영상처리

HW #04

2014253005 박세현

The purpose of this assignment is to implement DCT algorithm for a given still image. In order to evaluate the performance, we can compare the subjective quality reconstructed by inverse transform as well as the objective criterion such as MSE (mean square error). Furthermore, we can exploit the frequency spectrum in which the energy compaction performance could be compared. For this purpose, carry out the following assignments.

1. For the given 512x512 image ‘Lena’,

(a) Perform 8x8 forward DCT and plot the frequency spectrum on the monitor in proper scale for easy observation.

(b) Perform 8x8 inverse DCT to obtain the reconstructed image. Compute the MSE.

[소스코드]

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iostream>
#include <random>
#include <string>
#include <windows.h>
#include <wingdi.h>
#define MAX 512
#define B_SIZE 8
#define pi 3.141592653589793238
#define RADIAN 40
#define nint(x) ((x) < 0. ? (int)((x)-0.5) : (int)((x) + 0.5))
using namespace std;

typedef struct headers {
  BITMAPFILEHEADER hFile;
  BITMAPINFOHEADER hInfo;
  RGBQUAD hRGB[256];
} BITMAPHEADERS;

void generate_headers(BITMAPHEADERS &bh) {
  bh.hFile.bfType = 0x4D42;
  bh.hFile.bfReserved1 = 0;
  bh.hFile.bfReserved2 = 0;
  bh.hFile.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER) +
                       sizeof(RGBQUAD) * 256;
  bh.hFile.bfSize = bh.hFile.bfOffBits + MAX * MAX;

  bh.hInfo.biSize = 40;
  bh.hInfo.biWidth = MAX;
  bh.hInfo.biHeight = MAX;
  bh.hInfo.biPlanes = 1;
  bh.hInfo.biBitCount = 8;
  bh.hInfo.biCompression = 0;
  bh.hInfo.biSizeImage = MAX * MAX;
  bh.hInfo.biXPelsPerMeter = 0;
  bh.hInfo.biYPelsPerMeter = 0;
  bh.hInfo.biClrUsed = 0;
  bh.hInfo.biClrImportant = 0;

  for (int i = 0; i < 256; i++) {
    bh.hRGB[i].rgbBlue = i;
    bh.hRGB[i].rgbGreen = i;
    bh.hRGB[i].rgbRed = i;
    bh.hRGB[i].rgbReserved = 0;
  }
}

void reverse_raw_data(BYTE *image) {
  for (int i = 0; i < MAX; i++) {
    for (int j = 0; j < MAX; j++) {
      if (i < MAX / 2) {
        swap(image[i * MAX + j], image[(MAX - i - 1) * MAX + j]);
      }
    }
  }
}

void make_bmp(BYTE *output_image, string output_name) {
  reverse_raw_data(output_image);

  BITMAPHEADERS bh;
  generate_headers(bh);
  string PATH = "outputs/" + output_name + ".bmp";
  FILE *output_file = fopen(PATH.c_str(), "wb");

  fwrite(&bh.hFile, sizeof(BITMAPFILEHEADER), 1, output_file);
  fwrite(&bh.hInfo, sizeof(BITMAPINFOHEADER), 1, output_file);
  fwrite(bh.hRGB, sizeof(RGBQUAD), 256, output_file);

  fwrite(output_image, sizeof(BYTE), MAX * MAX, output_file);
  // fclose(output_file);
  return;
}
```

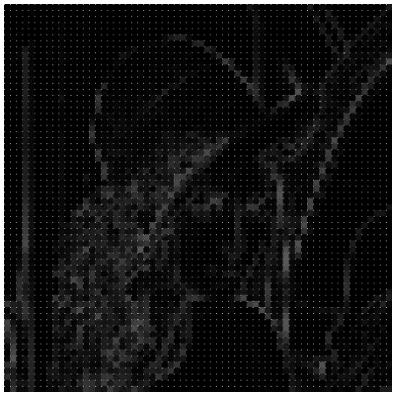
```
void LUT_based_DCT(double ix[][B_SIZE]) {
  double x[B_SIZE][B_SIZE], z[B_SIZE][B_SIZE], y[B_SIZE], yy[B_SIZE];
  double c[RADIAN], s[RADIAN], ft[4], fxy[4], zz;

  // cos, sin 값 미리 계산
  for (int i = 0; i < RADIAN; i++) {
    zz = pi * (double)(i + 1) / 64.0;
    c[i] = cos(zz);
    s[i] = sin(zz);
  }

  // x에 ix에 저장된 값들을 B_SIZE * B_SIZE 만큼 떼어와 저장한다.
  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      x[i][j] = (double)ix[i][j];
    }
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      y[j] = x[i][j];
    }

    for (int j = 0; j < 4; j++) {
      ft[j] = y[j] + y[7 - j];
    }

    fxy[0] = ft[0] + ft[3];
    fxy[1] = ft[1] + ft[2];
    fxy[2] = ft[1] - ft[2];
    fxy[3] = ft[0] - ft[3];

    ft[0] = c[15] * (fxy[0] + fxy[1]);
    ft[2] = c[15] * (fxy[0] - fxy[1]);
    ft[1] = s[7] * fxy[2] + c[7] * fxy[3];
    ft[3] = -s[23] * fxy[2] + c[23] * fxy[3];

    for (int j = 4; j < 8; j++) {
      yy[j] = y[7 - j] - y[j];
    }

    y[4] = yy[4];
    y[7] = yy[7];
    y[5] = c[15] * (-yy[5] + yy[6]);
    y[6] = c[15] * (yy[5] + yy[6]);

    yy[4] = y[4] + y[5];
    yy[5] = y[4] - y[5];
    yy[6] = -y[6] + y[7];
    yy[7] = y[6] + y[7];

    y[0] = ft[0];
    y[4] = ft[2];
    y[2] = ft[1];
    y[6] = ft[3];
    y[1] = s[3] * yy[4] + c[3] * yy[7];
    y[5] = s[19] * yy[5] + c[19] * yy[6];
    y[3] = -s[11] * yy[5] + c[11] * yy[6];
    y[7] = -s[27] * yy[4] + c[27] * yy[7];

    for (int j = 0; j < B_SIZE; j++) {
      z[i][j] = y[j];
    }
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      y[j] = z[j][i];
    }

    for (int j = 0; j < 4; j++) {
      ft[j] = y[j] + y[7 - j];
    }

    fxy[0] = ft[0] + ft[3];
    fxy[1] = ft[1] + ft[2];
    fxy[2] = ft[1] - ft[2];
    fxy[3] = ft[0] - ft[3];

    ft[0] = c[15] * (fxy[0] + fxy[1]);
    ft[2] = c[15] * (fxy[0] - fxy[1]);
    ft[1] = s[7] * fxy[2] + c[7] * fxy[3];
    ft[3] = -s[23] * fxy[2] + c[23] * fxy[3];

    for (int j = 4; j < 8; j++) {
      yy[j] = y[7 - j] - y[j];
    }

    y[4] = yy[4];
    y[7] = yy[7];
    y[5] = c[15] * (-yy[5] + yy[6]);
    y[6] = c[15] * (yy[5] + yy[6]);

    yy[4] = y[4] + y[5];
    yy[5] = y[4] - y[5];
    yy[6] = -y[6] + y[7];
    yy[7] = y[6] + y[7];

    y[0] = ft[0];
    y[4] = ft[2];
    y[2] = ft[1];
    y[6] = ft[3];
    y[1] = s[3] * yy[4] + c[3] * yy[7];
    y[5] = s[19] * yy[5] + c[19] * yy[6];
    y[3] = -s[11] * yy[5] + c[11] * yy[6];
    y[7] = -s[27] * yy[4] + c[27] * yy[7];

    for (int j = 0; j < B_SIZE; j++) {
      y[j] = y[j] / 4.0;
    }

    for (int j = 0; j < B_SIZE; j++) {
      z[j][i] = y[j];
    }
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      ix[i][j] = nint(z[i][j]);
    }
  }
}
```

```
void LUT_based_IDCT(double ix[][B_SIZE]) {
  double x[B_SIZE][B_SIZE], z[B_SIZE][B_SIZE], y[B_SIZE], yy[B_SIZE];
  double c[RADIAN], s[RADIAN], ait[4], aixy[4], zz;

  for (int i = 0; i < RADIAN; i++) {
    zz = pi * (double)(i + 1) / 64.0;
    c[i] = cos(zz);
    s[i] = sin(zz);
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      x[i][j] = (double)ix[i][j];
    }
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      y[j] = x[j][i];
    }

    ait[0] = y[0];
    ait[1] = y[2];
    ait[2] = y[4];
    ait[3] = y[6];

    aixy[0] = c[15] * (ait[0] + ait[2]);
    aixy[1] = c[15] * (ait[0] - ait[2]);
    aixy[2] = s[7] * ait[1] - s[23] * ait[3];
    aixy[3] = c[7] * ait[1] + c[23] * ait[3];

    ait[0] = aixy[0] + aixy[3];
    ait[1] = aixy[1] + aixy[2];
    ait[2] = aixy[1] - aixy[2];
    ait[3] = aixy[0] - aixy[3];

    yy[4] = s[3] * y[1] - s[27] * y[7];
    yy[5] = s[19] * y[5] - s[11] * y[3];
    yy[6] = c[19] * y[5] + c[11] * y[3];
    yy[7] = c[3] * y[1] + c[27] * y[7];

    y[4] = yy[4] + yy[5];
    y[5] = yy[4] - yy[5];
    y[6] = -yy[6] + yy[7];
    y[7] = yy[6] + yy[7];

    yy[4] = y[4];
    yy[7] = y[7];
    yy[5] = c[15] * (-y[5] + y[6]);
    yy[6] = c[15] * (y[5] + y[6]);

    for (int j = 0; j < 4; j++) {
      y[j] = ait[j] + yy[7 - j];
    }

    for (int j = 4; j < 8; j++) {
      y[j] = ait[7 - j] - yy[j];
    }

    for (int j = 0; j < B_SIZE; j++) {
      z[j][i] = y[j];
    }
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      y[j] = z[i][j];
    }

    ait[0] = y[0];
    ait[1] = y[2];
    ait[2] = y[4];
    ait[3] = y[6];

    aixy[0] = c[15] * (ait[0] + ait[2]);
    aixy[1] = c[15] * (ait[0] - ait[2]);
    aixy[2] = s[7] * ait[1] - s[23] * ait[3];
    aixy[3] = c[7] * ait[1] + c[23] * ait[3];

    ait[0] = aixy[0] + aixy[3];
    ait[1] = aixy[1] + aixy[2];
    ait[2] = aixy[1] - aixy[2];
    ait[3] = aixy[0] - aixy[3];

    yy[4] = s[3] * y[1] - s[27] * y[7];
    yy[5] = s[19] * y[5] - s[11] * y[3];
    yy[6] = c[19] * y[5] + c[11] * y[3];
    yy[7] = c[3] * y[1] + c[27] * y[7];

    y[4] = yy[4] + yy[5];
    y[5] = yy[4] - yy[5];
    y[6] = -yy[6] + yy[7];
    y[7] = yy[6] + yy[7];

    yy[4] = y[4];
    yy[7] = y[7];
    yy[5] = c[15] * (-y[5] + y[6]);
    yy[6] = c[15] * (y[5] + y[6]);

    for (int j = 0; j < 4; j++) {
      y[j] = ait[j] + yy[7 - j];
    }

    for (int j = 4; j < 8; j++) {
      y[j] = ait[7 - j] - yy[j];
    }

    for (int j = 0; j < B_SIZE; j++) {
      z[i][j] = y[j] / 4.0;
    }
  }

  for (int i = 0; i < B_SIZE; i++) {
    for (int j = 0; j < B_SIZE; j++) {
      // ix[i][j] = ceil(z[i][j]);
      ix[i][j] = nint(z[i][j]);
    }
  }
}
```

```c
int main() {
  FILE *input_file = fopen("lena_raw_512x512.raw", "rb");
  if (input_file == NULL) {
    printf("FILE ERROR\n");
    return 1;
  }

  BYTE *image = (BYTE *)malloc(sizeof(BYTE) * MAX * MAX);
  double *transformed_image = (double *)malloc(sizeof(double) * MAX * MAX);
  BYTE *DCT_image = (BYTE *)malloc(sizeof(BYTE) * MAX * MAX);
  BYTE *restored_image = (BYTE *)malloc(sizeof(BYTE) * MAX * MAX);

  size_t n_size = fread(image, sizeof(BYTE), MAX * MAX, input_file);
  fclose(input_file);

  for (int i = 0; i < MAX; i += B_SIZE) {
    for (int j = 0; j < MAX; j += B_SIZE) {
      double copied[B_SIZE][B_SIZE];
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          copied[a - i][b - j] = image[MAX * a + b];
        }
      }
      LUT_based_DCT(copied);
      double max_val = -2048.0;
      double min_val = 2048.0;
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          transformed_image[MAX * a + b] = copied[a - i][b - j];
          max_val = max(max_val, copied[a - i][b - j]);
          min_val = min(min_val, copied[a - i][b - j]);
        }
      }

      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          double temp = (copied[a - i][b - j] - min_val) * 255;
          temp /= (max_val - min_val);
          if (temp > 255) {
            temp = 255;
          } else if (temp < 0) {
            temp = 0;
          }
          DCT_image[MAX * a + b] = temp;
        }
      }
    }
  }

  for (int i = 0; i < MAX; i += B_SIZE) {
    for (int j = 0; j < MAX; j += B_SIZE) {
      double copied[B_SIZE][B_SIZE];
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          copied[a - i][b - j] = transformed_image[MAX * a + b];
        }
      }
      LUT_based_IDCT(copied);
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          restored_image[MAX * a + b] = ceil(copied[a - i][b - j]);
        }
      }
    }
  }

  make_bmp(DCT_image, "DCT_image_Lena_CPP");
  make_bmp(restored_image, "Restored_DCT_Lena_CPP");

  // MSE값을 구한다.
  long long int nTmp = 0;
  double dmse = 0;
  for (int i = 0; i < MAX * MAX; i++) {
    nTmp += (image[i] - restored_image[i]) * (image[i] - restored_image[i]);
  }

  dmse = (double)nTmp / (MAX * MAX);
  printf("MSE 값 : %lf\n", dmse);

  return 0;
}
```

[실행 결과]

[그림 1-1]



[그림 1-2]

```
MSE 값 : 3709.364647
PS E:\User\Code\ImageProcessing_HW\HW04_vscode>
```

먼저 대부분의 소스코드는 이전 HW들과 유사하다. DCT와 IDCT에 대한 코드는 과제의 부록으로 첨부된 코드를 이용하였다. 부록의 DCT, IDCT 코드는 8*8 크기를 대상으로 DCT를 수행한다. 따라서 512*512 크기의 DCT와 IDCT를 진행하기 전에 8*8 크기로 잘라서 입력으로 보내주어야 한다. 여기서는 8*8 블록이라 부른다.

main에서 8*8 크기의 double[][]의 copied를 선언하고 여기에 원본 영상을 8*8블록만큼 복사하여 저장한다. 그리고 copied를 DCT 함수의 인자로 넘겨주어 DCT를 수행하고, 그 결과를 copied에 갱신한다. 갱신된 copied의 값을 저장을 하는데 이때 미리 선언해둔 원본과 같은 사이즈를 가진 double* 타입의 transformed_image에 원본 이미지에서 8*8 블록을 가져온 곳에 해당하는 위치에 옮겨준다. 8*8블록을 가져오는 곳을 옮겨가며 반복하여 512*512크기의 이미지에 모두 DCT를 수행한다.

IDCT를 하는 방식은 DCT를 수행하는 방식과 같다. IDCT 역시 8*8 블록을 만들고 이를 IDCT 함수의 인자로 넘겨준다. 이후 갱신된 값을 원본 이미지에서 8*8블록을 가져온 위치에 해당하는 BYTE* 타입의 restored_image에 저장한다. 소스 코드의 수행 결과 [그림 1-1] Lena의 모습이 원본 영상을 DCT와 IDCT를 한 후 복원되어 나타난 이미지이다. 그 과정에서 mse의 값이 3709로 계산되었다.

transformed_image을 바로 이미지로 만들면 원래의 형태를 잘 알아볼 수 없으므로 형태를 알아보기 쉽게 하기 위해 재구성하는 작업을 진행하였다. DCT를 수행하여 나온 8*8블록을 단위로 평활화 작업을 진행하여 0 ~ 255 사이의 값으로 만들었고, BYTE* 타입의 DCT_image에 저장하였다. 이후 이미지의 모든 픽셀에 대해 작업이 끝난 다음 DCT_image를 bmp 파일로 만든 것이 [그림 1-2]이다.

[그림 1-2]에서 흰 부분이 8*8블록의 왼쪽 상단에 모이게 되는 것을 확인할 수 있고, 우측 하단으로 갈수록 어두워지는 것을 볼 수 있다. 이는 중요한 정보를 갖고 있는 저주파 영역을 왼쪽 상단으로 보내고 상대적으로 불필요한 정보인 고주파 영역은 우측 하단으로 보내는 DCT의 효과를 뚜렷하게 볼 수 있다.

2. For the given 512x512 image ‘Boat’,

(a) Perform 8x8 forward DCT and plot the frequency spectrum on the monitor in proper scale for easy observation.

(b) Perform 8x8 inverse DCT to obtain the reconstructed image. Compute the MSE.

[소스 코드]

```c
int main() {
  FILE *input_file = fopen("BOAT512.raw", "rb");
  if (input_file == NULL) {
    printf("FILE ERROR\n");
    return 1;
  }

  BYTE *image = (BYTE *)malloc(sizeof(BYTE) * MAX * MAX);
  double *transformed_image = (double *)malloc(sizeof(double) * MAX * MAX);
  BYTE *DCT_image = (BYTE *)malloc(sizeof(BYTE) * MAX * MAX);
  BYTE *restored_image = (BYTE *)malloc(sizeof(BYTE) * MAX * MAX);

  size_t n_size = fread(image, sizeof(BYTE), MAX * MAX, input_file);
  fclose(input_file);

  for (int i = 0; i < MAX; i += B_SIZE) {
    for (int j = 0; j < MAX; j += B_SIZE) {
      double copied[B_SIZE][B_SIZE];
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          copied[a - i][b - j] = image[MAX * a + b];
        }
      }
      LUT_based_DCT(copied);
      double max_val = -2048.0;
      double min_val = 2048.0;
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          transformed_image[MAX * a + b] = copied[a - i][b - j];
          max_val = max(max_val, copied[a - i][b - j]);
          min_val = min(min_val, copied[a - i][b - j]);
        }
      }

      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          double temp = (copied[a - i][b - j] - min_val) * 255;
          temp /= (max_val - min_val);
          if (temp > 255) {
            temp = 255;
          } else if (temp < 0) {
            temp = 0;
          }
          DCT_image[MAX * a + b] = temp;
        }
      }
    }
  }

  for (int i = 0; i < MAX; i += B_SIZE) {
    for (int j = 0; j < MAX; j += B_SIZE) {
      double copied[B_SIZE][B_SIZE];
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          copied[a - i][b - j] = transformed_image[MAX * a + b];
        }
      }
      LUT_based_IDCT(copied);
      for (int a = i; a < i + B_SIZE; a++) {
        for (int b = j; b < j + B_SIZE; b++) {
          restored_image[MAX * a + b] = ceil(copied[a - i][b - j]);
        }
      }
    }
  }

  make_bmp(DCT_image, "DCT_image_BOAT512_CPP");
  make_bmp(restored_image, "Restored_DCT_BOAT512_CPP");

  // MSE값을 구한다.
  long long int nTmp = 0;
  double dmse = 0;
  for (int i = 0; i < MAX * MAX; i++) {
    nTmp += (image[i] - restored_image[i]) * (image[i] - restored_image[i]);
  }

  dmse = (double)nTmp / (MAX * MAX);
  printf("MSE 값 : %lf\n", dmse);

  return 0;
}
```

[그림 2-1]

[그림 2-2]

```
MSE 값 : 8427.397522
PS E:\User\Code\ImageProcessing_HW\HW04_vscode>
```

Boat512.raw의 경우에는 위에서 진행한 Lena.raw를 이용하여 수행한 것과 동일하게 진행되었다.

코드의 수행 결과로 원본 이미지를 DCT와 IDCT를 수행한 다음 [그림 2-1]과 같이 복원되어 나타난다. mse는 8427로 출력되며, 원본 이미지를 DCT로 주파수 영역으로 변환하였을 때의 상황을 이해하기 편하게 이미지로 재구성한 것이 [그림 2-2]이다. 이 경우 역시 이미지를 통해 DCT의 효과를 명확하게 이해할 수 있다.

3. Discuss the results.

처음 과제를 수행하였을 때 C++을 이용하여 작성하였으나 문제를 다르게 이해하여 Python으로 다시 작성하였다. 그과정에서 같은 코드를 C++로 수행하였을 때와 Python으로 수행하였을 때 mse의 차이가 크게 나타난 것을 확인할 수 있었다.

```
[Running] python -u "e:\User\Code\ImageProcessing_HW\HW04_vscode\Problem_01.py"
DCT start
DCT clear
IDCT start
IDCT clear
0.08361053466796875
```

[Lena]

```
[Running] python -u "e:\User\Code\ImageProcessing_HW\HW04_vscode\Problem_02.py"
DCT start
DCT clear
IDCT start
IDCT clear
0.08406829833984375
```

[Boat]

C++로 작성한 경우와 Python을 이용하여 작성하였을 때 mse의 차이가 크게 나타났다. 이러한 차이는 자료형에 따른 문제로 확인할 수 있었다. Python으로 작성한 경우에 restored_image에 저장된 값이 float형으로 저장된다. 반면에 C++로 작성한 경우에는 BYTE 타입으로 선언하였기 때문에 저장을 하는 과정에서 자료형 변환이 발생하면서 손실이 발생한다. 실제로 C++에서 restored_image를 실수 자료형으로 선언하고 이를 이용하여 mse를 계산하는 경우 Python에서 계산한 결과값과 유사하게 출력된다.

과제의 결과로 DCT를 수행하고 IDCT를 수행하였을 때 이론상으로는 오차가 발생하지 않아야 하지만 소수점 계산 문제로 오차가 발생하게되는 것을 확인할 수 있었다. 그리고 그 오차는 실수 자료형에서 BYTE로 형변환 하면서 발생하는 오차보다 더 작은 것을 확인할 수 있다. 또한 DCT가 후의 값을 8*8 블록 단위 평활화 후 이미지로 만드는 작업을 통해 DCT의 효과를 명확하게 확인할 수 있었다.

Reference

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html

※ Please make sure to enclose programming source codes when submitting the report.