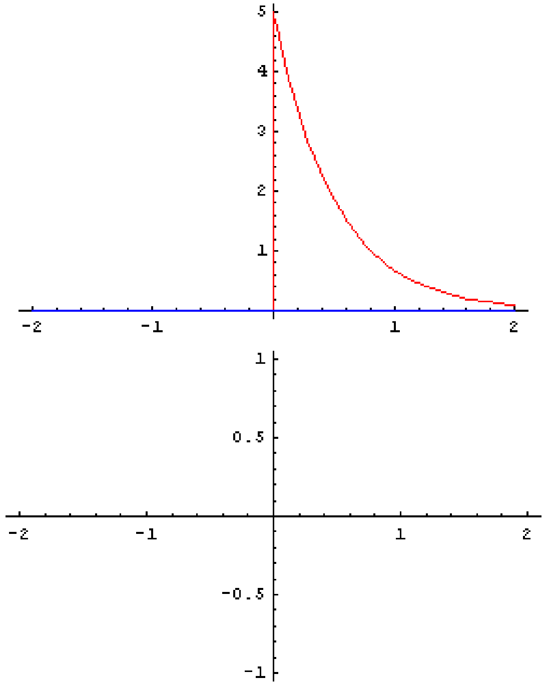
Fourier Transform: Moments (expressing moments in terms of Fourier transform)

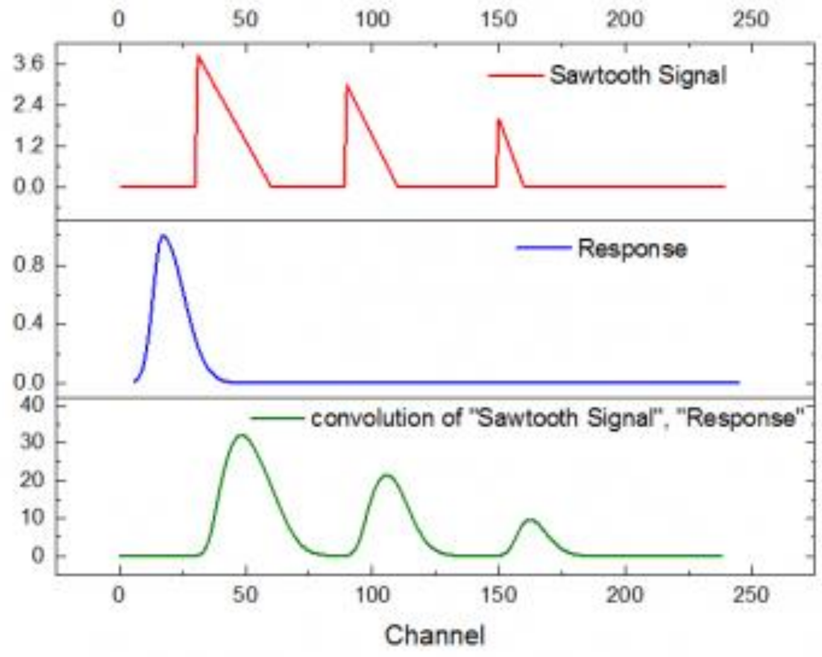


Convolution

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

Commutativity: $f * g = g * f$
Associativity: $f * (g * h) = (f * g) * h$ and also distributivity.

$$\mathcal{F}\{f * g\} = F(\omega)G(\omega)$$

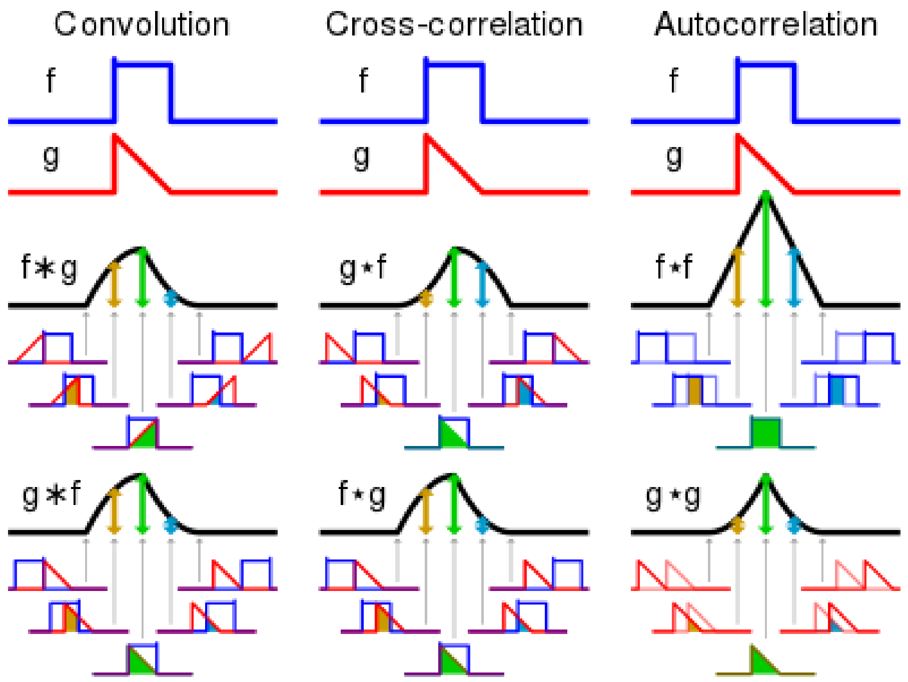


Convolution theorem: convolution in the time domain is equal to multiplication in the frequency domain, and vice-versa!

https://www.originlab.com/doc/Origin-Help/Convolution

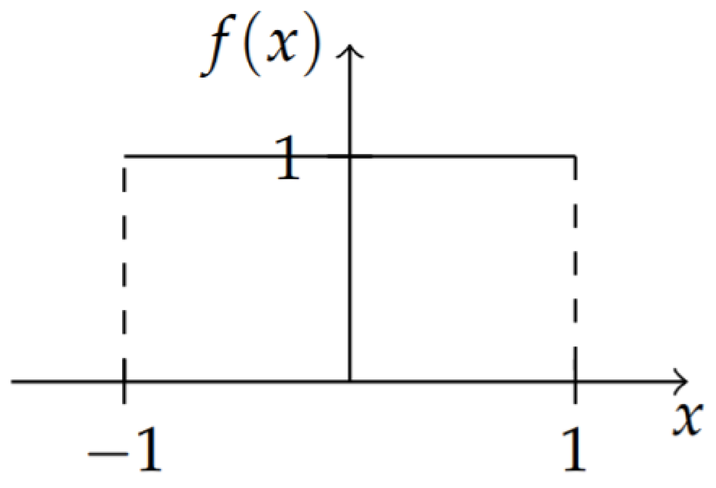
Correlation: two signal sequences are simply compared. The goal is to measure the degree to which two signals are similar and thus extract information about the underlying processes.

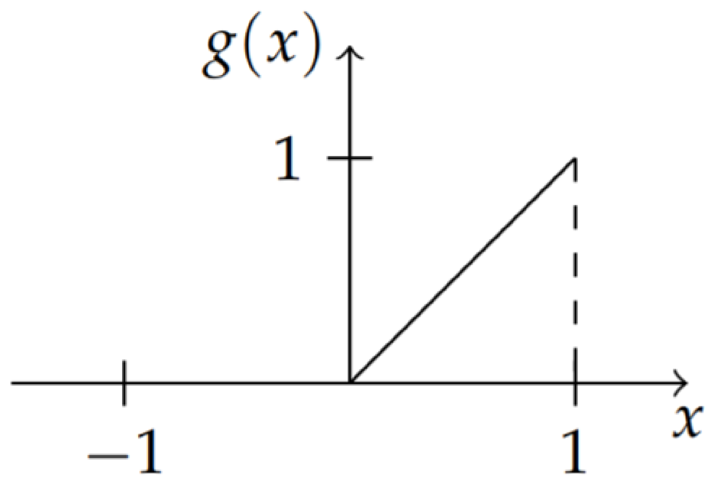
Convolution: a mathematical way of combining two signal sequences (e.g., an input signal and an impulse/kernel response) to form a third signal.

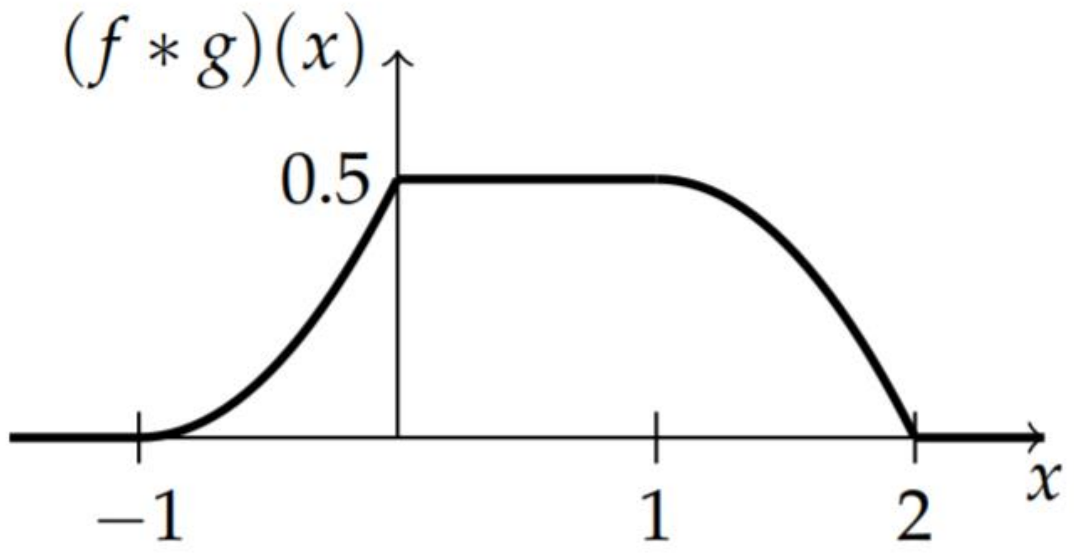


https://en.wikipedia.org/wiki/Cross-correlation

Convolution



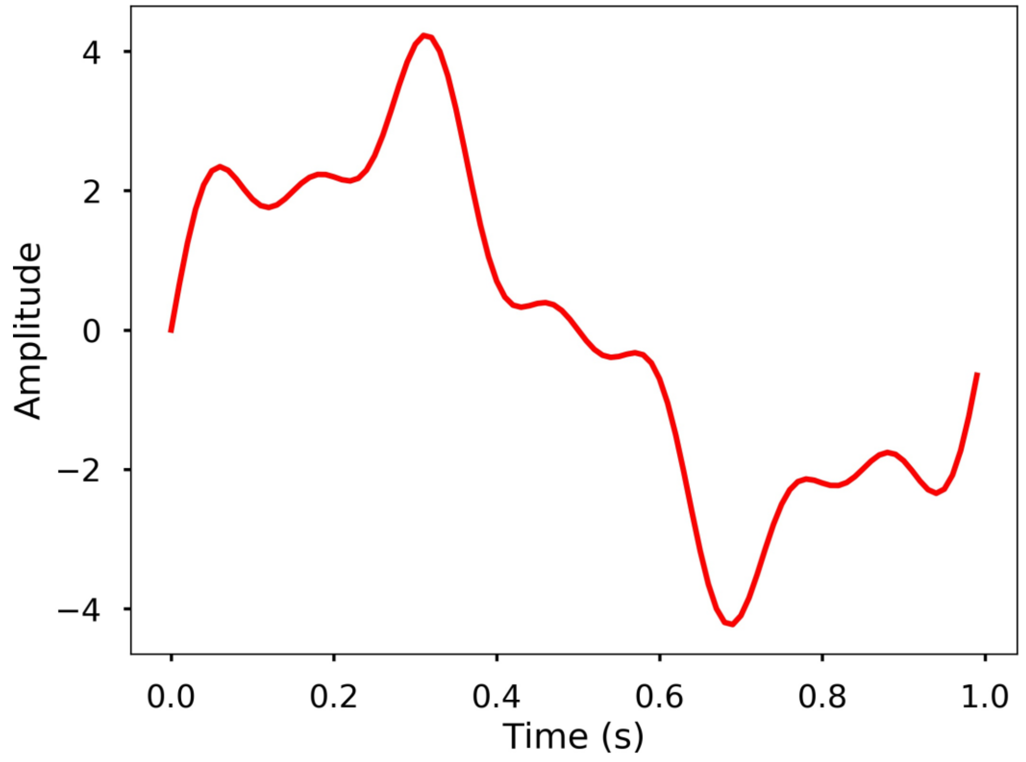


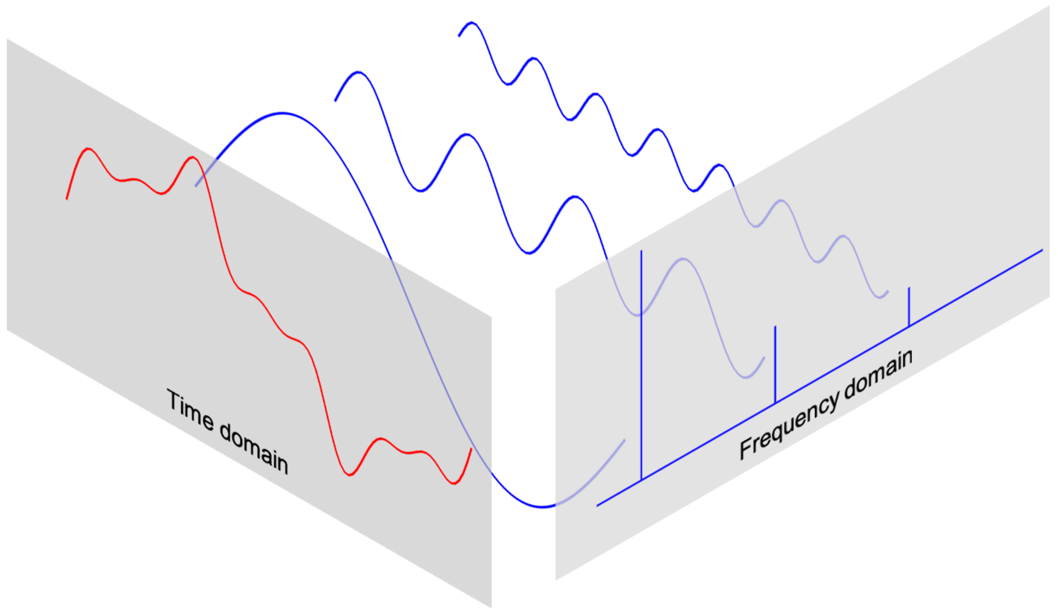


https://math.libretexts.org/Bookshelves/Differential_Equations/Introduction_to_Partial_Differential_Equations_%28Herman%29/09%3A_Transform_Techniques_in_Physics/9.06%3A_The_Convolution_Operation

Discrete Fourier Transform (DFT)

The Fourier Transform can be used to decompose any signal into a sum of simple sine and cosine waves that we can easily measure the frequency, amplitude and phase. The Fourier transform can be applied to continuous or discrete waves and an easy computational Fourier transform can be implemented using the Discrete Fourier Transform (DFT).





https://pythonnumericalmethods.berkeley.edu/notebooks/chapter24.02-Discrete-Fourier-Transform.html

Discrete Fourier Transform (DFT)

The DFT can transform a sequence of evenly spaced signal points to the information about the frequency of all the sine waves that needed to sum to the time domain signal. It is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi kn/N} = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi kn}{N}\right) - i\sin\left(\frac{2\pi kn}{N}\right)\right]$$

N = number of samples (how we sample the signal)
n = current sample (index in the sum)
k = current frequency, where $k \in [0, N-1]$
x_n = the signal value at n
X_k = DFT coefficient at k

$$Re\{X_k\} = \sum_{n=0}^{N-1} x_n \cos\left(\frac{2\pi kn}{N}\right)$$

$$Im\{X_k\} = \sum_{n=0}^{N-1} x_n \sin\left(\frac{2\pi kn}{N}\right)$$

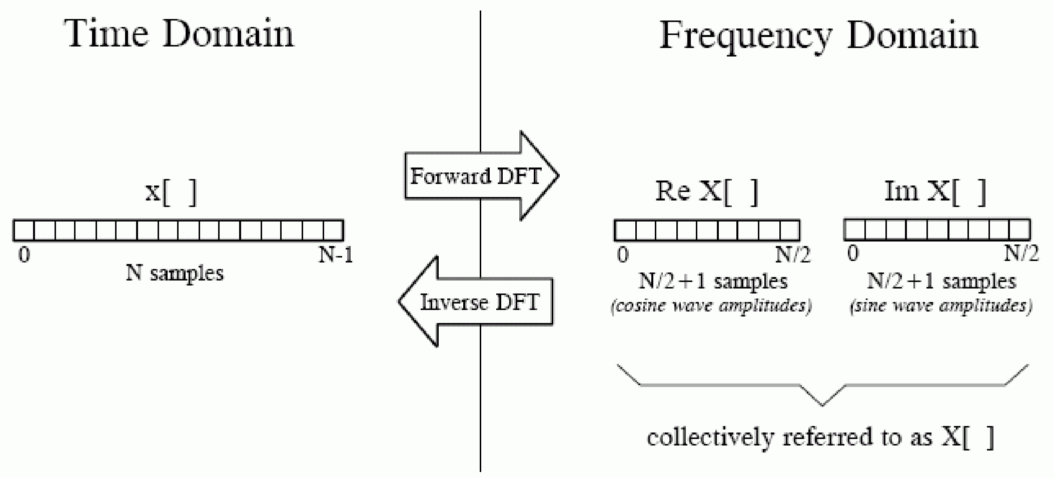


FIGURE 8-3
DFT terminology. In the time domain, *x*[ ] consists of *N* points running from 0 to *N*-1. In the frequency domain, the DFT produces two signals, the real part, written: *Re X*[ ], and the imaginary part, written: *Im X*[ ]. Each of these frequency domain signals are *N*/2 + 1 points long, and run from 0 to *N*/2. The Forward DFT transforms from the time domain to the frequency domain, while the Inverse DFT transforms from the frequency domain to the time domain. (Take note: this figure describes the **real DFT**. The **complex DFT**, discussed in Chapter 31, changes *N* complex points into another set of *N* complex points).

The Scientist and Engineer's Guide to Digital Signal Processing
By Steven W. Smith

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter24.02-Discrete-Fourier-Transform.html

Discrete Fourier Transform (DFT)

The DFT can transform a sequence of evenly spaced signal points to the information about the frequency of all the sine waves that needed to sum to the time domain signal. It is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-i2\pi kn/N} = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi kn}{N}\right) - i\sin\left(\frac{2\pi kn}{N}\right)\right]$$

N = number of samples (how we sample the signal)
n = current sample (index in the sum)
k = current frequency, where $k \in [0, N-1]$
x_n = the signal value at n
X_k = DFT coefficient at k

$$A_k = \frac{|X_k|}{N} = \frac{1}{N}\sqrt{(Re\{X_k\})^2 + (Im\{X_k\})^2}$$

$$Re\{X_k\} = \sum_{n=0}^{N-1} x_n \cos\left(\frac{2\pi kn}{N}\right)$$

$$\phi_k = \tan^{-1}(Re\{X_k\}/Im\{X_k\})$$

$$Im\{X_k\} = \sum_{n=0}^{N-1} x_n \sin\left(\frac{2\pi kn}{N}\right)$$

How much does a particular frequency contribute to the signal?

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter24.02-Discrete-Fourier-Transform.html

The DC Term

If the average value of a signal over one period is non-zero, i.e., the data has a constant offset, the signal will have a 0 frequency coefficient X_0. This is also called the “DC” term, by analogy to electrical circuits, where “DC” refers to a constant voltage.

Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is an efficient algorithm to calculate the DFT of a sequence. It is a divide-and-conquer algorithm that recursively breaks the DFT into smaller DFTs to bring down the computation. As a result, it successfully reduces the complexity of the DFT from $O(N^2)$ to $O(N \log N)$, where N is the size of the data. This reduction in computation time is significant, especially for data with large N, therefore, making FFT widely used in engineering, science and mathematics. The FFT algorithm is the Top 10 algorithm of the 20th century by the Journal Computing in Science & Engineering (https://www.computer.org/csdl/magazine/cs/2000/01/c1022/13rRUxBJhBm).

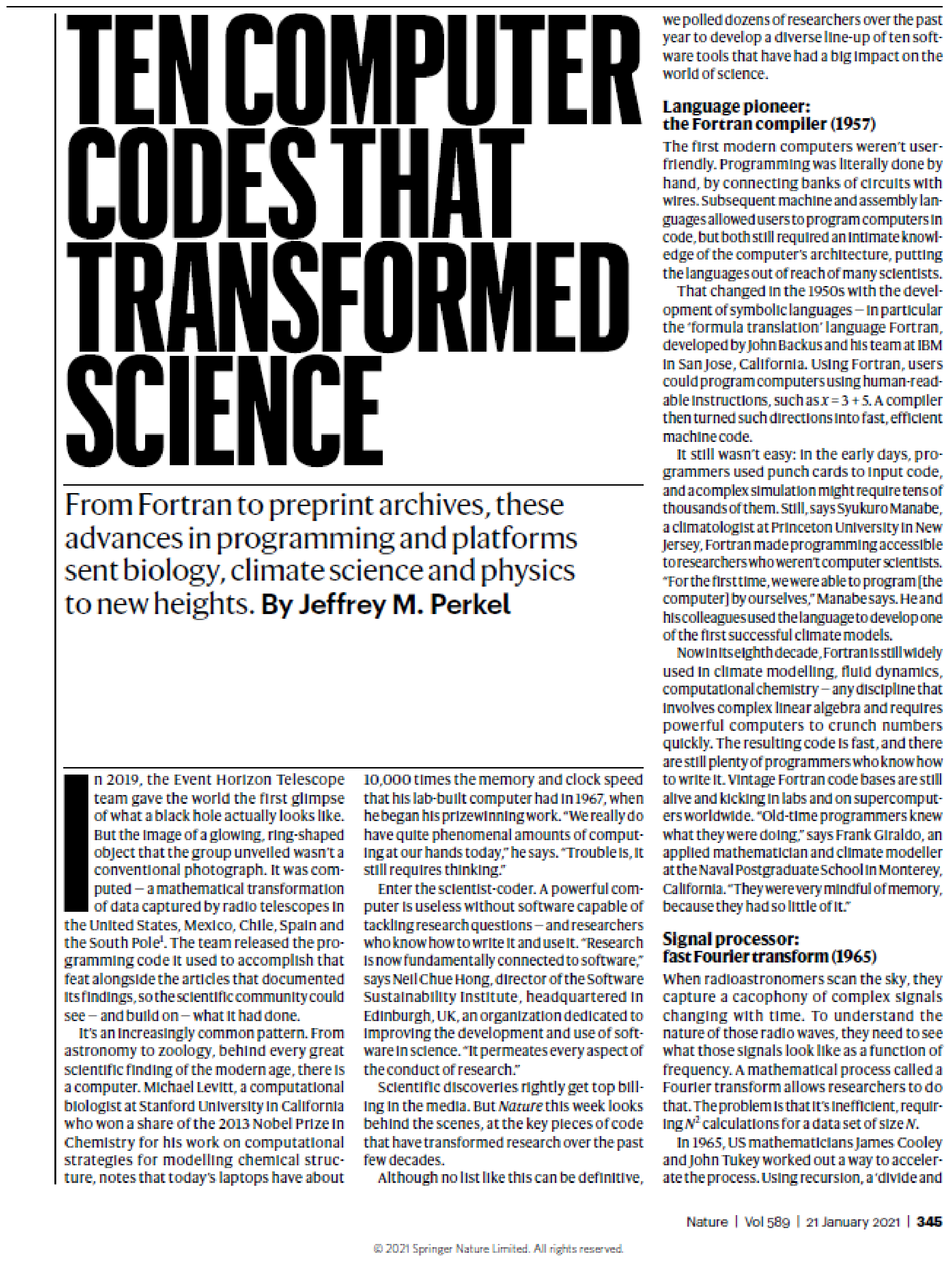
See the main tricks that FFT adds to speed computation in:

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter24.03-Fast-Fourier-Transform.html

and Dr. Jackel's notes, chapter 23.

Most commonly used FFT is the *Cooley–Tukey algorithm:* uses recursion, a ‘divide and conquer’ programming approach in which an algorithm repeatedly reapplies itself.

“An algorithm for the machine calculation of complex Fourier series”, James W. Cooley and John W. Tukey Math. Comp. 19 (1965), 297-301.



TEN COMPUTER CODES THAT TRANSFORMED SCIENCE

From Fortran to preprint archives, these advances in programming and platforms sent biology, climate science and physics to new heights. **By Jeffrey M. Perkel**

In 2019, the Event Horizon Telescope team gave the world the first glimpse of what a black hole actually looks like. But the image of a glowing, ring-shaped object that the group unveiled wasn't a conventional photograph. It was computed – a mathematical transformation of data captured by radio telescopes in the United States, Mexico, Chile, Spain and the South Pole[1]. The team released the programming code it used to accomplish that feat alongside the articles that documented its findings, so the scientific community could see – and build on – what it had done.

It's an increasingly common pattern. From astronomy to zoology, behind every great scientific finding of the modern age, there is a computer. Michael Levitt, a computational biologist at Stanford University in California who won a share of the 2013 Nobel Prize in Chemistry for his work on computational strategies for modelling chemical structure, notes that today's laptops have about 10,000 times the memory and clock speed that his lab-built computer had in 1967, when he began his prizewinning work. “We really do have quite phenomenal amounts of computing at our hands today,” he says. “Trouble is, it still requires thinking.”

Enter the scientist-coder. A powerful computer is useless without software capable of tackling research questions – and researchers who know how to write it and use it. “Research is now fundamentally connected to software,” says Neil Chue Hong, director of the Software Sustainability Institute, headquartered in Edinburgh, UK, an organization dedicated to improving the development and use of software in science. “It permeates every aspect of the conduct of research.”

Scientific discoveries rightly get top billing in the media. But *Nature* this week looks behind the scenes, at the key pieces of code that have transformed research over the past few decades.

Although no list like this can be definitive, we polled dozens of researchers over the past year to develop a diverse line-up of ten software tools that have had a big impact on the world of science.

Language pioneer: the Fortran compiler (1957)

The first modern computers weren't user-friendly. Programming was literally done by hand, by connecting banks of circuits with wires. Subsequent machine and assembly languages allowed users to program computers in code, but both still required an intimate knowledge of the computer's architecture, putting the languages out of reach of many scientists.

That changed in the 1950s with the development of symbolic languages – in particular the ‘formula translation’ language Fortran, developed by John Backus and his team at IBM in San Jose, California. Using Fortran, users could program computers using human-readable instructions, such as $x = 3 + 5$. A compiler then turned such directions into fast, efficient machine code.

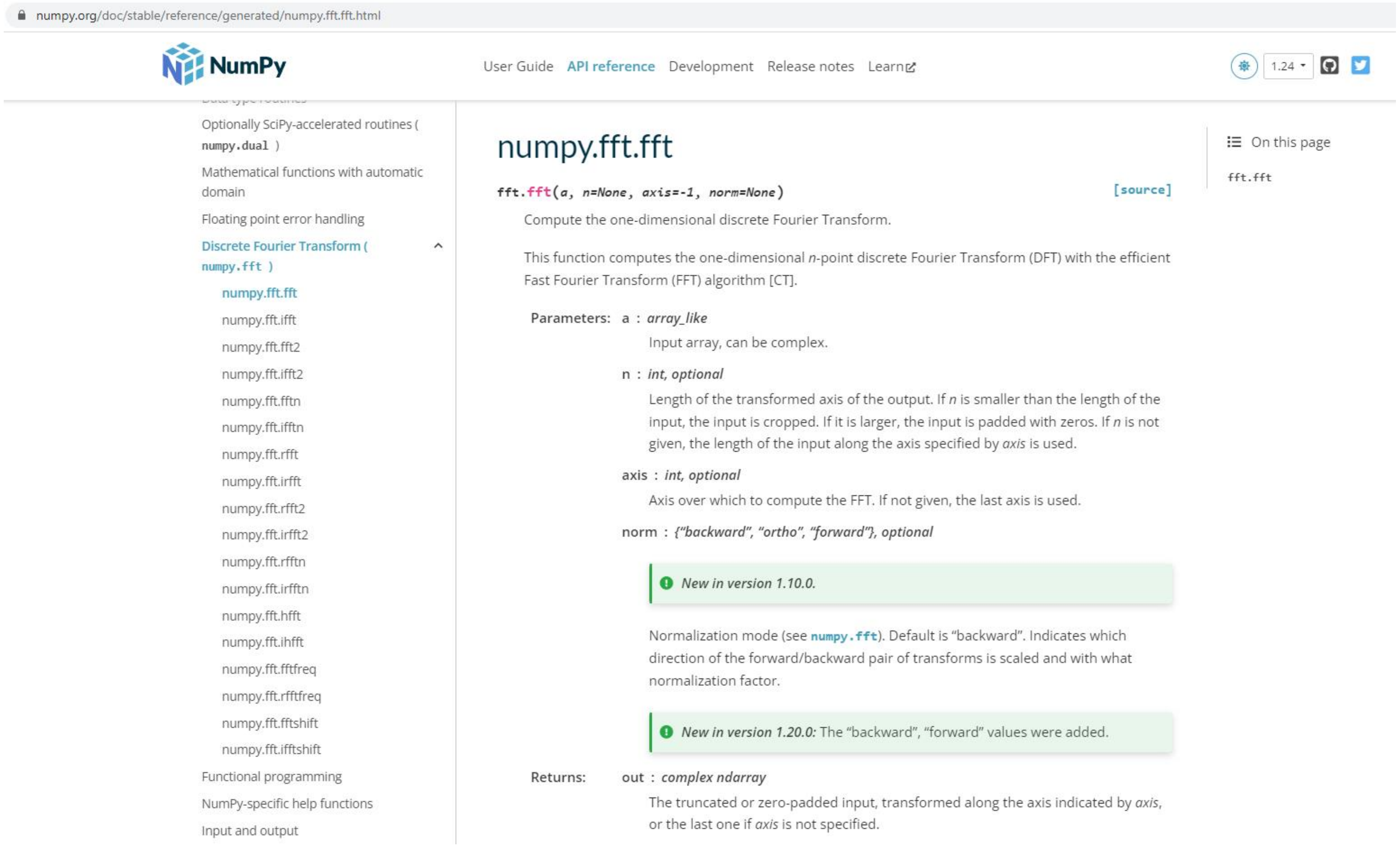
It still wasn't easy: in the early days, programmers used punch cards to input code, and a complex simulation might require tens of thousands of them. Still, says Syukuro Manabe, a climatologist at Princeton University in New Jersey, Fortran made programming accessible to researchers who weren't computer scientists. “For the first time, we were able to program [the computer] by ourselves,” Manabe says. He and his colleagues used the language to develop one of the first successful climate models.

Now in its eighth decade, Fortran is still widely used in climate modelling, fluid dynamics, computational chemistry – any discipline that involves complex linear algebra and requires powerful computers to crunch numbers quickly. The resulting code is fast, and there are still plenty of programmers who know how to write it. Vintage Fortran code bases are still alive and kicking in labs and on supercomputers worldwide. “Old-time programmers knew what they were doing,” says Frank Giraldo, an applied mathematician and climate modeller at the Naval Postgraduate School in Monterey, California. “They were very mindful of memory, because they had so little of it.”

Signal processor: fast Fourier transform (1965)

When radioastronomers scan the sky, they capture a cacophony of complex signals changing with time. To understand the nature of those radio waves, they need to see what those signals look like as a function of frequency. A mathematical process called a Fourier transform allows researchers to do that. The problem is that it's inefficient, requiring N^2 calculations for a data set of size N.

In 1965, US mathematicians James Cooley and John Tukey worked out a way to accelerate the process. Using recursion, a ‘divide and


numpy.org/doc/stable/reference/generated/numpy.fft.fft.html

NumPy User Guide API reference Development Release notes Learn 1.24

numpy.fft.fft

`fft.fft(a, n=None, axis=-1, norm=None)` [source]

Compute the one-dimensional discrete Fourier Transform.

This function computes the one-dimensional *n*-point discrete Fourier Transform (DFT) with the efficient Fast Fourier Transform (FFT) algorithm [CT].

Parameters: **a** : *array_like*

Input array, can be complex.

n : *int, optional*

Length of the transformed axis of the output. If *n* is smaller than the length of the input, the input is cropped. If it is larger, the input is padded with zeros. If *n* is not given, the length of the input along the axis specified by *axis* is used.

axis : *int, optional*

Axis over which to compute the FFT. If not given, the last axis is used.

norm : *{"backward", "ortho", "forward"}, optional*

New in version 1.10.0.

Normalization mode (see `numpy.fft`). Default is "backward". Indicates which direction of the forward/backward pair of transforms is scaled and with what normalization factor.

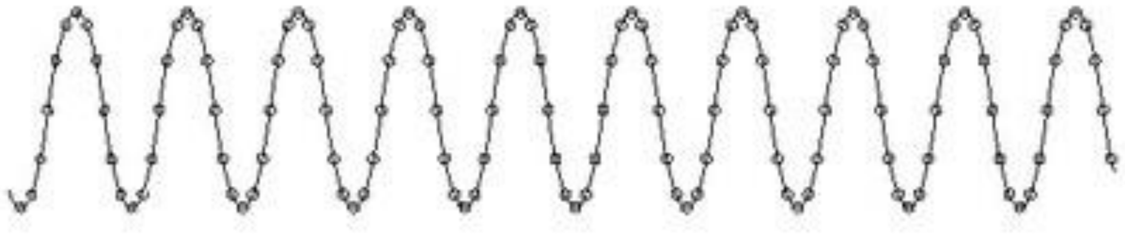
New in version 1.20.0: The "backward", "forward" values were added.

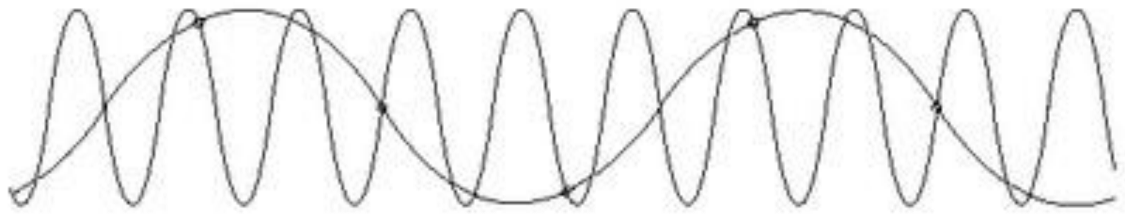
Returns: **out** : *complex ndarray*

The truncated or zero-padded input, transformed along the axis indicated by *axis*, or the last one if *axis* is not specified.

On this page
fft.fft

Choice of N and Δt (or how to sample your signal) is very important to reconstruct the signal otherwise we may see **aliasing** phenomenon!


Adequately sampled signal


Aliased signal due to undersampling

A continuous time signal can be represented in its samples and can be recovered back when sampling frequency ν_s is greater than or equal to twice the highest frequency component of the signal, i.e.

$$\boxed{\nu_s \geq 2\nu_{max} \longleftrightarrow \nu_{max} \leq \nu_{Nyquist}}$$

Time domain

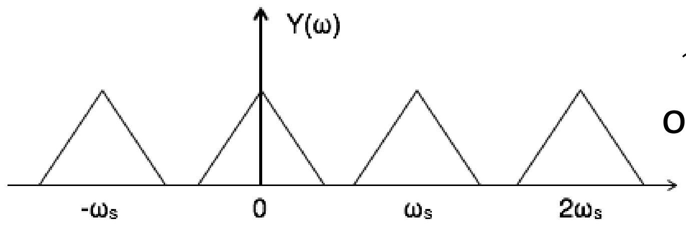
$\Delta t = h$

Frequency domain

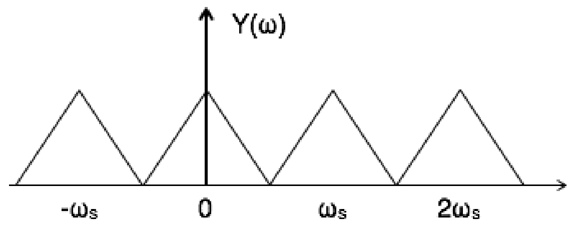
$$\nu_s = \frac{1}{h}$$

$$\nu_{Nyquist} = \frac{\nu_s}{2}$$

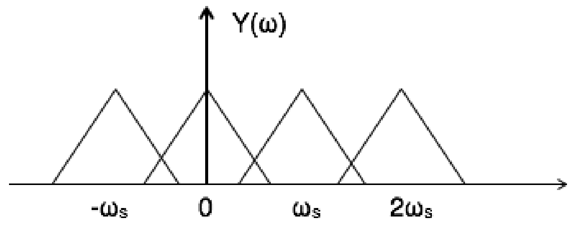
$$\omega_s = 2\pi\nu_s$$



$\nu_s > 2\nu_{max}$
over sampling



$\nu_s = 2\nu_{max}$
matching sampling



$\nu_s < 2\nu_{max}$
under sampling
Aliasing!

- Repetitions in the frequency domain overlap.
- Distortion (aliasing) in frequency domain.

Nyquist

The Nyquist frequency is defined as

$$\nu_{Nyquist} = \frac{\nu_s}{2} = \frac{1}{2\Delta t}$$

And it is the highest frequency that can be resolved for a given sampling spacing Δt.