Introduction

The project compares the performance of RGB and YUV color based VQ compression on test images using codebooks generated from training data. It reports compression ratios and quality (PSNR), helping evaluate which color space is better suited for compression.

Code

```
public class Main {
    public static void main(String[] args) throws Exception {
        String trainingDir = "training";
        String testDir     = "test";

        System.out.println("=== Generating RGB codebooks ===");
        List<double[]> redCB   = CodebookGenerator.generate(trainingDir,  comp: 'R');
        List<double[]> greenCB = CodebookGenerator.generate(trainingDir,  comp: 'G');
        List<double[]> blueCB  = CodebookGenerator.generate(trainingDir,  comp: 'B');
        CodebookIO.saveCodebook(redCB, filename: "codebooks/redCB.txt");
        CodebookIO.saveCodebook(greenCB,  filename: "codebooks/greenCB.txt");
        CodebookIO.saveCodebook(blueCB, filename: "codebooks/blueCB.txt");
```

Doing 2 files(training and testing that include animals , faces and natures images
For RGB codebooks we generate codebook for Red , Green and Blue separately with size 256 codebook size for each color and 2x2 blocksize

```
    // Codebook sizes
    int cbY   = 256;   // Y luminance
    int cbUV  = 64;    // U/V chroma subsampled

    System.out.println("=== Generating YUV codebooks ===");
    List<double[]> yCB = CodebookGeneratorYUV.generate(trainingDir,  channel: 'Y', cbY);
    List<double[]> uCB = CodebookGeneratorYUV.generate(trainingDir,  channel: 'U', cbUV);
    List<double[]> vCB = CodebookGeneratorYUV.generate(trainingDir,  channel: 'V', cbUV);
    CodebookIO.saveCodebook(yCB,  filename: "codebooks/yCB.txt");
    CodebookIO.saveCodebook(uCB,  filename: "codebooks/uCB.txt");
    CodebookIO.saveCodebook(vCB,  filename: "codebooks/vCB.txt");
```

We transform RGB to YUV ,Y has codebook 256 while U and V have a codebook of 64

1.RGB Compression

ImageUtilsQV

```java
//Extract R G B from image
public static int[][] extractComponent(BufferedImage img, char component) {  4 usages
    int w = img.getWidth(), h = img.getHeight();
    int[][] comp = new int[h][w];
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            int rgb = img.getRGB(x, y);
            int r = (rgb >> 16) & 0xFF;
            int g = (rgb >> 8) & 0xFF;
            int b = rgb & 0xFF;
            if (component == 'R') comp[y][x] = r;
            else if (component == 'G') comp[y][x] = g;
            else comp[y][x] = b;
        }
    }
    return comp;
}
```

Purpose:

Extracts a specific color component (`'R'`, `'G'`, or `'B'`) from a given image.

How it works:

- Iterates over each pixel of the image.
- Extracts the red, green, and blue channels.
- Returns a 2D array of the selected component values.

```java
//Compress to Codebook
public static int[][] compressComponent(int[][] comp, List<double[]> codebook) {  3 usages
    int h = comp.length, w = comp[0].length;
    int[][] idx = new int[h][w];
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            idx[y][x] = nearest(comp[y][x], codebook);
        }
    }
    return idx;
}
```

```java
public static List<double[]> getBlocks(int[][] comp) {  1 usage
    int blockSize = 2;
    int h = comp.length;
    int w = comp[0].length;
    int cropH = (h / blockSize) * blockSize;
    int cropW = (w / blockSize) * blockSize;
    List<double[]> blocks = new ArrayList<>();

    for (int y = 0; y < cropH; y += blockSize) {
        for (int x = 0; x < cropW; x += blockSize) {
            double[] block = new double[blockSize * blockSize];
            int idx = 0;
            for (int dy = 0; dy < blockSize; dy++) {
                for (int dx = 0; dx < blockSize; dx++) {
                    block[idx++] = comp[y + dy][x + dx];
                }
            }
            blocks.add(block);
        }
    }
    return blocks;
}
```

Purpose:
Divides the image component into 2×2 blocks and flattens each block into a 1D array for use in training or building the codebook.

How it works:

- Crops the image to be divisible by 2 (block size).
- Traverses the image block by block.
- Converts each 2×2 block into a 1D array (length = 4).
- Returns a list of these blocks.

```java
//decompress codebook into RGB again
public static int[][] decompressComponent(int[][] idx, List<double[]> codebook) {
    int h = idx.length, w = idx[0].length;
    int[][] comp = new int[h][w];
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            comp[y][x] = (int) codebook.get(idx[y][x])[0];
        }
    }
    return comp;
}
```

Purpose:
Reconstructs an image component from the compressed codebook indices.

How it works:

- Uses the index array to fetch the original value from the codebook.
- Assumes each codebook vector represents a single pixel (not a full block).

```java
public static BufferedImage mergeComponents(int[][] r, int[][] g, int[][] b) {
    int h = r.length, w = r[0].length;
    BufferedImage img = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            int pixel = (r[y][x] << 16) | (g[y][x] << 8) | b[y][x];
            img.setRGB(x, y, pixel);
        }
    }
    return img;
}
```

Purpose:
Reconstructs a full RGB image from its separate R, G, and B components.

How it works:

- Combines the three channels using bit-shifting and bitwise OR operations.
- Creates a `BufferedImage` with combined RGB values.

```java
//calculate the nearest codebook to the pixel
private static int nearest(int value, List<double[]> codebook) {  1 usage
    double minDist = Double.MAX_VALUE;
    int best = 0;
    for (int i = 0; i < codebook.size(); i++) {
        double d = Math.abs(value - codebook.get(i)[0]);
        if (d < minDist) {
            minDist = d;
            best = i;
        }
    }
    return best;
}
```

Purpose:
Finds the index of the codebook entry closest to the given pixel value.

How it works:

- Computes absolute distance between the pixel value and each codebook entry.
- Returns the index with the smallest distance.

```java
//calculate PNSR
public static double computePSNR(int[][] orig, int[][] recon) {  1 usag
    double mse = 0.0;
    int h = orig.length, w = orig[0].length;
    for (int y = 0; y < h; y++) {
        for (int x = 0; x < w; x++) {
            double diff = orig[y][x] - recon[y][x];
            mse += diff * diff;
        }
    }
    mse /= (h * w);
    return 10 * Math.log10(255 * 255 / mse);
}
```

Purpose:
Computes the Peak Signal to Noise Ratio (PSNR) between the original and reconstructed image components.

How it works:

- Calculates Mean Squared Error (MSE).

Why PSNR?

It is a common metric to evaluate the quality of image compression-> the higher the PSNR, the better the reconstruction quality.

2. YUV Compressor

CodebookGenerator

```
public class CodebookGeneratorYUV {  3 usages

    private static final int BLOCK_SIZE = 2;  8 usages


    public static List<double[]> generate(String trainingDir,char channel,int codebookSize) throws IOException {

        List<double[]> allBlocks = new ArrayList<>();

        File dir = new File(trainingDir);
        if (!dir.isDirectory()) {
            throw new IllegalArgumentException("Not a folder: " + trainingDir);
        }
        // Queue for recursive folder traversal
        Queue<File> queue = new LinkedList<>();
        queue.add(dir);

        while (!queue.isEmpty()) {
            File f = queue.poll();
            if (f.isDirectory()) {
                Collections.addAll(queue, f.listFiles());
            } else {
                // Accept only image files
                String name = f.getName().toLowerCase();
                if (name.endsWith(".jpg") || name.endsWith(".jpeg")
                        || name.endsWith(".png") || name.endsWith(".bmp")) {

                    BufferedImage img = ImageIO.read(f);
                    if (img == null) continue;
```

ImageUtilsYUV

```java
public class ImageUtilsYUV {  13 usages
    private static final int BLOCK_SIZE = 2;  18 usages


     // Convert an RGB image to separate Y,U,V 2D arrays
    public static int[][][] rgbToYuv(BufferedImage image) {  2 usages
        int width = image.getWidth();
        int height = image.getHeight();
        int[][] Y = new int[width][height];
        int[][] U = new int[width][height];
        int[][] V = new int[width][height];

        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                int rgb = image.getRGB(x, y);
                int r = (rgb >> 16) & 0xFF;
                int g = (rgb >> 8)  & 0xFF;
                int b = rgb & 0xFF;
                // standard conversion
                Y[x][y] = (int)(0.299 * r + 0.587 * g + 0.114 * b);
                U[x][y] = (int)(-0.14713 * r - 0.28886 * g + 0.436 * b);
                V[x][y] = (int)( 0.615   * r - 0.51499 * g - 0.10001 * b);
            }
        }
        return new int[][][]{ Y, U, V };
    }
```

Converts a given RGB image to separate Y, U, and V channels.

Y (luminance), U, and V (chrominance) values are calculated using standard formulas.

Returns a 3D array: `int[][][] { Y, U, V }`.

```java
public static int[][] subSample(int[][] channel) {  2 usages
    int width  = channel.length;
    int height = channel[0].length;
    int newW = width  / BLOCK_SIZE;
    int newH = height / BLOCK_SIZE;
    int[][] sub = new int[newW][newH];

    for (int x = 0; x < newW; x++) {
        for (int y = 0; y < newH; y++) {
            int sum = 0;
            // sum over 2×2
            for (int i = 0; i < BLOCK_SIZE; i++) {
                for (int j = 0; j < BLOCK_SIZE; j++) {
                    sum += channel[x*BLOCK_SIZE + i][y*BLOCK_SIZE + j];
                }
            }
            sub[x][y] = sum / (BLOCK_SIZE * BLOCK_SIZE);
        }
    }
    return sub;
}
```

Performs 2×2 block averaging on a channel (Y, U, or V).

Reduces width and height by half by averaging every 2×2 block.

Useful for compressing chrominance channels (U and V) which are less sensitive to human vision.

```java
public static int[][] compressChannel(int[][] channel, List<double[]> codebook, int blockSize) {
    int width  = channel.length;
    int height = channel[0].length;
    int bxCount = width  / blockSize;
    int byCount = height / blockSize;
    int[][] indices = new int[bxCount][byCount];

    for (int bx = 0; bx < bxCount; bx++) {
        for (int by = 0; by < byCount; by++) {
            double[] block = new double[blockSize * blockSize];
            int idx = 0;
            for (int i = 0; i < blockSize; i++) {
                for (int j = 0; j < blockSize; j++) {
                    block[idx++] = channel[bx*blockSize + i][by*blockSize + j];
                }
            }
            // find nearest codebook vector
            int   bestK = 0;
            double bestD = Double.MAX_VALUE;
            for (int k = 0; k < codebook.size(); k++) {
                double dist = 0;
                double[] vec = codebook.get(k);
                for (int d = 0; d < vec.length; d++) {
                    double diff = block[d] - vec[d];
                    dist += diff * diff;
                }
                if (dist < bestD) {
                    bestD = dist;
                    bestK = k;
                }
```

- Splits the input channel into blocks of `blockSize × blockSize`.
- Each block is converted into a vector.
- It finds the closest match (by Euclidean distance) from the codebook.
- Stores the index of the closest vector in a 2D array.

```java
public static int[][] decompressChannel(int[][] indices, List<double[]> codebook, int blockSize) {
    int bxCount = indices.length;
    int byCount = indices[0].length;
    int width  = bxCount * blockSize;
    int height = byCount * blockSize;
    int[][] channel = new int[width][height];

    for (int bx = 0; bx < bxCount; bx++) {
        for (int by = 0; by < byCount; by++) {
            double[] vec = codebook.get(indices[bx][by]);
            int idx = 0;
            for (int i = 0; i < blockSize; i++) {
                for (int j = 0; j < blockSize; j++) {
                    channel[bx*blockSize + i][by*blockSize + j] = (int)vec[idx++];
                }
            }
        }
    }
    return channel;
}
```

- Reconstructs the channel using the stored indices.
- Each index corresponds to a vector in the codebook, which is expanded back into a block.
- Blocks are placed into the final image grid.

```java
public static int[][] upsample(int[][] channel) {  2 usages
    int w = channel.length;
    int h = channel[0].length;
    int[][] full = new int[w*BLOCK_SIZE][h*BLOCK_SIZE];
    for (int x = 0; x < w; x++) {
        for (int y = 0; y < h; y++) {
            int v = channel[x][y];
            full[x*BLOCK_SIZE    ][y*BLOCK_SIZE    ] = v;
            full[x*BLOCK_SIZE + 1][y*BLOCK_SIZE    ] = v;
            full[x*BLOCK_SIZE    ][y*BLOCK_SIZE + 1] = v;
            full[x*BLOCK_SIZE + 1][y*BLOCK_SIZE + 1] = v;
        }
    }
    return full;
}
```

Duplicates each pixel in a 2×2 block to restore the original resolution.

This is the reverse of subsampling.

```java
public static BufferedImage yuvToRgb(int[][] Y, int[][] U, int[][] V) {  1 usage
    int width = Y.length;
    int height = Y[0].length;
    BufferedImage out = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);

    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            int yv = Y[x][y];
            int uv = U[x][y];
            int vv = V[x][y];
            int r = clamp( v: yv + (int)(1.13983 * vv));
            int g = clamp( v: yv - (int)(0.39465 * uv) - (int)(0.58060 * vv));
            int b = clamp( v: yv + (int)(2.03211 * uv));
            int rgb = (r << 16) | (g << 8) | b;
            out.setRGB(x, y, rgb);
        }
    }
    return out;
}

// keep value in [0..255]
private static int clamp(int v) {  3 usages
    return v < 0 ? 0 : (v > 255 ? 255 : v);
}
```

Reconstructs an RGB image from YUV channels.

Applies reverse YUV-to-RGB conversion formulas.

Uses `clamp()` to ensure values stay in the valid RGB range [0, 255].

Ensures a value is between 0 and 255.

Necessary to avoid overflow or underflow when converting back to RGB.

Key Finding

Category	Image	RGB VQ PSNR (dB)	RGB Compression	YUV Compression	RGB/YUV Ratio	YUV Better By Factor
Animals	animal1.jpeg	48.57	3.00×	14.67×	0.20	4.89×
Animals	animal2.jpg	49.83	3.00×	14.67×	0.20	4.89×
Animals	animal3.jpg	49.58	3.00×	14.67×	0.20	4.89×
Animals	animal4.jpg	48.18	3.00×	14.67×	0.20	4.89×
Animals	animal5.jpg	47.68	3.00×	14.67×	0.20	4.89×
Faces	face1.jpg	48.37	3.00×	14.67×	0.20	4.89×
Faces	face2.jpg	49.65	3.00×	14.67×	0.20	4.89×
Faces	face3.jpg	50.54	3.00×	14.67×	0.20	4.89×

UV VQ consistently outperforms RGB VQ in terms of compression ratio by a factor of 4.89 across all images.

PSNR values for RGB VQ are high (47–51 dB), indicating good reconstruction quality despite stronger compression from YUV.

RGB/YUV ratio is stable at 0.20, showing consistent performance differences across image types and resolutions.

```
[YUV] Converting YUV → RGB...
[YUV] Saving output to output\yuv_decoded\nature5.jpg
   YUV VQ Compression Ratio: 14.67
   RGB/YUV Ratio: 0.20
   => YUV VQ compresses better by a factor of 4.89
--- All done! ---
```

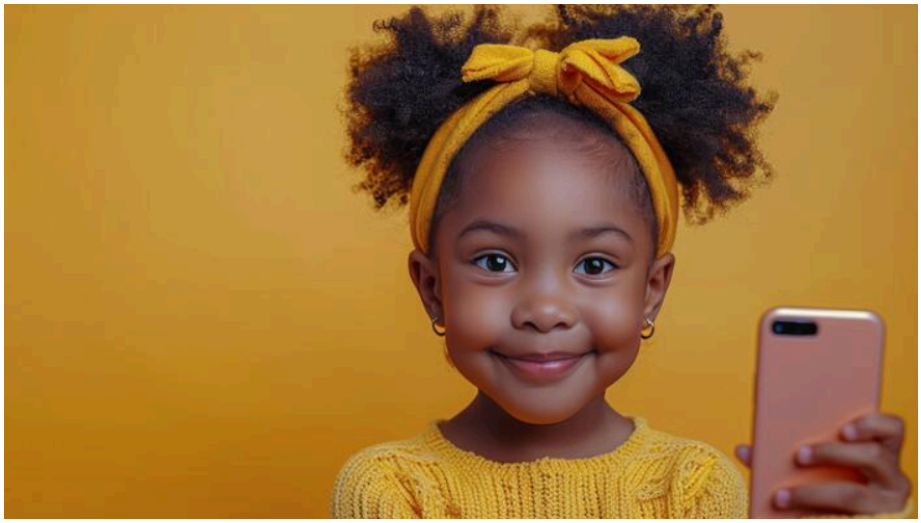
Original Image

YUV O/P

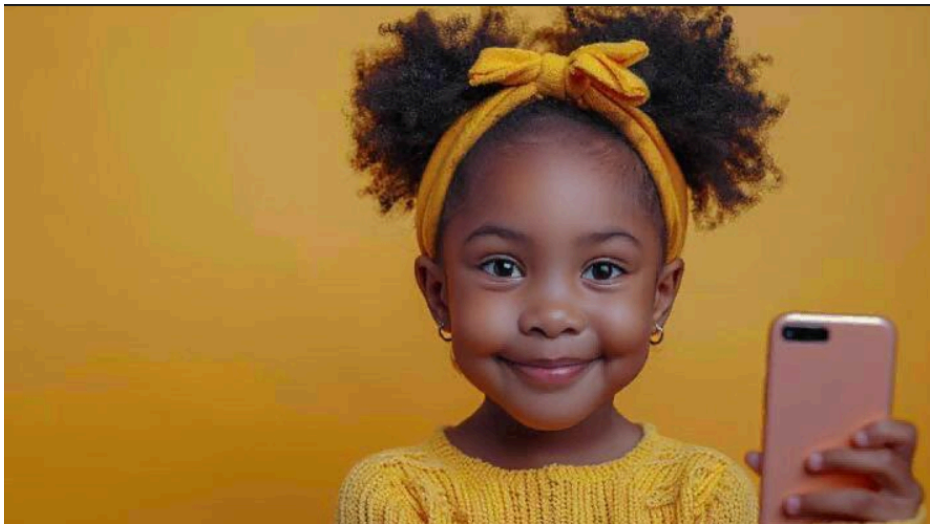
RGB O/P

AS We see the yuv lost some details compared to RGB compression but overall yuv provides higher compression.

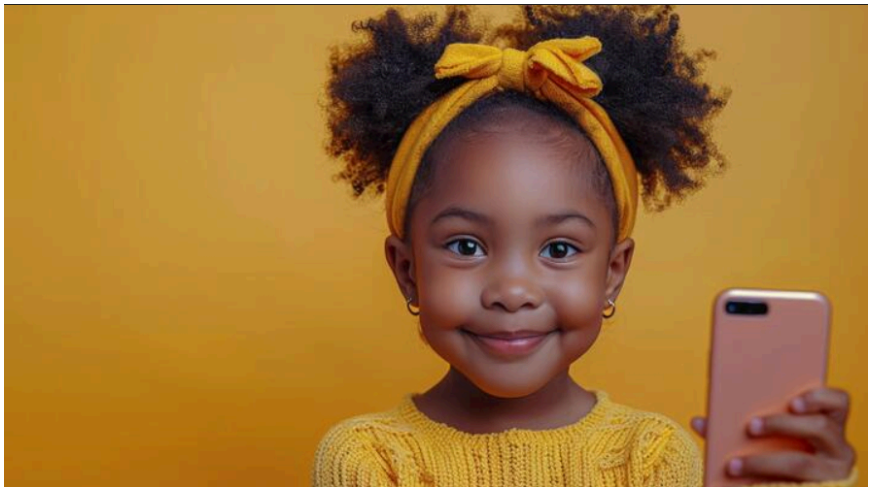
Example 2
Original Image



YUVCompressor



RGB Compression



It didn’t differ too much but if we look at the background colors there are some abnormalities from yuv compression.
For RGB compression it looks the same as the original image.