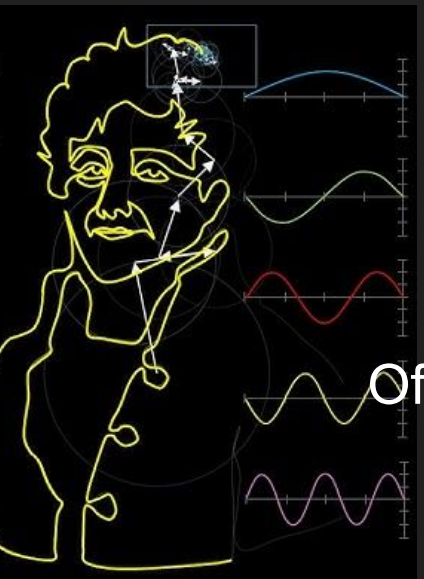
Fourier
PHYS4840
Analysis, Transform, Series…



github.com/nialljmiller/PHYS4840_Fourier

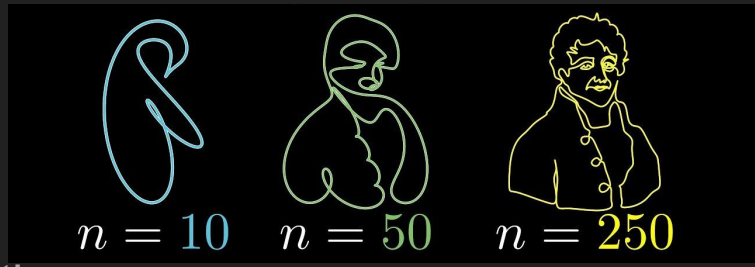
(This is currently the pinned repo on my page)

Dr Niall Miller

AG 404

Office hours - 2 hours before class or by request

Fourier



- Fourier series
  Using lots of trig functions to make any other function

- Fourier transform
  Using lots of trig to represent the frequency space of a complex signal

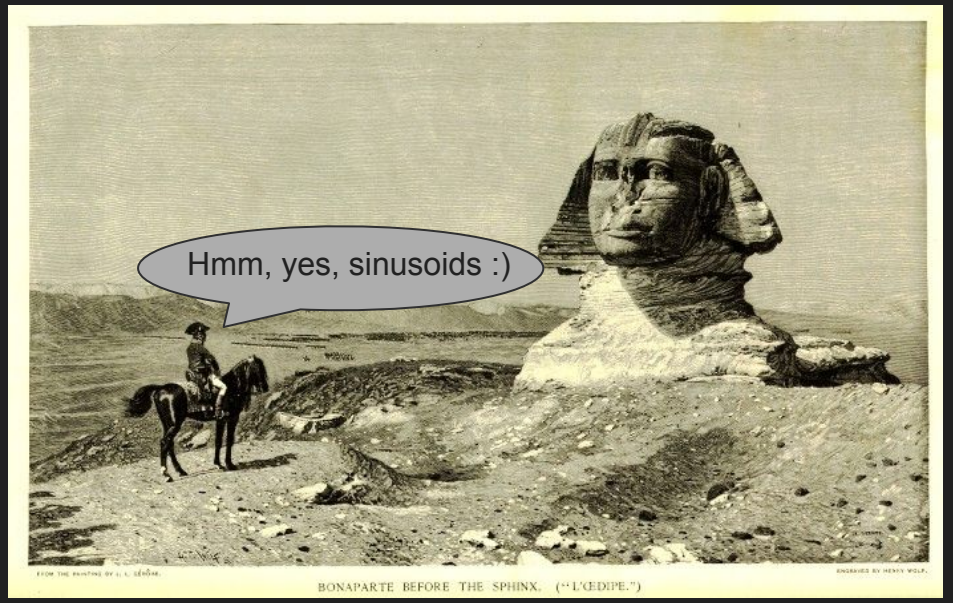
- Discrete Sine and Cosine transforms (jpegs)

- Fast Fourier Transforms
  Using lots of trig to represent the frequency space of a complex signal
  but on a computer and fast :)

  And other things like reverse DFT, Gibbs, window functions…

Fourier

- What is Fourier analysis? (Signal analysis)
- Why is Fourier? (19th century scholars wanted to represent functions as sums of trig)
- Who is Fourier? (A French chap who was quite good with maths and also ran parts of Egypt for a bit and also Napoleon's m8)
- How is Fourier? (Dead)
- When is Fourier? (~ 1800s)



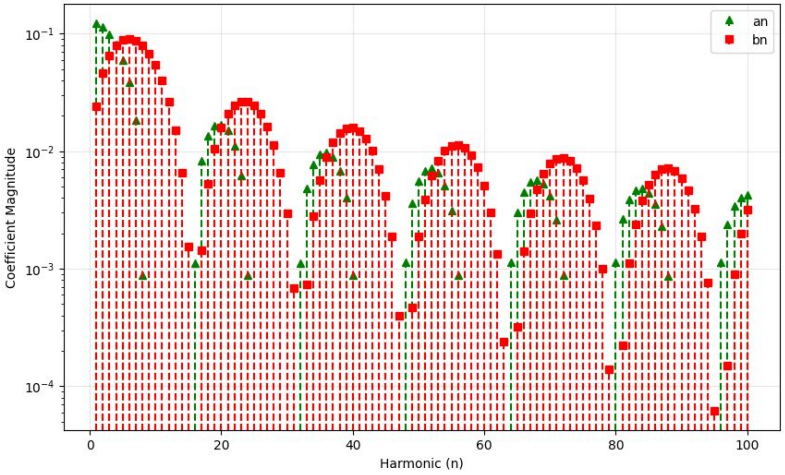
Fourier Series

- Any periodic function can be represented as a sum of sines and cosines
- So we can model complex signals with *lots* of simple components

F(x) = a0/2 + a1*cos(x) + b1*sin(x) + a2*cos(2x) + b2*sin(2x) + a3*cos(3x) + b3*sin(3x) + …

an and bn = Coefficients derived from :

Values for an and bn fitting a pulse wave



$$a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x), dx$$

$$a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx), dx$$

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx), dx$$

Fourier Series maths

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(nx) + b_n \sin(nx)]$$

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x), dx$$

$$a_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \cos(nx), dx$$

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx), dx$$

Fourier Series code

- Any periodic function can be represented as a sum of sines and cosines
- So we can model complex signals with *lots* of simple components

```
# Compute Fourier series coefficients
a0, an, bn = fs.compute_coefficients(wave, TERMS)

# Calculate the Fourier approximation
y_approx = fs.fourier_series_approximation(x, a0, an, bn)
```

Fourier Series code

```
# Compute Fourier series coefficients
a0, an, bn = fs.compute_coefficients(wave, TERMS)
```

```
a0 = compute_a0(func, period, num_points)
an = np.zeros(n_terms)
bn = np.zeros(n_terms)

for n in range(1, n_terms + 1):
    an[n-1] = compute_an(func, n, period, num_points)
    bn[n-1] = compute_bn(func, n, period, num_points)

return a0, an, bn
```

Fourier Series code

```
# Compute Fourier series coefficients
a0, an, bn = fs.compute_coefficients(wave, TERMS)
```

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x), dx$$

```
def compute_a0(func, period=2*np.pi, num_points=1000):
    x = np.linspace(0, period, num_points)
    y = func(x)

    result = np.trapz(y, x)
    return (1 / period) * result
```

Fourier Series code

$$b_n = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x) \sin(nx), dx$$

```python
def compute_bn(func, n, period=2*np.pi, num_points=1000):
    x = np.linspace(0, period, num_points)
    y = func(x)

    integrand = y * np.sin(2 * np.pi * n * x / period)
    result = np.trapz(integrand, x)
    return (2/period) * result
```

Fourier Series code

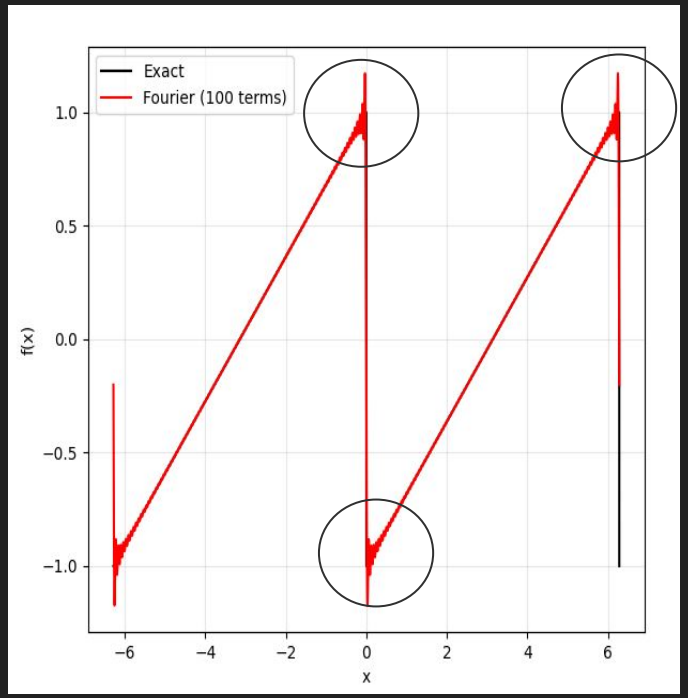
```python
# Calculate the Fourier approximation
y_approx = fs.fourier_series_approximation(x, a0, an, bn)
```

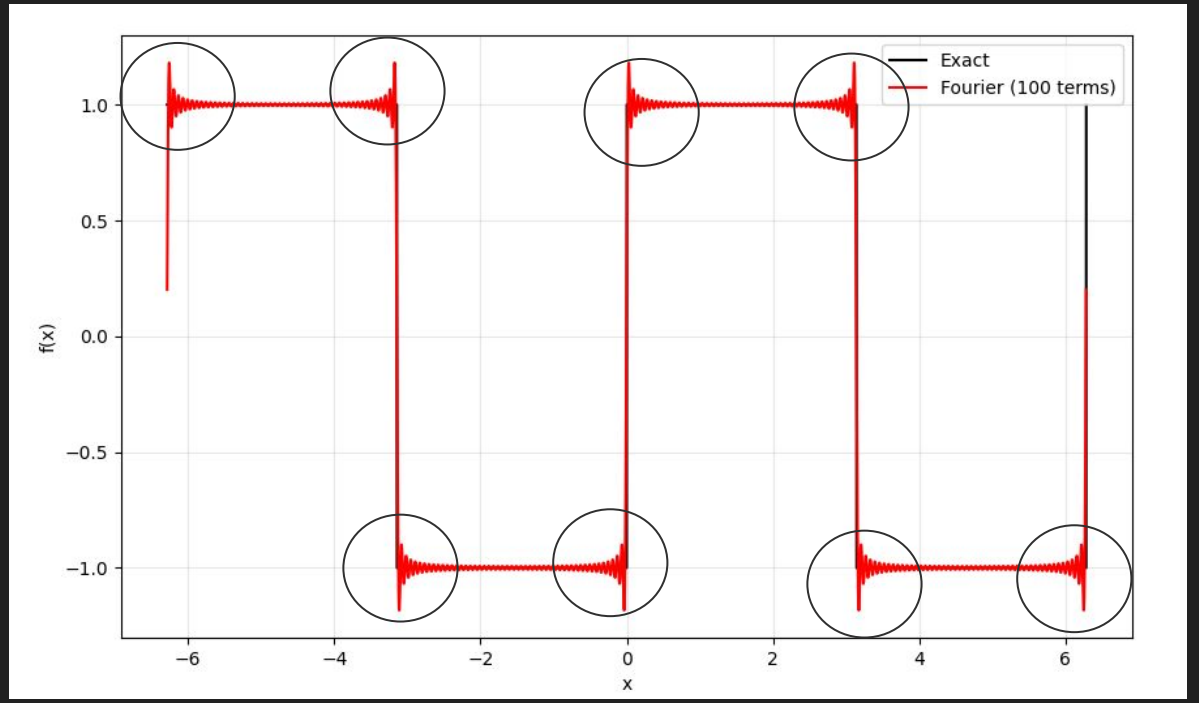
$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos(nx) + b_n \sin(nx)]$$

```python
def fourier_series_approximation(x, a0, an, bn, period=2*np.pi):
    result = np.ones_like(x) * a0

    for n in range(1, len(an) + 1):
        result += an[n-1] * np.cos(2 * np.pi * n * x / period)
        result += bn[n-1] * np.sin(2 * np.pi * n * x / period)

    return result
```

Gibbs

The Gibbs phenomenon is the overshoot (or undershoot) that happens when you approximate a function with a discontinuity (like a step) using a finite number of terms in its Fourier series.



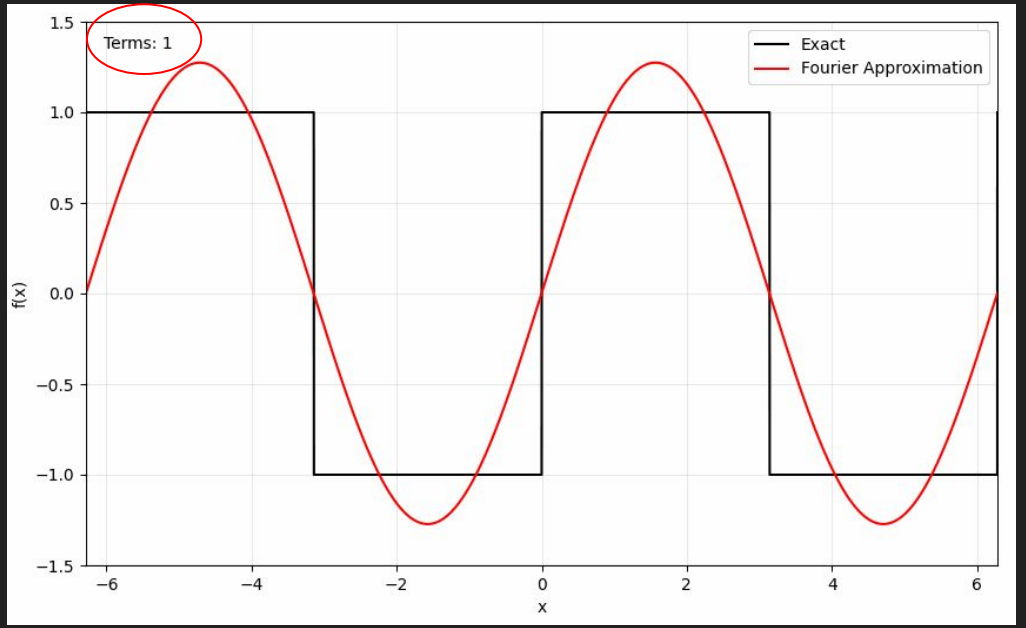


Gibbs

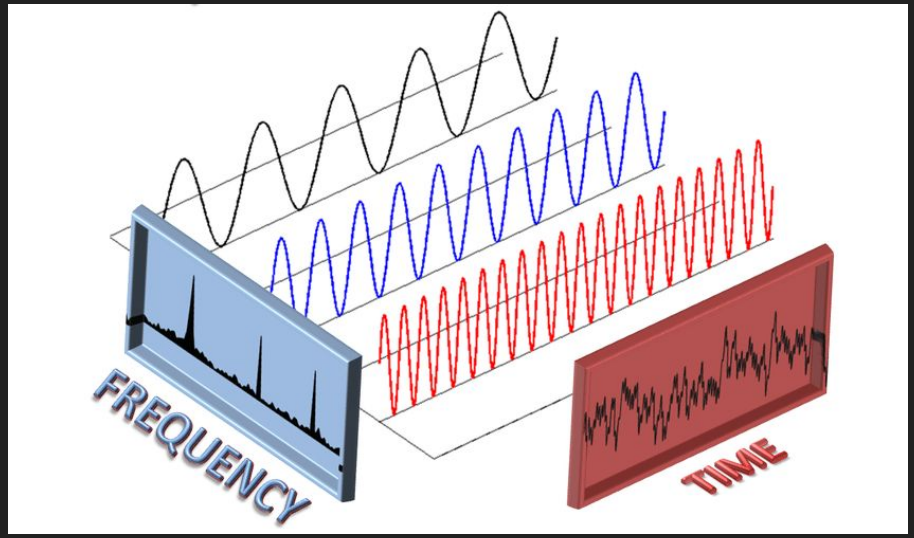
The oscillations don't go away as you add more terms — they just get narrower and more localized near the discontinuity.

The max overshoot settles around 9% of the jump, no matter how many terms you add.

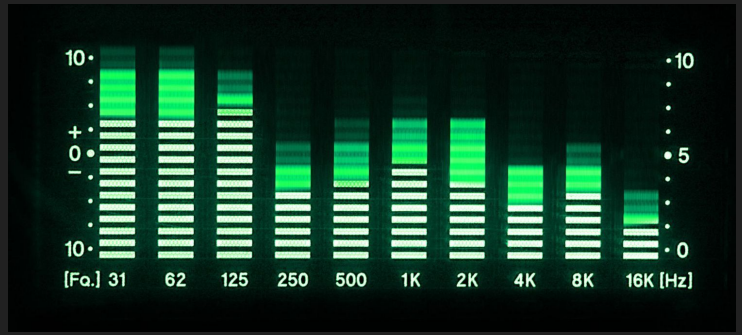
Even though the full Fourier series converges in the mean (L^2 norm), pointwise convergence fails at the jump



Fourier transform



- This is the one you know!
- All the audio visualisers work like this
- Has many applications
- Is reversible
- High compute complexity O(N^2)
- Sum of Fourier transforms = fourier transform of sum
- It is the frequency space representation of a signal



Fourier Series is nice if we know the periodicity

While the Fourier series applies to continuous, periodic functions
The Discrete Fourier Transform (DFT) is designed for sampled, finite-length signals.

The DFT transforms a sequence of N complex numbers into another sequence of complex numbers representing frequency components.

For a sequence $x_0, x_1, \ldots, x_{N-1}$, the DFT is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i \frac{2\pi}{N} kn} \quad \text{for } k = 0, 1, \ldots, N-1$$

For a sequence $x_0, x_1, \ldots, x_{N-1}$, the DFT is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i \frac{2\pi}{N} kn} \quad \text{for } k = 0, 1, \ldots, N-1$$

In terms of sines and cosines:

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

Intuition Behind the Fourier Transform

- Think of it as measuring how much of each frequency component exists in the signal
- For each frequency ω, we multiply the signal by e^(-iωt) and integrate
- This is like computing a "correlation" between the signal and each frequency
- The result F(ω) gives amplitude and phase information at frequency ω

```
for k in range(N):
    for n in range(N):
        X[k] += x[n] * np.exp(-2j * np.pi * k * n / N)
```

Reversing the Fourier Transform

- Recall that Fourier transform is linear!
- We should be able to construct a signal from any FT spectra as long as our function is nicely behaved and we didn't undersample

For a sequence $x_0, x_1, \ldots, x_{N-1}$, the DFT is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i \frac{2\pi}{N} kn} \quad \text{for } k = 0, 1, \ldots, N-1$$

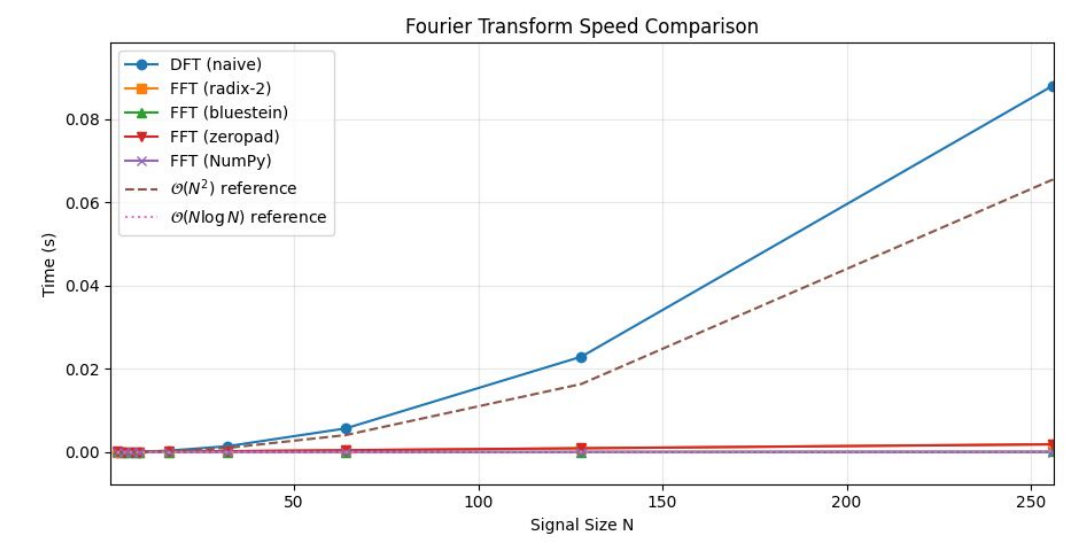
Reversing the Fourier Transform

- Recall that Fourier transform is linear!
- We should be able to construct a signal from any FT spectra as long as our function is nicely behaved and we didn't undersample

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i \frac{2\pi}{N} kn} \quad \text{for } n = 0, 1, \ldots, N-1$$

FFT

The Fast Fourier Transform computes the Discrete Fourier Transform (DFT) in O(N log N) time (vs O(N²) for DFT), making Fourier analysis practical for real-world applications.



A brief detour on Marks ears…

[5]The notable exception is the CD, which was introduced about ten years before the invention of MP3s. CDs use uncompressed audio—just raw sound samples—which is extremely inefficient, although in theory it also gives higher sound quality because there are no compression artifacts. Good luck hearing the difference though; these days audio compression is extremely good and it's rare that one can actually hear compression artifacts. Double-blind listening tests have been conducted where listeners are played a pristine, uncompressed recording of a piece of music, followed either by a compressed version of the same recording or by the uncompressed version again. When the best modern compression techniques are used, even expert listeners have been unable to reliably tell the difference between the two.

A brief detour on Marks ears…

[5]The notable exception is the CD, which was introduced about ten years before the invention of MP3s. CDs use uncompressed audio—just raw sound samples—which is extremely inefficient, although in theory it also gives higher sound quality because there are no compression artifacts. Good luck hearing the difference though; these days audio compression is extremely good and it's rare that one can actually hear compression artifacts. Double-blind listening tests have been conducted where listeners are played a pristine, uncompressed recording of a piece of music, followed either by a compressed version of the same recording or by the uncompressed version again. When the best modern compression techniques are used, even expert listeners have been unable to reliably tell the difference between the two.

A brief detour on Marks ears…

HOW?!?!?!?!?!?!?

[5]The notable exception is the CD, which was introduced about ten years before the invention of MP3s. CDs use uncompressed audio—just raw sound samples—which is extremely inefficient, although in theory it also gives higher sound quality because there are no compression artifacts. Good luck hearing the difference though; these days audio compression is extremely good and it's rare that one can actually hear compression artifacts. Double-blind listening tests have been conducted where listeners are played a pristine, uncompressed recording of a piece of music, followed either by a compressed version of the same recording or by the uncompressed version again. When the best modern compression techniques are used, even expert listeners have been unable to reliably tell the difference between the two.

A brief detour on Marks ears…

Understanding Computational Complexity - O

Big `O' notation

O(N^2): The computational cost grows quadratically with input size N (naive DFT)

O(N log N): The computational cost grows much more slowly (FFT)

Or 👽(👶log👶) in emoji notation

Consider the computational requirements for a signal with N = 1,000,000 points:

- Naive DFT: ~1,000,000,000,000 (trillion) operations

- FFT: ~20,000,000 (20 million) operations

This approximately 50,000× speedup is what made many modern digital signal processing applications feasible.

What is an FFT, how many are there? why?

There isn’t just one FFT — there are many FFT algorithms, including:

- Radix-2 Cooley-Tukey (for powers of 2)
- Bluestein’s algorithm (for any size, even prime)
- Mixed-radix FFTs, split-radix, Rader’s FFT (for special cases)

What is an FFT, how many are there? why?

FFTs are algorithms for computing the DFT efficiently.

There are many versions because different algorithms are better for different input sizes, hardware, and constraints.

You pick the one that fits your signal size and performance needs.

Cooley-Tukey (Radix-2)

Calculating 2 small DFT is faster than one big DFT…

Classic recursive FFT, only works when input length is a power of 2.

Very fast and elegant — the go-to algorithm in clean, controlled cases.

Often used in hardware, textbooks, and libraries when signal size is fixed.

Cooley-Tukey (Radix-2)

Classic recursive FFT, only works when input length is a power of

Very fast and elegant — the go-to algorithm in clean, controlled ca

Often used in hardware, textbooks, and libraries when signal size

The algorithm, along with its recursive application, was invented by ***Carl Friedrich Gauss.***

Cooley and Tukey independently rediscovered and popularized it 160 years later.

What if our input is not a power of 2?

Zero Padding

Option 1 - Just make it a power of 2, lol

Pads the input to the next power-of-2 size and runs a radix-2 FFT.

Simple trick to use a fast algorithm even when the input size isn’t ideal.

Can cause spectral leakage, but it's often “good enough.”

What if our input is not a power of 2?

Bluestein

Option 2 - add a chirp - obviously…

Bluestein works for any signal — even prime numbers.

Bluestein’s Algorithm rewrites the DFT as a convolution using a special "chirp" signal.

A “Chirp” signal is just a oscillatory signal with a changing frequency

- think of a bird chirp where the pitch increases over its short time

This lets us use a standard FFT to handle any input length.

It’s slower than radix-2, but much more flexible

Discrete Sine and Cosine transforms

Why jpegs are a convenient evil!

DCT = Discrete Cosine Transform

DST = Discrete Sine Transform

Like Fourier transforms — but use only cosine or sine components

Real-valued output (unlike full DFT which is complex)

Super useful for compressing data where we want to keep the “important” frequency content

Why Use Cosine or Sine Only?

Cosine: even-symmetric functions (mirror at center)

Sine: odd-symmetric functions (zero at center)

These symmetries help with:

- Reducing artifacts
- Better compression
- Simpler maths (real instead of complex values)

What is the Discrete Sine/Cosine Transform (DST/DCT)?

Converts a sequence of data points into the sum of sine/cosine functions oscillating at different frequencies.

The DCT is closely related to the Discrete Fourier Transform (DFT) but uses only real-valued cosines.

- Flipping it at the center does not invert the function $f(-x) = f(x)$.

The DST is used when odd symmetry is important in the data

- Flipping it at the center inverts the function $f(-x) = -f(x)$.

What has this got to do with JPEG?

JPEG is a cosine transform of real data

- We do this because its faster, smaller and the human eye is stupid
- The photo I took of the squirrel on prexies will not be analysed, I don't care about the exact pixel values of his fluffy tail
- However, the exact pixel values on the fluffy tail of a dwarf galaxy…

Why do I care?

Rosa Menkman

JPEG from A Vernacular of File Formats, 2009 - 2010), 2023 revisitation with hidden message in DCT

Accepts Crypto | No reserve

Cryptocurrency Payments

No Reserves

Lot Closed

April 26, 12:01 PM MDT

Estimate

7,000 - 10,000 USD

Log in to view results

How to: Make JPEG

1 - Break image into 8×8 pixel blocks

Each block is treated independently

Operates on grayscale or on YCbCr color space (luma/chroma channels)

2 - Apply 2D DCT to each block

Turns 64 spatial pixel values into 64 frequency coefficients

Low-frequency terms (top-left) = broad patterns

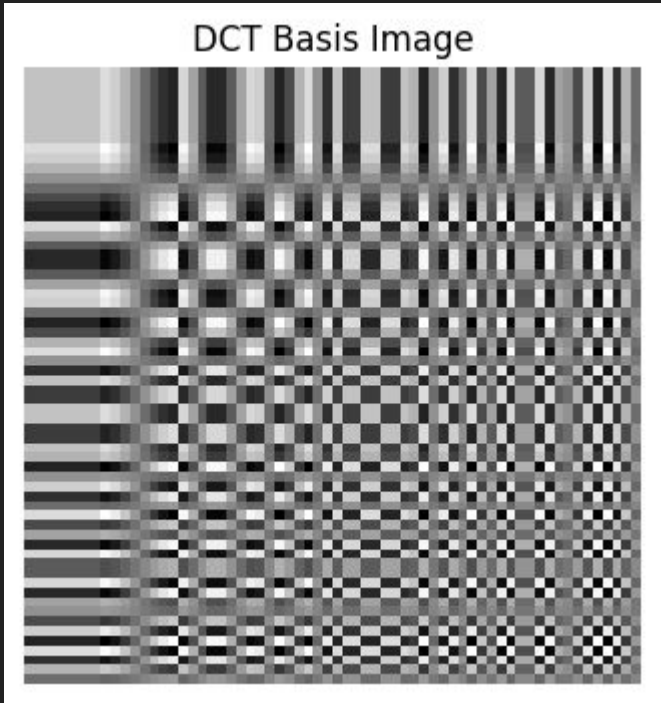
High-frequency terms (bottom-right) = tiny details

3 - Quantization

Human eyes are bad at high-frequency changes

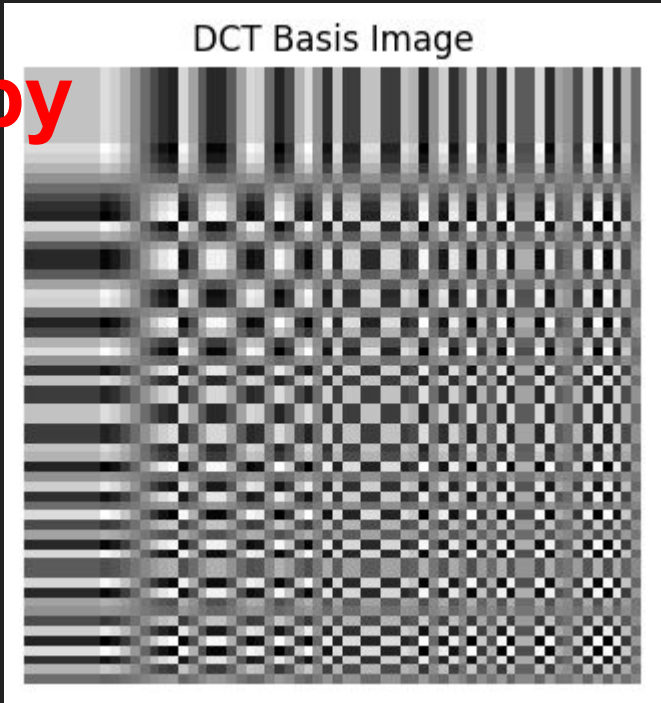
JPEG aggressively reduces (or zeroes) high-frequency coefficients

This is where lossy compression happens



How to: Make JPEG

dct_jpeg.py



1 - Break image into 8×8 pixel blocks

Each block is treated independently

Operates on grayscale or on YCbCr color space (luma/chroma channels)

2 - Apply 2D DCT to each block

Turns 64 spatial pixel values into 64 frequency coefficients

Low-frequency terms (top-left) = broad patterns

High-frequency terms (bottom-right) = tiny details

3 - Quantization

Human eyes are bad at high-frequency changes

JPEG aggressively reduces (or zeroes) high-frequency coefficients

This is where lossy compression happens

How to: Make JPEG

1 - Break image into 8×8 pixel blocks

Each block is treated independently

Operates on grayscale or on YCbCr color space (luma/chroma channels)

2 - Apply 2D DCT to each block

Turns 64 spatial pixel values into 64 frequency coefficients

Low-frequency terms (top-left) = broad patterns

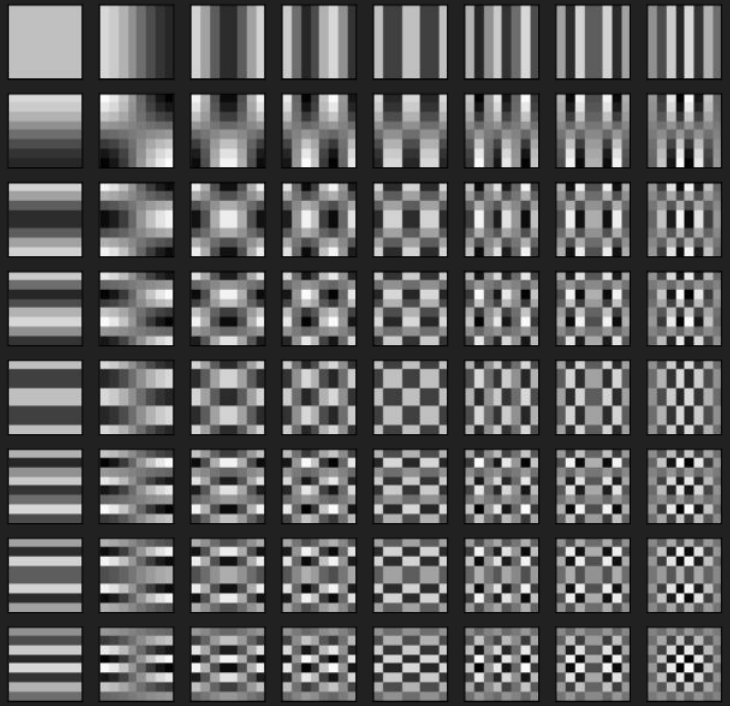
High-frequency terms (bottom-right) = tiny details

3 - Quantization

Human eyes are bad at high-frequency changes

JPEG aggressively reduces (or zeroes) high-frequency coefficients

This is where lossy compression happens



How to: Make JPEG

1 - Break image into 8×8 pixel blocks

Each block is treated independently

Operates on grayscale or on YCbCr color space (luma/chroma channels)

2 - Apply 2D DCT to each block

Turns 64 spatial pixel values into 64 frequency coefficients

Low-frequency terms (top-left) = broad patterns

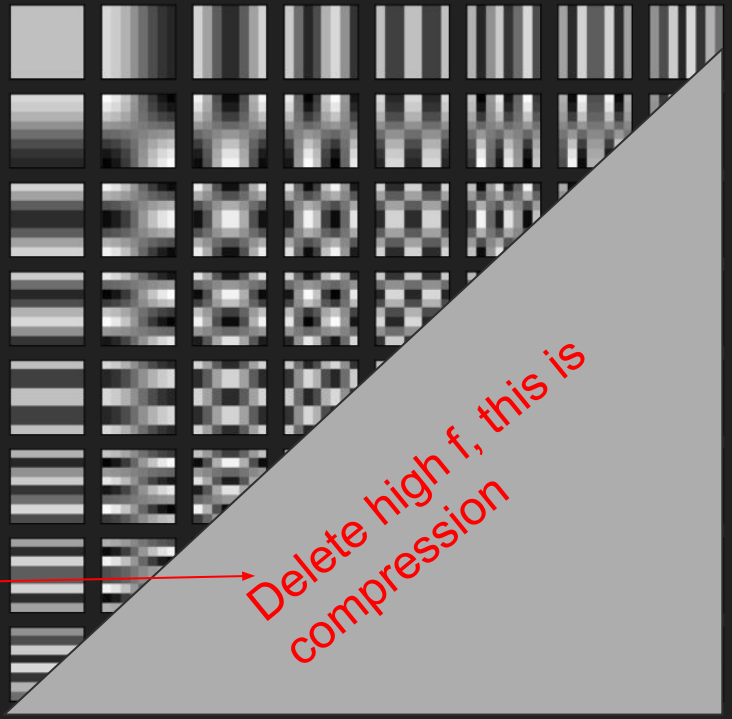
High-frequency terms (bottom-right) = tiny details

3 - Quantization

Human eyes are bad at high-frequency changes

JPEG aggressively reduces (or zeroes) high-frequency coefficients

*This is where **lossy** compression happens*



Lets code!

github.com/nialljmiller/PHYS4840_Fourier

ft_demo.py & ft_timing