IMAGE AND VIDEO TECHNOLOGY

INFO-Y093

Practical report of "Image and Video Technology" exercises

Authors:
Maxime BOLLENGIER

Professor:
Schelkens PETER

Assistant
Schretter COLAS

Academic Year: 2021-2022

Contents

1 WPO1 **1**
- 1.1 Explore images with ImageJ . . . 1
- 1.2 Create a RAW 32bpp grayscale image . . . 2
- 1.3 Modify a RAW 32bpp grayscale image . . . 4

2 WPO2 **5**
- 2.1 Uniform and gaussian random white noise . . . 5
- 2.2 Blur and additive white noise using ImageJ . . . 6
- 2.3 Additive white noise and blur using ImageJ . . . 7

3 WPO3 **8**
- 3.1 Matrix of orthogonal basis functions . . . 8
- 3.2 Discrete cosine transforms(DCT) . . . 9

4 WPO4 **11**
- 4.1 Lossy JPEG image approximation . . . 11
- 4.2 Packing DCT coefficients . . . 13

5 WPO5 **14**
- 5.1 Delta encoding of DC coefficients . . . 14
- 5.2 Run-length encoding(RLE) of AC coefficients . . . 16
- 5.3 Discrete probability density functions (PDF) . . . 19

6 WPO6 **20**
- 6.1 Exponential-Golomb variable-length code (VLC) . . . 20
- 6.2 Lossy grayscale image compression . . . 22

Appendix **24**

A output values **24**

1 WPO1

1.1 Explore images with ImageJ

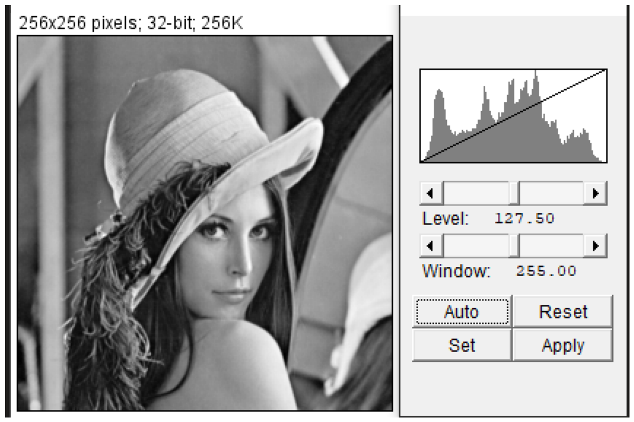
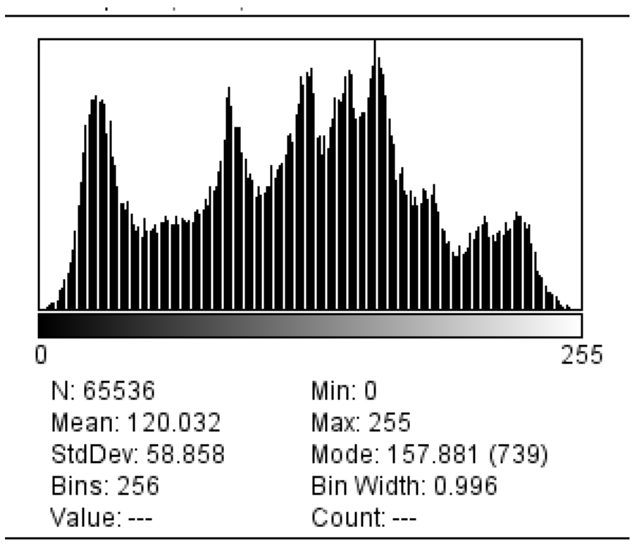


Figure 1: lena 256x256 in ImageJ viewer and Window/Level

Figure 1 shows that the window which encloses all pixel values is 255. This result corresponds to the range of grayscale intensity level which is between 0 and 256 for each pixel. These values are also highlighted in the **Min** and **Max** values in the histogram of Figure 2 shown below. 0 represents an image composed of white pixels and 255 an image composed of black pixels.



```
N: 65536            Min: 0
Mean: 120.032       Max: 255
StdDev: 58.858      Mode: 157.881 (739)
Bins: 256           Bin Width: 0.996
Value: ---          Count: ---
```

Figure 2: lena 256x256 in ImageJ viewer and its histogram

Figure 2 shows the distribution of the image pixels intensity. The images have 256×256 pixels. Then the histogram has a **Count** of 65536 values. The **Mean** tells us the average brightness of the image. The standard deviation (**StdDev**) represents the mean variation of the pixel intensity values compared to the mean. The **Mode** corresponds to the dominant pixel intensity value (peak on the histogram). The **Bin Width** represents the space between each value of pixel intensity. It can be seen as a quantifier because each value present in this range will be associated to the corresponding value.

1.2 Create a RAW 32bpp grayscale image

By creating a cosine pattern following the equation 1 below,

$$I(x,y) = \frac{1}{2} + \frac{1}{2}cos(x\frac{\pi}{32})cos(y\frac{\pi}{64}) \quad (1)$$

the figure 3 and its histogram (Figure 4) were obtained and shown below :

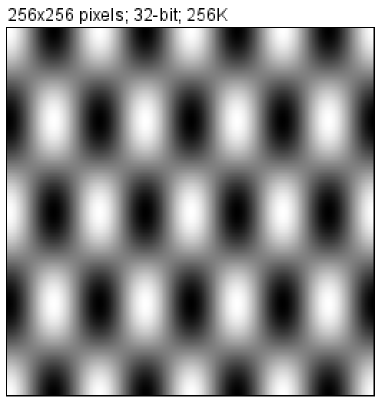
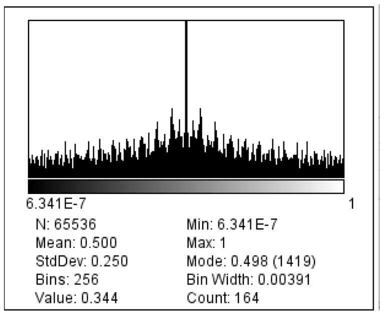


Figure 3: Visualization of the cosine pattern in ImageJ for a Window = 1 and level = 0.5



```
N: 65536            Min: 6.341E-7
Mean: 0.500         Max: 1
StdDev: 0.250       Mode: 0.498 (1419)
Bins: 256           Bin Width: 0.00391
Value: 0.344        Count: 164
```

Figure 4: Visualization of the histogram of the cosine pattern in ImageJ for a Window = 1 and level = 0.5

The figure 4 confirms that we have a **Mean** $\approx \frac{1}{2}$ which is the theoretical mean that we expect. By setting the window as close as possible to the minimum and by sliding different levels, we can observe the cosine pattern without any blurring as shown in Figure 5, Figure 6 and Figure 7. This visual effect is due to the fact that all the pixels present outside the window level are rounded to white or black pixel depending the side. Since we have approximately the same number of pixels before and after the mean, the Figure 5 shows some square shapes for a small window and mean intensity level (= 0.5). In opposite, by sliding the level up/down to the mean, the square's corners are rounded to black/white pixel as shown in Figure 6 and Figure 7.

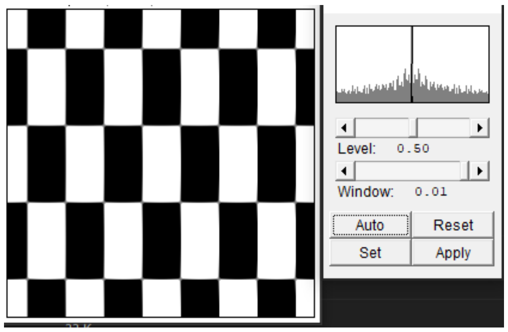


Figure 5: Visualization of the cosine pattern in ImageJ for a Window = 0.01 and level = 0.5

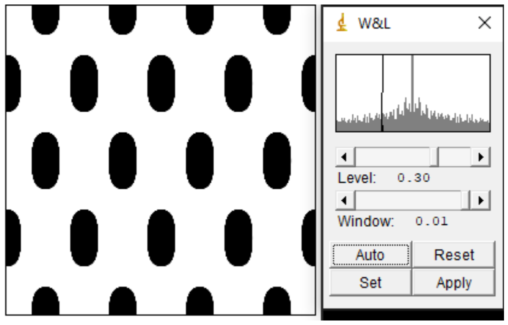


Figure 6: Visualization of the cosine pattern in ImageJ for a Window = 0.01 and level = 0.3

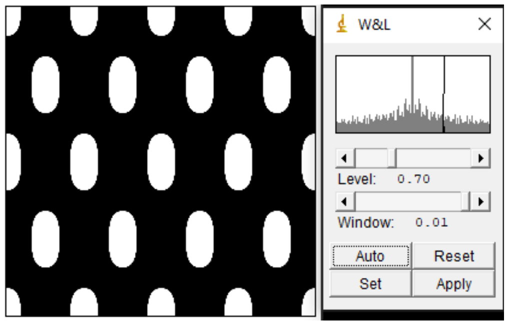


Figure 7: Visualization of the cosine pattern in ImageJ for a Window = 0.01 and level = 0.7

The store function is detailed in Listing 1. To store images in RAW files, first the data type **ofstream** from the **fstream** library is used to create the file in which will be written the information of the image. The argument **ios::binary** is specified to be sure that the only written characters are the ones that are given. the total size in the memory will be the number of pixels multiplied by the size of each of them (here a float which is coded on 4bytes). Then, for a 256x256 pixels images, the size of the file will be 262 144 bytes.

```cpp
void Store(float* a, string fname, int size) {
  // fstream is Stream class to both
  // read and write from/to files.
  // file is object of fstream class
  ofstream file;

  // opening file "fname"
  // in out(write) mode
  // ios::out Open for output operations.
  file.open(fname, ios::binary);

  // If no file is created, then
  // show the error message.
  if (!file)
  {
    cout << "Error in creating file!!!" << endl;

  }
  cout << fname << "File created successfully." << endl;

  file.write((char*)a, size * size * 4); //adress in memory

  cout << fname << "writting is good" << endl;
  // closing the file.
  // The reason you need to call close()
  // at the end of the loop is that trying
  // to open a new file without closing the
  // first file will fail.
  file.close();
}
```

Listing 1: Store function

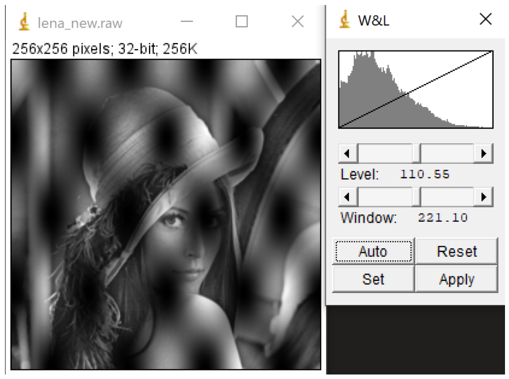
1.3 Modify a RAW 32bpp grayscale image

In the same way, the store function is made with the difference that the data type **ifstream** is used as shown in Listening 2.

```
float* load(const char* filename) {
  const int size_in_bytes = 256 * 256 * 4;
  ifstream file(filename, ios::in | ios::binary | ios::ate);// ate:: read at the end

  if (file.is_open())
  {
    streampos size = file.tellg(); //get the size
    char* memblock = new char[size]; // allocate memory
    file.seekg(0, ios::beg);// tell the adress of the position 0
    file.read(memblock, size);//
    file.close();

    cout << "the entire file content is in memory with size =" << size << endl;;

    return (float*)memblock;
    //delete[] memblock;
  }
  else cout << "Unable to open file";
  return 0;
}
```

Listing 2: Load function

By applying the cosine pattern to ”lena” image with a window of 221.1 and a level of 110.55, we obtain the following result:



256x256 pixels; 32-bit; 256K

Level: 110.55

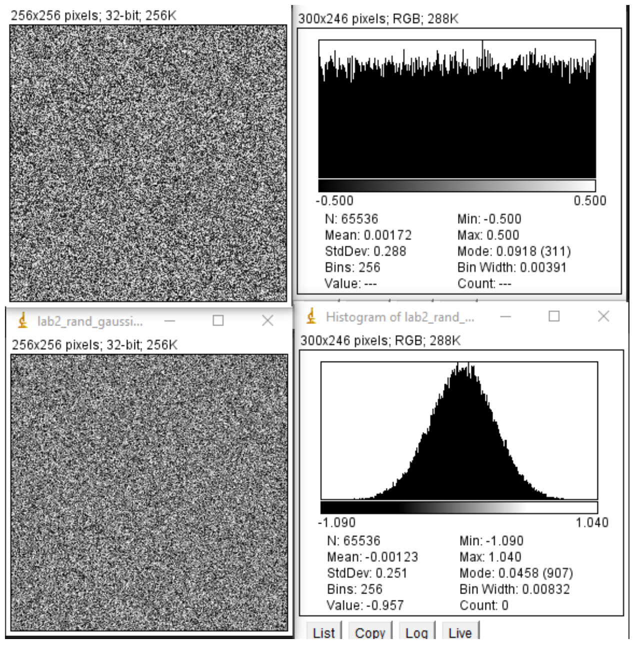
Window: 221.10

Figure 8: lena 256x256 with additive cosine pattern visualization on ImageJ

2 WPO2

2.1 Uniform and gaussian random white noise

The Figure 9 shows uniform and Gaussian distributed random noise. It's difficult to see the difference on the image but the histogram tells us clearly the difference between them. The first shows that the noise is uniformly distributed on each bin in opposite of the second which has a gaussian shape. The expected mean is 0 and the expected variance 0.25. The value on the histogram is not exactly the same due to the random process to generate them. To get the expected mean and variance, we have to make a large number of realizations and average them (law of large numbers).



256x256 pixels; 32-bit; 256K

300x246 pixels; RGB; 288K

```
N: 65536            Min: -0.500
Mean: 0.00172       Max: 0.500
StdDev: 0.288       Mode: 0.0918 (311)
Bins: 256           Bin Width: 0.00391
Value: ---          Count: ---
```

256x256 pixels; 32-bit; 256K

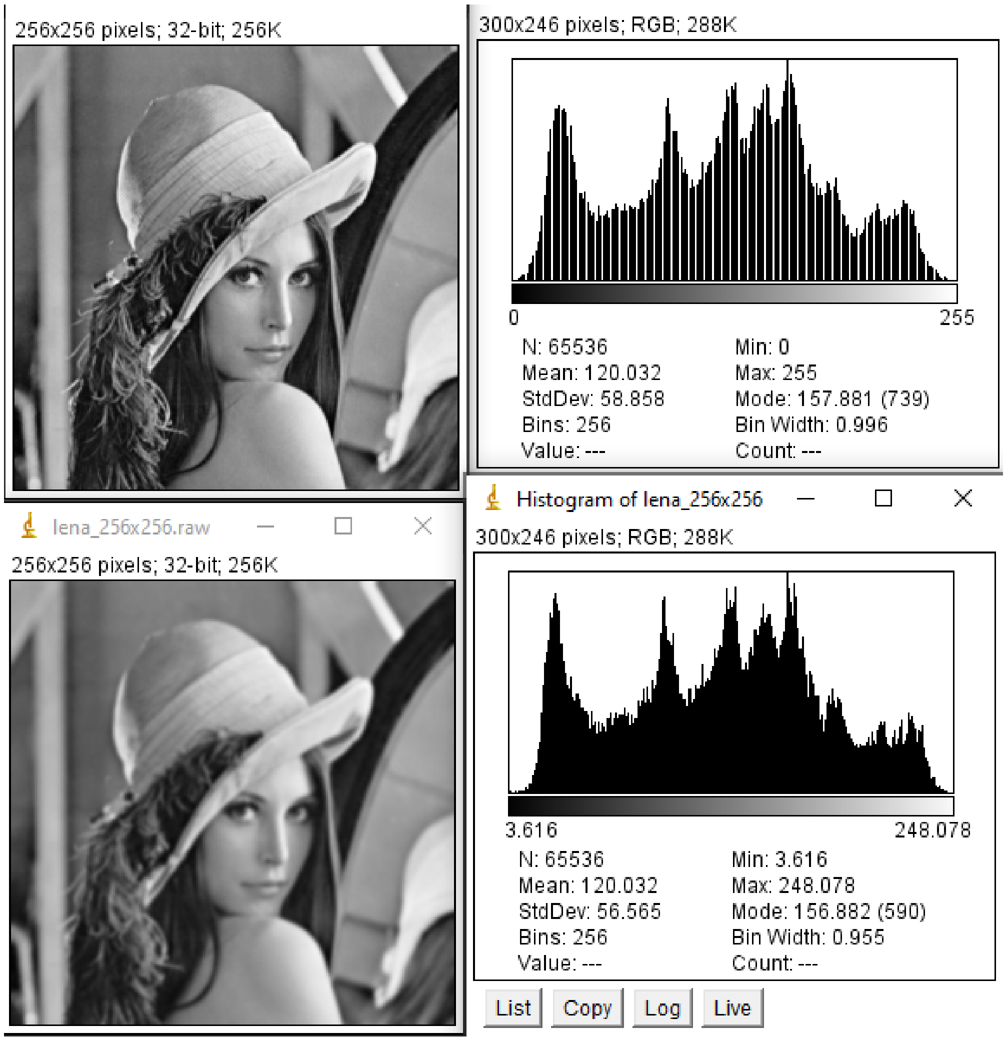
300x246 pixels; RGB; 288K

```
N: 65536            Min: -1.090
Mean: -0.00123      Max: 1.040
StdDev: 0.251       Mode: 0.0458 (907)
Bins: 256           Bin Width: 0.00832
Value: -0.957       Count: 0
```

Figure 9: Uniform and gaussian random with noise and its histogram visualization on ImageJ

2.2 Blur and additive white noise using ImageJ

The Figure 10 shows the effect of gaussian blur filter with a std-dev (sigma) equal to 1. Firstly we can see that the image is blurred and secondly that the histogram is smoothed with respect to the original one.


256x256 pixels; 32-bit; 256K
300x246 pixels; RGB; 288K
0
255
lena_256x256.raw
256x256 pixels; 32-bit; 256K
Histogram of lena_256x256
300x246 pixels; RGB; 288K
3.616
248.078
List
Copy
Log
Live

```
N: 65536          Min: 0
Mean: 120.032     Max: 255
StdDev: 58.858    Mode: 157.881 (739)
Bins: 256         Bin Width: 0.996
Value: ---        Count: ---
```

```
N: 65536          Min: 3.616
Mean: 120.032     Max: 248.078
StdDev: 56.565    Mode: 156.882 (590)
Bins: 256         Bin Width: 0.955
Value: ---        Count: ---
```

Figure 10: original and blurred image and its histogram visualization on ImageJ

The Listings 3 and 4 show the implementation of the Mean Square Error (MSE) and the Peak Signal-to-Noise Ratio (PSNR), respectively computed as follows:

$$MSE = \frac{\sum_{I,J}(I_1(i,j) - I_2(i,j))^2}{I * J} \tag{2}$$

where I, J correspond to the number of pixels i,j.

$$PSNR = 10log_{10}(\frac{MAX^2}{MSE}) \tag{3}$$

where MAX corresponds to the max possible value of one pixel (here 255).

```
float MSE(const float* img1,const float* img2, const int size) {
  float mse = 0;
  int x;
  for (x = 0; x < size * size; x++) {

    mse += (img1[x] - img2[x]) * (img1[x] - img2[x]) / (size * size);
  }

  return mse;
}
```

Listing 3: Store function

```
float PSNR(const int max, const float* img1, const float* img2, const int size) {

  float mse = MSE(img1, img2, size);

  float psnr = 10 * log_{10}(max * max / mse);

  return psnr;
}
```

Listing 4: Store function

By applying the above gaussian blur filter on the original image, a value of 64.36 is obtained for MSE and 30.04 for the PSNR.

2.3 Additive white noise and blur using ImageJ

By adding gaussian noise with variance equal to 65, the same MSE as previous is obtained. Now, by manually setting different values of gaussian blur standard deviation (sigma), a value of 0.6 was found to minimize the MSE between the restored noisy image w.r.t the original as shown in Figure 1. The MSE found is 0.32 (so one half of the previous value).

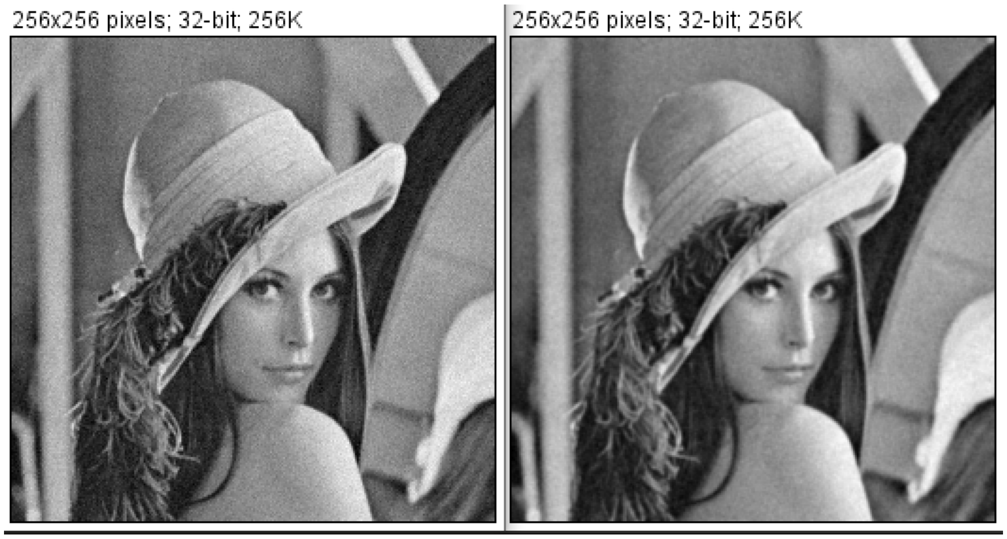


Figure 11: Noisy image ($mse \approx 64$) and the restored version ($mse \approx 32$) by applying gaussian blurring

In conclusion, the blurring has the capacity to partially restore a noisy image.

3 WPO3

3.1 Matrix of orthogonal basis functions

The DCT basis vectors for a 1D signal of length N is computed by the following equation :

$$b_k(n) = c(k)cos(\frac{\pi}{N}(n + \frac{1}{2})k) \tag{4}$$

with the scale factors $c(k)$ defined by :

$$c(k) = \frac{1}{\sqrt{N}} \forall k = 0 \tag{5}$$

and

$$c(k) = \frac{2}{\sqrt{N}} \forall k = 1, .., N - 1 \tag{6}$$

This version of DCT basis is called DCT-II. This formula comes from `https://fr.wikipedia.org/wiki/Transform%C3%A9e_en_cosinus_discr%C3%A8te`.The figure 12 shows the DCT-basis for N = 256x256 and its transposed.

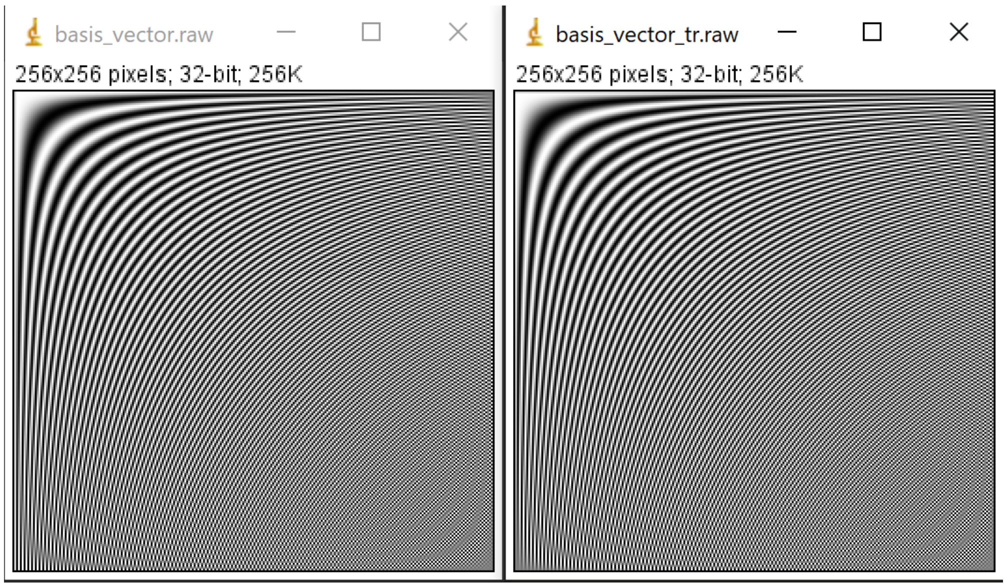


Figure 12: DCT-basis for N = 256x256 and its transposed visualization on ImageJ

The DCT transform of the image is performed by matrix multiplication between the image and the basis. This operation is done two times, one time for the rows and one time for the columns of the images as shown in Listing 6. To obtain the IDCT, the same process is done by transposing the DCT basis with the transposed function found in Listing 5 as shown in Listing 8. The DCT/IDCT basis are orthonormal : $AA^T = A^T A = I$ as shown in Figure 13

```
float* transpose(const float* matrix, int size) {
  float* transpose = new float[size * size];
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      transpose[i + j * size] = matrix[j + i * size];
    }
  }
  return transpose;
}
```

Listing 5: Transpose function

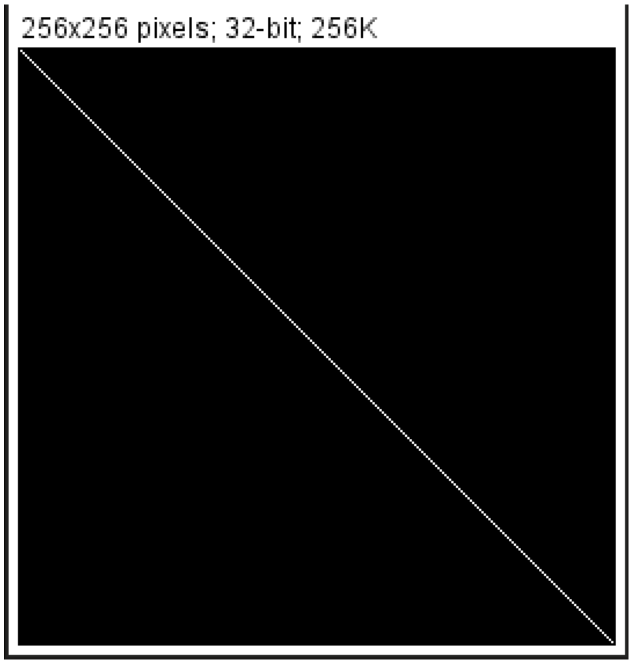


Figure 13: Visualiztion of matrix multiplication of DCT basis and its transpose on ImageJ

3.2 Discrete cosine transforms(DCT)

```
float* transform(const float* image, const float* basis, const int size) {

  float* transformed_image = matmul(image, basis, size);
  return transformed_image;
}
float* dct_transform2d(const float* image, const float* basis, const int size) {
  float* dct = transform(image, basis, size);
  dct = transpose(dct, size);
  float* dct2d = transform(dct, basis, size);
  dct2d = transpose(dct2d, size);
  return dct2d;
}
float* matmul(const float* mat1, const float* mat2,const int size) {
  float* mul = new float[size * size];
  for (int i = 0; i < size; i++)
  {
    for (int j = 0; j < size; j++)
    {
      mul[i + size * j] = 0;
      for (int k = 0; k < size; k++)
      {
        mul[i + size * j] += mat1[i + size * k] * mat2[k + size * j];
      }
    }
  }
  return mul;
}
```

Listing 6: DCT transform functions

```
float* inverse_dct_transform2d(const float* dct2d, const float* basis, const int size) {
  const float* basis_t = transpose(basis, size);
  dct2d = transpose(dct2d, size);
  float* dct = transform(dct2d, basis_t, size);
  dct = transpose(dct, size);
  float* image = transform(dct, basis_t, size);
  return image;
}
```

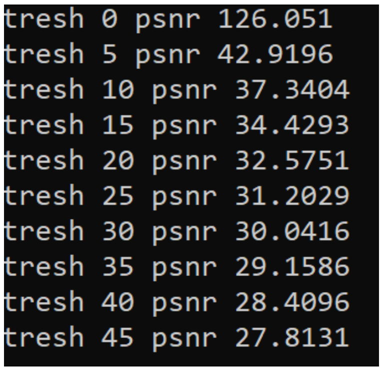
Listing 7: Inverse DCT transform

In order to remove small coefficients after DCT transformation, a threshold function is implemented in Listing 7.

```
void threshold(float* coeff, const float tresh, const int size) {
  //float* thresholded = new float[size * size];
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      if (abs(coeff[i + size * j]) > tresh) {
        coeff[i + size * j] = coeff[i + size * j];
      }
      else {
        coeff[i + size * j]= 0;
      }
    }
  }
}
```

Listing 8: threshold function

By testing some values of threshold, we can see (on Figure 14) that the psnr decreases when the threshold increases. This is due to the loss of information by discarding some elements in the frequency domain. A threshold value of 10 seems to be good for the following step. The result in image is shown on Figure15.



```
tresh 0 psnr 126.051
tresh 5 psnr 42.9196
tresh 10 psnr 37.3404
tresh 15 psnr 34.4293
tresh 20 psnr 32.5751
tresh 25 psnr 31.2029
tresh 30 psnr 30.0416
tresh 35 psnr 29.1586
tresh 40 psnr 28.4096
tresh 45 psnr 27.8131
```

Figure 14: Various value of psnr for various values of threshold

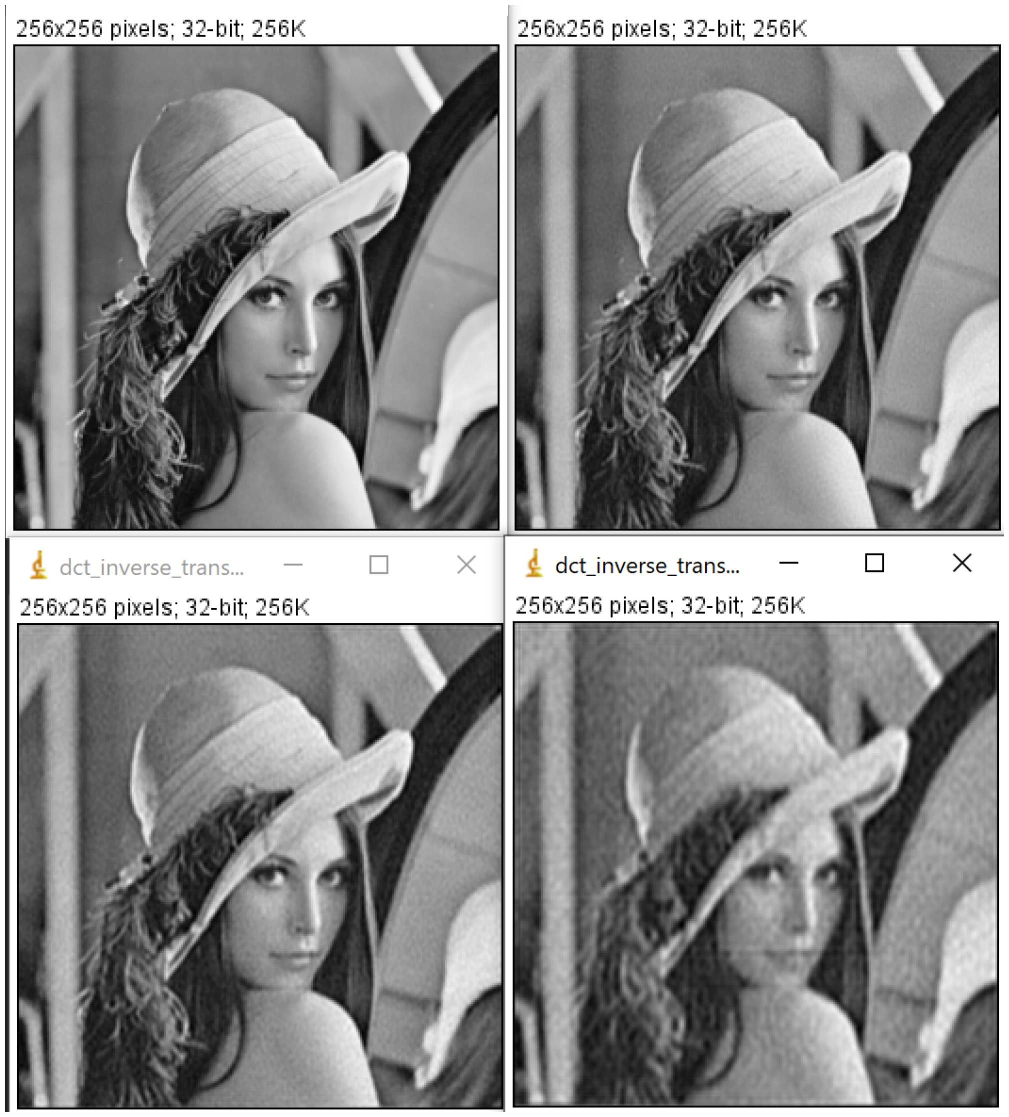

256x256 pixels; 32-bit; 256K
256x256 pixels; 32-bit; 256K
dct_inverse_trans...
256x256 pixels; 32-bit; 256K
dct_inverse_trans...
256x256 pixels; 32-bit; 256K

Figure 15: Visualization of reconstructed images for threshold values(= 0,10,20,50)

4 WPO4

4.1 Lossy JPEG image approximation

The Figure 16 corresponds to the standard JPEG quantization matrix at 50% quality and was designed and optimized by hand. We can observe that the low frequency components have more weight than the higher frequency components.

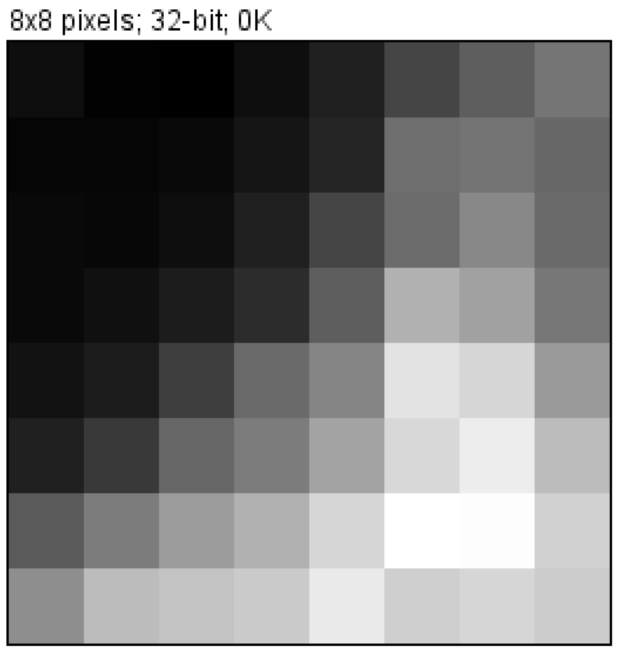


Figure 16: Standard JPEG quantization matrix at 50% quality visualization on ImageJ

For each 8x8 block, the quantization step produces the quantized DCT components by first dividing them with the corresponding weight and then rounding the result to minimize the quantization error. The implementation of this process is shown in Listing 9.

```c
void quantize(float* dct, const int* Q) {
  for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
      dct[i + 8 * j] = round(dct[i + 8 * j] / Q[i + 8 * j]);
}

void inverse_quantize(float* dct, const int* Q) {
  for (int i = 0; i < 8; i++)
    for (int j = 0; j < 8; j++)
      dct[i + 8 * j] = dct[i + 8 * j] * Q[i + 8 * j];
}

float* approximate(const float* image, const float* basis, const int* Q, int size) {
  float* dct = dct_transform2d(image, basis, size);
  threshold(dct, 10, size);
  quantize(dct, Q);
  Store(dct, "quantized.raw", size);
  inverse_quantize(dct, Q);
  return inverse_dct_transform2d(dct, basis, size);
}
```

Listing 9: Approximate functions

After the approximation process, the following figure (Figure 17) is obtained. It corresponds to the difference between the original image and the reconstructed one after approximation. We would expect an image that is zero everywhere, but as we can see this is not the case. We can observe the edges of lena from the original image. This is the residual corresponding to the loss of high frequency content during the quantization process.

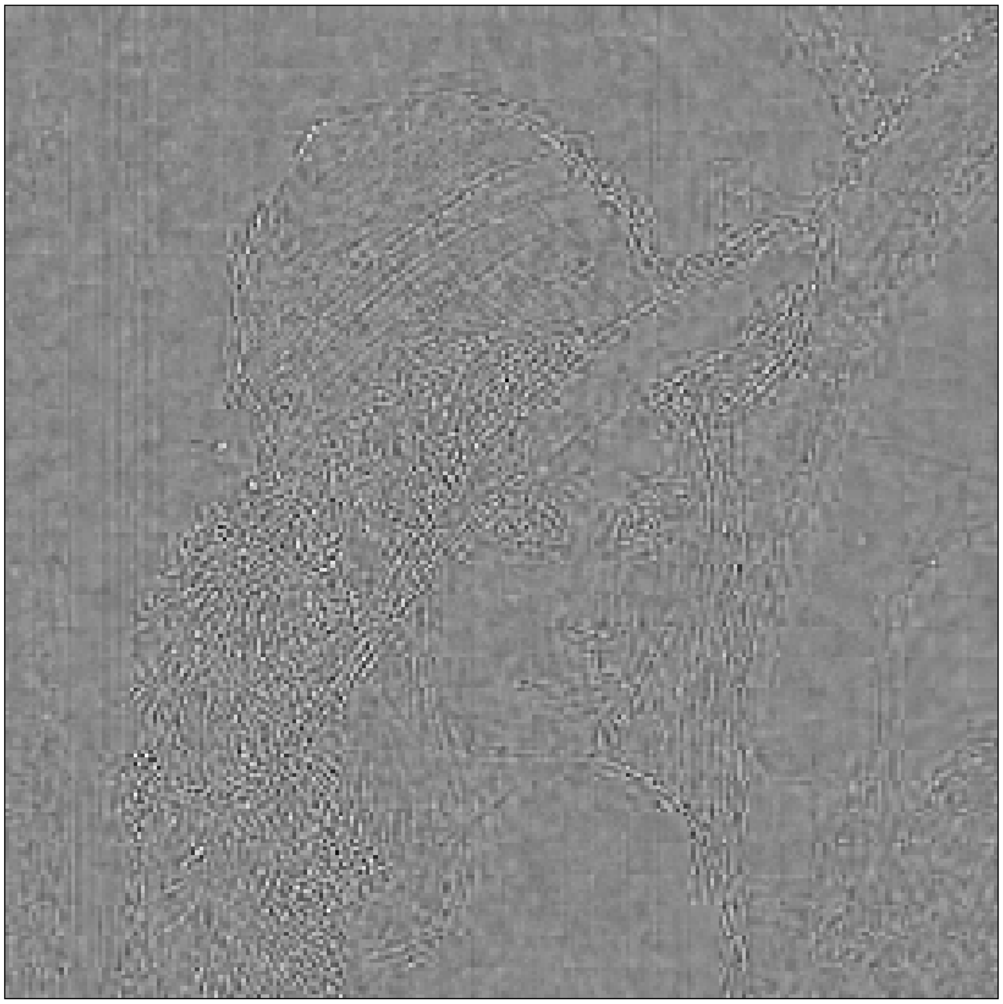


Figure 17: Visualization of the difference between the original image and the reconstructed image after approximation

4.2 Packing DCT coefficients

The Figure 18 represents the different ways to layout the DCT coefficients. The interleaved layout consists of grouping each pixel of each 8x8 blocks in one 32x32 block. Then, the first 32x32 block being composed of the DC components. The Contiguous simply consists of grouping each 8x8 block one by one by keeping its 8x8 structure.

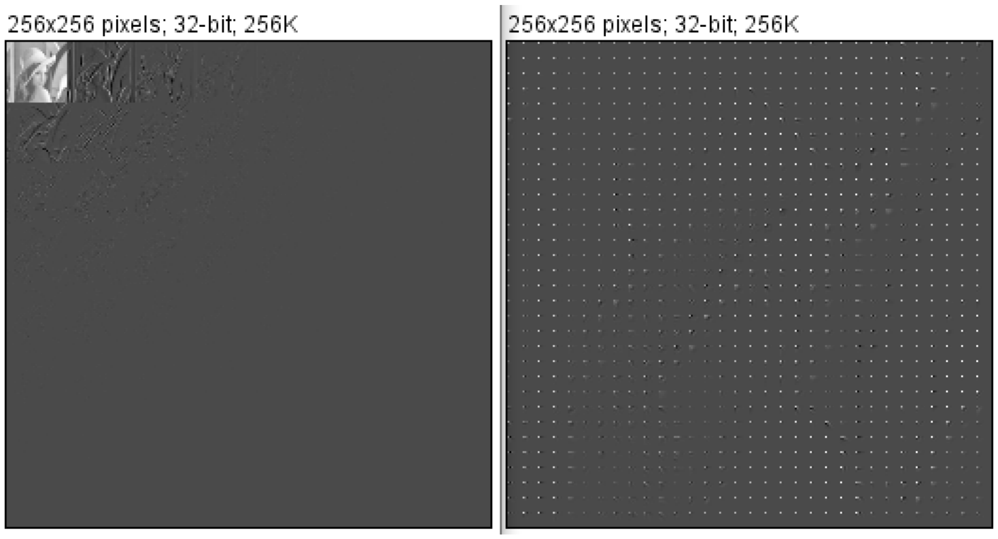
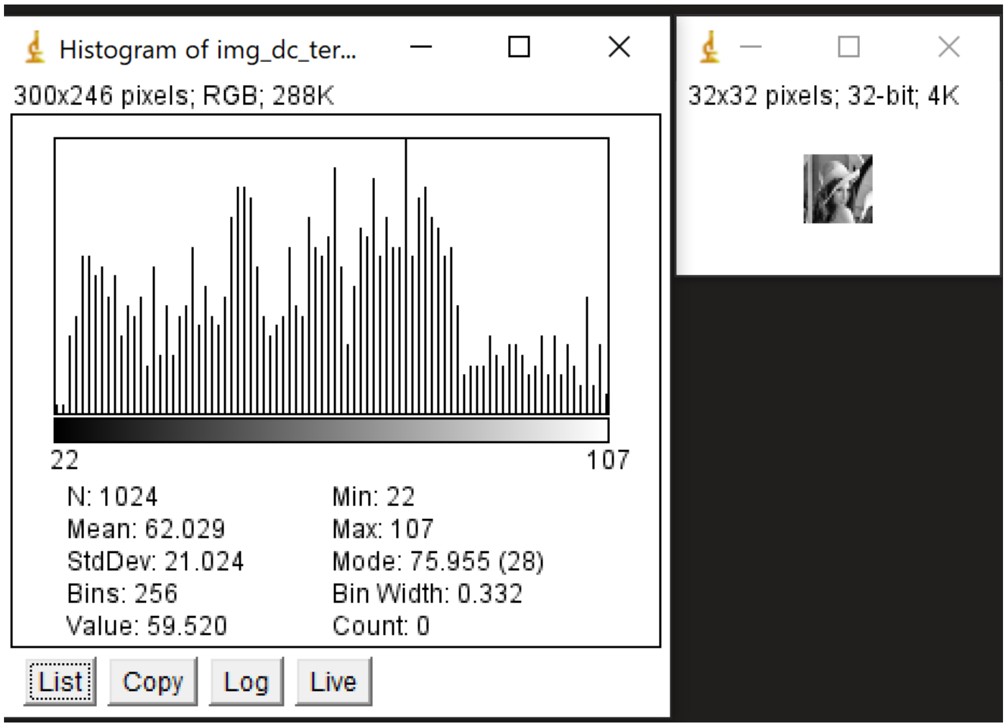


Figure 18: **Left** Contiguous **Right** Interleaved

5 WPO5

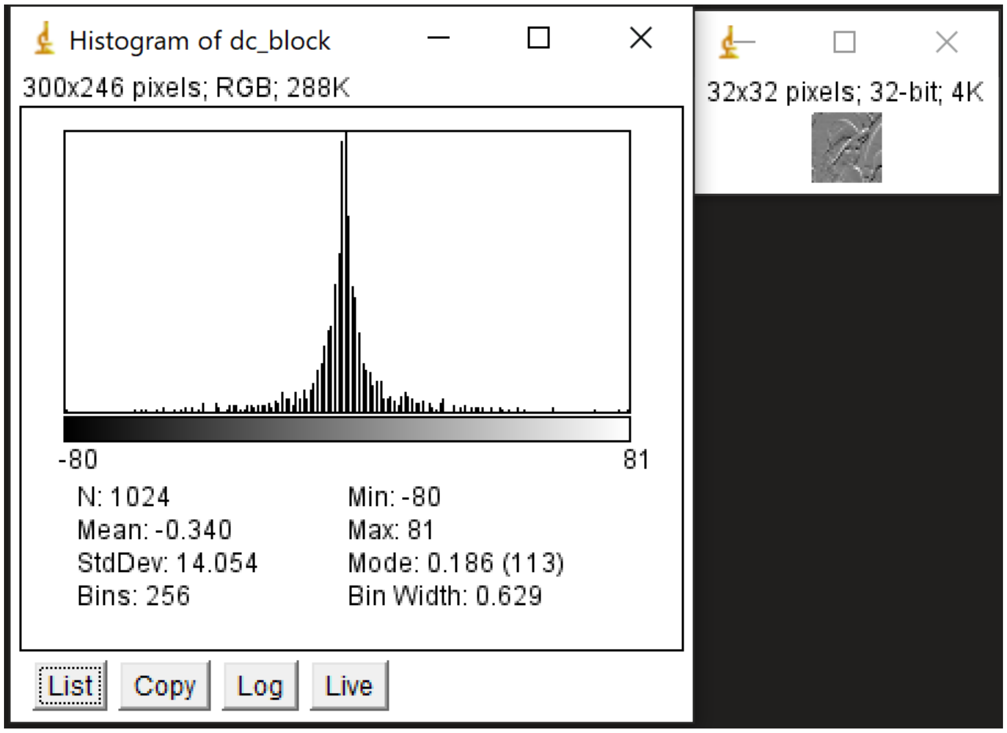
5.1 Delta encoding of DC coefficients

Delta encoding of DC coefficients consists of replacing each coefficient by the difference with the previous one. It allows to have a small number and reduces the variance of the pixels intensity distribution without any loss. Figures 19 and 20 show the 32x32 block containing the DC component. We can observe that the delta length encoding reduces drastically the variance of the histogram distribution and allows to get a laplacian shape with a mean ≈ 0. The implementation of delta encoding is presented in Listing 10 and the decoding in Listing 11.



N: 1024 Min: 22
Mean: 62.029 Max: 107
StdDev: 21.024 Mode: 75.955 (28)
Bins: 256 Bin Width: 0.332
Value: 59.520 Count: 0

Figure 19: Visualization of DC components without delta encoding



N: 1024 Min: -80
Mean: -0.340 Max: 81
StdDev: 14.054 Mode: 0.186 (113)
Bins: 256 Bin Width: 0.629

Figure 20: Visualization of DC components with delta encoding

```
float* delta_encoding(float* img_dc_terms, int size) {
  float* block = new float[size * size];
  ofstream myfile("delta_encoding_dc.txt");
  int l = 0;
  float* delta = new float[size * size];
  for (int n = 0; n < size; n++) {
    for (int k = 0; k < size; k++) {

      if (k == 0) {
        if (n == 0) {
          delta[l] = img_dc_terms[n + size * k];
          myfile << delta[l] << endl;
          block[n + k * size] = delta[l];
        }
        else {
          delta[l] = (img_dc_terms[n + size * k] - img_dc_terms[(n - 1) + size * k]);
          myfile << delta[l] << endl;
          block[n + k * size] = delta[l];
        }

      }
      else {
        delta[l] = (img_dc_terms[n + size * k] - img_dc_terms[n + size * (k - 1)]);
        myfile << delta[l] << endl;
        block[n + k * size] = delta[l];
      }
      l++;
    }
  }
  myfile.close();
  return block;
}
```

Listing 10: delta encoding function

```
float* delta_decoding(const float*block,const int size) {
  float* delta = new float[size * size];
  float* dc_terms = new float[size * size];
  for (int n = 0; n < size; ++n) {
    for (int k = 0; k < size; k++) {
      if (k == 0) {
        dc_terms[n + size * k] = block[n + size * k];
        if (n == 0) {
          dc_terms[n + size * k] = block[n + size * k];
        }
        else {
          dc_terms[n + size * k] = block[n + size * k] + dc_terms[n - 1 + size * k];
        }
      }
      else {
        dc_terms[n + size * k] = block[n + size * k] + dc_terms[n + size * (k - 1)];
      }
    }
  }
  return dc_terms;
}
```

Listing 11: delta decoding function

5.2 Run-length encoding(RLE) of AC coefficients

To encode the AC terms with run length encoding, we take the interleaved layout described above. Next, we encode each 32x32 blocks (without the first one (DC term)) by writing in a text file each non-zero term and for zero terms, I write a zero, followed by the number of zeros. This method is very useful for low frequency block due to the fact that they are full zero due to the threshold. Then, for the last block, the run length encoding is ”0 1024” which consists of two terms instead of 1024 (32x32) terms. The implementation is presented in Listing 12 and the decoding function simply reads elements until the count becomes equal to 1024 as shown in Listing 13.

```
void run_length_encoding(const float* img_ac_terms, const int size, const int block_size) {
  float* block = new float[block_size * block_size];
  int run_length = 0;
  int max_run_length = 0;
  ofstream myfile("run_length_encoding_ac_decompress.txt");
  long long int count_zeros = 0;
  for (int n = 0; n < size / block_size; n++) {
    for (int k = 0; k < size / block_size; k++) {
      block = dicing_window(img_ac_terms, block_size * n, block_size * k, block_size);
      count_zeros = 0;
      for (int x = 0; x < block_size; x++) {
        for (int y = 0; y < block_size; y++) {


          if ((block[x + block_size * y]) != 0) {
            if (count_zeros > 0) {
              myfile << 0 << endl;
              myfile << count_zeros << endl;
              myfile << block[x + block_size * y] << endl;
              run_length += 3;
              count_zeros = 0;
            }
            else {
              myfile << block[x + block_size * y] << endl;
              run_length++;
              count_zeros = 0;

            }

          }
          else {
            if (x == block_size - 1 && y == block_size - 1) {
              count_zeros += 1;
              myfile << 0 << endl;
              myfile << count_zeros << endl;
              run_length += 2;
              count_zeros = 0;


            }
            else {
              count_zeros += 1;

            }
          }
        }
      }
      count_zeros = 0;
      if (run_length > max_run_length) {
        max_run_length = run_length;
      }
      run_length = 0;

    }
  }
  myfile.close();
  cout << "max run length =      " << max_run_length << endl;

}
```

Listing 12: run length encoding function

```
float* run_length_decoding(const int size, const int block_size) {
  string* elemen = new string[size * size];
  ifstream infile;
  infile.open("run_length_encoding_ac_decompress.txt");
  string a;
  int l = 0;
  float* ac_term = new float[size * size];
  float* block = new float[block_size * block_size];
  while (getline(infile, a))
  {
    elemen[l] = a;
    l++;
  }
  infile.close();
  int i = 0;
  int index = 0;

  int idx = 0;
  int idy = 0;
  int count = 0;
  while (i < l) {
    if (count < block_size*block_size) {
      if (elemen[i] != "0") {
        block[index] = std::stoi(elemen[i]);
        count++;
        i++;
        index++;
      }
      else {
        for (int x = 0; x < std::stoi(elemen[i + 1]); x++) {

          block[index + x] = 0;
          count++;

        }
        index = index + std::stoi(elemen[i + 1]);
        i += 2;
      }
    }
    if (count == block_size * block_size) {
      //cout << idx << "    " << idy << endl;
      insert_block3(ac_term, block, idx, idy, block_size);

      idy++;
      if (idy == (size / block_size)) {
        idy = 0;
        idx++;
      }
      index = 0;
      count = 0;
    }
  }

  return ac_term;
}
```

Listing 13: run length decoding function

5.3 Discrete probability density functions (PDF)

The Listing 14 shows how to calculate the entropy (average minimum number of bits/symbole) and other interesting values.

```cpp
void count_value_rle(int size) {
  string* elem = new string[size * size];
  ifstream ac;
  ac.open("run_length_encoding_ac_decompress.txt");
  string a;
  int l = 0;
  int total_symbol = 0;

  while (getline(ac, a))
  {
    elem[l] = a;
    l++;
  }
  total_symbol = l;
  map<string, int> dict;
  map<int, int>::iterator it;
  int i = 1097; //begin from ac coeff
  while (i < l) {
    ++dict[elem[i]];
    i++;
  }
  int Histsize = dict.size();
  int totalelements = 0;
  for (auto iter = dict.begin(); iter != dict.end(); ++iter)
  {
    cout << "element: " << iter->first << "  frequency: " << iter->second << endl;
    totalelements = totalelements + (int)(iter->second);
  }
  string* elements = new string[Histsize];
  float* Probabilities = new float[Histsize];
  int n = 0;
  int intvalue;
  float Entropy = 0;
  for (auto iter = dict.begin(); iter != dict.end(); ++iter)
  {
    cout << "element: " << iter->first << "  frequency: " << iter->second;

    elements[n] = (iter->first);
    intvalue = (iter->second);
    Probabilities[n] = (float)((intvalue + 0.0) / (totalelements + 0.0));
    cout << " P(i) " << Probabilities[n] << endl;
    Entropy = Entropy - (Probabilities[n] * log2(Probabilities[n]));
    n++;
  }
  // Printing Output values
  cout << "The total number of symbols is: " << totalelements << endl;
  cout << "The Number of symbols (Histogram) is: " << Histsize << endl;
  cout << "The Average bpsymbol is: " << Entropy << endl;
  cout << "The File size is: " << ((Entropy)*totalelements) / 8.0 << " bytes" << endl;
}
```

Listing 14: count rle values function

The Listing 15 shows a part of the output of the previous code.

```
//output_value
//......
element: 0  frequency: 2994 P(i) 0.217745
element: 1  frequency: 2738 P(i) 0.199127
element: -1  frequency: 1776 P(i) 0.129164
element: 2   frequency: 1163 P(i) 0.0845818
```

```
element:  -2   frequency: 664 P(i) 0.0482909
//......
The total number of symbols is: 13750
The Number of symbols (Histogram) is: 173
The Average bpsymbol is: 3.98772
The File size is: 6853.89 bytes
```

Listing 15: output of count value rle function

The Listing 15 shows that the theoretical minimum bits to encode all symbols is 3.98772 resulting in a minimum possible file size = 6853.89 bytes. Another interesting fact, is that the number of elements -2,-1,0,1,2 constitute approximately 68% of the total numbers of symbols. The full output is shown in AppendixA. By taking the delta encoded DC coefficients into account (run length encoding for all coefficients) we obtain the following output :

```
The total number of symbols is: 14847
The Number of symbols (Histogram) is: 188
The Average bpsymbol is: 4.09941
The File size is: 7607.99 bytes
```

The max run length for AC coefficient is 1053 corresponding to the block (1,0) (and 1022 corresponding to the block (0,1)). By taking into account the DC coefficients we obtain a maximum run length of 1097 (corresponding to the block (0,0)). In conclusion, the run length encoding has good results for low frequency coefficient (with a lot of zeros) and it's degraded by high frequency coefficients.

6 WPO6

6.1 Exponential-Golomb variable-length code (VLC)

The Exponential-Golomb coding pre-assumes that the probability distribution of the symbols corresponds to a Laplacian distribution, assuming that the values close to zero are more represented than the ones far from zero. The numbers are encoded with a certain number of zeros before the significant binary value in the aim to facilitate the decoding. The Listing 18 shows the validation of the Golomb (Listing16) and inverse Golomb (Listing 17) functions. To encode a negative number, I just assigned each number to an other (if n is negative, it will be the n^{th} even number and the same with odd numbers for the non-negative). The implementation of coding and decoding follows the steps presented in `https://titanwolf.org/Network/Articles/Article?AID=52e4ca0b-f224-4f86-839b-50201928a581`

```
string golomb(int x) {
  if (x > 0) {
    x = x * 2 - 1;
  }
  else {
    x = (-x) * 2;
  }
  string string_bit;
  int y = x + 1;
  int num = floor(log2(y)) + 1;
  int m = num - 1;
  int* array_of_bit = new int[num];
  while (m > 0) {
    string_bit += std::to_string(0);
    m--;
  }
  int i = 0;
  while (y > 0) {
    array_of_bit[i] = y % 2;
    y = y / 2;
    i++;
  }
  for (int j = i - 1; j >= 0; j--) {
    string_bit += std::to_string(array_of_bit[j]);
  }
  return string_bit;
}
```

Listing 16: golomb function

```
int inv_golomb(string binary_string) {
  int x = 0;
  int offset = 0;
  int N;
  int m = 0;
  int i = 0;
  bool stop = 1;
  int temp;
  while (binary_string[i] == '0') {
    m++;
    i++;
  }
  for (int k = 0; k < m; k++) {
    if (binary_string[2 * m - k] == '1') {
      temp = 1;

    }
    else {
      temp = 0;
    }

    offset += pow(2, k) * temp;
    //cout << offset << endl;
  }
  x = pow(2, m) - 1 + offset;


  if (x % 2 == 0) {
    x = -x / 2;
  }
  else {
    x = (x + 1) / 2;
  }
  return x;
}
```

Listing 17: inverse golomb function

```

int i = -20;
for (i; i < 20; i++) {
  cout << "number:  " << i << "      codeword golomb:   " << (golomb(i)) << "    result -->
    " << inv_golomb(golomb(i)) << endl;
}

//--------output--------
number:  -20     codeword golomb:   00000101001    result -->  -20
number:  -19     codeword golomb:   00000100111    result -->  -19
number:  -18     codeword golomb:   00000100101    result -->  -18
number:  -17     codeword golomb:   00000100011    result -->  -17
number:  -16     codeword golomb:   00000100001    result -->  -16
number:  -15     codeword golomb:   000011111    result -->  -15
number:  -14     codeword golomb:   000011101    result -->  -14
number:  -13     codeword golomb:   000011011    result -->  -13
number:  -12     codeword golomb:   000011001    result -->  -12
number:  -11     codeword golomb:   000010111    result -->  -11
number:  -10     codeword golomb:   000010101    result -->  -10
number:  -9     codeword golomb:   000010011    result -->  -9
```

```
number:  -8     codeword golomb:   000010001    result -->  -8
number:  -7     codeword golomb:   0001111    result -->  -7
number:  -6     codeword golomb:   0001101    result -->  -6
number:  -5     codeword golomb:   0001011    result -->  -5
number:  -4     codeword golomb:   0001001    result -->  -4
number:  -3     codeword golomb:   00111    result -->  -3
number:  -2     codeword golomb:   00101    result -->  -2
number:  -1     codeword golomb:   011    result -->  -1
number:  0     codeword golomb:   1    result -->  0
number:  1     codeword golomb:   010    result -->  1
number:  2     codeword golomb:   00100    result -->  2
number:  3     codeword golomb:   00110    result -->  3
number:  4     codeword golomb:   0001000    result -->  4
number:  5     codeword golomb:   0001010    result -->  5
number:  6     codeword golomb:   0001100    result -->  6
number:  7     codeword golomb:   0001110    result -->  7
number:  8     codeword golomb:   000010000    result -->  8
number:  9     codeword golomb:   000010010    result -->  9
number:  10     codeword golomb:   000010100    result -->  10
number:  11     codeword golomb:   000010110    result -->  11
number:  12     codeword golomb:   000011000    result -->  12
number:  13     codeword golomb:   000011010    result -->  13
number:  14     codeword golomb:   000011100    result -->  14
number:  15     codeword golomb:   000011110    result -->  15
number:  16     codeword golomb:   00000100000    result -->  16
number:  17     codeword golomb:   00000100010    result -->  17
number:  18     codeword golomb:   00000100100    result -->  18
number:  19     codeword golomb:   00000100110    result -->  19
```

Listing 18: Validation of golomb function

6.2 Lossy grayscale image compression

The Listings 19 and 20 summarize all functions detailed in the previous sections. Finally, the decompressed image is found back with MSE = 22.67 and PSNR = 34.58 (due to the threshold of 10 applied on the DCT coefficients and also due to the quantization). The compressed file has a size of 96.733 octets which gives us a compression rate ≈ 2.7.

```cpp
void compress(float* img, int img_size) {
  const int block_size = 8;
  const float* basis = get_basis(block_size);
  float* img_dc_term = new float[(img_size / block_size) * (img_size / block_size)];
  float* img_ac_term = new float[(img_size) * (img_size)];
  for (int x = 0; x < (img_size / block_size); x++) {
    for (int y = 0; y < (img_size / block_size); y++) {
      float* block = dicing_window(img, x * block_size, y * block_size, block_size);
      float* block_approx = encode(block, basis, Q, block_size);
      insert_block1(img_ac_term, block_approx, x, y, block_size, img_size);

    }
  }
  Store(img_ac_term, "img_ac_term.raw", img_size);
  img_dc_term = dicing_window(img_ac_term, 0, 0, img_size / block_size);
  float*img_dc_term_delta = delta_encoding(img_dc_term, (img_size / block_size));
  insert_block(img_ac_term, img_dc_term_delta, 0, 0, img_size / block_size);
  Store(img_ac_term, "img_ac_term_dc_delta.raw", img_size);
  run_length_encoding(img_ac_term, img_size,(img_size/ block_size));
  string* elemen = new string[img_size * img_size];
  ifstream infile;
  infile.open("run_length_encoding_ac_decompress.txt");
  string a;
  int i = 0;
  int l = 0;
  int ac_term = 0;
  ofstream myfile("compressed.txt");
  while (getline(infile, a))
  {
    elemen[l] = a;
    l++;
  }
  infile.close();
  while (i < l) {
    ac_term = std::stoi(elemen[i]);
    i++;
    myfile << golomb(ac_term) << endl;
  }

  myfile.close();


}
```

Listing 19: compress function

```
float* decompress(int img_size,int  block_size) {
  ifstream infile;
  string a;
  string* elemen = new string[img_size * img_size];
  int i = 0;
  int l = 0;
  infile.open("compressed.txt");
  while (getline(infile, a))
  {
    elemen[l] = a;
    l++;
  }
  infile.close();
  ofstream myfile("run_length_encoding_ac_decompress.txt");
  while (i < l) {
    myfile << inv_golomb(elemen[i]) << endl;
    i++;
  }
  myfile.close();
  float* img_ac_term_reconstruct = run_length_decoding(img_size, img_size/block_size);
  Store(img_ac_term_reconstruct, "img_ac_term_reconstruct_without_delta_decoding.raw",
    img_size);
  float* dc_block = dicing_window(img_ac_term_reconstruct, 0, 0, img_size / block_size);
  Store(dc_block, "dc_block.raw", img_size / block_size);
  float* dc_block_decoded = delta_decoding2(dc_block, img_size / block_size);
  insert_block(img_ac_term_reconstruct, dc_block_decoded, 0, 0, img_size / block_size);
  Store(img_ac_term_reconstruct, "img_ac_term_reconstruct_with_delta_decoding.raw", img_size
    );
  float* layout_for_idct = new float[img_size * img_size];
  //refine original form 8x8 block
  for (int x = 0; x < block_size; x++) {
    for (int y = 0; y <  block_size; y++) {
      float* block = dicing_window(img_ac_term_reconstruct, x * img_size / block_size, y *
    img_size / block_size, img_size/block_size);
      insert_block1(layout_for_idct, block, x, y, img_size/block_size, img_size);
    }
  }
  Store(layout_for_idct, "layout_for_idct.raw", img_size);
  //decode
  float* img_retrivial = new float[img_size * img_size];
  for (int x = 0; x < (img_size / block_size); x++) {
    for (int y = 0; y < (img_size / block_size); y++) {
      float* block = dicing_window(layout_for_idct, x * block_size, y * block_size,
    block_size);
```

```
      float* block_approx = decode(block,get_basis(block_size), Q, block_size);
      insert_block(img_retrivial, block_approx, x, y, block_size);

    }
  }
  return img_retrivial;
}
```

Listing 20: decompress function

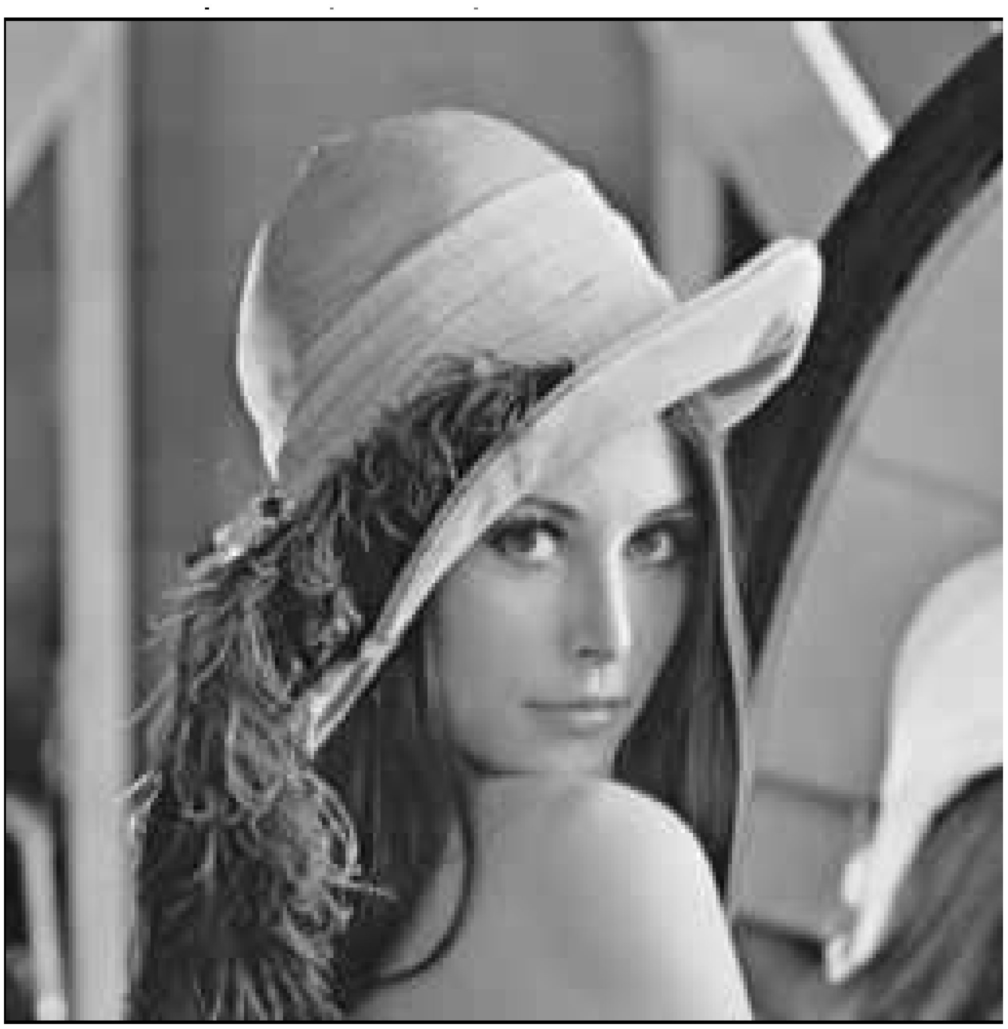


Figure 21: Visualization of decompressed image

Appendix

A output values

```

element: -1  frequency: 1776 P(i) 0.129164
element: -10  frequency: 48 P(i) 0.00349091
element: -11  frequency: 30 P(i) 0.00218182
element: -12  frequency: 26 P(i) 0.00189091
element: -13  frequency: 19 P(i) 0.00138182
element: -14  frequency: 21 P(i) 0.00152727
element: -15  frequency: 18 P(i) 0.00130909
element: -16  frequency: 21 P(i) 0.00152727
element: -17  frequency: 10 P(i) 0.000727273
element: -18  frequency: 20 P(i) 0.00145455
element: -19  frequency: 18 P(i) 0.00130909
element: -2  frequency: 664 P(i) 0.0482909
element: -20  frequency: 6 P(i) 0.000436364
element: -21  frequency: 8 P(i) 0.000581818
element: -22  frequency: 11 P(i) 0.0008
element: -23  frequency: 4 P(i) 0.000290909
element: -24  frequency: 6 P(i) 0.000436364
element: -25  frequency: 6 P(i) 0.000436364
element: -26  frequency: 5 P(i) 0.000363636
element: -27  frequency: 3 P(i) 0.000218182
element: -28  frequency: 3 P(i) 0.000218182
element: -29  frequency: 5 P(i) 0.000363636
element: -3  frequency: 404 P(i) 0.0293818
element: -30  frequency: 6 P(i) 0.000436364
element: -31  frequency: 3 P(i) 0.000218182
element: -32  frequency: 1 P(i) 7.27273e-05
element: -34  frequency: 1 P(i) 7.27273e-05
element: -35  frequency: 1 P(i) 7.27273e-05
element: -36  frequency: 2 P(i) 0.000145455
element: -37  frequency: 1 P(i) 7.27273e-05
element: -38  frequency: 3 P(i) 0.000218182
element: -39  frequency: 3 P(i) 0.000218182
element: -4  frequency: 239 P(i) 0.0173818
element: -40  frequency: 1 P(i) 7.27273e-05
element: -41  frequency: 2 P(i) 0.000145455
element: -44  frequency: 1 P(i) 7.27273e-05
element: -45  frequency: 1 P(i) 7.27273e-05
element: -48  frequency: 1 P(i) 7.27273e-05
element: -5  frequency: 186 P(i) 0.0135273
element: -6  frequency: 120 P(i) 0.00872727
element: -7  frequency: 94 P(i) 0.00683636
element: -8  frequency: 81 P(i) 0.00589091
element: -9  frequency: 59 P(i) 0.00429091
element: 0  frequency: 2994 P(i) 0.217745
element: 1  frequency: 2738 P(i) 0.199127
element: 10  frequency: 116 P(i) 0.00843636
element: 100  frequency: 1 P(i) 7.27273e-05
element: 101  frequency: 1 P(i) 7.27273e-05
element: 1024  frequency: 20 P(i) 0.00145455
element: 103  frequency: 1 P(i) 7.27273e-05
element: 106  frequency: 2 P(i) 0.000145455
element: 107  frequency: 1 P(i) 7.27273e-05
element: 109  frequency: 1 P(i) 7.27273e-05
element: 11  frequency: 68 P(i) 0.00494545
element: 113  frequency: 1 P(i) 7.27273e-05
element: 118  frequency: 1 P(i) 7.27273e-05
element: 12  frequency: 68 P(i) 0.00494545
element: 120  frequency: 2 P(i) 0.000145455
element: 121  frequency: 1 P(i) 7.27273e-05
element: 122  frequency: 1 P(i) 7.27273e-05
element: 123  frequency: 1 P(i) 7.27273e-05
element: 124  frequency: 1 P(i) 7.27273e-05
element: 127  frequency: 1 P(i) 7.27273e-05
element: 13  frequency: 47 P(i) 0.00341818
```

```
element: 14  frequency: 54 P(i) 0.00392727
element: 142  frequency: 1 P(i) 7.27273e-05
element: 147  frequency: 2 P(i) 0.000145455
element: 149  frequency: 2 P(i) 0.000145455
element: 15  frequency: 48 P(i) 0.00349091
element: 150  frequency: 1 P(i) 7.27273e-05
element: 152  frequency: 3 P(i) 0.000218182
element: 153  frequency: 4 P(i) 0.000290909
element: 154  frequency: 1 P(i) 7.27273e-05
element: 155  frequency: 1 P(i) 7.27273e-05
element: 156  frequency: 1 P(i) 7.27273e-05
element: 16  frequency: 43 P(i) 0.00312727
element: 17  frequency: 64 P(i) 0.00465455
element: 18  frequency: 58 P(i) 0.00421818
element: 180  frequency: 1 P(i) 7.27273e-05
element: 181  frequency: 1 P(i) 7.27273e-05
element: 183  frequency: 1 P(i) 7.27273e-05
element: 19  frequency: 40 P(i) 0.00290909
element: 196  frequency: 1 P(i) 7.27273e-05
element: 2  frequency: 1163 P(i) 0.0845818
element: 20  frequency: 30 P(i) 0.00218182
element: 208  frequency: 1 P(i) 7.27273e-05
element: 209  frequency: 1 P(i) 7.27273e-05
element: 21  frequency: 24 P(i) 0.00174545
element: 210  frequency: 1 P(i) 7.27273e-05
element: 212  frequency: 2 P(i) 0.000145455
element: 213  frequency: 1 P(i) 7.27273e-05
element: 214  frequency: 2 P(i) 0.000145455
element: 22  frequency: 25 P(i) 0.00181818
element: 223  frequency: 2 P(i) 0.000145455
element: 23  frequency: 26 P(i) 0.00189091
element: 233  frequency: 1 P(i) 7.27273e-05
element: 24  frequency: 21 P(i) 0.00152727
element: 242  frequency: 1 P(i) 7.27273e-05
element: 243  frequency: 1 P(i) 7.27273e-05
element: 245  frequency: 1 P(i) 7.27273e-05
element: 25  frequency: 16 P(i) 0.00116364
element: 26  frequency: 23 P(i) 0.00167273
element: 263  frequency: 1 P(i) 7.27273e-05
element: 269  frequency: 1 P(i) 7.27273e-05
element: 27  frequency: 13 P(i) 0.000945455
element: 271  frequency: 1 P(i) 7.27273e-05
element: 275  frequency: 1 P(i) 7.27273e-05
element: 276  frequency: 1 P(i) 7.27273e-05
element: 28  frequency: 14 P(i) 0.00101818
element: 284  frequency: 1 P(i) 7.27273e-05
element: 29  frequency: 14 P(i) 0.00101818
element: 3  frequency: 653 P(i) 0.0474909
element: 30  frequency: 18 P(i) 0.00130909
element: 31  frequency: 20 P(i) 0.00145455
element: 312  frequency: 1 P(i) 7.27273e-05
element: 316  frequency: 1 P(i) 7.27273e-05
element: 318  frequency: 1 P(i) 7.27273e-05
element: 32  frequency: 7 P(i) 0.000509091
element: 33  frequency: 8 P(i) 0.000581818
element: 336  frequency: 1 P(i) 7.27273e-05
element: 339  frequency: 1 P(i) 7.27273e-05
element: 34  frequency: 5 P(i) 0.000363636
element: 347  frequency: 1 P(i) 7.27273e-05
element: 35  frequency: 6 P(i) 0.000436364
element: 36  frequency: 4 P(i) 0.000290909
element: 38  frequency: 1 P(i) 7.27273e-05
element: 39  frequency: 2 P(i) 0.000145455
element: 4  frequency: 405 P(i) 0.0294545
element: 40  frequency: 1 P(i) 7.27273e-05
```

```
element: 41  frequency: 1 P(i) 7.27273e-05
element: 42  frequency: 3 P(i) 0.000218182
element: 43  frequency: 1 P(i) 7.27273e-05
element: 432  frequency: 1 P(i) 7.27273e-05
element: 433  frequency: 1 P(i) 7.27273e-05
element: 44  frequency: 2 P(i) 0.000145455
element: 45  frequency: 2 P(i) 0.000145455
element: 46  frequency: 1 P(i) 7.27273e-05
element: 47  frequency: 1 P(i) 7.27273e-05
element: 48  frequency: 3 P(i) 0.000218182
element: 49  frequency: 1 P(i) 7.27273e-05
element: 499  frequency: 1 P(i) 7.27273e-05
element: 5  frequency: 273 P(i) 0.0198545
element: 50  frequency: 3 P(i) 0.000218182
element: 51  frequency: 1 P(i) 7.27273e-05
element: 52  frequency: 2 P(i) 0.000145455
element: 53  frequency: 2 P(i) 0.000145455
element: 54  frequency: 2 P(i) 0.000145455
element: 56  frequency: 2 P(i) 0.000145455
element: 57  frequency: 1 P(i) 7.27273e-05
element: 58  frequency: 2 P(i) 0.000145455
element: 59  frequency: 4 P(i) 0.000290909
element: 590  frequency: 1 P(i) 7.27273e-05
element: 6  frequency: 206 P(i) 0.0149818
element: 60  frequency: 5 P(i) 0.000363636
element: 61  frequency: 3 P(i) 0.000218182
element: 62  frequency: 1 P(i) 7.27273e-05
element: 63  frequency: 2 P(i) 0.000145455
element: 644  frequency: 2 P(i) 0.000145455
element: 69  frequency: 1 P(i) 7.27273e-05
element: 7  frequency: 146 P(i) 0.0106182
element: 76  frequency: 1 P(i) 7.27273e-05
element: 77  frequency: 1 P(i) 7.27273e-05
element: 8  frequency: 104 P(i) 0.00756364
element: 800  frequency: 1 P(i) 7.27273e-05
element: 82  frequency: 2 P(i) 0.000145455
element: 84  frequency: 1 P(i) 7.27273e-05
element: 87  frequency: 1 P(i) 7.27273e-05
element: 871  frequency: 2 P(i) 0.000145455
element: 89  frequency: 1 P(i) 7.27273e-05
element: 9  frequency: 98 P(i) 0.00712727
element: 91  frequency: 2 P(i) 0.000145455
element: 92  frequency: 1 P(i) 7.27273e-05
element: 93  frequency: 1 P(i) 7.27273e-05
The total number of symbols is: 13750
The Number of symbols (Histogram) is: 173
The Average bpsymbol is: 3.98772
The File size is: 6853.89 bytes
```