JPEG COMPRESSION ALGORITHM

Group project for the Course in Multimedia Systems 2019-20
Bachelor's Degree in Computer Science and Engineering
Faculty of Computer Science - Free University of Bolzano

Simonluca Merlante
Ulrike Niederstätter
Stephan Unterrainer

Table of Contents

Bolzano, June 9th 2020

1. Introduction
The goal of our project was to reproduce the JPEG-compression algorithm using the Java environment. This was chosen because all group members were already familiar with Java and because of integration with the Image-processing library OpenCV.
In order to provide some basic interaction with the application, we developed a simple GUI which allows the user to select a file from local storage, compress it and save it automatically. The application was developed only for grey scale images.

2. GUI
The GUI was developed using basic operations of JavaSwing. It contains a simple Menu where the user can open a new file for compression. Although, the user can also select for compression the default image saved within the application (see Figure 1) .

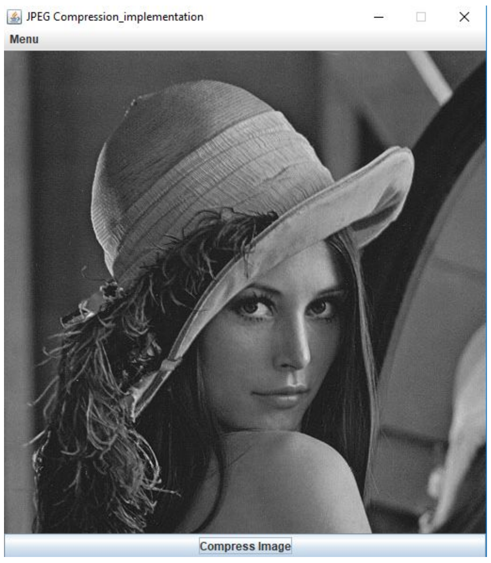


Figure 1

When the user clicks on the "Compress Image" button, the chosen file is used in the JPEG compression algorithm and the output, i.e., the compressed image, is saved in the main folder of the application. The image will be saved with the name *compressed_image.png*. If a file is already present, the application overwrites it.

3. Compression
As we know, the compression algorithm is divided into various steps. In order to simplify things, it was chosen to skip the step of the color-downsampling.
Therefore, the remaining steps that were implemented are the following:

1. As a first step we developed a utility-method:

   ```
   public static Mat imgToMat(String path);
   ```

   which is used to read an image and convert into a `Mat` matrix. This was later divided into blocks of 8 x 8 elements. Each of these blocks was then saved in an ArrayList. This data structure was selected because of its flexible behaviour.

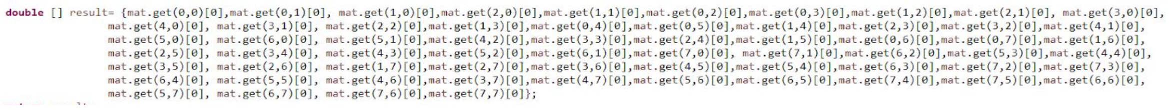
2. For each of the blocks in the `ArrayList` from 1.) we performed the Discrete Cosine Transform (DCT) . This was accomplished by using a build-in function of OpenCV:

   ```
   Core.dct(m, dct_convert);
   ```

   This method takes each block m and returns a matrix dct_convert of the same size and same type with the frequency values as elements.
   It was necessary to convert the original matrix to floating points, using the CvType `CV_64FC`1, because the function `Core.dct(src, dest)` expects floating point values for the source matrix.
3. The next step is quantization. For that we have defined a standard quantization matrix and performed single point multiplication with a variable called `qualityFactor`, which represents the level of quality used in the compression algorithm. The method

   ```
   public static List<Mat> quantise(List<Mat> mat)
   ```

   performs quantization for each of the blocks obtained at 2.), applying a punctual division of the blocks elements with the quantization matrix elements. The results were, again, stored in an `ArrayList` of `Mat` objects.
4. The following step is the zig-zag scanning of the matrix. For that we have used a similar technique shown in the laboratory.

   ```
   double [] result= {mat.get(0,0)[0],mat.get(0,1)[0], mat.get(1,0)[0],mat.get(2,0)[0],mat.get(1,1)[0],mat.get(0,2)[0],mat.get(0,3)[0],mat.get(1,2)[0],mat.get(2,1)[0], mat.get(3,0)[0],
                   mat.get(4,0)[0], mat.get(3,1)[0], mat.get(2,2)[0],mat.get(1,3)[0],mat.get(0,4)[0],mat.get(0,5)[0],mat.get(1,4)[0],mat.get(2,3)[0],mat.get(3,2)[0],mat.get(4,1)[0],
                   mat.get(5,0)[0], mat.get(6,0)[0], mat.get(5,1)[0],mat.get(4,2)[0],mat.get(3,3)[0],mat.get(2,4)[0],mat.get(1,5)[0],mat.get(0,6)[0],mat.get(0,7)[0],mat.get(1,6)[0],
                   mat.get(2,5)[0], mat.get(3,4)[0], mat.get(4,3)[0],mat.get(5,2)[0],mat.get(6,1)[0],mat.get(7,0)[0], mat.get(7,1)[0],mat.get(6,2)[0],mat.get(5,3)[0],mat.get(4,4)[0],
                   mat.get(3,5)[0], mat.get(2,6)[0], mat.get(1,7)[0],mat.get(2,7)[0],mat.get(3,6)[0],mat.get(4,5)[0],mat.get(5,4)[0],mat.get(6,3)[0],mat.get(7,2)[0],mat.get(7,3)[0],
                   mat.get(6,4)[0], mat.get(5,5)[0], mat.get(4,6)[0],mat.get(3,7)[0],mat.get(4,7)[0],mat.get(5,6)[0],mat.get(6,5)[0],mat.get(7,4)[0],mat.get(7,5)[0],mat.get(6,6)[0],
                   mat.get(5,7)[0], mat.get(6,7)[0], mat.get(7,6)[0],mat.get(7,7)[0]};
   ```

   This produced an array of floating point values representing each block and organised in zig-zag fashion.
5. Now, the actual encoding of each element is performed. We separated the encoding of the first element of each block which corresponds to the DC-element from all other elements, the AC-elements.
   - Encoding of DC-Element: For the DC-element we developed a method called

     ```
     public static JPEGCategory RLEDC(double DCElement)
     ```

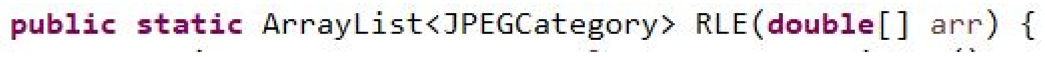
     - This method takes a floating point value and computes the difference from a forecast using the following function of the `Utils` class:
       `Utils.setEncodedPred(DCElement)`
       This function was created to ensure correctness of the prediction mechanism for both encoding and decoding.
       Hence, only the difference between the previous DC-Element and the actual one is encoded. This method produces an object of type `JPEGCategory`. This is a custom object that we have created which represents an AC or a DC-element and has the following properties.
     - The coefficient: the actual value of the element.
     - The category: which is one of the ten possible categories that non-zero coefficients can get.
     - The precision: which specifies the actual position of the element inside the assigned category.
     - The runlength: which is the number of zero-elements proceeding that element.

     With this method:

     ```
     public static JPEGCategory assignCategory(double coeff)
     ```

We take the coefficient and compute the correct category and precision for each `JPEGCategory` object.

- Encoding AC-Element: the encoding of all other elements works in a very similar way, the method works a bit differently:



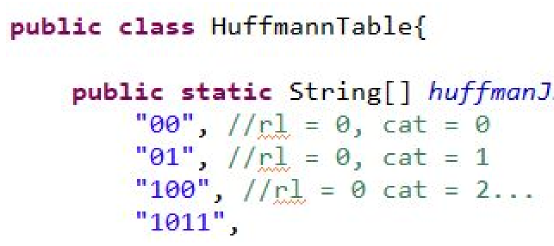
```
public static ArrayList<JPEGCategory> RLE(double[] arr) {
```

It takes an array of coefficients, checks them one by one and increased the runlength when it encounters zeros. Finally, it returns a `JPEGCategory` object containing all necessary information including the runlength. This is different to the encoding of the DC-Element since such elements have runlength zero. Moreover, we take in consideration special cases such as:

  - The End of Blocks (EoB) signaled with a special symbol
  - A runlength of 16 zeros signaled with a special symbol as well.

The result is a list of `JPEGCategory` objects, each containing the information to encode a meaningful symbol.

6. Now we have reached the Huffman Encoding part of the algorithm. Depending on the runlength and on the category of each `JPEGCategory` object, we assign to it a string, in the Huffman table. For the table we have used the default Huffman table seen also in the laboratory part for this exercise.



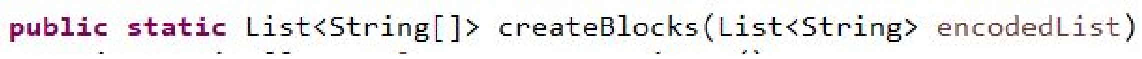
```
public class HuffmannTable{

    public static String[] huffmanJPG={
        "00", //rl = 0, cat = 0
        "01", //rl = 0, cat = 1
        "100", //rl = 0 cat = 2...
        "1011",
```

The result is a list of Strings of binary symbols which is ready to be sent to the channel.

4. Decompression

To simulate the reception of the signal and the steps in this case, we performed on the encoded picture also the decompression part.

1. The first in the decompression part is the organisation the list of Strings of binary symbols, into blocks of Strings. This is done in order to reconstruct the original blocks. We wrote a method that performs this task by identifying the End of Block symbols.



```
public static List<String[]> createBlocks(List<String> encodedList)
```

2. For each of the blocks identified in 1., we isolate the first element, which corresponds to the DC-Element, i.e. the error of the actual DC-Element and the previous one. At this step we need to recover the actual element by adding the prediction. This is done using:

   `this.coeff = error + Utils.getDecodedPred();`

   Now, for this value we perform the Huffman decoding using the lookup table. This method checks the position of the Huffman-symbol and depending on that, it returns the category, runlength and the coefficient. The precision was sent without Huffman encoding as per JPEG standard and does not need decoding.The same is then repeated for all AC-Elements in a similar way.

Finally, we need to set again the prediction value to the current DC-Element using:

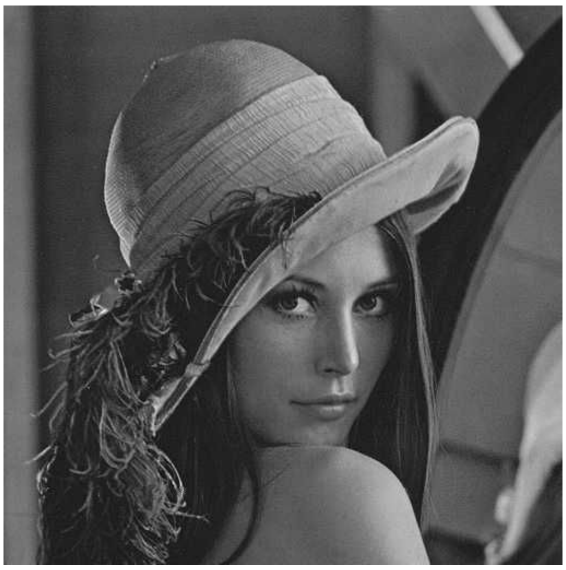
```
Utils.setDecodedPred(this.coeff);
```

3. Now, we need to perform the inverse operation of the zig-zag scanning. It works similarly as for the forward zig-zag using a matrix:
   ```
   public static Mat invert(JPEGCategory[] input)
   ```
   But before doing the inverse zig-zag, we need to convert the `JPEGCategory` array into a full array of 64 elements. This done by the method:
   ```
   public static double[] convertToDouble(JPEGCategory input)
   ```
   which evaluates the various runlengths, assigning the coefficients and filling out the remaining positions with zero depending on the runlengths. The results of the inverse zig-zag is a list of `Mat` objects, each representing a block.
4. Dequantization: Here we simply inverted the process of quantization, multiplying each block for the quantisation matrix which was previously multiplied by the `qualityFactor` (See point 3. of compression)
5. Inverse DCT: Also in the inverse case we used the built-in function of OpenCV to perform the inverse discrete cosine transform to get back the values in grey-scale, i.e., move from the frequency domain back to the space domain.
6. As the final step, we needed to reconstruct the original matrix (image) since at point 5. We still had a list of `Mat` objects. This was accomplished through a triple for loop which wrote all blocks over rows and columns of the final matrix. In order to perform that, we needed to first save all data into a simple array of doubles and use that as input to fill the final matrix. The only thing missing is then, to convert the final matrix to the dedicated type of elements (`CvType.CV_32FC3`)and write it to disk in image format:
   ```
   Imgcodecs.imwrite("compressed_image.png", finalIMG);
   ```
   Now, the image saved under *compressed_image.png* is a compressed version of the original image.

5. Evaluation

The algorithm depends on various parameters, so we know for different images we will not achieve the same compression rate even though we used the same quality factor. We have built some test cases to show what results the algorithm achieves:

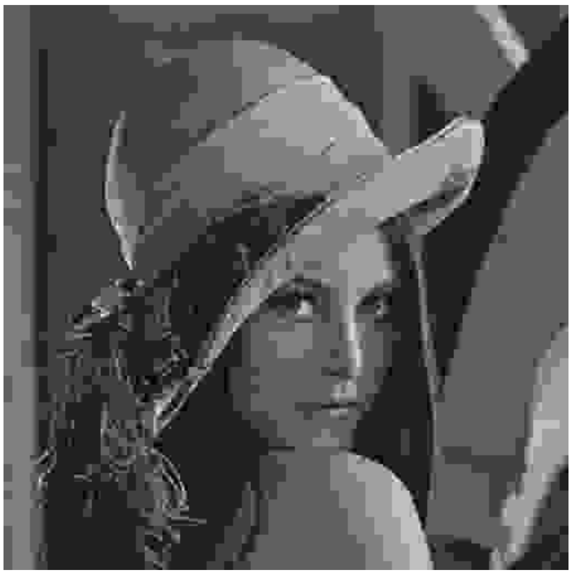
Test Case 1: Lena-grey.png q=100



Original image | *Compressed image*

Compression power: 165/148 KB	1,11
Bitrate: 8/1,11	7,20 bpp
Peak Signal-to-noise ratio	34,43 dB

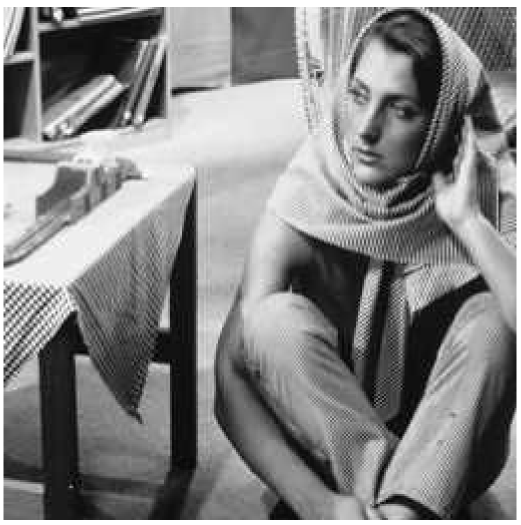
Test Case 2: Lena_grey.png q=85



Original image | Compressed image

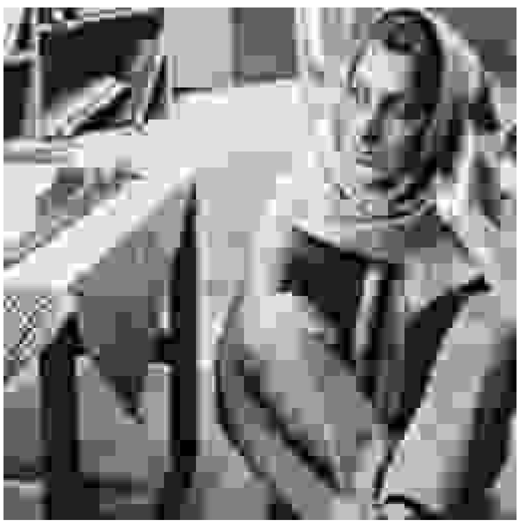
Compression power: 165/62,1 KB	2,65
Bitrate: 8/2,65	3,01 bpp
Peak Signal-to-noise ratio	25,09 dB

Test Case 3: Barbara.tif q=100



Original image	*Compressed image*
Compression power: 89/50KB	1,78
Bitrate: 8bpp/1,78	4,494 bpp
Peak Signal-to-Noise Ratio	30,727 dB

Test case 4: Barbara.tif q=85



Original image	*Compressed image*
Compression power: 89KB/32KB	2,781
Bitrate: 8bpp/2,781	2,877 bpp
Peak Signal-to-noise ratio	21,355 dB