$$X[k] = \sum_{m=0}^{\frac{N}{2}-1} x[2m]\, W_N^{k(2m)} + \sum_{m=0}^{\frac{N}{2}-1} x[2m+1]\, W_N^{k(2m+1)}.$$

Since $W_N^{2k} = W_{N/2}^k$, this can be rewritten as

$$X[k] = E[k] + W_N^k\, O[k],$$

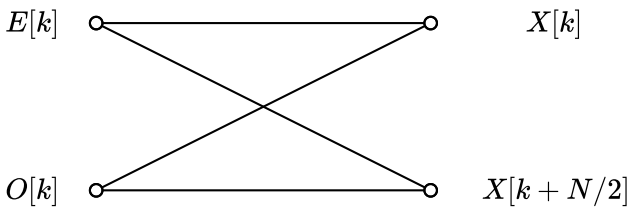
where

$$E[k] = \sum_m x[2m]\, W_{N/2}^{km}, \qquad O[k] = \sum_m x[2m+1]\, W_{N/2}^{km}.$$

Thus, the N-point DFT has been reduced to two DFTs of size $N/2$ (the even and odd parts), followed by simple additions and multiplications by powers of W_N^k (the so-called twiddle factors).
For $k = 0, \ldots, \frac{N}{2} - 1$, we obtain

$$X[k] = E[k] + W_N^k O[k], \qquad X[k + \tfrac{N}{2}] = E[k] - W_N^k O[k].$$

Each pair $(E[k], O[k])$ generates two outputs and forms a basic computational motif called a ***butterfly***. By recursively applying this decomposition to the E and O sequences, we arrive at the complete FFT structure.



Each butterfly requires one complex multiplication and two complex additions. Since there are $N/2$ butterflies per stage and $\log_2 N$ stages in total, the number of operations grows proportionally to $N \log_2 N$, a dramatic improvement over the naive N^2 complexity.

Bit-Reversed Ordering

The DIT FFT rearranges the input sequence before processing so that data are accessed locally at each stage. This reordering is called ***bit-reversal***. The index n is written in binary form, the order of bits is reversed, and the sequence is permuted accordingly. For example, for $N = 8$:

$$000, 001, 010, 011, 100, 101, 110, 111 \;\Rightarrow\; 000, 100, 010, 110, 001, 101, 011, 111.$$

After this reordering, the butterflies at each stage operate on contiguous elements.

Example

To illustrate how the radix-2 Decimation-in-Time FFT works in practice, let us compute the 8-point transform of the sequence

$$x[n] = [1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0],$$

using the forward FFT convention without $1/N$ scaling and with

$$W_N = e^{-j2\pi/N}.$$

We will explicitly go through the three stages of computation and the bit-reversal reordering that precedes them.
At each stage of the FFT, the variable m denotes the *current butterfly length*, i.e. the number of samples combined by a single butterfly block at that stage. In a radix-2 DIT FFT, m doubles with every stage:

$$m = 2,\ 4,\ 8,\ \ldots,\ N.$$

For $N = 8$, Stage 1 uses $m = 2$ (2-point butterflies), Stage 2 uses $m = 4$ (4-point groups), and Stage 3 uses $m = 8$ (the final 8-point combination).
Within each group of length m, the twiddle factors applied to the "odd" branches are determined by

$$w_r = W_N^{r\cdot(N/m)} = e^{-j2\pi r/m} = W_m^r, \qquad r = 0, 1, \ldots, \frac{m}{2} - 1.$$

This rule shows that at every stage, the global N-point roots of unity reduce to the local m-point ones. For example, for $N = 8$ and $m = 4$:

$$W_8^0 = 1, \qquad W_8^2 = e^{-j\pi/2} = -j,$$

which are exactly the twiddles used in Stage 2.

Stage 0 (Bit-Reversal Reordering)

For $N = 8$, the index n has three binary digits. Bit-reversal means that we reverse the order of these bits to obtain the new order in which samples should appear before Stage 1. The mapping is:

$$[0, 1, 2, 3, 4, 5, 6, 7] \;\Rightarrow\; [0, 4, 2, 6, 1, 5, 3, 7].$$

Reordering $x[n]$ according to this permutation gives

$$x^{(0)} = [x_0, x_4, x_2, x_6, x_1, x_5, x_3, x_7] = [1, 0, 1, 0, 1, 0, 1, 0].$$

This arrangement ensures that each butterfly in the following stages works with contiguous elements, enabling efficient in-place computation.

Stage 1 (Length-2 Butterflies)

In the first stage, the butterflies operate on consecutive pairs of elements:

$$(0, 1),\ (2, 3),\ (4, 5),\ (6, 7).$$

Since all twiddle factors are $W_8^0 = 1$, only additions and subtractions are needed:

$$(a, b) \longrightarrow (a + b,\ a - b).$$

Each pair here is $(1, 0)$, which yields the same outputs for every butterfly:

$$1 + 0 = 1, \quad 1 - 0 = 1.$$

Hence, after Stage 1 the entire sequence becomes

$$x^{(1)} = [1, 1, 1, 1, 1, 1, 1, 1].$$

Stage 2 (Length-4 Butterflies)

The second stage processes the data in groups of four samples. There are two such groups: [0..3] and [4..7]. At this stage ($m = 4$), each group forms a 4-point butterfly, with the twiddle factors:

$$W_8^0 = 1, \qquad W_8^2 = e^{-j\pi/2} = -j.$$

Using $x^{(1)} = [1, 1, 1, 1,\ 1, 1, 1, 1]$, we compute the transformed values within each group as follows:

$$\begin{aligned}
x'_0 &= x_0 + W_8^0 x_2 = 1 + 1 = 2, \\
x'_1 &= x_1 + W_8^2 x_3 = 1 + (-j) \cdot 1 = 1 - j, \\
x'_2 &= x_0 - W_8^0 x_2 = 1 - 1 = 0, \\
x'_3 &= x_1 - W_8^2 x_3 = 1 - (-j) \cdot 1 = 1 + j.
\end{aligned}$$

Both groups are identical and produce the same set of four results. Thus, after Stage 2 we have

$$x^{(2)} = [\,2,\ 1 - j,\ 0,\ 1 + j,\ 2,\ 1 - j,\ 0,\ 1 + j\,].$$

Stage 3 (Length-8 Butterfly = Final Combination)

In the final stage, the two 4-point groups are combined into a single 8-point FFT. The butterflies now connect pairs separated by four indices:

$$(0,4),\ (1,5),\ (2,6),\ (3,7).$$

The corresponding twiddle factors are:

$$W_8^0 = 1, \qquad W_8^1 = e^{-j\pi/4} = \tfrac{1}{\sqrt{2}}(1-j),$$

$$W_8^2 = e^{-j\pi/2} = -j, \qquad W_8^3 = e^{-j3\pi/4} = -\tfrac{1}{\sqrt{2}}(1+j).$$

Each pair (a, b) is transformed according to

$$(a, b) \to (a + wb,\ a - wb),$$

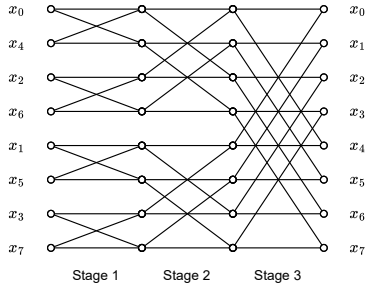
where w is the appropriate twiddle factor. Applying this to our data:
$X_0 = 2 + 1 \cdot 2 = 4$,
$X_1 = (1-j) + W_8^1(1-j) = (1-j) + \frac{1}{\sqrt{2}}(1-j)^2 = 1 - 2.4142j$,
$X_2 = 0$,
$X_3 = (1+j) + W_8^3(1+j) = (1+j) - \frac{1}{\sqrt{2}}(1+j)^2 = 1 - 0.4142j$,
$X_4 = 2 - 1 \cdot 2 = 0$,
$X_5 = (1-j) - W_8^1(1-j) = 1 + 0.4142j$,
$X_6 = 0$,
$X_7 = (1+j) - W_8^3(1+j) = 1 + 2.4142j$.

Final Spectrum and Interpretation

The final FFT result is therefore

$$X = [\,4,\ 1 - 2.4142j,\ 0,\ 1 - 0.4142j,\ 0,\ 1 + 0.4142j,\ 0,\ 1 + 2.4142j\,].$$

The large DC term $X_0 = 4$ indicates that most of the signal's energy lies in the constant (zero-frequency) component. The remaining nonzero values occur at small indices $k = 1$ and $k = 7$, which correspond to low frequencies. This matches our expectation, since the input $x[n]$ consists of four ones followed by four zeros — a slowly varying signal dominated by low-frequency content.
The complex conjugate symmetry of $X[k]$ around $k = 0$ and $k = 4$ reflects the fact that the input signal is real-valued.

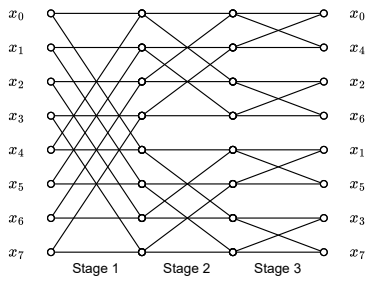


Decimation in Frequency (DIF)

The *Decimation in Frequency (DIF)* algorithm is an alternative factorization of the Discrete Fourier Transform. Whereas the DIT version splits the input signal into even and odd time indices, the DIF variant instead splits the *output spectrum* into even and odd frequency indices. This means that each stage of the computation separates contributions to even and odd frequency bins, performing a butterfly operation across increasingly smaller blocks as the algorithm proceeds.
At the beginning, butterflies connect elements that are far apart (separated by $N/2$), and with every subsequent stage, the block size m is halved:

$$N \to \frac{N}{2} \to \frac{N}{4} \to \cdots \to 2.$$

The twiddle factors multiply the lower (odd-frequency) outputs. Unlike DIT, the DIF FFT assumes that the *input* is in natural order and that bit-reversal reordering appears only at the end, on the output spectrum.



Butterfly Operation

For a block of size m, with base index b and offset $r = 0, 1, \ldots, m/2 - 1$, the butterfly operation is defined as:

$$\begin{aligned} u &= x[b + r], \quad v = x[b + r + m/2], \\ s &= u + v, \qquad d = (u - v)\, W_m^r, \quad W_m = e^{-j\frac{2\pi}{m}}, \\ x[b + r] &\leftarrow s, \quad x[b + r + m/2] \leftarrow d. \end{aligned}$$

The upper outputs (s) correspond to even frequency bins, and the lower outputs (d) to odd frequency bins, which are multiplied by the corresponding twiddle factor W_m^r.
Each subsequent stage halves the butterfly length, progressively refining the decomposition of the frequency spectrum.

Example

Let us again consider the example signal

$$x = [1,\ 1,\ 1,\ 1,\ 0,\ 0,\ 0,\ 0],$$

and compute its FFT using the DIF scheme. We will go through three stages of computation and conclude with the bit-reversal reordering of the final output.

Stage 1 (Length-8 Butterflies)

At the first stage, the butterfly length is $m = 8$. Pairs of elements are separated by four indices: $(0, 4)$, $(1, 5)$, $(2, 6)$, $(3, 7)$. The twiddle factors are given by:

$$W_8^r = e^{-j\frac{2\pi r}{8}}, \qquad r = 0, 1, 2, 3.$$

The computation for each pair is:

$$\begin{aligned}
&r{=}0 : u = 1,\ v = 0 \Rightarrow s = 1,\ d = (1-0)W_8^0 = 1,\\
&r{=}1 : u = 1,\ v = 0 \Rightarrow s = 1,\ d = (1-0)W_8^1 = \tfrac{1}{\sqrt{2}}(1-j),\\
&r{=}2 : u = 1,\ v = 0 \Rightarrow s = 1,\ d = (1-0)W_8^2 = -j,\\
&r{=}3 : u = 1,\ v = 0 \Rightarrow s = 1,\ d = (1-0)W_8^3 = -\tfrac{1}{\sqrt{2}}(1+j).
\end{aligned}$$

After Stage 1:

$$x^{(1)} = [\,1,\ 1,\ 1,\ 1,\ 1,\ \tfrac{1}{\sqrt{2}}(1-j),\ -j,\ -\tfrac{1}{\sqrt{2}}(1+j)\,].$$

Stage 2 (Length-4 Butterflies)

The second stage processes two groups of four samples each: [0..3] and [4..7]. Now $m = 4$, and the corresponding twiddles are:

$$W_4^r = e^{-j\frac{2\pi r}{4}} = [1, -j].$$

For the upper group $[1, 1, 1, 1]$, we obtain:

$$\begin{aligned}
&r{=}0 : u = 1,\ v = 1 \Rightarrow s = 2,\ d = (1-1)W_4^0 = 0,\\
&r{=}1 : u = 1,\ v = 1 \Rightarrow s = 2,\ d = (1-1)W_4^1 = 0.
\end{aligned}$$

For the lower group $[1,\ \frac{1}{\sqrt{2}}(1-j),\ -j,\ -\frac{1}{\sqrt{2}}(1+j)]$:

$$\begin{aligned}
&r{=}0 : u = 1,\ v = -j \Rightarrow s = 1-j,\ d = (1+j)W_4^0 = 1+j,\\
&r{=}1 : u = \tfrac{1}{\sqrt{2}}(1-j),\ v = -\tfrac{1}{\sqrt{2}}(1+j) \Rightarrow s = -\sqrt{2}\,j,\\
&\qquad d = (\tfrac{1}{\sqrt{2}}(1-j) - (-\tfrac{1}{\sqrt{2}}(1+j)))W_4^1 = \sqrt{2}(-j) = -\sqrt{2}\,j.
\end{aligned}$$

After Stage 2:

$$x^{(2)} = [\,2,\ 2,\ 0,\ 0,\ 1-j,\ -\sqrt{2}\,j,\ 1+j,\ -\sqrt{2}\,j\,].$$

Stage 3 (Length-2 Butterflies)

In the final stage, $m = 2$, with pairs $(0, 1)$, $(2, 3)$, $(4, 5)$, $(6, 7)$. All twiddles are $W_2^0 = 1$, so only additions and subtractions occur:

$$\begin{aligned}
&(0,1) : (2,2) \Rightarrow s = 4,\ d = 0,\\
&(2,3) : (0,0) \Rightarrow s = 0,\ d = 0,\\
&(4,5) : (1-j,\ -\sqrt{2}j) \Rightarrow s = 1-(1+\sqrt{2})j,\ d = 1+(\sqrt{2}-1)j,\\
&(6,7) : (1+j,\ -\sqrt{2}j) \Rightarrow s = 1+(1-\sqrt{2})j,\ d = 1+(1+\sqrt{2})j.
\end{aligned}$$

After Stage 3:

$$x^{(3)} = [\,4,\ 0,\ 0,\ 0,\ 1-2.4142j,\ 1+0.4142j,\ 1-0.4142j,\ 1+2.4142j\,].$$

Bit-Reversal and Final Output

The DIF FFT produces its outputs in bit-reversed order of frequency indices. For $N = 8$, the bit-reversal permutation is:

$$\pi = [0, 4, 2, 6, 1, 5, 3, 7].$$

Reordering $x^{(3)}$ according to this permutation yields the final spectrum:

$$X[k] = x^{(3)}[\pi(k)] = [\,4,\ 1 - 2.4142j,\ 0,\ 1 - 0.4142j,\ 0,\ 1 + 0.4142j,\ 0,\ 1 + 2.4142j\,].$$

Interpretation

The result is identical to that obtained via the DIT FFT, confirming that both algorithms compute the same DFT. The key difference lies in data flow: DIT requires bit-reversal of the input and processes from small to large butterflies, whereas DIF keeps the input in natural order and performs bit-reversal on the output. Both achieve the same $\mathcal{O}(N \log N)$ complexity and produce identical numerical results.

FFT and Input Length

The radix-2 FFT relies on the assumption that the sequence length is a power of two:

$$N = 2^m.$$

If this condition is not met, the binary splitting pattern cannot continue until only 2-point butterflies remain. In practice, many signals or images have arbitrary lengths, and several methods exist to handle these cases.
If $N = 10$, we can still compute the DFT directly as

$$X[k] = \sum_{n=0}^{9} x[n] e^{-j2\pi kn/10},$$

but we must use one of three strategies: zero-padding, mixed-radix FFT, or a prime-length algorithm such as Rader's or Bluestein's.

What Is Radix?

The term ***radix*** denotes the branching factor of the FFT decomposition. In the Cooley–Tukey algorithm we express

$$N = R \times M,$$

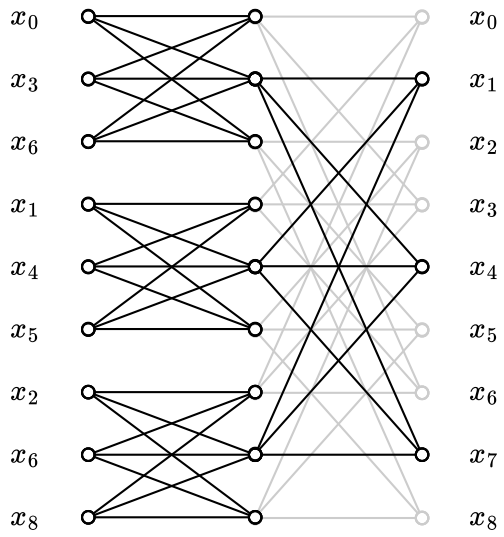
where R is the radix—the number of subtransforms per stage. For $R = 2$ we obtain the radix-2 FFT (splitting even and odd samples), for $R = 3$ the radix-3 FFT, and for $R = 4$ the radix-4 FFT. Each stage processes R inputs per butterfly and recursively computes smaller DFTs. All of these achieve the same asymptotic complexity $\mathcal{O}(N \log N)$ but differ in structure and efficiency.

Radix-3 FFT

In a radix-3 FFT, each butterfly combines three inputs (x_0, x_1, x_2):

$$\begin{aligned} X_0 &= x_0 + x_1 + x_2, \\ X_1 &= x_0 + W_3^1 x_1 + W_3^2 x_2, \\ X_2 &= x_0 + W_3^2 x_1 + W_3^1 x_2, \end{aligned}$$

where $W_3^1 = e^{-j2\pi/3} = -\frac{1}{2} - j\frac{\sqrt{3}}{2}$ and $W_3^2 = e^{-j4\pi/3} = -\frac{1}{2} + j\frac{\sqrt{3}}{2}$. Each output is a weighted sum of all three inputs, with the weights corresponding to complex rotations evenly spaced around the unit circle.

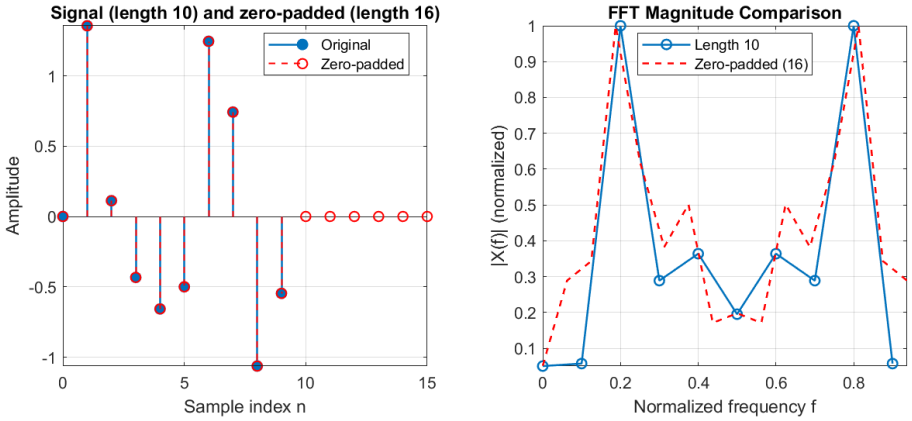


Zero Padding

Zero-padding extends a signal by appending zeros at the end before taking the FFT. This operation does not change the actual frequency content of the signal, but it increases the density of frequency samples in the discrete spectrum. If the zero-padded length is $N_p > N$, we have

$$X_p[k] = \sum_{n=0}^{N_p-1} x_p[n] e^{-j2\pi kn/N_p}.$$

Zero-padding is therefore useful for visualization or interpolation of spectra, though the added resolution is purely interpolated rather than new information.

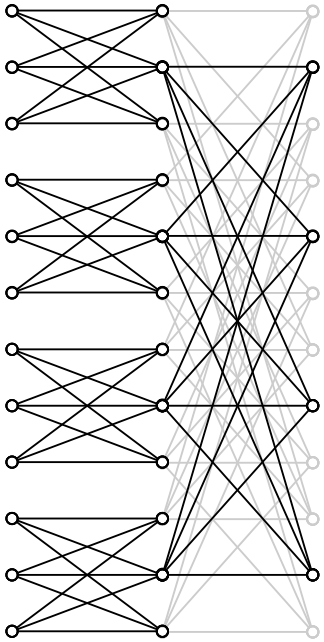


Mixed-Radix FFTs

For lengths that are not powers of two, the FFT can be generalized by factorizing N into smaller integers:

$$N = N_1 \times N_2 \times \dots$$

and using a different radix at each stage. This is known as the ***mixed-radix FFT***. For example, $N = 12 = 4 \times 3$ can be computed by performing three 3-point FFTs followed by four 4-point FFTs. The complexity remains $\mathcal{O}(N \log N)$, although the constant factors depend on the particular factorization.



Practical Recommendations

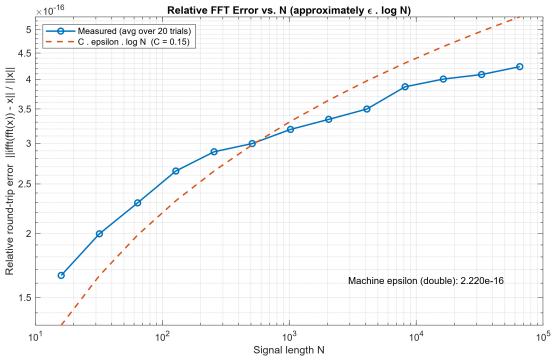
Whenever possible, use data lengths that are powers of two. These allow the use of radix-2 FFTs, which are both the simplest and the most efficient, thanks to hardware and software optimizations. For arbitrary lengths, one can either zero-pad to the next power of two or rely on high-performance libraries such as FFTW, NumPy, or MATLAB, which automatically select optimal mixed-radix or Bluestein algorithms.

Numerical Aspects of FFT

The Fast Fourier Transform operates on complex numbers using finite-precision arithmetic. Each arithmetic operation introduces a tiny rounding error, which cannot be avoided because of the limited number of bits available to represent real and imaginary values. During the FFT computation, these small errors propagate through successive stages. Fortunately, their accumulation is mild: the total numerical error typically grows only proportionally to the logarithm of the transform length,

$$\| \hat{X} - X \| \approx \mathcal{O}(\varepsilon \log N),$$

where ε denotes the machine precision (around 10^{-16} in double precision). In practice, for real-world signals or images, such numerical inaccuracies are completely negligible compared with sensor noise or quantization effects introduced during acquisition.



The FFT algorithm is considered numerically stable, or more precisely, backward stable. This means that the result computed by the FFT is mathematically identical to the exact DFT of slightly perturbed input data. In other words, FFT does not amplify roundoff errors in a harmful way.

From the implementation point of view, performance of FFT depends largely on how the data are stored and accessed in memory. Efficient implementations exploit in-place computation, minimizing memory transfers, and use cache-friendly radix blocks such as radix-2, radix-4, or radix-8 to achieve good locality of reference. Power-of-two data lengths are particularly advantageous, as they allow optimal reuse of twiddle factors and regular memory access patterns, leading to maximal computational efficiency.

Two-Dimensional FFT

The two-dimensional Discrete Fourier Transform (2D DFT) generalizes the 1D case to two spatial dimensions and is fundamental for image analysis. For an image $f(x, y)$ of

size $M \times N$, the transform is defined as

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y)\, e^{-j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}.$$

The quantity $F(u,v)$ describes the frequency content of the image in both directions, where u corresponds to horizontal and v to vertical spatial frequency. The inverse transform reconstructs the image:

$$f(x,y) = \frac{1}{MN} \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u,v)\, e^{j2\pi\left(\frac{ux}{M}+\frac{vy}{N}\right)}.$$

The 2D DFT therefore provides a complete spectral representation of the image, showing how strongly each spatial frequency contributes to its overall structure.
An important property of the 2D FFT is its separability. The transform can be computed as two successive one-dimensional FFTs:

$$F(u,v) = \mathrm{FFT}_x\big(\mathrm{FFT}_y(f(x,y))\big).$$

In practice, one first computes FFTs along all rows (horizontal direction) and then along all columns (vertical direction). This separable structure makes the 2D FFT efficient to implement. The computational complexity is

$$\mathcal{O}(MN \log N + MN \log M) \approx \mathcal{O}(MN \log N),$$

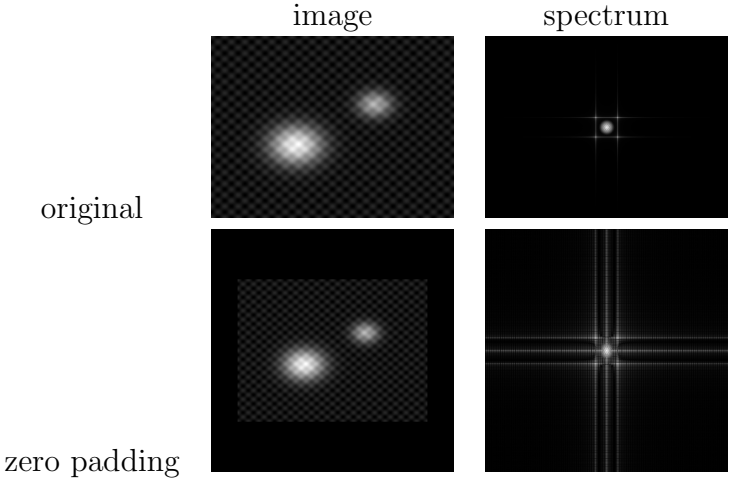
which for square images simplifies to $\mathcal{O}(N^2 \log N)$.

Zero Padding in 2D FFT

In image processing, zero padding means extending an image with zeros to dimensions that are convenient for FFT computation. Most FFT algorithms, especially those based on radix-2 decomposition, achieve optimal speed when each dimension is a power of two:

$$M' = 2^{\lceil \log_2 M \rceil}, \qquad N' = 2^{\lceil \log_2 N \rceil}.$$

Zero padding does not alter the image content; it merely provides more frequency samples in the resulting spectrum, leading to a visually smoother appearance. Moreover, padding can significantly accelerate computation since power-of-two sizes align perfectly with the recursive structure of the FFT algorithm.



Practical use – Filtering

Filtering in the Spatial Domain

Image filtering is commonly formulated as two-dimensional convolution,

$$g(x,y) = (f * h)(x,y) = \sum_{m=-M_h}^{M_h} \sum_{n=-N_h}^{N_h} f(x-m, y-n)\, h(m,n),$$

where $f(x,y)$ is the input image, $h(m,n)$ the filter kernel (impulse response), and $g(x,y)$ the output image. This operation has computational complexity $\mathcal{O}(N^2K^2)$ for an image of size $N \times N$ and a kernel of size $K \times K$. For small filters, such as 3×3 or 5×5, direct convolution is acceptable, but for larger filters it becomes prohibitively slow. Typical examples of spatial-domain filters include Gaussian blurring, edge detection, sharpening, and directional filters.

The Convolution Theorem

A fundamental result linking spatial and frequency domains is the Convolution Theorem:

$$f * h \;\leftrightarrow_{\mathrm{FFT}}\; F(u,v)\, H(u,v).$$

This theorem states that convolution in one domain corresponds to multiplication in the other. Consequently, spatial filtering can be implemented much more efficiently by transforming the image and the kernel into the frequency domain, multiplying them elementwise, and transforming the result back. The convolution theorem thus provides the theoretical foundation for FFT-based filtering methods.

Circular Convolution and Wrap-Around Error

Since the FFT implicitly assumes periodic boundary conditions, convolution computed via FFT corresponds to circular convolution,

$$g_{\mathrm{circ}}(x,y) = (f * h)_{\mathrm{circular}}.$$

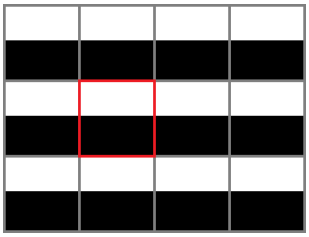
This leads to wrap-around artifacts, where parts of the filtered image from one edge appear wrapped to the opposite edge. To obtain the correct linear convolution, both the image and the filter must be extended by zero padding:

$$f_p = \mathrm{pad}(f, K-1), \qquad h_p = \mathrm{pad}(h, N-1),$$

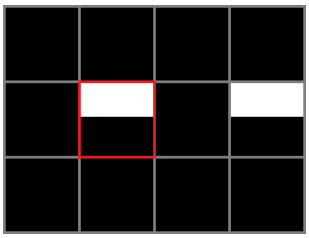
so that the padded sizes satisfy

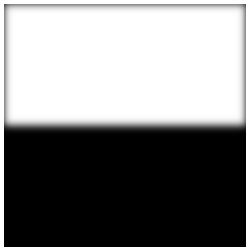
$$M' \geq M + P - 1, \qquad N' \geq N + Q - 1,$$

where $M \times N$ is the image size and $P \times Q$ is the filter size.



Proper padding avoids this problem and ensures that FFT-based filtering gives the same result as direct spatial convolution.



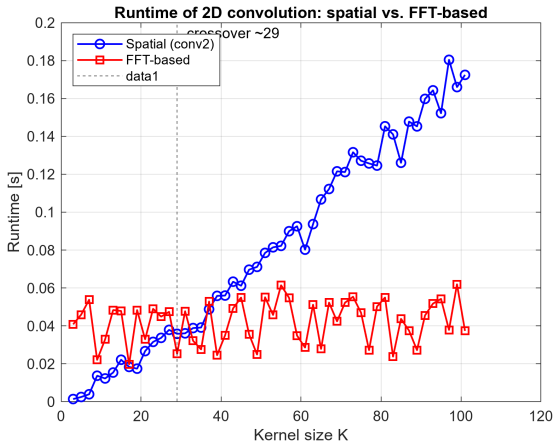


FFT-Based Filtering in 2D

The convolution theorem enables an efficient filtering procedure based entirely on FFTs:

1. Zero-pad both f and h to prevent circular wrap-around.
2. Compute their Fourier transforms: $F = \text{fft2}(f_p)$, $H = \text{fft2}(h_p)$.
3. Multiply them element-wise: $G = F \cdot H$.
4. Apply the inverse transform: $g = \text{ifft2}(G)$.

This procedure reduces computational complexity from $\mathcal{O}(N^2K^2)$ for direct convolution to $\mathcal{O}(N^2 \log N)$ for FFT-based convolution, which is a substantial improvement for large images and filters.
Runtime comparison between direct convolution and FFT-based filtering. FFT becomes advantageous for larger filter sizes



Summary

The Fourier Transform decomposes a signal or image into its frequency components, and the Fast Fourier Transform provides an efficient way to compute it with $\mathcal{O}(N \log N)$ complexity. Power-of-two data lengths and appropriate zero-padding are key to computational efficiency and numerical stability. In image processing, FFTs make it possible to implement complex filters, edge enhancement, deblurring, and frequency-domain operations that would otherwise be computationally infeasible in the spatial domain. By turning convolution into simple multiplication, FFT-based filtering combines theoretical elegance with practical speed and remains one of the most powerful tools in modern image analysis.