CS 240 – Data Structures and Data Management

Module 10: Compression

A. Hunt, A. Jamshidpey O. Veksler

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Winter 2023

version 2023-03-28 10:02

Fourier Analysis – Basic Idea

Convert time or space-dependent data into the representation

$$f(t) = a_0 + a_1 \cos(qt) + b_1 \sin(qt) + a_2 \cos(2qt) + b_2 \sin(2qt) + \cdots$$

i.e. infinite sum of increasingly high frequency **sine/cosines**.

The a_i and b_i coefficients now determine the function.

Could view this as:

- a weird (non-polynomial) interpolation problem: what coefficients let us fit this particular summation form to a given function?
- an expansion/approximation of the function as an infinite sum of sines/cosines. (e.g., compare with our old friend, the Taylor series.)

Continuous Fourier Series

Goal is to represent any $f(t)$ as an infinite sum of trig functions:

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos\left(\frac{2\pi kt}{T}\right) + \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi kt}{T}\right)$$

a_k, b_k indicate the "information"/amplitude for each sinusoid of a specific period $\frac{T}{k}$, or frequency $\frac{k}{T}$.

Higher integer k indicates shorter period & higher wave frequency.

Continuous Fourier Series

We can write any periodic function over a period T as

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos\left(\frac{2\pi kt}{T}\right) + \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi kt}{T}\right)$$

with coefficients are given by solving the following integrals:

$$a_0 = \frac{\int_0^{2\pi} f(t)dt}{2\pi}$$

$$a_k = \frac{\int_0^{2\pi} f(t)\cos(kt)dt}{\int_0^{2\pi} \cos^2(kt)dt}$$

$$b_k = \frac{\int_0^{2\pi} f(t)\sin(kt)dt}{\int_0^{2\pi} \sin^2(kt)dt}$$

Fourier series with complex exponentials

Now, given our earlier sinusoidal expression of a function $f(t)$

$$f(t) = a_0 + \sum_{k=1}^{\infty} a_k \cos(kt) + \sum_{k=1}^{\infty} b_k \sin(kt),$$

we can express it more concisely as

$$f(t) = \sum_{k=-\infty}^{+\infty} c_k e^{ikt}$$

where the c_k coefficients are **complex numbers**.

There exists a simple conversion: $c_k \iff a_k, b_k$.

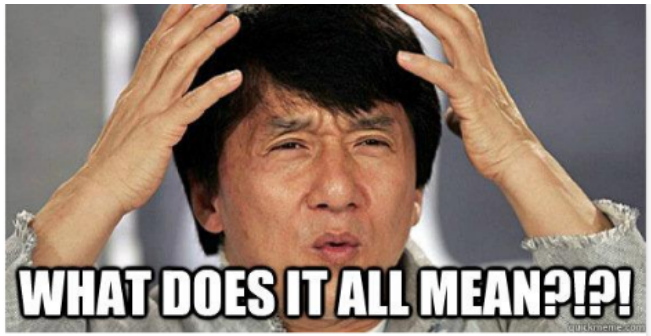
Fourier Series – Truncating

An *approximation* of a function could be achieved by **truncating** the series to a **finite** number of sinusoids:

$$f(t) \approx \sum_{k=-M}^{+M} c_k e^{ikt}$$

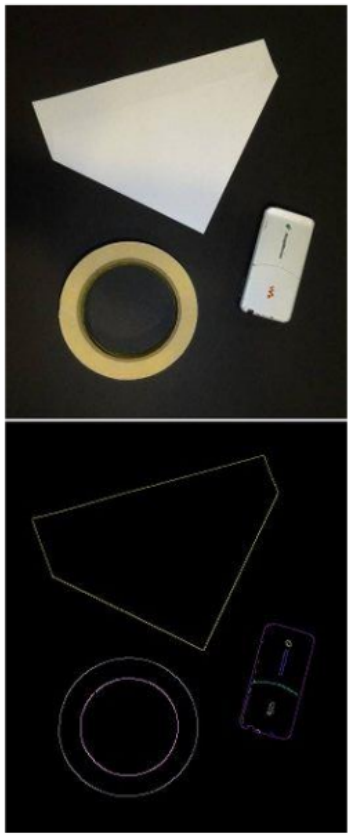
Today, we'll extend Fourier ideas to discrete data, rather than functions.

Extracting Meaning



But what does it all *mean?* Various "physical" interpretations, depending on context:

- In electrical signals, the $\left|c_k\right|$ describe the power at given frequency $\frac{k}{T}$.
- High frequencies (i.e., large k terms) are often noise in a signal. Filtering out (dropping) these frequencies may clean the data.
- Or, high frequency image components might suggest edges (discontinuities). Can be used to detect features, or sharpen/enhance edges.



Edge Detection

Discrete Input Data

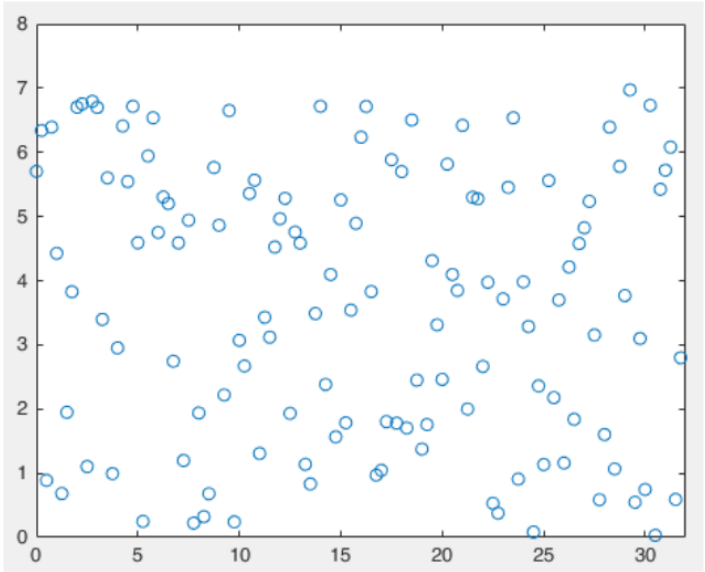
Consider now a vector of **discrete data**.

e.g., $f_0, f_1, f_2, \ldots, f_{N-1}$ for N uniformly spaced data points (N assumed even).

Assume data are from an *unknown* function $f(t)$, evaluated at each

$t_n = n\Delta t = \frac{nT}{N}$, for $n = 0, 1, \ldots, N-1$.

i.e. $f_n = f(t_n)$.



N=128 discrete data samples over a domain [0,32].

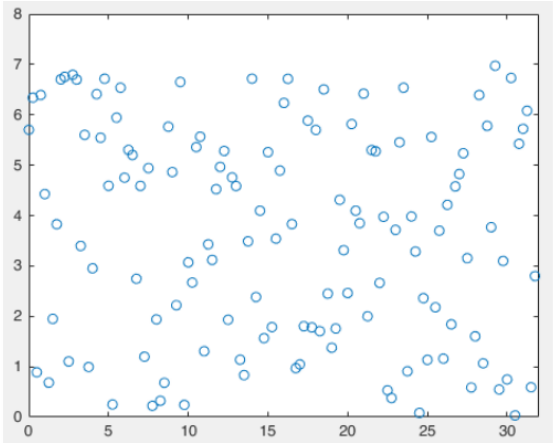
Discrete Fourier Transform as interpolation

We have $N = 128$ points, and period $T = 32$.

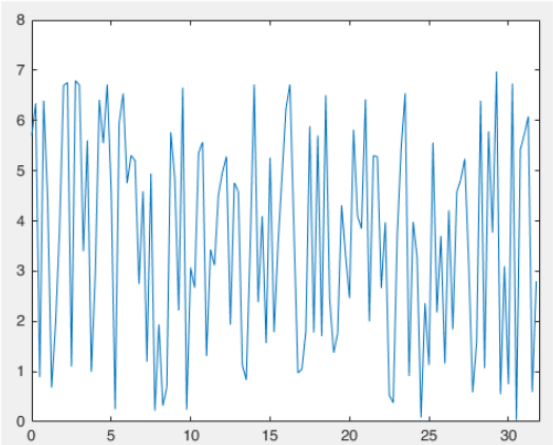
For N points, we will use N degrees of freedom (i.e., N coefficients) to exactly **interpolate** the data.

Assuming N is even, we can approximate with a truncated Fourier series as:

$$f(t) \approx \sum_{-\frac{N}{2}+1}^{N/2} c_k e^{\frac{(2\pi i)kt}{T}}$$



N=128 discrete data samples over a domain [0,32].



Piecewise linear plot.

12

Discrete Fourier Transform

Plugging in each of our N data points (t_n, f_n) into the expression

$$f(t) \approx \sum_{k=-\frac{N}{2}+1}^{N/2} c_k e^{\frac{(2\pi i)kt}{T}}$$

will give us N equations, involving unknowns coefficients, c_k.

This will lead towards our Discrete Fourier Transform.

Discrete Fourier Transform

For notational convenience we defined:

$$W = e^{\left(\frac{2\pi i}{N}\right)}$$

W is an Nth **Root of Unity**, since it satisfies

$$W^N = e^{2\pi i} = 1$$

So

$$f_n = \sum_{k=0}^{N-1} F_k e^{i\left(\frac{2\pi nk}{N}\right)} = \sum_{k=0}^{N-1} F_k W^{nk}$$

Discrete data f_n is expressed as a sum of coefficients, F_k, times complex exponentials, W^{nk}.

Discrete Fourier Transform

Typically, we want to learn/achieve something by processing the data (Image data, audio samples, prices, intensities, etc.).

In theory, the time-domain data tells us everything!

In practice, Fourier coefficients provide easier access to useful insights/information for certain problems.

Note: A Lack of Standardization

Various definitions of DFT/IDFT pairs can be found in the literature/code.

We use the following, with 0-based indexing:

$$f_n = \sum_{k=0}^{N-1} F_k W^{nk} \qquad \text{and} \qquad F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk}$$

SciPy (and some other sources) use:

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} F_k W^{nk} \qquad \text{and} \qquad F_k = \sum_{n=0}^{N-1} f_n W^{-nk}$$

Note the different placement of the constant scaling by $\frac{1}{N}$.

So be careful (a) when coding in Jupyter, and (b) reading other sources!

Slow Fourier Transform

A direct implementation of $F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk}$ takes $O(N^2)$ complex floating-point operations.

Essentially two nested **for** loops:

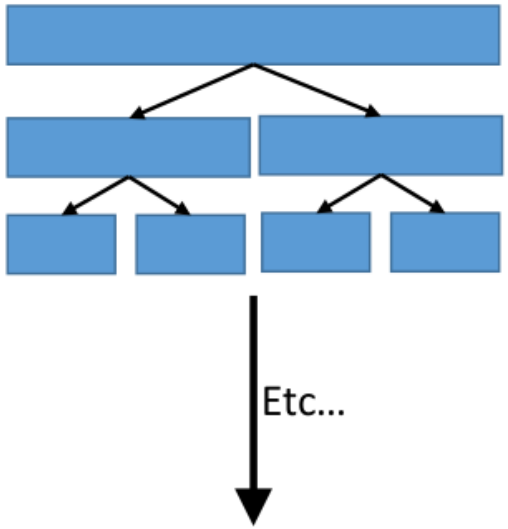
For $k = 0 : N - 1$ //iterate over all k unknown coeffs
    $F_k = 0$ //initialize coefficient to zero
    For $n = 0 : N - 1$ //iterate over all n data values
        $F_k += f_n W^{-nk}$ //increment by scaled data value
    End
    $F_k = F_k / N$ //normalize
End

A Faster Fourier Transform

Design a *divide and conquer* strategy.

We'll:

1. Split the full DFT into two DFT's of half the length.
2. Repeat recursively.
3. Finish at the base case of individual numbers.



(If $N \neq 2^m$ for some m, we can pad our initial data with zeros.)

Key question: *How* can we split up the DFT?

Dividing it up

The usual DFT of the sequence f_n is:

$$F_k = \frac{1}{N} \sum_{n=0}^{N-1} f_n W^{-nk}$$

We'll show we can express it with DFTs of two new arrays of **half the length** $(N/2)$:

$$g_n = \frac{1}{2}(f_n + f_{n+\frac{N}{2}})$$

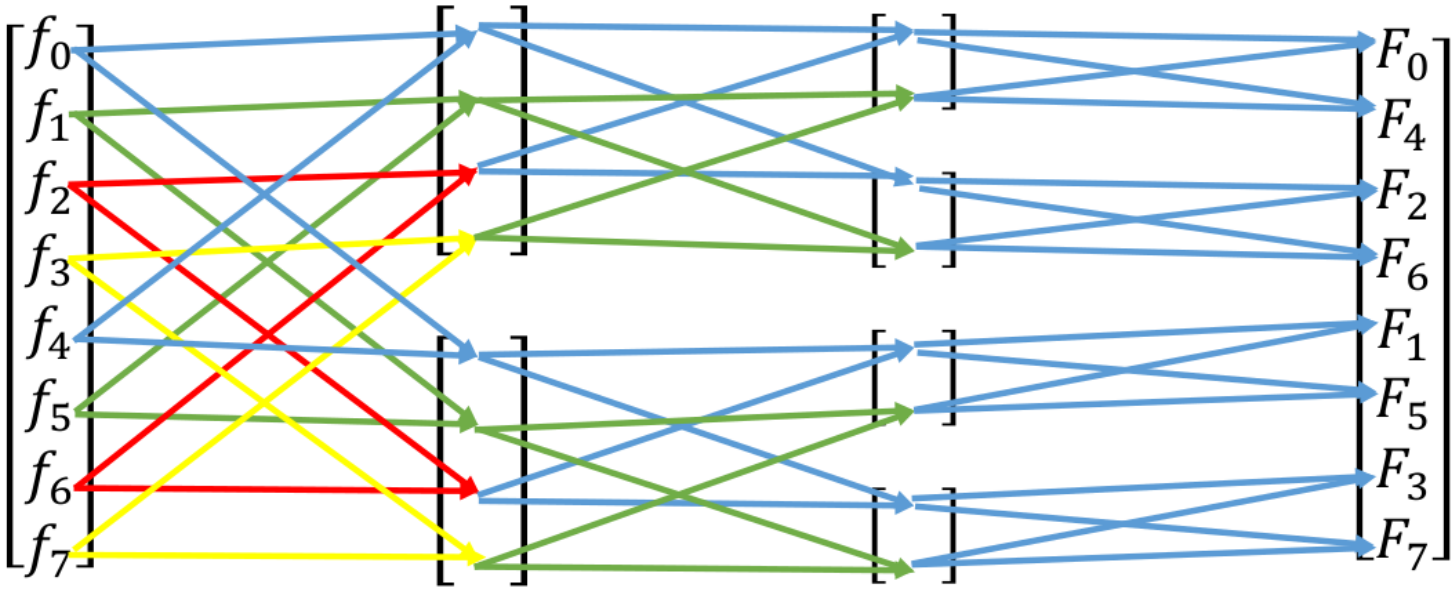
$$h_n = \frac{1}{2}(f_n - f_{n+\frac{N}{2}})W^{-n}$$

where $n \in \left[0, \frac{N}{2} - 1\right]$.

Then

$$F_{\text{even}} = G = DFT(g) \qquad \text{and} \qquad F_{\text{odd}} = H = DFT(h)$$

Big Picture – Recursive Butterfly FFT algorithm



$N = 8 = 2^3$, so we have 3 recursive stages.

Note: But coefficient output order is scrambled.

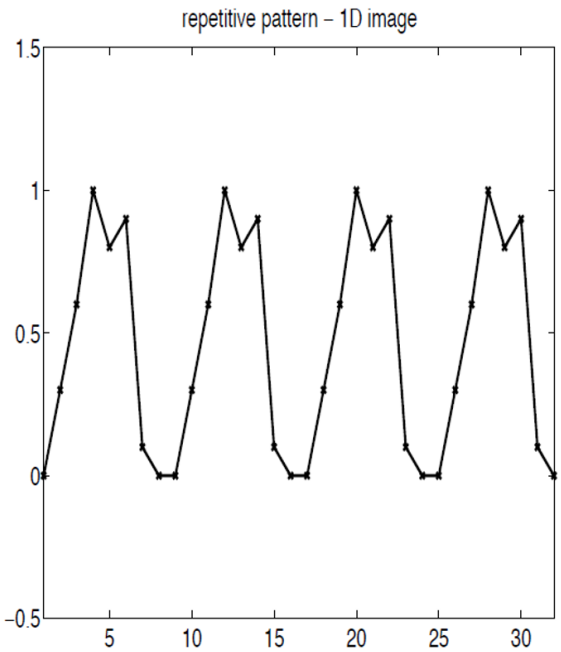
1D Grayscale Image

We can think of a 1D array of grayscale values as a "1D image".

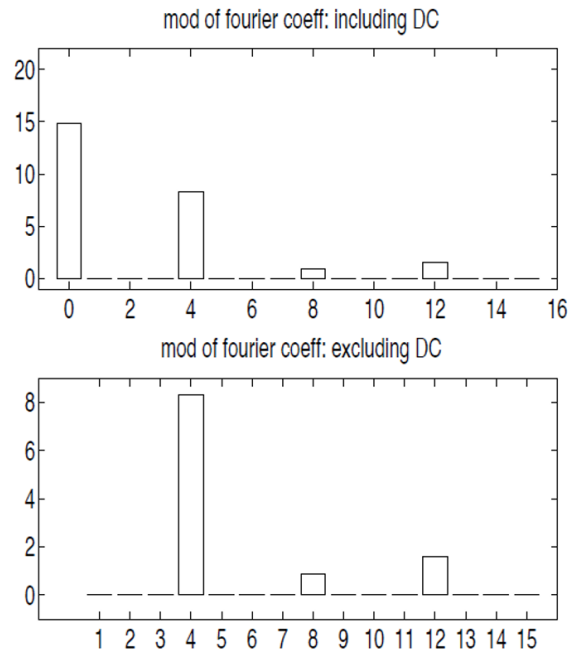
Each entry gives the pixels' intensity/gray values

Processing "1D Images"

For grayscale images, we can perform a DFT on the pixels' intensity/gray values.



1-D image x



Size of Fourier Coefficients

Notice: Plot is only the first 16 coefficients, since real data implies conjugate symmetry!

Processing "1D Images"

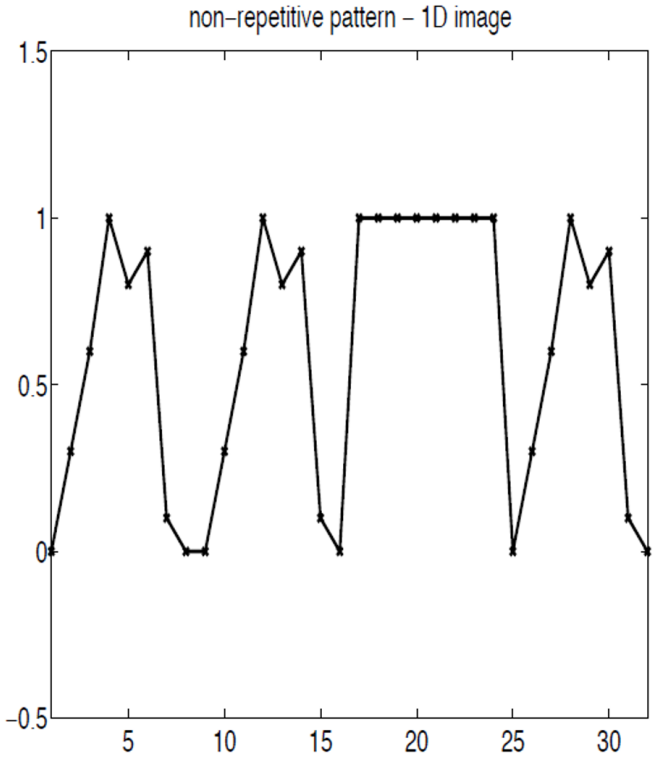
In this example

- Average is about 0.5, so $F_0 \approx 0.5$. (Well, actually ~ 15 since plot uses Jupyter's convention.)

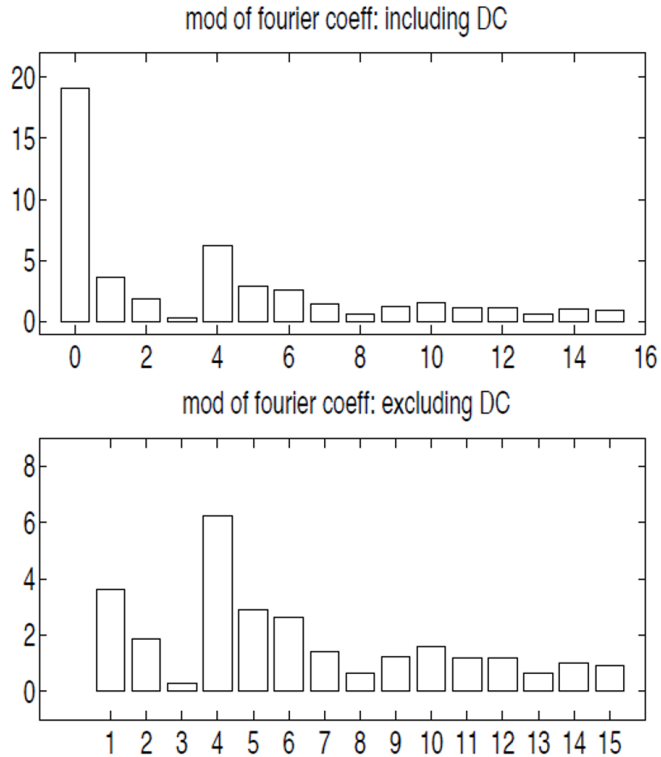
- Overall pattern has few coefficients active. Can store it cheaply just with F_0, F_4, F_8 and F_{12}.

If we instead replace the 3^{rd} bump's top with a flat line, the simple repetition is destroyed.

- Many more Fourier coefficients will become active / non-zero. Becomes expensive to store!



1-D image y



Size of Fourier Coefficients

Compression of 1D Images

- Although many coefficients are non-zero, many still have fairly small moduli/magnitude...
- ...so those frequencies contribute less to the image.

Compression Strategy:

- Create an (approximate) compressed version of the image, f_n, by throwing away "small" Fourier coefficients: $|F_k| < tol$
- To reconstruct the image, run the **inverse** DFT to get modified data (pixels), $\hat{f}_n$.
- Discard the imaginary parts of $\hat{f}_n$, to ensure new data is strictly real.

Comparison

Comparison:

Recovered "image" hopefully has fairly small deviations from the original "true" data.

But! We store fewer Fourier coefficients and so save space!

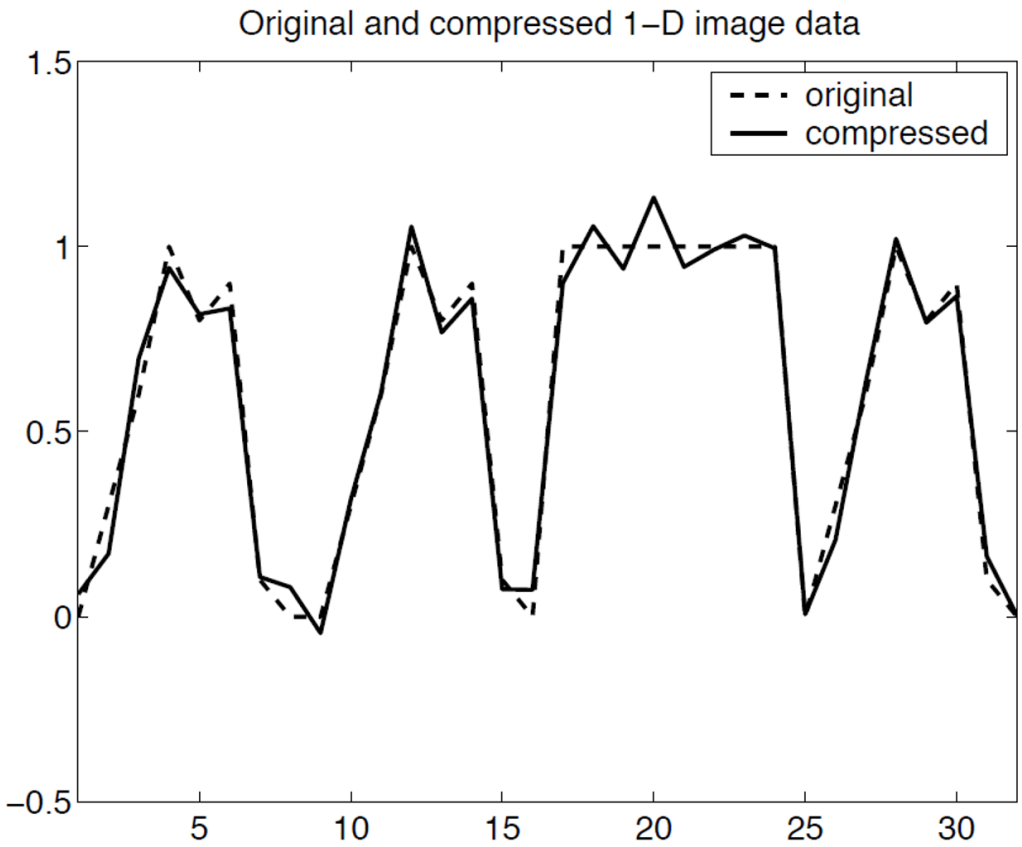


Image Compression in 2D

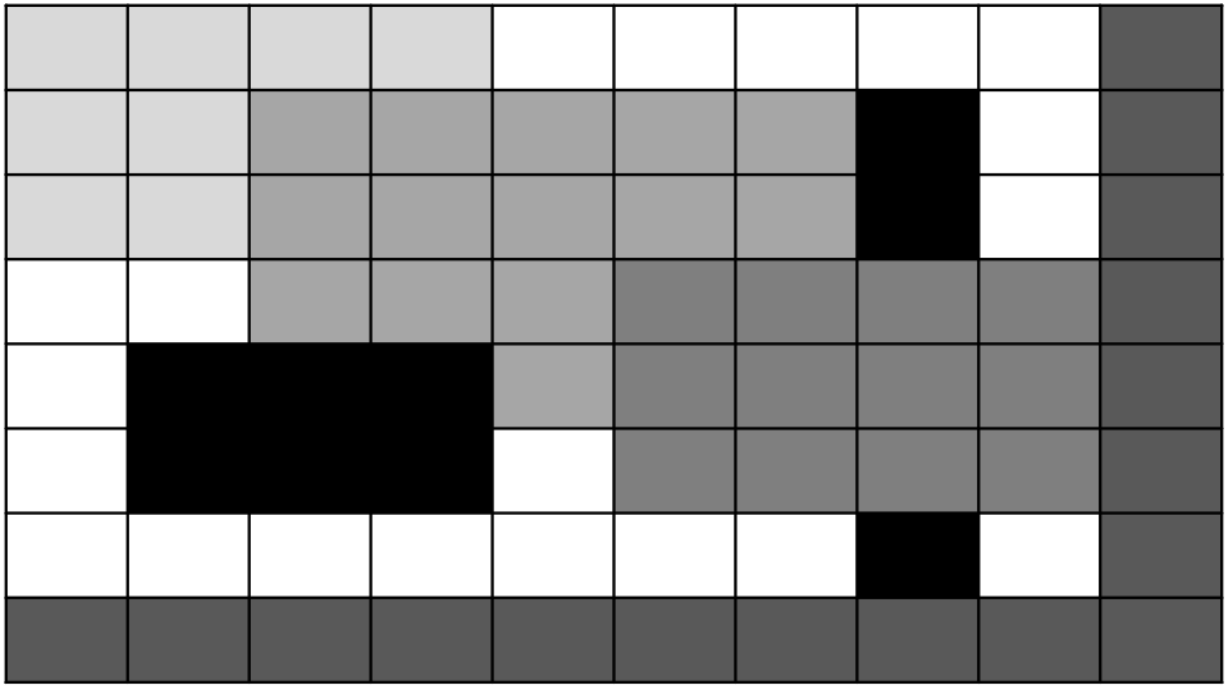
Ultimately, this is what we would like to achieve...

(a) Original

(b) Compressed by 85%

Image Processing in 2D

How does the DFT work for 2D (grayscale) image data? i.e., we have a 2D array X of per-pixel intensities, of size $M \times N$. Assume we have scaled image data, so $0 \leq X(i,j) \leq 1$.



2D Fast Fourier Transform - Complexity

Performing M 1D FFT's of length N: $M.O(N \log_2 N) = O(MN \log_2 N)$.

Performing N 1D FFT's of length M: $N.O(M \log_2 M) = O(MN \log_2 M)$.

So total complexity is $O(MN(\log_2 M + \log_2 N))$.

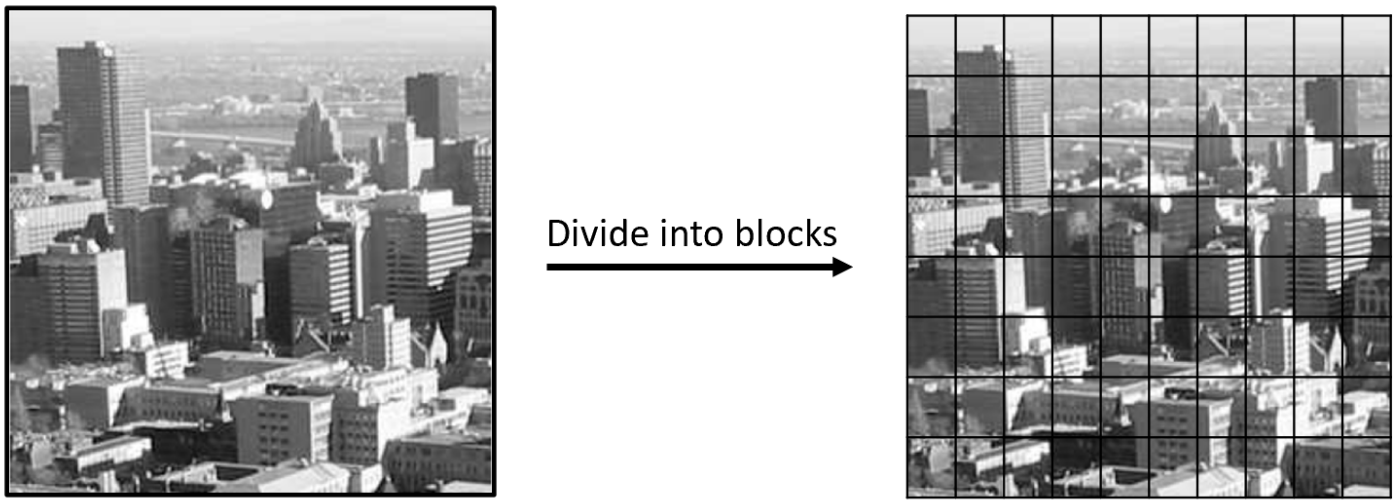
Simple Image Compression in 2D:

Take the 2D FFT of image intensities, threshold the magnitudes of the Fourier coefficients and discard small ones, as in 1D. Store only coefficients, and do IFFT when you want to view it again.

Block-based Image Compression

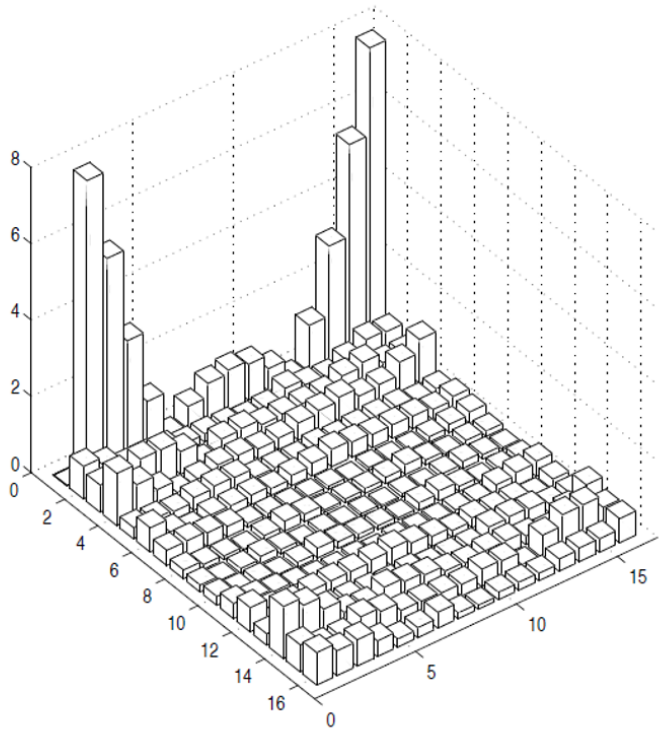
Often an image is subdivided into smaller blocks: 16×16 or 8×8 pixels.

Compression is performed on each block independently.

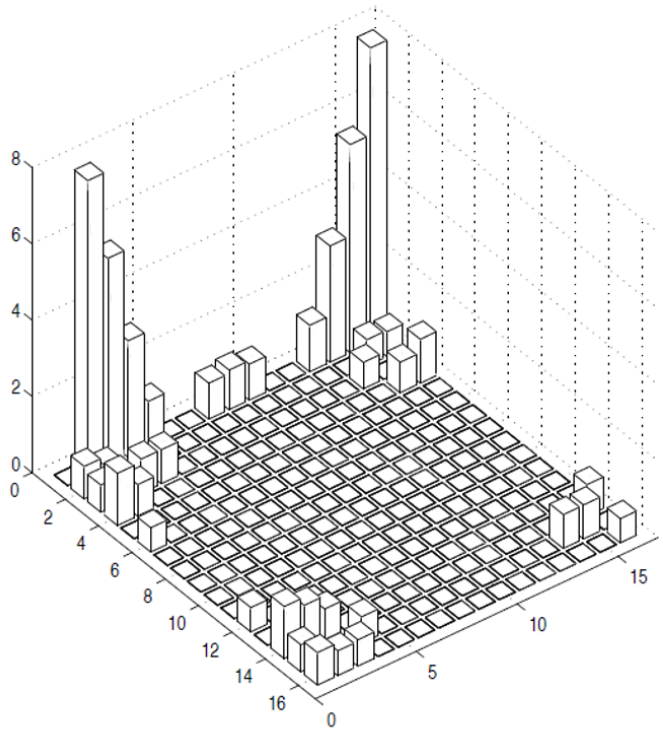


Pixels within a block often have similar/related data, and hopefully compress more effectively.

2D Fourier Plot for a 16×16 block



(a) Original



(b) Compressed by 85%