# JPEG Recompression Detection

Digital Signal Processing – Assignment 4b

Samuel Stark (sws35)

18/01/2022

2835 words

**Word Count Begins Here**

---

## 1 Introduction

DSP Assignment 4b presented students with six images, and tasked them with implementing a technique to classify each image as either

- 0x JPEG compressed, e.g. originated directly from a camera
- 1x JPEG compressed, e.g. compressed and then decompressed once
- 2x JPEG compressed, e.g. compressed and decompressed more than once

This report demonstrates and explains my implementation, which automatically classifies an image *and* guesses the compression parameters, as well as the intuition informing the technique. Section 4 describes a test framework which generates similar images to those provided, and shows my implementation has a high degree of classification accuracy (98%) on these images. The code for my implementation, as well as my generated test images, was uploaded to Moodle along with this PDF. It can also be found on my GitHub[1] once the deadline has passed.

[1] https://github.com/theturboturnip/cambs-dsp/tree/main/assignment4

| | Compression | Quantization | Quality |
|---|---|---|---|
| Test 1 | 0x | - | - |
| Test 1C | 2x | 6, 8 | 80-83, 73-77 |
| Test 2 | 1x | 9 | 70-73 |
| Test 2C | 2x | 8, 8 | 73-77, 73-77 |
| Test 3 | 2x | 4, 10 | 86-89, 67-70 |
| Test 3C | 0x | - | - |

Table 1: Results

### 1.1 Background

The assignment noted three articles [1, 2, 3] which could help with the classification task. I also found four other articles [4, 5, 6, 7] which specifically acknowledge detecting re-compression.

Fan and de Queiroz (2003) introduced a statistical method for classifying compressed vs. uncompressed grayscale images[1]. Neelamani et al. (2006) refined this technique and expanded it to support color images. Their technique estimates not just the quantization levels, but also the color spaces and chroma subsampling methods used in previous JPEG compression[2], which was unnecessary for my purposes. I replicated the technique for grayscale images, using log-probabilities to accurately store very small probability values, but it turned out to not be useful.

Lewis and Kuhn (2010) demonstrate an "exact, complete, stable recompressor" which exactly inverts each stage of JPEG decompression to determine the original JPEG file for a decompressed bitmap. This process would likely find more accurate DCT values than using a standard DCT function. However it initially seemed too complicated to implement, especially considering how much precision was required, so I used the standard MATLAB `dct` function instead (see Section 2.2).

[4, 5] rely on machine learning constructs to identify re-compression, and focus on identifying useful features for SVM and PCA analysis rather than directly inferring compression level from those features. Popescu and Farid (2004) demonstrate that different levels of compression produce distinct artifacts in the frequency domain. Chen and Hsu (2011) exploit power-spectrum artifacts to detect recompression, including when the image is cropped after the first compression[6], but only describe how to extract these features rather than how to actually use them.

### 1.2 JPEG Compression Tools

The assignment webpage[2] notes the five processing pipelines used to generate the test images, and provides other raw images captured around the West Cambridge site. The processing pipelines used the standard Unix tools `cjpeg` and `djpeg` to compress and decompress JPEG data, respectively. In Section 3.3 I exploit this knowledge to reverse the transformation from quality to quantization level. In Section 4 I use the raw images to generate training data and gauge the accuracy of the classifier on known data.

## 2 Undoing JPEG decompression

The loss of quality during JPEG compression takes place at the DCT level. The DCT is evaluated for each 8x8 block in the image, and each of the 64 components are *quantized* - e.g. rounded to a multiple of some value $q$ that can be different for each coefficient. Decompression then evaluates the IDCT of those quantized coefficients to generate pixel values, which may be rounded to integers. The key to detecting multiple compressions is identifying multiple levels of quantization, and thus requires the quantized DCT terms. These are found by splitting the provided bitmaps into 8x8 blocks of pixels, and applying a DCT.

For color images, the RGB data is converted to YCrCb, the color space JPEG performs compression in, with the MATLAB `rgb2ycbcr` function. Due to chroma subsampling, where the CrCb components are filtered and compressed at half resolution, applying DCT to the raw CrCb components doesn't produce clearly quantized values (as noted in [6]). A technique for undoing the subsampling is shown in [3] but not implemented here. Instead, only the Y component is analysed.

[2] https://www.cl.cam.ac.uk/teaching/2021/L314/assignment4b/

## 2.1 Splitting into Blocks

Once the raw image pixels have been imported, they must be split into 8-by-8 pixel blocks for the DCT to operate on. The `blocks_8x8` function takes in the 2D image data, and outputs a linear array of 8-by-8 blocks. Most of this function's code is just to prepare suitable arguments to `mat2cell`, which produces a 2D array of 8x8 blocks (for example, a 16x8 image would produce a 2x1 array of blocks, which is `reshape`-d into a linear array).

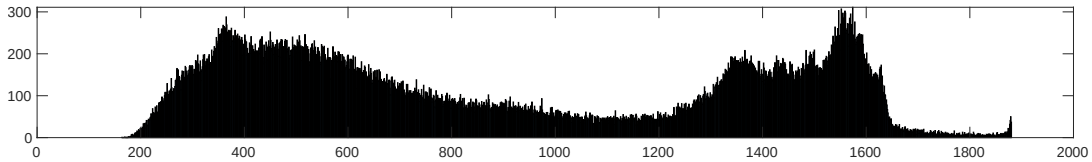
```
function bs=blocks_8x8(dat)
    n_blocks = size(dat)/8;
    % Create a 1xn_blocks(1) vector of 8s
    blocks_x = repmat(8, 1, n_blocks(1));
    blocks_y = repmat(8, 1, n_blocks(2));
    % Get a 2D array of blocks
    block_2d_array = mat2cell(dat, blocks_x, blocks_y);
    % Convert to a linear array of blocks
    bs = reshape(block_2d_array, [1 numel(block_2d_array)]);
end
```

## 2.2 Acquiring the DCT histogram

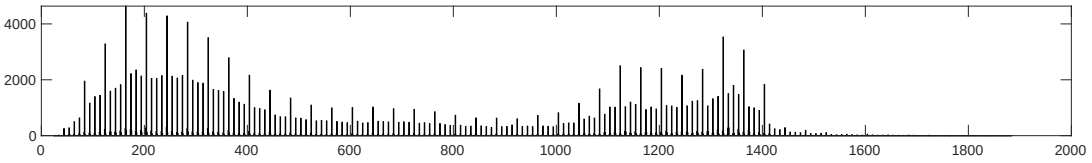
JPEG encoders quantize the coefficients calculated from a DCT on each block. In order to find the quantization, and thus the compression, the DCT must be calculated. The `DCT_params` function takes in a linear array of 8x8 blocks and returns a 8x8 array $D$ (lifting the naming convention from [2]). Each element, $D\{i, j\}$, contains a vector of all values the $i, j$th DCT coefficient has taken in each block. For example, $D\{1, 1\}$ holds the value of the 1st DCT component in the first block, the value in the second block, and so on. Fan and de Queiroz note that uniform blocks (consisting of a single value) and potentially-truncated blocks (those with values which may have been clamped between 0 and 255) could throw off their algorithm. Such blocks are excluded from this computation, although the variant on this algorithm presented by Neelamani et al. doesn't explicitly require it.

```
function Di=DCT_params(bs)
    % Di = 8x8 cell array, each cell = 1xlength(bs) vector
    Di = cell(8, 8);
    % Preallocate enough memory for all blocks on each coefficient
    % ... omitted for brevity ...
    n_dcts = 1;
    for i = 1:length(bs)
        b = bs{i};
        % Do Fan2003 filtering here
        if ismember(255, b) || ismember(0, b) || length(unique(b)) == 1
            % This block is potentially-truncated or uniform
            % => DCT could throw off results, don't do it
        else
            % For unsigned 8-bit input, DCT coefficients are signed 11-bit ints
            % => it's OK to round them
            d = round(dct2(b));
            for j = 1:8
                for k = 1:8
                    Di{j,k}(n_dcts) = d(j,k);
                end
            end
            n_dcts = n_dcts + 1;
        end
    end
    % Shrink the cell vectors to only the DCTs we calculated
    % ... omitted for brevity ...
end
% Example of generating a histogram
h = histcounts(abs(Di{1,1}),'BinMethod','Integer');
```

# 3 Detecting quantized DCT values




(a) Raw Unquantized Histogram




(b) 2x Quantized Histogram

Quantizing a value once by $q$ leaves an easily noticeable artifact on the histogram (see Section 3). Each bin $b_i$ where $i \neq nq$ has its values moved to the nearest multiple of q $b_{nq}$. To put it another way, $b_{nq}$ is equal to the sum of the surrounding bins, and all other bins are zero:

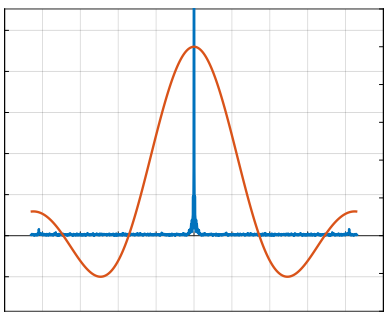
$$b\_quantized(i) = \begin{cases} \sum_{j=-q/2}^{q/2} b(nq+j) & i = nq \; \forall \; n \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

This is equivalent to convolving with a box filter of width $q$, and then sampling with a Dirac Comb of period $q$:

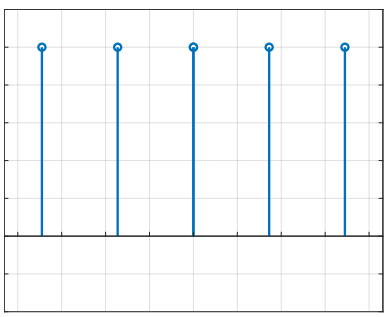
$$b\_filtered(i) = \sum_{j=-q/2}^{q/2} b(i+j) \tag{3.2}$$

$$\begin{aligned} b\_quantized(i) &= b\_filtered(i) * \sum_{n=-\infty}^{\infty} \delta(i-nq) \\ &= \begin{cases} \sum_{j=-q/2}^{q/2} b(nq+j) & i = nq \; \forall \; n \in \mathbb{Z} \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.3}$$

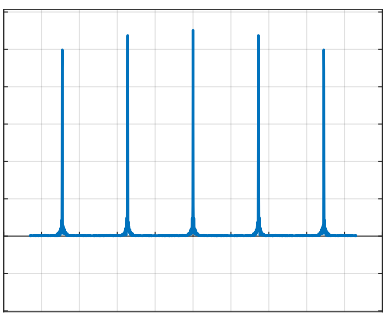
This maps intuitively to the frequency domain: a single quantization step is equivalent to multiplying $B(i)$ by a $\mathcal{F}\{rect(t/q)\} = q * sinc(fq)$ function, then convolving with a Dirac Comb of period $1/q$. Crucially, the FFT of a previously-quantized histogram will have multiple peaks, even if the initial histogram only has one (as is typical for the raw images provided in the assignment).



(a) $B(i)$ and $q * sinc(fq)$



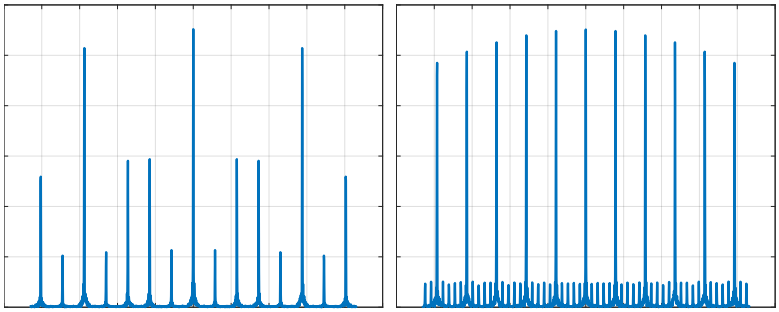
(b) Dirac Comb ($f = 1/q$)



(c) 1x Quantized Result

Figure 2: Frequency-space quantization steps ($q = 5$)

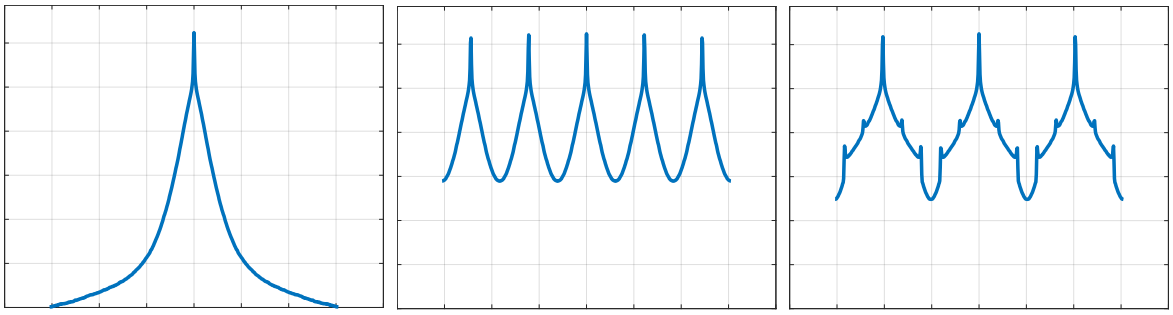
Applying another quantization to the new histogram can further distort the FFT, adding multiple levels of peaks unique to a doubly-quantized histogram. However, if the second value $q_2$ is a multiple of $q_1$ (or vice versa), the quantization in histogram space is equivalent to a single quantization by $min(q_1, q_2)$, and the FFT will be indistinguishable from a singly-quantized one. In all other cases the second quantization will introduce extra peaks of varying heights, where the peaks are at intervals of $1/(q_1 q_2)$ and the relative heights of peaks change depending on if $q_2 > q_1$. Notably, the second-tallest peak is likely an image of the centre peak resulting from the **second** round of sampling[3], which can be exploited for determining compression order.



(a) $q_1 = 5, q_2 = 3$ (b) $q_1 = 5, q_2 = 11$

Figure 3: Examples of frequency-space 2x quantization artifacts

The above figures were generated from the histogram of the DC component of the DCT - i.e. the very first element. For the raw images provided on the assignment webpage, this has a very tight frequency distribution, but the AC components can have much wider frequency distributions and thus when sampled further the images merge together. However, the crucial element is the same: the extra peaks introduced are of significantly different heights to the originals. [6] use this as the basis for detecting aligned-block recompression, using the standard-deviation of the peak heights as measured in the power spectrum (which for our purposes is equivalent to the DFT squared). Another important element to note is that there are more transformations within compression and decompression, e.g. multiple levels of rounding, so real-world data will not look as clean as the examples thus far.



(a) 0x, $D\{1, 2\}\ B(i)$ (b) 1x, $q = 5$ (c) 2x, $q_1 = 5, q_2 = 3$

Figure 4: AC frequency-space quantization

The final algorithm I use analyses the peaks in the DC component, attempts to differentiate between the 0x, 1x, 2x scenarios, and use that information to determine the quantization ratios (and thus the initial compression quality).

[3]Peak images from the first round of sampling are multiplied by a $sinc$ before re-sampling, and will be smaller

### 3.1 Identifying Peaks

[6] uses a simple search window to detect peaks. For element $B(i)$ to be considered a peak, it must be larger than a threshold value, and must not be the smallest value in the window over $B(i-n)$ to $B(i+n)$.

$$B(i) > \frac{max(B)}{\alpha} \tag{3.4}$$

$$B(i) > min(B(j)\ \forall\ j \in \{i-n..i+n\}) + \delta \tag{3.5}$$

$$\alpha = 5, \quad n = 1, \quad \delta = 0$$

Peak constraints for [6]

$n$, $\delta$, and $\alpha$ are empirically chosen. I take a slightly different approach, but retain the overall threshold and empirically set parameters.

```
[ps, locs] = findpeaks(y, ...
    'MinPeakHeight', max(y)/15, ...
    'MinPeakDistance', max(length(y)/100, 1));
```

The above MATLAB code uses the `findpeaks` function from the Signal Processing Toolbox to do the grunt work of finding peaks, and specifies two conditions the peaks must meet. Like [6], a minimum peak height is set based on the maximum value in the FFT. Because the FFT data (especially for the DC components) can be noisy, especially near the tails of large peaks, a minimum peak distance is applied based on the size of the data. MATLAB enforces this constraint by removing all peaks within a fixed distance of a larger peak.

$$B(i) > \frac{max(B)}{\alpha} \tag{3.6}$$

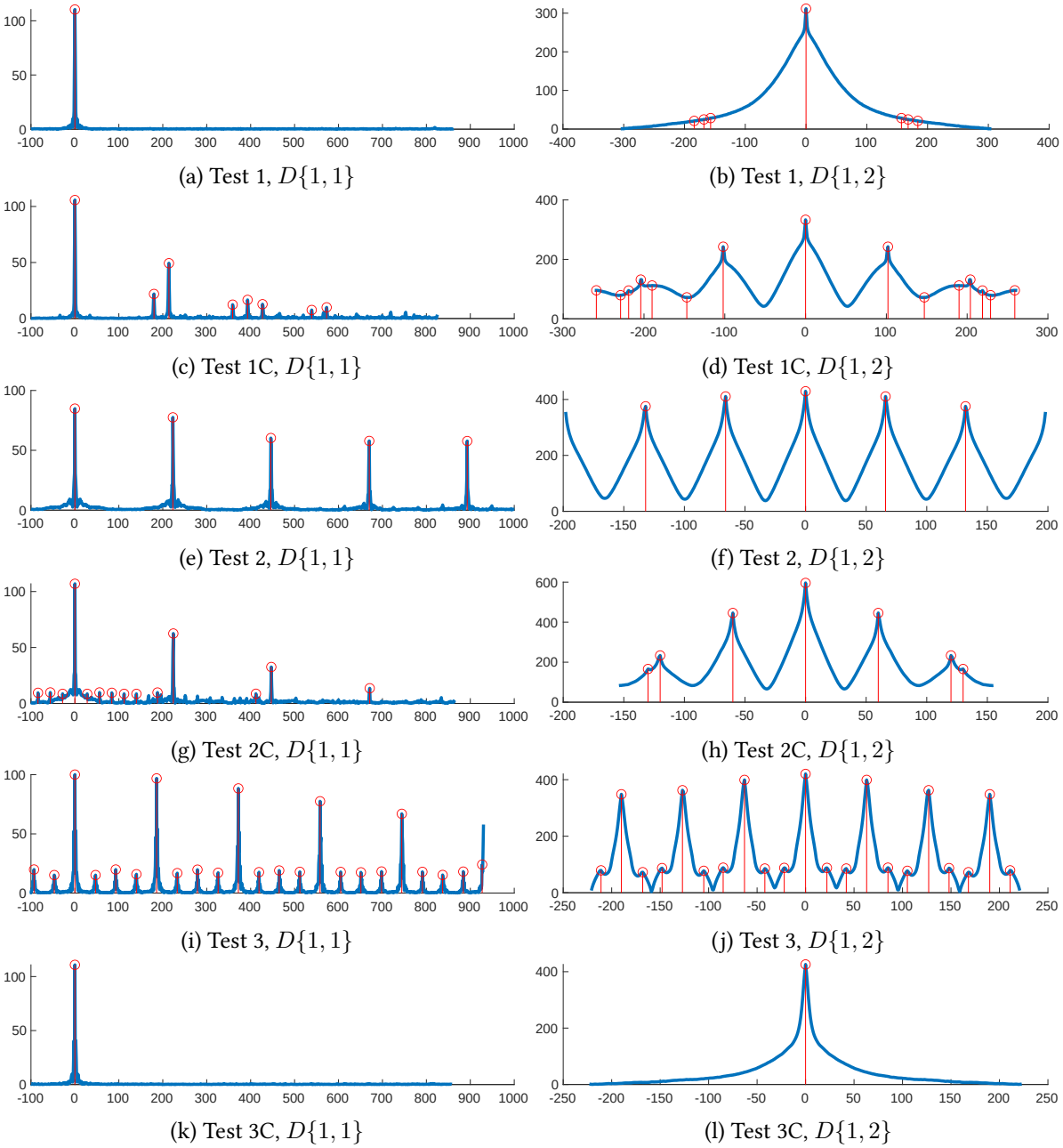
$$B(i) = max(B(j)\ \forall\ j \in \{i-n..i+n\}) \tag{3.7}$$

$$\alpha = 15, \quad n = max\left(\frac{length(B)}{100}, 1\right)$$

Peak constraints for my classifier

This identifies intuitively correct peaks on the DC components of the provided images. In the case of test 2C, there are some smaller peaks around $x = 300$ that aren't caught, but enough minor peaks are caught for the detection to work correctly. Some image's AC components produce extra peaks on seemingly smooth areas due to noise (e.g. tests 1 and 1C), demonstrating why the DC component is better for detection[4].

[4] Applying a filter beforehand could help with this, but may then also prevent e.g. the low peaks in Fig. 5j from being detected.



(a) Test 1, $D\{1,1\}$

(b) Test 1, $D\{1,2\}$

(c) Test 1C, $D\{1,1\}$

(d) Test 1C, $D\{1,2\}$

(e) Test 2, $D\{1,1\}$

(f) Test 2, $D\{1,2\}$

(g) Test 2C, $D\{1,1\}$

(h) Test 2C, $D\{1,2\}$

(i) Test 3, $D\{1,1\}$

(j) Test 3, $D\{1,2\}$

(k) Test 3C, $D\{1,1\}$

(l) Test 3C, $D\{1,2\}$

Figure 5: Peak Identification Results

### 3.2 Identifying Quantization Levels

Once peaks have been found, they can be used to classify the compression level. The easiest one to identify is 0x compression, as in these cases there should be no sampling and only one peak (e.g. Test 1 and Test 3C). If there are multiple peaks, the image has been sampled at least once. A singly-compressed image will have evenly-spaced peaks, which slowly decrease in amplitude as they move further from the centre (e.g. Test 2). The remaining cases have peaks with periodic jumps in amplitude (Tests 1C, 2C, 3), which correspond to 2x quantization.

The first stage of the algorithm decides between 0x, 1x, 2x compression. If one peak is present 0x compression is assumed, otherwise the difference in heights between consecutive peaks is calculated. If these differences in heights are large and positive ("large" here defined as a percentage of the tallest peak) then 2x compression is assumed, otherwise 1x compression.

```
if length(ps) <= 1
    % One peak = the centre
    % no other peaks => no compression
    qs_h_space_est = [];
else
    % Assume odd amount of peaks (centre + even amounts on both sides)
    centre_peak = ceil(length(ps)/2);
    % Remove peaks to the left-of-centre
    ps_right = ps(centre_peak:end);
    centred_locs_right = centred_locs(centre_peak:end);

    % First scan - if we're just 1x compressed, won't have major increases in height.
    % peak_spacing(i) = location(i+1) - location(i)
    % height_diffs(i) = ps(i+1) - ps(i)
    peak_spacing = diff(centred_locs_right);
    height_diffs = diff(ps_right);
    if all(height_diffs < max(y)/5)
        % 1x compressed
        qs_h_space_est = [..];
    else
        % 2x compressed
        qs_h_space_est = [.., ..];
    end
end
```

If the image was compressed, the intervals between peaks can be used to find the quantization values. In the 1x compression case, the frequency-space Dirac comb interval used for sampling can be found by measuring the distance between adjacent peaks, and converted into "histogram space" with a division.

```
qs_h_space_est = [length(h)./peak_spacing(1)];
```

In the 2x compression case, I rely on two properties noted at the start of Section 3:

- The second-tallest peak likely originates from the second round of sampling
- Peaks should be evenly distributed with interval $(1/q_1 q_2)$

$q_2$ is estimated by finding the second-highest peak, measuring the distance to the centre for the frequency-space interval, and translating to histogram-space.

```
% Find the second-highest peak - this corresponds to the
% *second* compression interval.
% Exclude the first peak, it's the centre peak so will be the highest.
[~, i_second_highest] = max(ps_right(2:end));
i_second_highest = i_second_highest + 1;
q2_f_space = centred_locs_right(i_second_highest);
q2_h_space = length(h)./q2_f_space;
```

The minimum distance between peaks is assumed to be $1/q_1q_2$, and with this relationship and the knowledge of $q_2$ we can find $q_1$.

```
% The minor peak spacings are prop to 1/(q1_hist*q2_hist) =
% q1_fft * q2_fft
q1_q2_combined_q = min(peak_spacing);
% q2_fft / (q1_fft * q2_fft) = 1/q1_fft = q1_histogram
q1_h_space = q2_f_space / q1_q2_combined_q;

qs_h_space_est = [q1_real_space q2_real_space];
```

This algorithm produces expected results on the test data. The number of compressions is correct, and the quality values match the histogram data. Interestingly, 2C seems to have been quantized twice with the *same* $q$. If the quantizations were performed directly after one another, this would not be distinguishable from a single quantization. However, the rounding and IDCT/DCT steps after the first quantization introduce noise which is picked up in the second quantization. This doesn't have a large impact, as shown by how small the minor peaks are in the Test 2C histogram, but it is noticeable. Therefore, it's likely Test 2C was compressed twice with very similar quality levels. These quality levels are estimated in the next subsection.

### 3.3 Identifying Compression Quality

From the quantization levels, the compression quality can be estimated. JPEG encoders include default quantization tables, which specify the quantization levels for each DCT coefficient. These quantizations are scaled based on the compression quality, potentially in an implementation-defined manner. As noted in Section 1.2 the test images were compressed with the Independent JPEG Group's `cjpeg` utility, the source code for which is available on GitHub[5]. Compression parameters are calculated in `jcparam.c`, which specifies the luminance quantization table, a function for converting a 0-100 quality metric to a "percentage scaling factor" (`jpeg_quality_scaling()`), and a function that applies the scaling to quantization values (`jpeg_add_quant_table()`. Overall, the calculation for the luminance DC component is as shown below:

quality = value from 1 to 100
**if** quality $< 50$ **then**
    scaling_factor $= 5000/$quality
**else**
    scaling_factor $= 200 - ($quality $* 2)$
**end if**
base_quant = 16
quant = (base_quant $*$ scaling_factor $+ 50)/100$

[5] https://github.com/libjpeg-turbo/ijg

| | $q_1, q_2$ |
|---|---|
| Test 1 | - |
| Test 1C | 6, 8 |
| Test 2 | 9 |
| Test 2C | 8, 8 |
| Test 3 | 4, 10 |
| Test 3C | - |

Table 2: Automated quantization detection results

Because this uses integer arithmetic, which automatically rounds towards zero, there are a range of quality values which would result in a specific quantization value. To try and capture this range, for each estimated quantization $q$ the process is reversed for both $q$ and $q+1$, and both quality values are displayed. This range could be narrowed down further using terms from AC components, but I didn't have time to implement this. The final results are shown in Table 3.

```
function qual=estimate_qual_from_q(q)
    % cjpeg can clamp the quantization at 255 or 32767.
    % If q was clamped, we can't reverse the computation.
    if q == 255 || q == 32767
        warning("q-value %d may have been clamped", q)
    end
    original_q = 16;
    scaling_factor = (q * 100 - 50)/original_q;
    % for quality in [1, 50], scaling factor in [5000, 100]
    % for quality in [50, 100], scaling factor in [100, 0]
    if scaling_factor <= 100
        % Invert the bottom branch - scale = 200 - quality*2
        qual = (200 - scaling_factor)/2;
    else
        % Invert the top branch - scaling_factor = 5000 / quality
        qual = 5000 / scaling_factor;
    end
    qual = round(qual);
end
```

## 4 Testing against generated data

To measure the accuracy of my classifier, I replicated the image creation process from Section 1.2 on the raw images provided on the assignment webpage. Because the raw images are captured from the same camera, and the processing pipelines are identical, these "training" images are suitable for gauging performance on the test data, rather than making the predictor more generalized.

I wrote a Python script that randomly generated a 0x, 1x, or 2x compressed test image for each raw image (in color and grayscale) and wrote the compression levels out to a CSV. The compression qualities were the same for each permutation (see Table 4), and the script also downscaled the images slightly to improve classification speed. A MATLAB script then parsed the CSV and ran the predictor on each image, printing whether it was correct or not. Overall, 47 out of 48 images were classified correctly.

| | Quantization | Quality |
|---|---|---|
| Test 1 | N/A | N/A |
| Test 1C | 6, 8 | 80-83, 73-77 |
| Test 2 | 9 | 70-73 |
| Test 2C | 8, 8 | 73-77, 73-77 |
| Test 3 | 4, 10 | 86-89, 67-70 |
| Test 3C | N/A | N/A |

Table 3: Final Results

| | | Quality | Quantization |
|---|---|---|---|
| Grayscale | 1x | 75 | 8 |
| | 2x | 80, 70 | 6, 10 |
| Color | 1x | 90 | 3 |
| | 2x | 90, 80 | 3, 6 |

Table 4: Compression qualities for different permutations

### 4.1 Results on Grayscale

All 24 grayscale images were correctly classified as 0x, 1x, and 2x. All 1x compression images correctly predicted the quality range $73 - 77$, and all 2x compression images correctly predicted the *second* quality range as $67 - 70$. However, 8 out of 11 of the 2x grayscale images incorrectly predicted the first quality range as $89 - 92$. An example is Fig. 6b, which detected a $q_2/q_1$ ratio of 3, but should have predicted 6.

The prediction for Fig. 6a relies on the extra peaks around $x = 595$, which aren't present in Fig. 6b. The loss of these minor peaks may be due to negative interference between signal images in the second compression: the singly-compressed signal is multiplied by $sinc$, which is negative in some regions, and peaks in those regions could destructively interfere with other, positive, peaks. One way to avoid this issue could be to sample the AC coefficients as well, and hope they don't all experience the same problem.

### 4.2 Results on Color

23 out of 24 color images were correctly classified as 0x, 1x, and 2x. One image was misclassified as 1x when it was actually 2x, e.g. Fig. 6c, which was due to the peak threshold being too high. It has a tiny peak just before $x = 300$, which was large enough in other images (e.g. Fig. 6d), but wasn't picked up here.

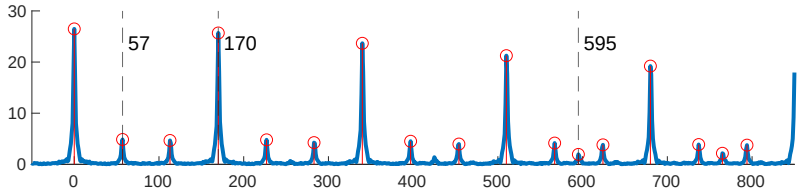
All of the correctly-classified 1x images correctly predicted the quality range $89 - 92$. However, all of the 2x images incorrectly predicted both quality ranges. This is in part due to an oversight when deciding the quality levels - the second quantization level is a multiple of the first. The only reason 2x compression is detected at all is because of noise added between quantizations. It's likely also another example of negative interference. Because $q_2$ is incorrectly predicted, it stands to reason there *should* be a different second-largest peak, which must have been reduced by destructive interference.

### 4.3 Conclusion

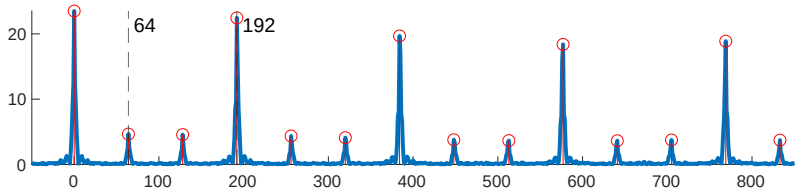
My implementation has proven itself to be very accurate at classifying the number of compressions. However in some cases, particularly those of 2x compressed images, it does not predict compression quality well, possibly due to destructive interference between sampling images. It could be further improved by sampling AC coefficients and combining these results with the DC coefficients, as [6] does. Additionally, the techniques in [3] could be used to extract more accurate initial DCT values, or allow sampling of chrominance information.

---

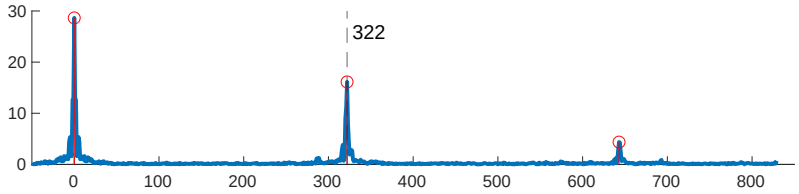
**Word Count Ends Here**



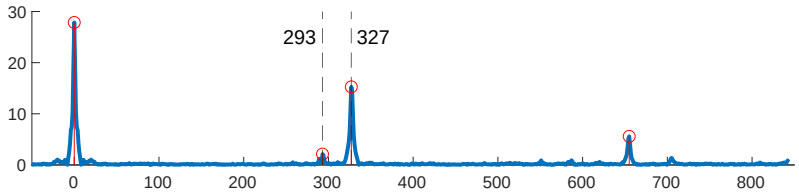

(a) 2x Grayscale, Correct $q = 6, 10$




(b) 2x Grayscale, Incorrect Quantization $q = 3, 10$




(c) 2x Color, Misclassified as 1x




(d) 2x Color, Incorrect Quantization $q = 10, 5$

Figure 6: FFTs for incorrectly predicted images

## References

[1] Zhigang Fan and R.L. de Queiroz. "Identification of Bitmap Compression History: JPEG Detection and Quantizer Estimation". In: *IEEE Transactions on Image Processing* 12.2 (February 2003), pp. 230–235. ISSN: 1941-0042. DOI: 10/bjrkbd.

[2] R. Neelamani et al. "JPEG Compression History Estimation for Color Images". In: *IEEE Transactions on Image Processing* 15.6 (June 2006), pp. 1365–1378. ISSN: 1941-0042. DOI: 10/bnm5nv.

[3] Andrew B. Lewis and Markus G. Kuhn. "Exact JPEG Recompression". In: *Visual Information Processing and Communication*. Vol. 7543. SPIE, 18th January 2010, pp. 256–264. DOI: 10/bpwf69.

[4] Xiaoying Feng and Gwenaël Doërr. "JPEG Recompression Detection". In: *Media Forensics and Security II*. Vol. 7541. SPIE, 27th January 2010, pp. 188–199. DOI: 10/fvhsck.

[5] Ali Taimori et al. "A Part-Level Learning Strategy for JPEG Image Recompression Detection". In: *Multimedia Tools and Applications* 80.8 (1st March 2021), pp. 12235–12247. ISSN: 1573-7721. DOI: 10/gk9j9m.

[6] Yi-Lei Chen and Chiou-Ting Hsu. "Detecting Recompression of JPEG Images via Periodicity Analysis of Compression Artifacts for Tampering Detection". In: *IEEE Transactions on Information Forensics and Security* 6.2 (June 2011), pp. 396–406. ISSN: 1556-6021. DOI: 10/ckj4hm.

[7] Alin C. Popescu and Hany Farid. "Statistical Tools for Digital Forensics". In: *Information Hiding*. Ed. by Jessica Fridrich. Red. by David Hutchison et al. Vol. 3200. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 128–147. ISBN: 978-3-540-24207-9 978-3-540-30114-1. DOI: 10.1007/978-3-540-30114-1_10.