CS 663: Fundamentals of Digital Image Processing
Project: Image Compression

Deeksha Dhiwakar (22B0988)
Sharvanee Sonawane (22B0943)

November 2024

Image Compression Engine using JPEG algorithm on grayscale images

The JPEG algorithm is used for compressing grayscale images as follows:

JPEG Encoder -

1. Subtract 128 (midpoint of the range 0 to 255) from every pixel value in the image so the range of intensity values changes from 0 to 255, to -128 to 127.
2. Pad modified array with zeros along rows and columns if required to ensure that the image dimensions are multiples of 8.
3. Divide the padded image into non-overlapping blocks of size 8×8 pixels.
4. Compute the 2D Discrete Cosine Transform (DCT) of each block. This produces a set of 64 "DCT coefficients" per block. The equation for the 2D DCT is

$$G(m,n) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} g(x,y) \cdot \cos\left(\pi \frac{m(2x+1)}{2M}\right) \cdot \cos\left(\pi \frac{m(2y+1)}{2N}\right)$$

   For a 2D DCT, we first compute the DCT along one dimension (rows: axis = 0), then apply it along the other dimension (columns, axis = 1) followed by orthogonal normalization ('norm='ortho'') to ensure perfect reconstruction during inverse DCT. The `dct` function in `SciPy` library is used to implement this.
5. Quantize these DCT coefficients (divide by an appropriate number and round off to the nearest integer) using a quantization matrix that depends on the quality factor Q. This causes many coefficients (higher frequency components less noticeable to the human eye) to become 0, hence, they need not be stored.
6. Transform each block of size 8×8 of the quantized DCT coefficients into a sequence using zigzag ordering.
7. Convert the ordered coefficients into a Run-length encoded vector, by replacing a subsequence of consecutive zeros with a zero followed by the count of zeros in the consecutive-zero subsequence.
8. Generate a Huffman dictionary for the symbols occurring in the Run-length encoded vector according to their frequencies. Use this dictionary to perform a lossless compression and return the final Huffman encoded stream.
9. The compressed file stores the Huffman-encoded stream, the Huffman dictionary which was used for encoding, and the dimensions of the original image into a `.bin` file (binary format). This is done using Python's `pickle` module, which serializes the data into a binary format.

JPEG Decoder -

1. The compressed data is read from the `.bin` file, which contains the Huffman-encoded bitstream, the Huffman dictionary, and the dimensions of the original image.
2. The Huffman-encoded stream is decoded using the Huffman dictionary to retrieve the Run-length encoded vector.
3. The Run-length encoded vector is decoded by expanding the zero-subsequences to recover the zigzag-ordered quantized DCT coefficients.
4. The zigzag-ordered DCT coefficients are reshaped back into 8×8 blocks.
5. The DCT coefficients are de-quantized (multiplying the coefficients point-wise with the entries in the quantization matrix) to restore the coefficients to their original values.
6. The 2D Inverse Discrete Cosine Transform (IDCT) is applied to each block to recover the image blocks in the spatial domain. For a 2D IDCT, we first compute the IDCT along one dimension (rows: axis = 0), then apply it along the other dimension (columns, axis = 1) followed by orthogonal normalization ('norm='ortho'') to ensure perfect reconstruction. The `dct` function in `SciPy` library is used to implement this.
7. The blocks are recombined to form the full image, and any padding that was added during the encoding process is removed to restore the original dimensions of the image.
8. Finally, the 128 offset (added during encoding) is reversed by adding 128 to each pixel value, transforming the pixel values back to the original range of 0 to 255.

Quantization matrix

The JPEG quantization matrix for quality factor = 50 is defined as:

$$M = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

The quality factor Q (ranging from 1 to 100) determines the scaling factor S as follows:

$$S = \begin{cases} \frac{5000}{Q}, & \text{if } Q < 50 \\ 200 - 2 \cdot Q, & \text{if } Q \geq 50 \end{cases}$$

The quantization matrix M_Q for quality factor Q is calculated using:

$$M_Q = \left\lfloor \frac{S \cdot M + 50}{100} \right\rfloor$$

Any element of M_Q that is zero is replaced with 1 to avoid division-by-zero errors during compression.

Testing on Dataset

We evaluated the performance of the JPEG compression algorithm for grayscale images on the `101_ObjectCategories` dataset which can be accessed at the following URL: `https://data.caltech.edu/records/mzrjq-6wc02`. This dataset consists of 10 categories of images, and we specifically selected the 10^{th} and 20^{th} images from each category for testing. The compression is performed at multiple quality factors, and the resulting images are compared based on the following two metrics:

1. **Relative Root Mean Squared Error (RMSE):** The RMSE measures the difference between the original image and its compressed version. It is calculated as the square root of the mean squared differences between the pixel values of the original and decompressed images, normalized by the average pixel value of the original image. N is the total number of pixels in the original image. The formula for RMSE is:

$$\text{RMSE} = \frac{\sqrt{\sum_{i,j}(I_{original}(i,j) - I_{decompressed}(i,j))^2}}{\sqrt{N \sum_{i,j} I^2_{original}(i,j)}}$$

2. **Bits Per Pixel (BPP):** The BPP metric indicates the amount of compression applied to an image. It is calculated by dividing the size of the compressed image (in bits) by the total number of pixels in the image. The formula for BPP is:

$$\text{BPP} = \frac{\text{Size of Compressed Image in Bits}}{\text{Number of Pixels in Image}}$$

For each selected image (image 10 and image 20 from each category), JPEG compression is applied with varying quality factors (Q) starting from 36 and ending at 74 in step 2. The quality factor Q controls the degree of compression, where higher values of Q result in less compression (higher quality) and lower values result in more compression (lower quality).
The compressed images are then decompressed, and their RMSE and BPP values are computed for each quality factor. For each selected image, the RMSE and BPP values are plotted for different quality factors. The graph of RMSE versus BPP illustrates the trade-off between image quality (as represented by RMSE) and compression ratio (represented by BPP).

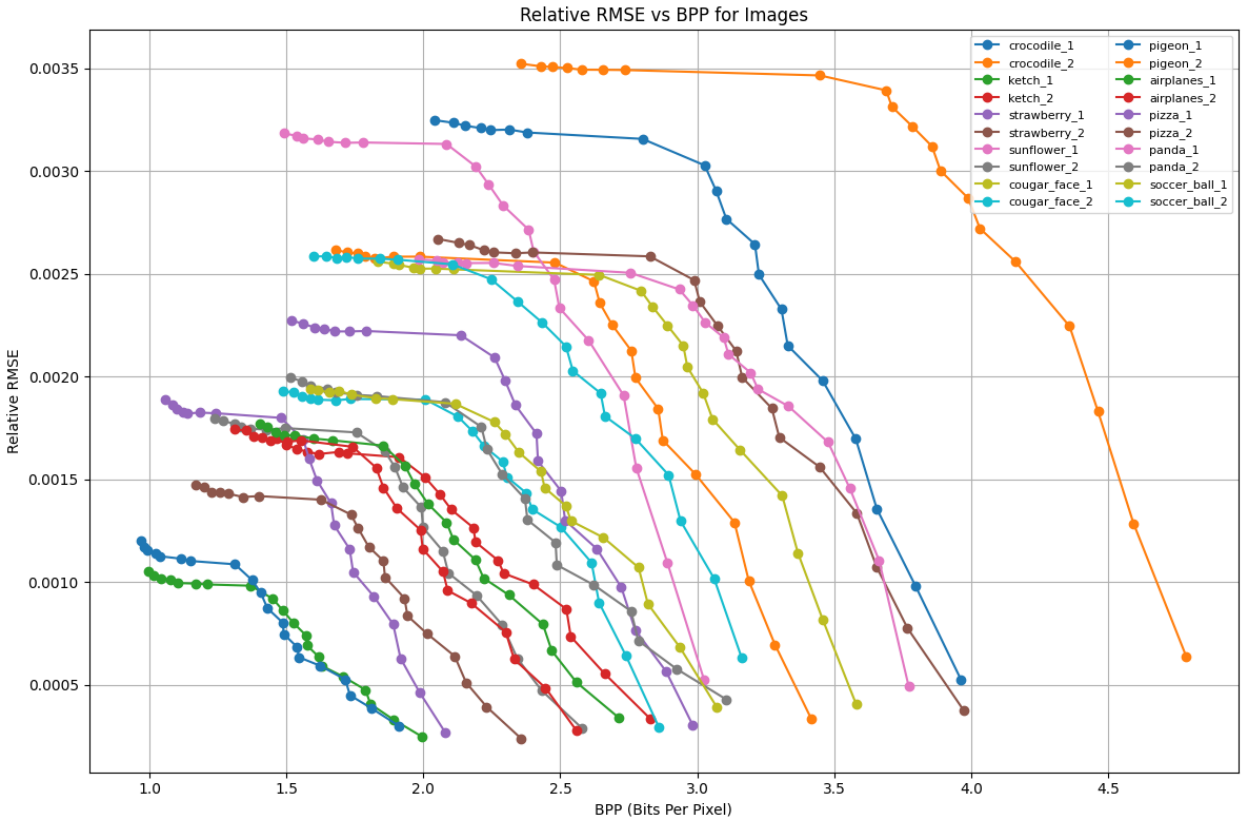


Figure 1: RMSE vs BPP for 20 different grayscale images

Results and Conclusion

As the quality factor Q decreases (i.e., higher compression), we achieve a lower BPP and the RMSE increases due to the loss of image details. The graph illustrates that while compression techniques may vary in their efficiency based on the content of the image, the overall trend holds. The larger the quality factor, the better the preservation of the image quality with a higher BPP. The smaller the quality factor, the more significant the loss in image quality (higher RMSE), but with a corresponding decrease in BPP.

Analysis of the Algorithm

- The implementation of JPEG compression provides a significant reduction in storage size, as evidenced by the BPP metric.
- The use of the quality factor Q allows control over the trade-off between compression and image quality. The ability to adjust Q provides flexibility for different use cases (e.g., needing higher quality for some professional applications or opting for higher compression in cases where quality can be compromised).
- At lower values of Q, the RMSE increases significantly, leading to visible artefacts in the image.
- The standard JPEG algorithm uses the same quantization matrix for all images irrespective of their content, which may not be optimal for all image types.
- JPEG compression can result in blocky artefacts at lower quality factors due to the block-based DCT (Discrete Cosine Transform) and quantization.

Extending the Grayscale JPEG Compression Algorithm to Color Images

Algorithm

The grayscale JPEG compression algorithm processes a single-channel image, where each pixel contains only one intensity value. To adapt the grayscale algorithm to compress colour images such as RGB images which contain three colour channels: Red (R), Green (G), and Blue (B), we need to apply the same steps to each of the individual colour channels (R, G, and B) while accounting for their interactions.

1. The colour image is converted from the RGB colour space to the YCbCr colour space. The YCbCr color space separates the luminance information (Y channel) from the chrominance information (Cb and Cr channels). The YCbCr channels are defined as:

$$Y = 0.299R + 0.587G + 0.114B \tag{1}$$

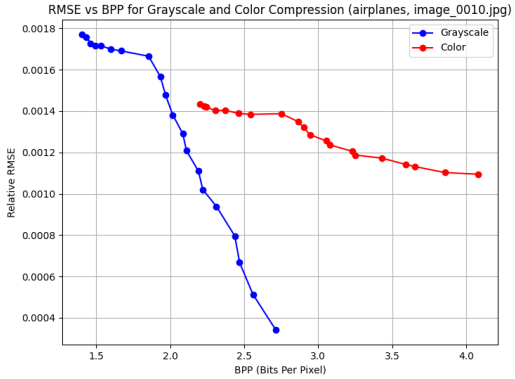
$$Cb = -0.1687R - 0.3313G + 0.5B \tag{2}$$

$$Cr = 0.5R - 0.4187G - 0.0813B \tag{3}$$

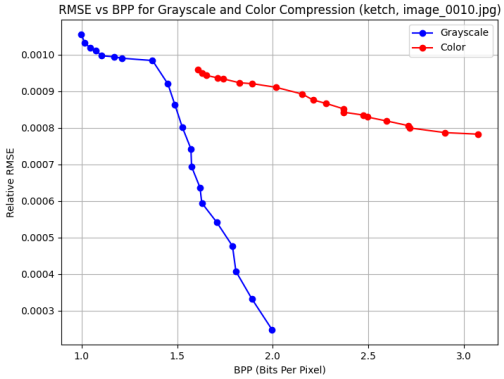
2. Once the image is in the YCbCr color space, we apply the grayscale JPEG compression algorithm to each channel separately. This means that each channel will have its own compressed bitstream. The final compressed image will contain three separate compressed streams: one for the Y channel, one for the Cb channel, and one for the Cr channel. The Cb and Cr channels are down-sampled by a factor of 2 in both the horizontal (X) and vertical (Y) directions. The Y channel (luminance) is not down-sampled. This is because the human eye is much more sensitive to luminance than to chrominance information.
3. The decompression process is also applied to each channel separately.
4. The image is reconstructed by combining the decompressed Y, Cb, and Cr channels and converting them back to the RGB colour space.

Results and Conclusion

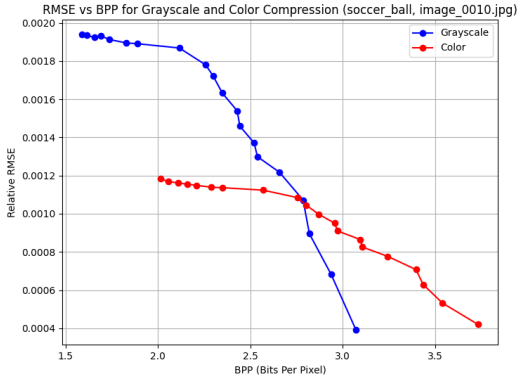
The Root Mean Square Error (RMSE) vs. Bits Per Pixel (BPP) curve for grayscale images is observed to lie below that of colour images.



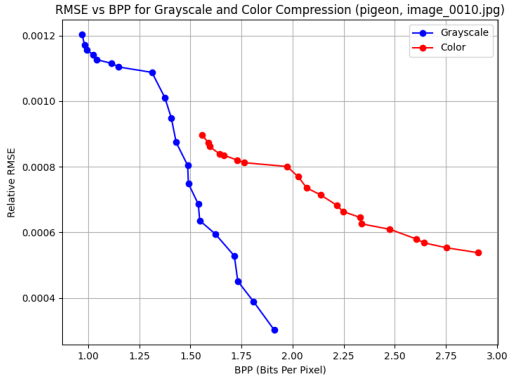
(a) RMSE vs BPP for aeroplane image



(b) RMSE vs BPP for ketch image



(a) RMSE vs BPP for soccer-ball image



(b) RMSE vs BPP for pigeon image

Figure 3: Comparison of Grayscale and Colour Compression

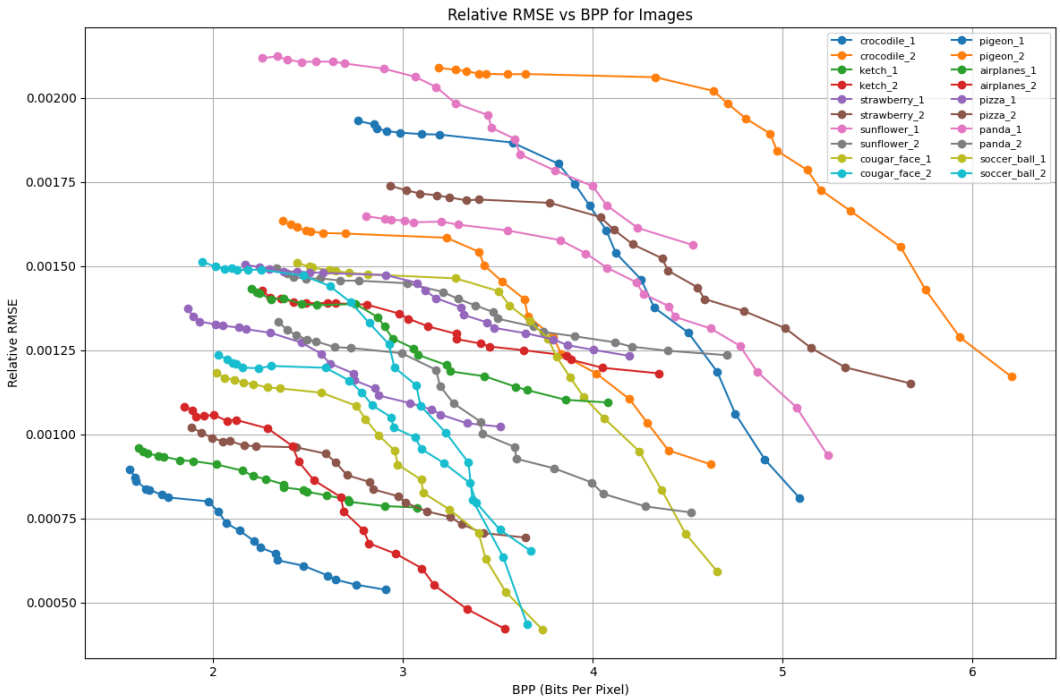


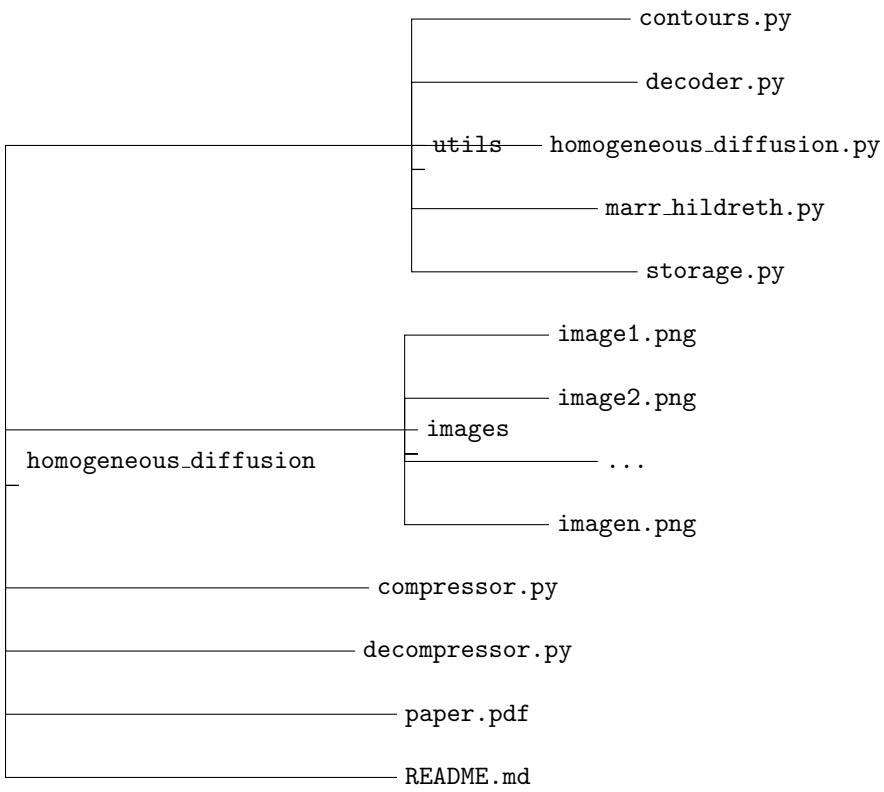
Figure 4: RMSE vs BPP for 20 different colour images

Grayscale images contain only one channel of luminance information, unlike color images which typically have three channels (e.g., Y, Cb, and Cr). So more efficient encoding is achieved in grayscale images due to lower spatial redundancy. Since the quantization is individually applied on each of the 3 channels in colour images, there is a higher loss of information leading to higher RMSE due to quantization errors. As a result, grayscale images usually achieve lower RMSE values at the same BPP compared to colour images. For the same quality factor, grayscale images achieve a higher compression.

Implementation of Research Paper

The paper we have chosen to implement is **Using partial differential equations (PDEs) for image compression: M. Mainberger and J.Weickert, "Edge-Based Image Compression with Homogeneous Diffusion", CAIP 2009**.

Directory Structure



The code is in the folder `homogeneous_diffusion.zip`. Unzip the folder to access the files. The folder `images` contains all the images for testing. `README.md` contains instructions to run the code which are also mentioned below.

Instructions to run the code

Compressing an image

1. Place the test image, say `image{image_number}.png` in the `images` folder.
2. Run the following command from inside the `homogeneous_diffusion` directory:
   `python3 compressor.py <image_number> <q> <d>`
   Here `q` is the quantization parameter that determines to how many levels the intensities will be quantized, and `d` is the subsampling parameter that determines how frequently edge pixels are sampled during compression.
3. The compressed file will be saved in `compressed/out{image_number}.bin`

Decompressing an image

1. Run the following command from inside the `homogeneous_diffusion` directory:
   `python3 decompressor.py <compressed file path>`
   From the previous compression, `compressed file path` will be `compressed/out{image_number}.bin`
2. The reconstructed image will be saved in `reconstructed_images/reconstructed_image{image_number}.png`

Explanation of the different files

To perform efficient image compression, `compressor.py` runs the following files:

marr_hildreth.py

This file is used to implement edge detection using the Marr-Hildreth algorithm. It takes as arguments the input and output file paths as well as optional parameters denoting the Gaussian standard deviation and the edge detection threshold. It stores the detected edges in bi-level format using the JBIG kit.

contours.py

This file is used to encode the contour pixel values. It takes as arguments the paths to the original image and edge map, and the quantization and subsampling parameters. It extracts the pixels along the edges, quantizes them to 2^q values, and subsamples by taking every d^{th} pixel, and stores the subsampled and quantized pixels in PAQ format.

storage.py

It is used to create the final compressed file. It takes as arguments the paths to the JBIG and PAQ files created, along with the parameters. It saves the final compressed file according to the format specified in the paper.

decoder.py

It is used to recover the subsampled pixel and edge information from the compressed file. It takes as argument the path to the compressed file and creates JBIG and PAQ files and extracts the metadata.

homogeneous_diffusion.py

It is used to reconstruct the image from the edges and contour pixels, whose paths are given as argument. It performs linear interpolation to recover the pixels along the edges and uses homogeneous diffusion to fill in the rest of the image using the Laplace equation, as described in the paper and finally saves the reconstructed image.

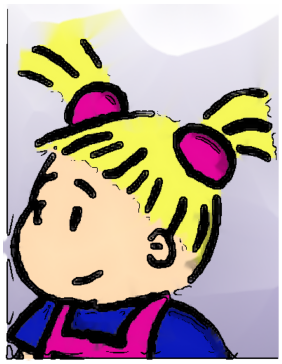
Experiments and Results

Following are the results of running the algorithm on the cartoon images provided in the paper.

(a) Original

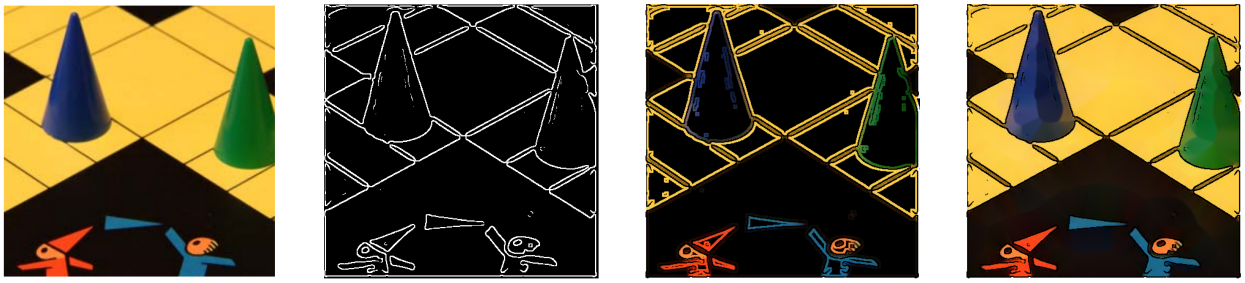
(b) Edges

(c) Contours



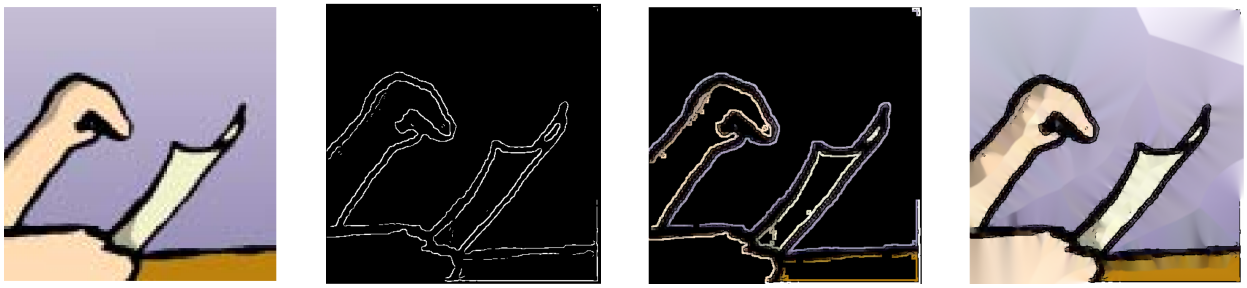
(d) Reconstructed

(a) Original (b) Edges (c) Contours (d) Reconstructed



(a) Original (b) Edges (c) Contours (d) Reconstructed

(a) Original (b) Edges (c) Contours (d) Reconstructed



(a) Original (b) Edges (c) Contours (d) Reconstructed

For each of the above images, we obtain a compression of atleast 50% for edge detection thresholds varying between 0.0005 and 0.001, q value 4 and d value 3. These values can be adjusted depending on the image to get better reconstruction and compression ratios, making it better than the JPEG and JPEG2000 algorithms, as discussed in the paper.