JPEG Compression

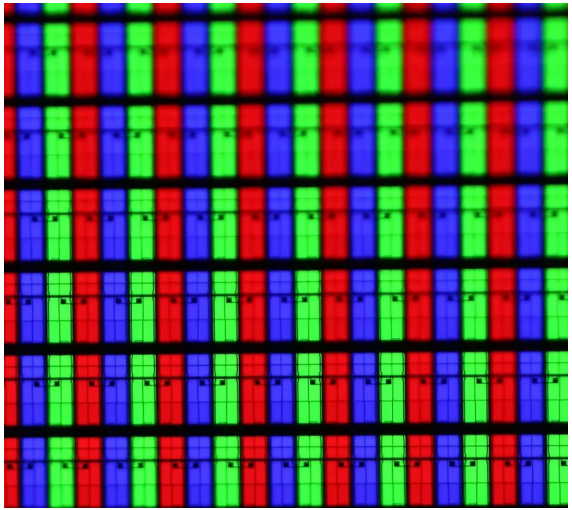
SIG Math Fall 2025

Motivating example

- A friend of mine is making a video game and wanted to implement a steganography feature where the player's user ID would be embedded in the pixels of all their screenshots
- This task is straightforward on iOS where photos are stored as png, but on some Android phones screenshots are stored as jpegs and suffer from compression
- Figuring out how to encode messages that persist through jpeg compression requires deep knowledge of how jpeg works!

How images work

- Images consist of a grid of pixels. Each pixel has a red, green, and blue component. These color components normally range from 0 to 255.
- The simplest file format is BMP (short for bitmap). If you open a bmp file and read it, it just lists out all the pixels' color components in order.
- While BMP's are very simple and precise, they are also very large! So they're not usually used except in places such as game textures.
- For reference: a 1920x1080 image is 6.2MB.



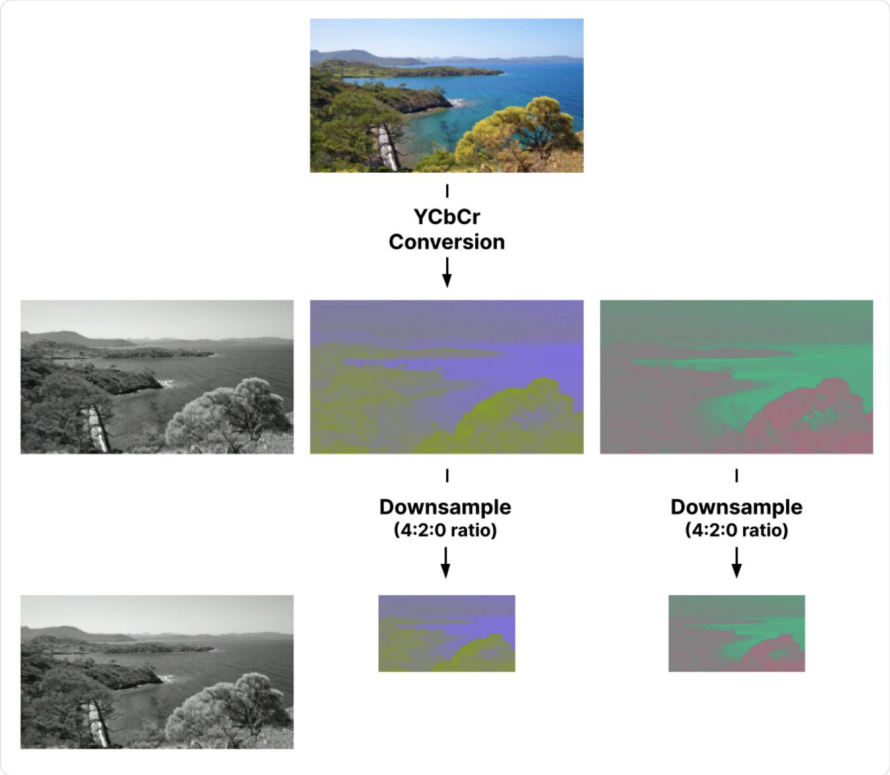
PNG vs JPEG

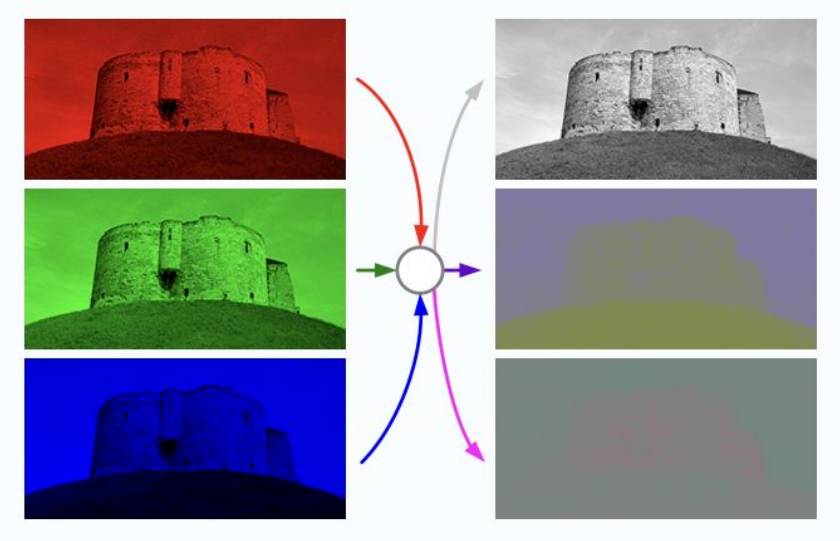
- PNG is a form of lossless compression, meaning that the RGB values will never be changed.
- JPEG is a form of lossy compression, so if you save your image as JPEG some of your pixel values may be subtly shifted.
- As a result, JPEGs are usually smaller than PNGs because they can discard some data – a 1920x1080 PNG is about 2MB, while an equivalent JPEG can be anywhere from 300KB to 1.5MB for the same amount of data.

Different color spaces

- At the beginning we talked about the RGB color space. This is great for a computer (since pixels have red, green, and blue lights), but the human eye is much more sensitive to brightness than color.
- JPEG exploits this by making significant changes to the picture's color information without affecting the picture's perceived quality by remapping the image from RGB to YCbCr.
  - Y is luminance and corresponds to how bright the pixel is.
  - Cb and Cr are chrominance blue and red channels that contain color information.
- JPEG averages the Cb and Cr values in each 2x2 pixel block, which saves space without significantly impacting image perception. This alone reduces file size by 50%!

YCbCr example





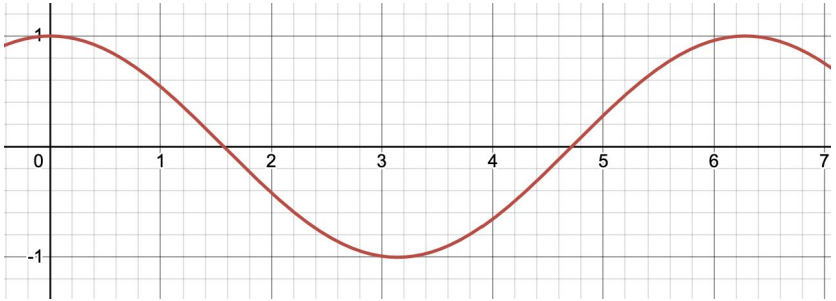
Frequency in images

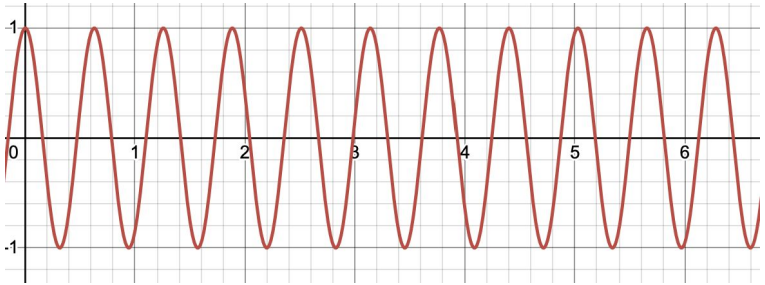
High vs low frequency

- In particular, a big distinction is between high and low frequency components. Intuitively, frequency is just "how much does the value change".
  - High frequency corresponds to fine details, sharp edges, textures, and intricate patterns.
    - This might be the stitches on your jeans or the details of a grassy field, and help make your image look "sharp".
  - Low frequency corresponds to smooth gradients, large color areas, and gradual changes.
    - These might be the overall color of your jeans or the field, and give the basic structure of an image.
- The big idea behind jpeg compression is that **your eye is much better at perceiving low frequency components than high frequency ones**.

Mathematically defining frequency

- Let's try to make a mathematical definition of frequency. One example is cosine waves. Look at the graphs of cos(x) and cos(10x):

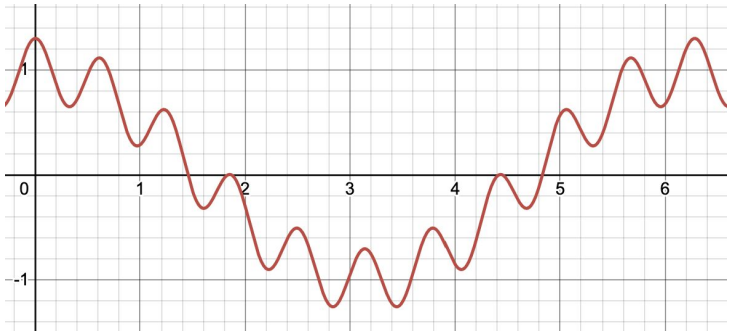




- Notice how the cos(10x) graph changes 10 times as much as the cos(x) graph! That's because its frequency is 10 times as high.
- You can think of the frequency of a cosine graph as the coefficient of the x term inside the cosine.

Breaking down graphs

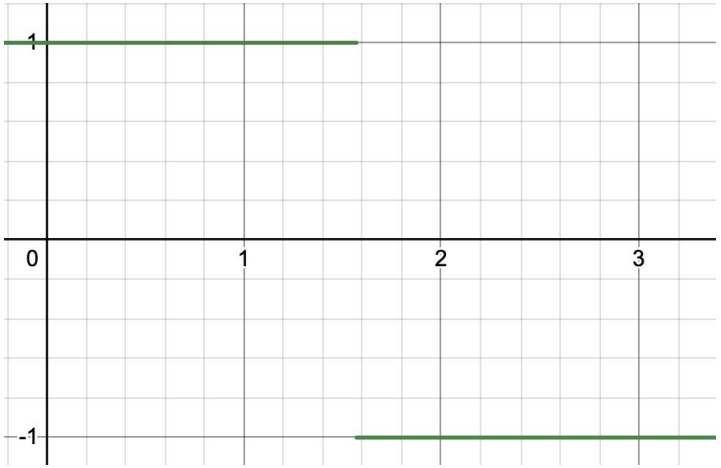
- Now let's look at this cosine graph. There's definitely a high frequency component, but there's also a low frequency component too.



- We can break this graph down into cos(x) + 0.3cos(10x). Now let's make a new graph, with the frequencies on the x axis and the corresponding amplitudes on the y axis:

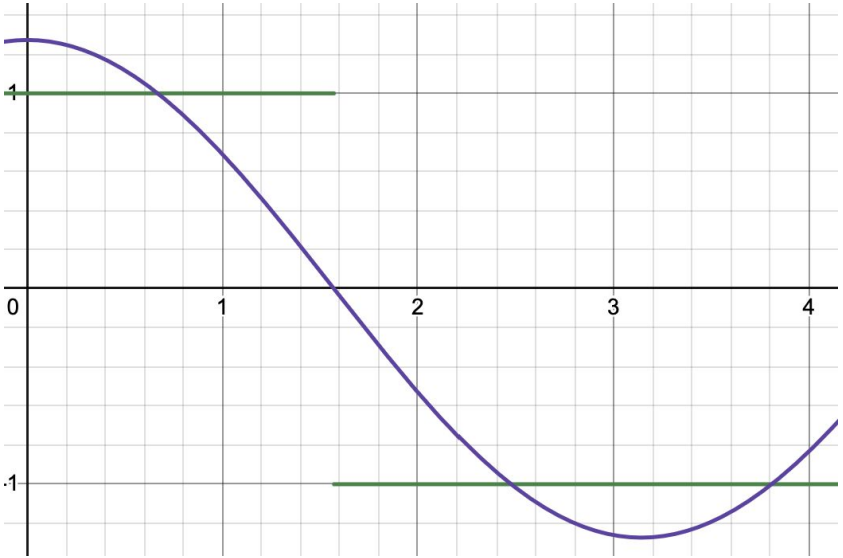
A more complex example

- One obvious problem is that not all graphs are made of cosines. Let’s look at this square wave – you can think of this as an image that's white on the left half and black on the right. It's obvious that we can't perfectly represent this with cosine, but we can approximate it.



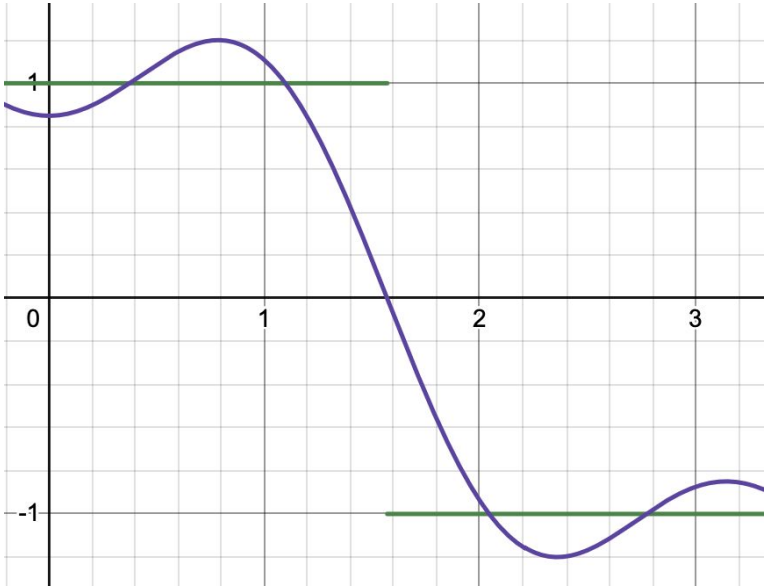
A more complex example (cont.)

- Here is the graph with only a cos(x) term. It's kind of got the vibes right, but the details are very off, which makes sense since this is a very low frequency approximation.



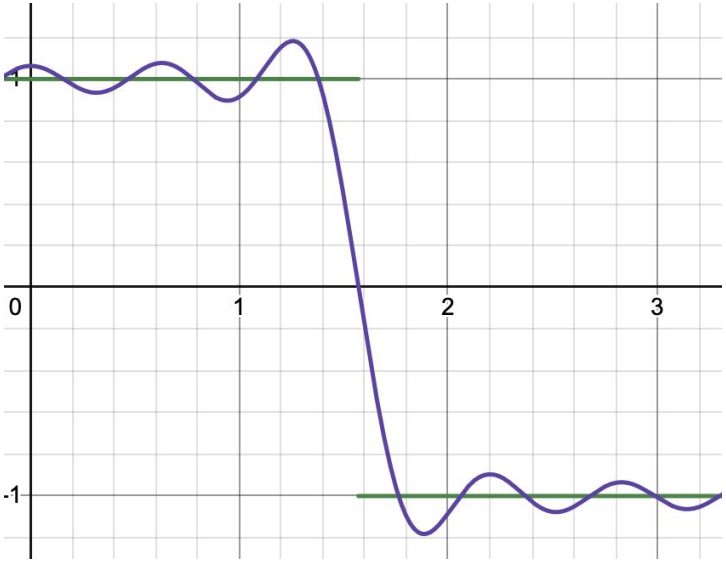
A more complex example (cont.)

- Adding a cos(3x) term, it's a bit better, but since we are only using low frequencies the details are still not very accurate.



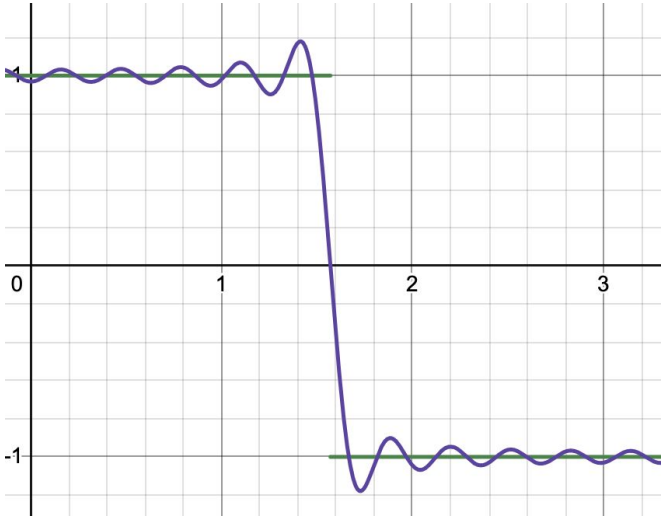
A more complex example (cont.)

- Let’s go up to a cos(9x) term. It’s definitely taking shape – the low frequency structure is all there, just not the high frequency details.



A more complex example (cont.)

- Now here is the sum up to the cos(19x) term. As you can see, the high-frequency functions are mostly for filling in the details, while the low-frequency functions are for the general structure.



The Direct Cosine Transform

The Direct Cosine Transform

- You might have heard of the Fourier Transform (FT). The idea is that similar to how Taylor Series represent any function as a polynomial, FTs let you represent any function as a sum of sines and cosines.
  - This is especially useful for sound processing where everything naturally is a sine wave.
- One important note is that the FT is used for continuous functions. For discrete functions (including images), the Discrete Fourier Transform (DFT) is used instead.
  - If you have ever heard of an FFT, that is just a Fast (discrete) Fourier Transform.
- The 1D Fourier Transform formula looks like this:

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi f t} dt$$

The Direct Cosine Transform (cont.)

- You'll notice that there is a complex exponential, so there are real and imaginary coefficients that have to be stored. For a computer this is unacceptable, so we have to pick either sine or cosine only.
  - Because cos(0) is 1, the first coefficient of DCT is the average of the pixels. That makes it a really good start. However, sin(0) is 0, so the DST's start as a gentle mount or flat plain which is uncommon in most images, and DST's generally take more coefficients than DCT's to encode most blocks. So we end up picking cosine.
- The end result is a formula that looks like this:

$$D_{h,v}(m,n) = 2\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\frac{1}{\sqrt{M}}\frac{1}{\sqrt{N}}\eta_h\eta_v\cos\left(\frac{\pi h}{2M}(2m+1)\right)\cos\left(\frac{\pi v}{2N}(2n+1)\right)$$

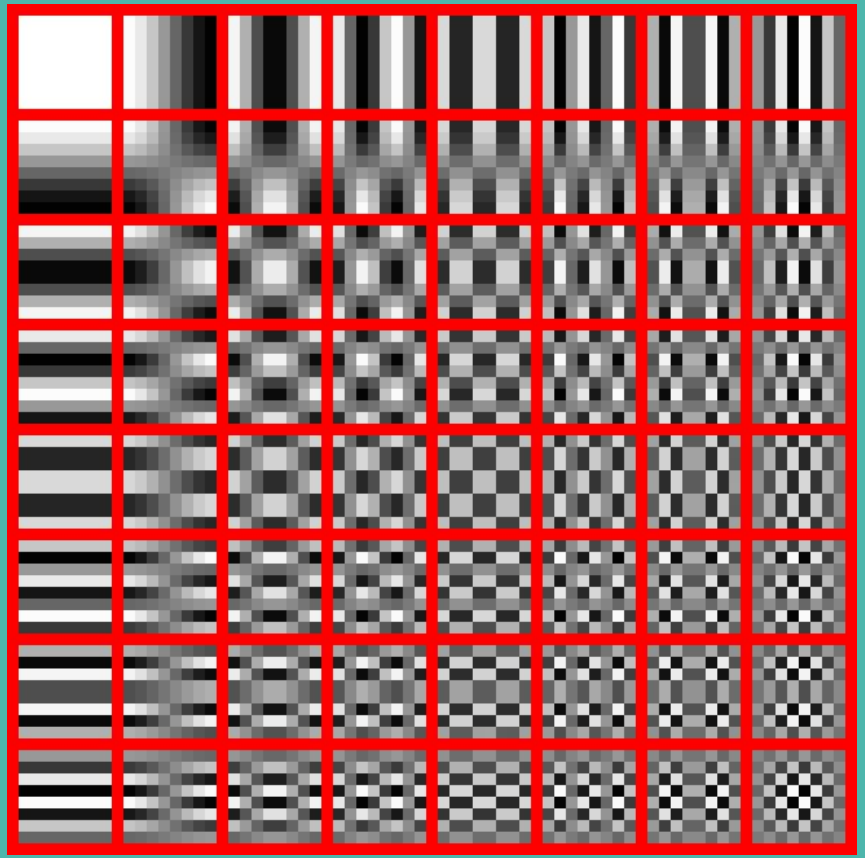
  - If you read about this, you may see DCT-II used sometimes, this is because DCT is a periodic extension and there are 16 ways to specify the boundary conditions.

The Direct Cosine Transform (cont.)

- A very important note is that we **only apply the DCT to 8x8 blocks of the image.**
- In any given image, there could be a lot going on in the whole thing, and some frequencies may only appear in some parts of the image.
- By contrast, in each 8x8 pixel block there's not much variation, so it's much easier to extract the frequency data.
- Everything in these next slides will be applied to one 8x8 block, which can be repeated to the entire image.

The Direct Cosine Transform (cont.)

This is the prototypical image when talking about the DCT. It's a little confusing at first, but we can break it down.



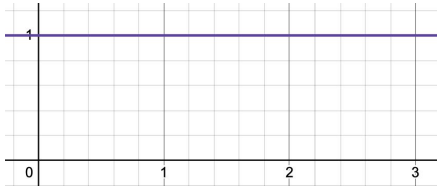
The Direct Cosine Transform (cont.)

- The first thing is that the chart is in x and y and is roughly symmetric, since images are 2D. So **if you understand what's happening in the 1D case, you can extrapolate to the 2D case**. Thus we only need to focus on the first row.

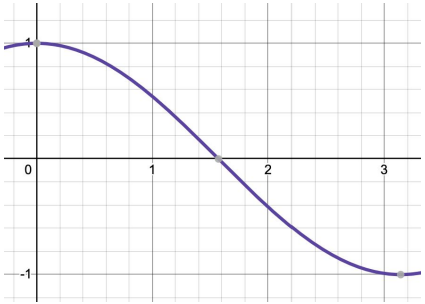
- Note the first cell is a solid square, corresponding to cos(0x). The second cell is a gradient and represents cos(x). The third cell goes up and back down, corresponding to cos(2x). As you move to the right, the coefficient of x increases.

The Direct Cosine Transform (cont.)

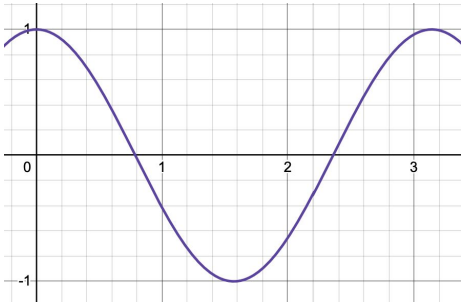
- cos(0x)



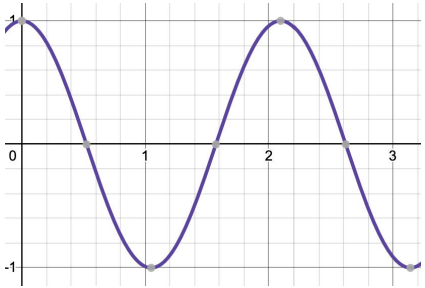
cos(x)



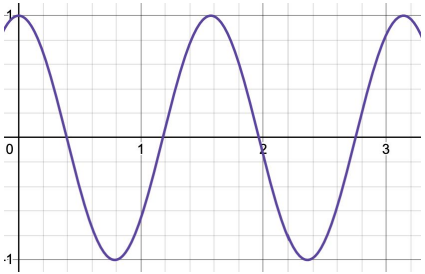
cos(2x)



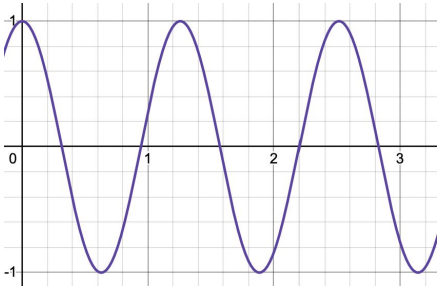
- cos(3x)



cos(4x)

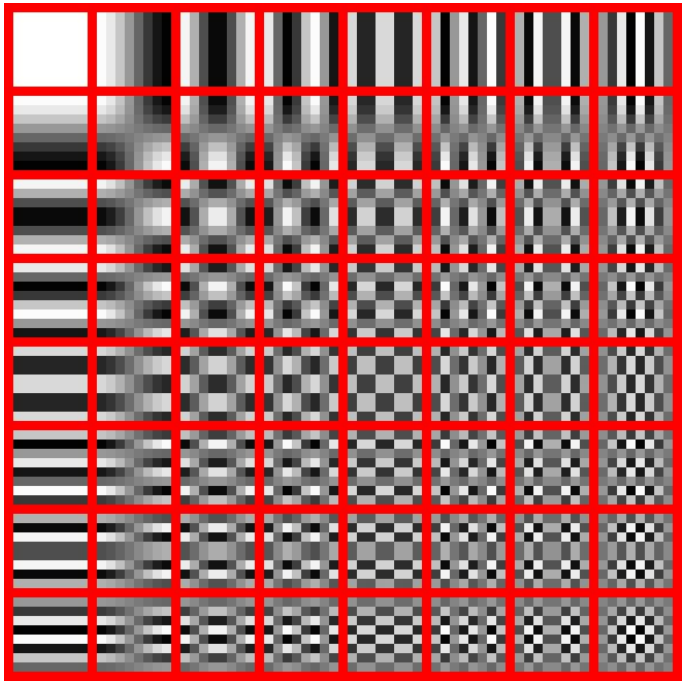


cos(5x)

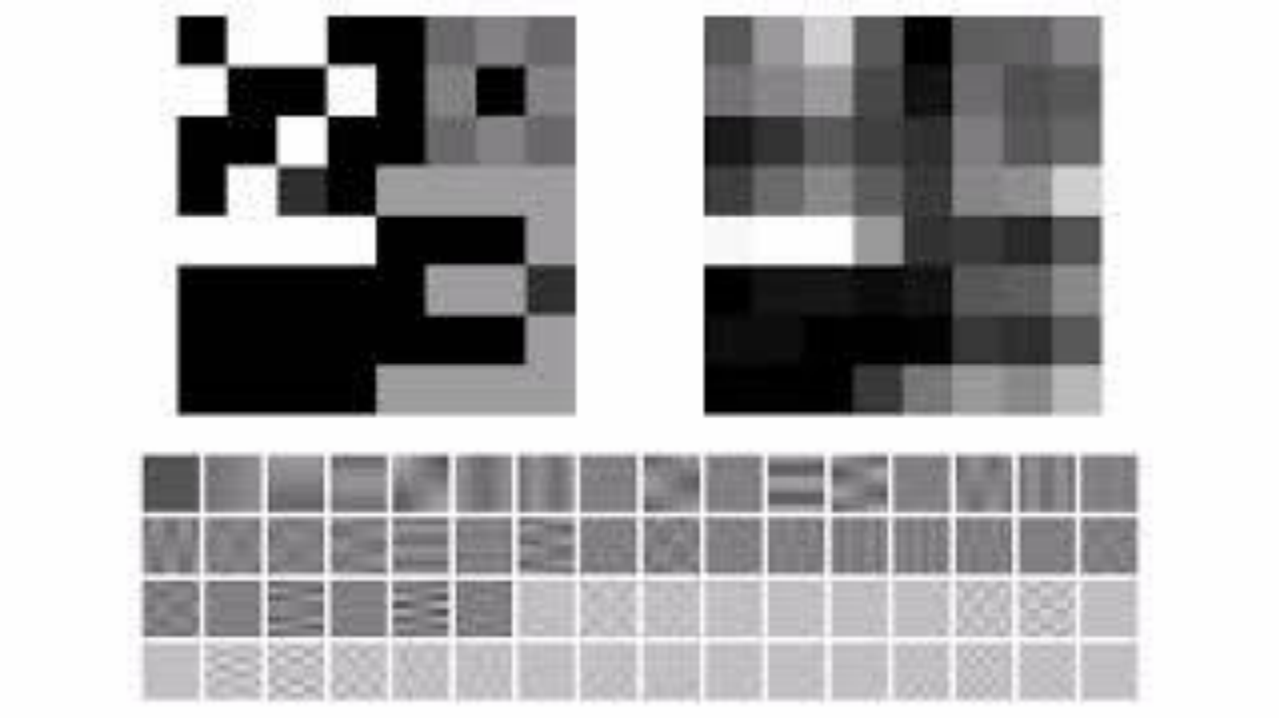


The Direct Cosine Transform (cont.)

- Once you have the frequency bands for the 1D case, you need them for the other direction too! You can multiply the two functions to get another cell, such as cos(4x)cos(5y).
- By adding all 64 of these cells multiplied by constants, **you can create any 8x8 pattern that you want!**

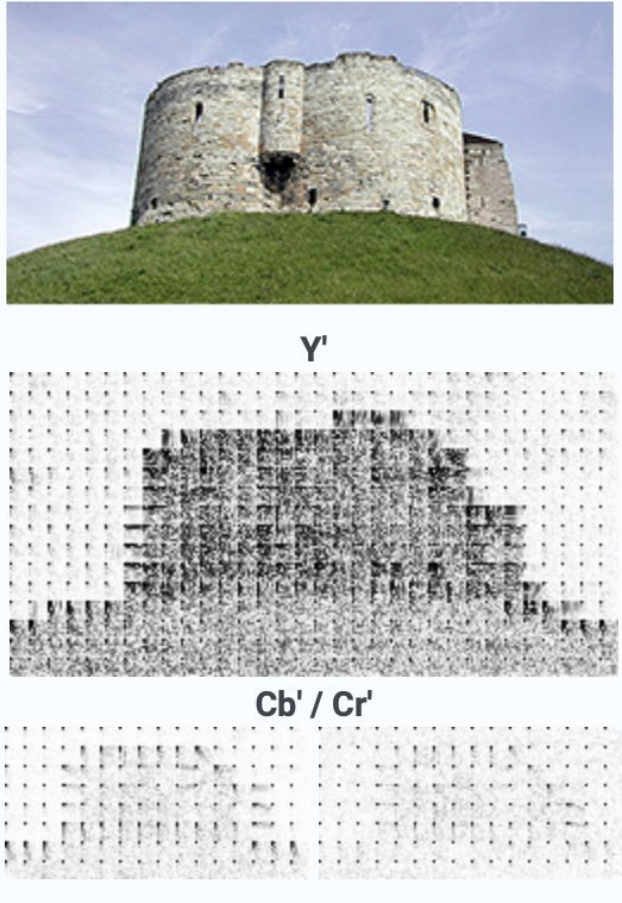


Animated Direct Cosine Transform Example



DCT Visualized

- Let’s take a look at this tower image. It’s been converted to YCbCr, and then DCT has been performed on it.
- You can see the top-left corner of each 8x8 block is filled due to the low-frequency info.
- For the blocks in the sky region of the image, the rest of the 8x8 block is mostly empty as there is not much high-frequency data.
- On the other hand, the bricks have busy texture and lots of high-frequency change, which means the 8x8 block is more filled.



Quantization

- If you look closely you may realize that the DCT hasn't actually saved any information yet, it's just turning an 8x8 block of pixels into a different 8x8 block of pixels.
- However, the data has been transformed into a form where it can be more easily compressed with traditional lossless compression!
    - The key to this is that DCT makes the image less noisy, which means it works much better with lossless compression.
- Let's give an example. Compare these three sequences:
    - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 (n)
    - 0, 0, 1, 1, 2, 2, 3, 3, 4, 4 (n/2)
    - 0, 0, 0, 0, 1, 1, 1, 1, 2, 2 (n/4)
- The first sequence could be the list of values in the Y frequency channel. Note that the higher the divisor, the more repetition there is in the data and the easier it is to compress.

Quantization (cont.)

- So how do we quantize our JPEG image data? If you've ever saved an image and chosen a quality value, this is where it comes into play.
- We start with 2 8x8 tables of numbers called the **quantization tables**, one for brightness and one for color.
- Each cell in the brightness and chrominance tables after DCT is divided by the corresponding cell in the quantization table.
- The quality corresponds to the values in the quantization tables – the lower the quality, the higher the values. **More data is lost by quantization, but the image becomes smaller.**
- This is where the real savings of JPEG come in, and also where the lossiness comes into play.

Very low quality quantization

Luminance (brightness) table							
93	70	81	81	104	139	284	418
64	70	75	99	128	203	371	534
58	81	93	128	215	319	452	551
93	110	139	168	325	371	505	568
139	151	232	296	394	470	597	650
232	336	331	505	632	603	702	580
296	348	400	464	597	655	696	597
354	319	325	360	447	534	586	574

Chrominance (colour) table							
99	104	139	273	574	574	574	574
104	122	151	383	574	574	574	574
139	151	325	574	574	574	574	574
273	383	574	574	574	574	574	574
574	574	574	574	574	574	574	574
574	574	574	574	574	574	574	574
574	574	574	574	574	574	574	574
574	574	574	574	574	574	574	574

Quantized Y'

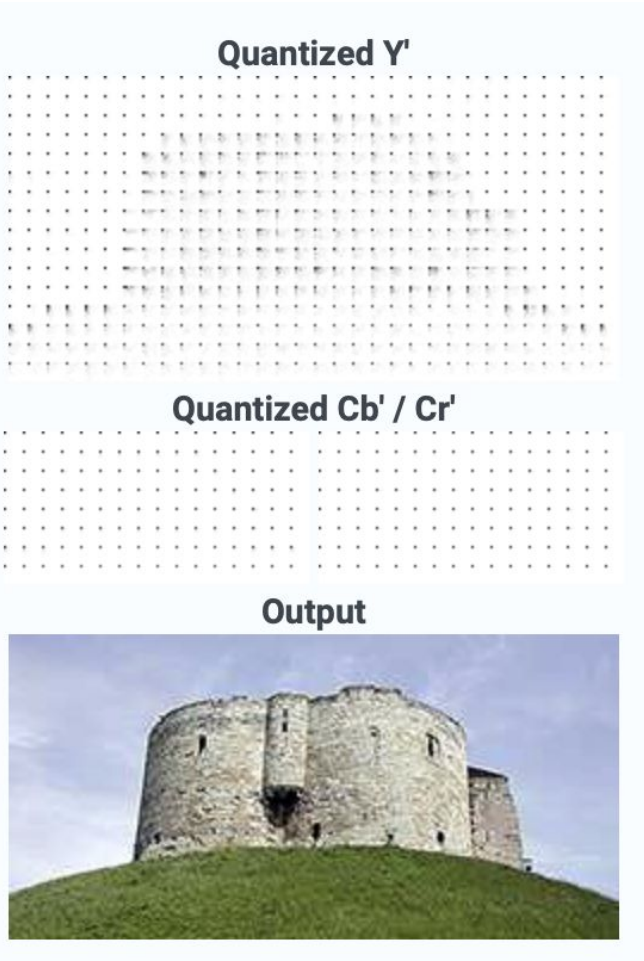
Quantized Cb' / Cr'

Output

Medium quality quantization

Luminance (brightness) table							
16	12	14	14	18	24	48	70
11	12	13	17	22	34	62	89
10	14	16	22	36	53	75	92
16	19	24	28	54	62	84	95
24	25	39	49	66	78	99	108
39	56	55	84	105	100	117	97
49	58	67	77	99	109	116	99
59	53	54	60	74	89	97	96

Chrominance (colour) table							
17	18	24	46	96	96	96	96
18	21	25	64	96	96	96	96
24	25	54	96	96	96	96	96
46	64	96	96	96	96	96	96
96	96	96	96	96	96	96	96
96	96	96	96	96	96	96	96
96	96	96	96	96	96	96	96
96	96	96	96	96	96	96	96

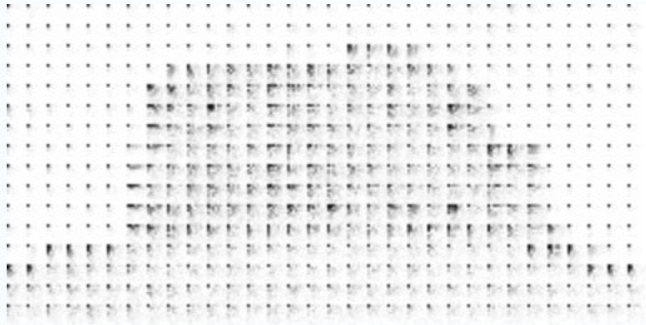


High quality quantization

Luminance (brightness) table							
4	3	4	4	4	6	11	15
3	3	3	4	5	8	14	19
3	4	4	5	8	12	16	20
4	5	6	7	12	14	18	20
6	6	9	11	14	17	21	23
9	12	12	18	23	22	25	21
11	13	15	17	21	23	25	21
13	12	12	13	16	19	21	21

Chrominance (colour) table							
4	4	6	10	21	21	21	21
4	5	6	14	21	21	21	21
6	6	12	21	21	21	21	21
10	14	21	21	21	21	21	21
21	21	21	21	21	21	21	21
21	21	21	21	21	21	21	21
21	21	21	21	21	21	21	21
21	21	21	21	21	21	21	21

Quantized Y'



Quantized Cb' / Cr'

Output

Putting it all together

Squeezing out the last bit of juice

- JPEG finally uses RLE and Huffman coding to compress the quantized image data losslessly.
- Quantization leaves a long list of 0's towards the bottom right corner. Imagine you have a string like 22888000000000. RLE compresses this as 223890.
- JPEG orders the bytes in a zigzag pattern that keeps the zeroes together. This is what the ffmpeg logo is based on!
- Finally, JPEG runs Huffman coding on the result.

Final compression algorithm

- Let's put all the pieces together. When we store an image as JPEG, we...
  - Convert from RGB to YCbCr
  - Downscale the chrominance channels
  - Split the image into 8x8 blocks and run the DCT
  - Quantize the DCT to reduce noise
  - Run RLE and Huffman encoding
- All of this put together gets us our final JPEG compression algorithm. We can also run it in reverse to decode the JPEG format into an RGB form that can be easily displayed, though some pixel data will be lost.

Thanks for listening!

Any questions?