情報システム工学実験II
フルレポート
画像処理（松岡担当）

氏名: 関川　謙人
学籍番号:2022531033

提出日:2023 年 12 月 22 日

1　目的

この講義は授業当日の課題に取り組んで簡単な画像処理を実装することで、日常で使われる画像処理の仕組みやアルゴリズムについて理解することを目的とする。

2　演習概要

今回取り組んだ課題は、以下のとおりである。

表 1: 課題演習

課題番号	画像処理技術	処理
1	線形量子化	画素値の階調の数を減らし、データ量を減らす。
2	二値化処理	閾値を基準に、入力画素の値を 0 か 255 に置き換える。
3-1	平均値を用いた縮小	縮小倍率 × 縮小倍率の範囲毎に平均値を求め、それを出力画素値として出力する。
3-2	線形補完法	拡大の際に生じる欠損画素を、画素間の画素値が線形に変化していると仮定して補う。
4	ガウシアンフィルタ	周辺画素 × $\exp(-2\frac{k^2+l^2}{2\omega^2})$ との平均を求め, 画像を平滑化する。ω は画素値の標準偏差。
5	ラプラシアンフィルタ	エッジの抽出。
	鮮鋭化	入力画素にラプラシアンフィルタで抽出したエッジを足し込み、エッジがはっきりとした画像を作る。

3　演習課題

課題 1: 画像の量子化

講義で配布した画像ファイル”lena256.ppm”を用いて，線形量子化処理を実行し，その結果をレポートに貼りつけよ．ただし，任意の N-bit に量子化できるようにプログラムを作成すること．N=2，4，7 のときの量子化結果を比較し考察せよ．

プログラム:

```
1     void quantize(ColorImage *outimage, const ColorImage *inimage)
2 {
3     int <variable>i,j,k;
4     int bit_shift;
5     printf("何ビットに量子化しますか? \n>>>");
6     scanf("%d",&bit_shift); // 量子化ビット
7     for(i = 0;i < inimage -> height; i++){
```

```
8           for(j = 0; j < inimage -> width; j++){
9               //量子化処理
10              outimage -> r[i][j] = (255/(pow(2,bit_shift)-1))*round(inimage -> r[i][j] /(255/(pow
                  (2,bit_shift)-1)));
11              outimage -> g[i][j] = (255/(pow(2,bit_shift)-1))*round(inimage -> g[i][j] /(255/(pow
                  (2,bit_shift)-1)));
12              outimage -> b[i][j] = (255/(pow(2,bit_shift)-1))*round(inimage -> b[i][j] /(255/(pow
                  (2,bit_shift)-1)));
13          }
14      }
15  }
```

実行結果:

N=7,N=4,N=2 といった風に量子化ビット数が下がっていくにつれて、画像が粗くなっているのが確認できる。

元画像　　N=7　　N=4　　N=2

考察:

入力画像を $f(i,j)$、出力画像を $g(i,j)$ とすると、この処理は、

$$g(i,j) = \frac{255}{2^N - 1} \times round\Big(\frac{2^N - 1}{255} f(i,j)\Big) \tag{1}$$

となる。

この時 $\frac{255}{2^N-1}$ で基準となる値を決めている。例えば N=2 であるとき、基準値は 85 である。また、$round(\frac{2^N-1}{255} f(i,j))$ の計算式でその画素をどの値とみなすかを決定している。

画素値は基準値の整数倍となり N=2 の場合、(0,85,170,255) のいずれかの値になる。

このときビット数が低くなるにつれて、画素値の種類が少なくなるので画像は粗くなる。

課題 2: 画像の二値化

講義で配布した画像ファイル”peppers.ppm”を用いて，以下の処理を実行し，その結果をレポートに貼りつけよ．ただし，２値化の結果が最も良くなる閾値を探し，その結果を閾値の値とともに報告すること．また，２値化の精度を向上するための方法を考察せよ．

・画像中の緑色のピーマンの領域を２値化で抽出せよ．

・画像中の赤色のピーマンの領域を 2 値化で抽出せよ.

プログラム:

```
    /*2 値化*/
void blackwhite(ColorImage *outimage, const ColorImage *inimage)
{
    int i,j,thresh,average;//thresh:閾値 average:RGB平均値
    printf("閾値を入力\n");
    scanf("%d",&thresh);
    for(i = 0; i < inimage -> height ; i++){
        for(j = 0; j < inimage -> width; j++){
            average = (inimage -> r[i][j] + inimage -> g[i][j] + inimage -> b[i][j]) / 3;
            if(average > thresh){
                //二値化処理
                outimage -> r[i][j] = 255;
                outimage -> g[i][j] = 255;
                outimage -> b[i][j] = 255;
            }
            else{
                outimage -> r[i][j] = 0;
                outimage -> g[i][j] = 0;
                outimage -> b[i][j] = 0;
            }
        }
    }
}\left\{\begin{array}{ll}
255 & (f(i,j) > thresh) \\
0 & (f(i,j) <= thresh)
\end{array}
\right.
```

実験結果:

緑は白、赤は黒として抽出を試みた結果。赤の抽出を白で試みたとき、同時に緑も抽出されてしまうことが多かった。また緑の部分の抽出には白が適しており、黒で抽出しようとすると赤の要素が混ざった。

元画像

赤抽出 閾値:123

緑抽出 閾値:95

考察:

二値化の式は、

$$g(i,j) = \begin{cases} 255 & (f(i,j) > thresh) \\ 0 & (f(i,j) \leqq thresh) \end{cases} \tag{2}$$

である。実際の処理ではそれぞれの RGB 値の平均を求めてそれを閾値と比べている。

閾値 95 で抽出できるピーマンの部分は RGB 合計が 285 以上に集まっており、黒で抽出した赤色は RGB 合計が 369 以下に集まっていると考えられる。

このことから、ピーマンの抽出は閾値 95 で白、パプリカは閾値 123 で黒で抽出するのが最適である。

課題 3-1: 画像の縮小

PPM ファイル “zoneplate256.ppm” の画像を入力として「平均値」を用いて画像を縦横それぞれ 1/4 倍のサイズ（64 × 64）に縮小せよ．縮小した結果画像をレポートに貼り，画質について考察せよ．また，その結果をダウンサンプリング法と比較せよ．ただし，縮小倍率は 1/4 倍とする．

プログラム

ダウンサンプリング法:

```
1      /* 縮小1（ダウンサンプリング） */
2  void scale_down_DS(ColorImage *outimage, const ColorImage *inimage)
3  {
4      int i,j;
5      int wide,high;//wide : 横の拡大率 high : 縦の拡大率
6      printf("縦の縮小率を入力\n>>>");
7      scanf("%d",&high);
8      printf("横の縮小率を入力\n>>>");
9      scanf("%d",&wide);
10     //演算
11     for(i=0;i < outimage -> width; i++){//一行ずらす
12         for(j=0;j < outimage -> height; j++){//一列ずらす
13             outimage -> r[i][j] = inimage -> r[(int)(i*wide)][(int)(j*high)];//位置の入れ替え、
                   赤
14             outimage -> g[i][j] = inimage -> g[(int)(i*wide)][(int)(j*high)];//位置の入れ替え、
                   緑
15             outimage -> b[i][j] = inimage -> b[(int)(i*wide)][(int)(j*high)];//位置の入れ替え、
                   青
16         }
17     }
18 }
```

平均値を用いた縮小:

```
1  void scale_down_Mean(ColorImage *outimage, const ColorImage *inimage)
2  {
3    int i,j,k,l;//i : 横の画素値  j:縦の画素値 k:横の平均を取る画素の座標 l:縦の平均を取る画素の座
       標
4      int wide,high;//wide : 横の縮小率 high : 縦の縮小率
5      int sumr,sumg,sumb;// sum: 画素地の合計
6      float red=0,green = 0,blue=0; //画素値を一時的に格納
7      //倍率入力
8      printf("縦の縮小倍率を入力\n>>>");
9      scanf("%d",&high);
10     printf("横の縮小倍率を入力\n>>>");
11     scanf("%d",&wide);
12     //演算
13     for(i=0;i < outimage -> width; i++){//一行ずらす
14         for(j = 0;j < outimage -> height; j++){//一列ずらす
15             sumr = 0;sumg = 0 ; sumb = 0;//sum値の初期化
16             for(k = 0;k < wide;k++){
17                 for(l = 0;l < high; l++){
18                     if(!(i*wide+k >= inimage -> width || j*high+l >= inimage -> height)){
19                         sumr = sumr + (inimage -> r[i*wide+k][j*high+l]); //r値の合計を変数sumr
                               に格納
```

```
20                          sumg = sumg + (inimage -> g[i*wide+k][j*high+l]); //g値の合計を変数sumg
                              に格納
21                          sumb = sumb + (inimage -> b[i*wide+k][j*high+l]);//b値の合計を変数sumbに
                              格納
22                      }
23                  }
24              }
25              red = sumr / (wide * high); //平均を取り、変数sumrに格納
26              green = sumg / (wide * high); //平均を取り、変数sumgに格納
27              blue = sumb / (wide * high); //平均を取り、変数sumbに格納
28              //変更を反映。
29              outimここge -> r[i][j] = red;
30              outimage -> g[i][j] = green;
31              outimage -> b[i][j] = blue;
32          }
33      }
34  }
```

実行結果:

ダウンサンプリング法で縮小した画像は中心の円の画像パターンが繰り返されているのに対し、平均値を用いた縮小では比較的元画像に対して忠実なまま縮小された。

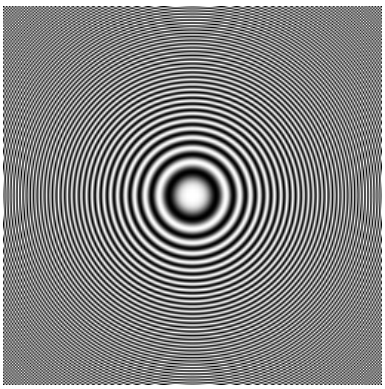


図 1: 元画像

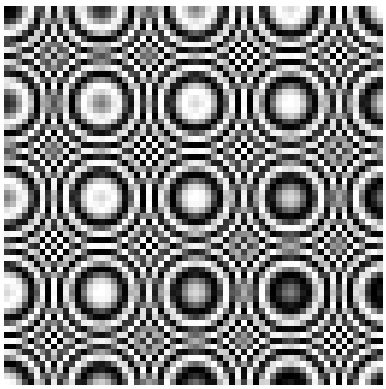


図 2: ダウンサンプリング

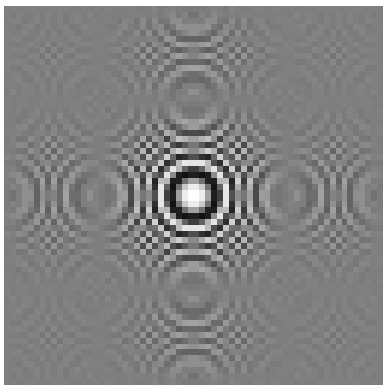


図 3: 平均値を用いた縮小

※組版の関係上、同じ大きさで表示しているが、実際は図 2,3 は図 1 の 1/16 の大きさである。

考察:

ダウンサンプリングの計算式は、縦横の縮小倍率をそれぞれ M, N、縦横の大きさをそれぞれ H, W とすると、

$$g(i,j) = f(Mi, Nj), 0 \leqq i < H, 0 \leqq j < W \tag{3}$$

また、平均値を用いた縮小の計算式は、

$$g_L(i,j) = \frac{1}{MN} \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} f_L(Mi+k, Nj+l) \tag{4}$$

平均値を用いた縮小では、縮小倍率に応じた範囲内にある画素を平均して出力画素としている。画素数が減る分、画質は悪くなる。しかし一部だけでなく周辺の画素も加味した値であるため出力す

る画素に偏りが生じにくく、ダウンサンプリング法と比較して元画像に忠実な縮小画像が出力できる。

一方ダウンサンプリング法では縮小倍率の倍数の位置にある画素のみ取り出し、出力画素とする。このためダウンサンプリングの画像が表しているのは、「4 の倍数の座標に位置する画素だけ集めた画像」である。このため元画像のようにパターンを含む画像を縮小しようとすると、同じような模様が並んでいる画像になる。

課題 3-2: 画像の拡大

線形補間法を用いて画像を縦横それぞれ 2 倍のサイズ（512 × 512）に拡大せよ．拡大した結果画像はレポートに貼り，画像について考察せよ．また, その結果を最近傍法と比較せよ．

プログラム:

線形補完法:

```
1      /* 拡大3（線形補間） */
2  void scale_up_IP(ColorImage *outimage, const ColorImage *inimage)
3  {
4      int i,j;
5      int wide,high;//wide : 横の拡大率  high : 縦の拡大率
6      float sep_top_r,sep_top_g,sep_top_b; //sep_top : 上の内分点の画素値
7      float sep_bot_r,sep_bot_g,sep_bot_b; //sep_bot:下の内分点の画素値
8      float alpha,beta; //alpha : i/wideの小数値  beta:j/highの小数値
9      int i_childa,j_childa; //i_childa : i/wide  j_childa : j/high
10     //倍率読み込み
11     printf("縦の拡大率を入力\n>>>");
12     scanf("%d",&high);
13     printf("横の拡大率を入力\n>>>");
14     scanf("%d",&wide);
15     //演算
16     for(i=0;i < outimage -> width - wide; i++){//一行ずらす
17         for(j=0;j < outimage -> height -high; j++){//一列ずらす
18             i_childa = i/wide;
19             j_childa = j/high;
20             //i/wide、j/wideが整数値であるとき
21             if(i%wide == 0 && j%high == 0){
22                 outimage -> r[i][j] = inimage -> r[(int)i_childa][(int)j_childa];
23                 outimage -> g[i][j] = inimage -> g[(int)i_childa][(int)j_childa];
24                 outimage -> b[i][j] = inimage -> b[(int)i_childa][(int)j_childa];
25             }
26             //i/wide,j/wideが整数値でないとき
27             else{
28                 //補完座標
29                 i_childa = floor(i_childa);
30                 beta = i/wide - i_childa;
31                 j_childa = floor(j_childa);
32                 alpha = j/high - j_childa;
33                 //上の内分点
34                 sep_top_r = (1-alpha)*inimage->r[(int)i_childa][(int)j_childa] + alpha* inimage
                     -> r[(int)i_childa][(int)(j_childa+1)];
35                 sep_top_g = (1-alpha)*inimage->g[(int)i_childa][(int)j_childa] + alpha* inimage
                     -> g[(int)i_childa][(int)(j_childa+1)];
36                 sep_top_b = (1-alpha)*inimage->b[(int)i_childa][(int)j_childa] + alpha* inimage
                     -> b[(int)i_childa][(int)(j_childa+1)];
37                 //下の内分点
38                 sep_bot_r = (1-alpha)*inimage->r[(int)(i_childa+1)][(int)j_childa] + alpha*
                     inimage -> r[(int)(i_childa+1)][(int)(j_childa+1)];
39                 sep_bot_r = (1-alpha)*inimage->r[(int)(i_childa+1)][(int)j_childa] + alpha*
                     inimage -> r[(int)(i_childa+1)][(int)(j_childa+1)];
40                 sep_bot_r = (1-alpha)*inimage->r[(int)(i_childa+1)][(int)j_childa] + alpha*
                     inimage -> r[(int)(i_childa+1)][(int)(j_childa+1)];
41                 //補完演算
```

```
                outimage -> r[i][j] = (1-beta)*sep_top_r + beta * sep_bot_r;
                outimage -> g[i][j] = (1-beta)*sep_top_g + beta * sep_bot_g;
                outimage -> b[i][j] = (1-beta)*sep_top_b + beta * sep_bot_b;
            }
        }
    }
}
```

最近傍法:

```
    void scale_up_IP(ColorImage *outimage, const ColorImage *inimage)
{
    int i,j;
    int wide,high;//wide : 横の拡大率  high : 縦の拡大率
    float sep_top_r,sep_top_g,sep_top_b; //sep_top : 上の内分点の画素値
    float sep_bot_r,sep_bot_g,sep_bot_b; //sep_bot:下の内分点の画素値
    float alpha,beta; //alpha : i/wideの小数値  beta:j/highの小数値
    int i_childa,j_childa; //i_childa : i/wide  j_childa : j/high
    //倍率読み込み
    printf("縦の拡大率を入力\n>>>");
    scanf("%d",&high);
    printf("横の拡大率を入力\n>>>");
    scanf("%d",&wide);
    //演算
    for(i=0;i < outimage -> width - wide; i++){//一行ずらす
        for(j=0;j < outimage -> height -high; j++){//一列ずらす
            i_childa = i/wide;
            j_childa = j/high;
            //i/wide、j/wideが整数値であるとき
            if(i%wide == 0 && j%high == 0){
                outimage -> r[i][j] = inimage -> r[(int)i_childa][(int)j_childa];
                outimage -> g[i][j] = inimage -> g[(int)i_childa][(int)j_childa];
                outimage -> b[i][j] = inimage -> b[(int)i_childa][(int)j_childa];
            }
            //i/wide,j/wideが整数値でないとき
            else{
                //補完座標
                i_childa = floor(i_childa);
                beta = i/wide - i_childa;
                j_childa = floor(j_childa);
                alpha = j/high - j_childa;
                //上の内分点
                sep_top_r = (1-alpha)*inimage->r[(int)i_childa][(int)j_childa] + alpha* inimage
                  -> r[(int)i_childa][(int)(j_childa+1)];
                sep_top_g = (1-alpha)*inimage->g[(int)i_childa][(int)j_childa] + alpha* inimage
                  -> g[(int)i_childa][(int)(j_childa+1)];
                sep_top_b = (1-alpha)*inimage->b[(int)i_childa][(int)j_childa] + alpha* inimage
                  -> b[(int)i_childa][(int)(j_childa+1)];
                //下の内分点
                sep_bot_r = (1-alpha)*inimage->r[(int)(i_childa+1)][(int)j_childa] + alpha*
                  inimage -> r[(int)(i_childa+1)][(int)(j_childa+1)];
                sep_bot_r = (1-alpha)*inimage->r[(int)(i_childa+1)][(int)j_childa] + alpha*
                  inimage -> r[(int)(i_childa+1)][(int)(j_childa+1)];
                sep_bot_r = (1-alpha)*inimage->r[(int)(i_childa+1)][(int)j_childa] + alpha*
                  inimage -> r[(int)(i_childa+1)][(int)(j_childa+1)];
                //補完演算
                outimage -> r[i][j] = (1-beta)*sep_top_r + beta * sep_bot_r;
                outimage -> g[i][j] = (1-beta)*sep_top_g + beta * sep_bot_g;
                outimage -> b[i][j] = (1-beta)*sep_top_b + beta * sep_bot_b;
            }
        }
    }
}
```

実行結果:
元画像と線形補完法の画像を比較した結果、元画像の要素のほとんどを残したまま拡大された画像が生成された。

図 4: 元画像

図 5: 線形補完法

図 6: 最近傍法

※組版の関係上同じ大きさで載せているが、図 5 と図 6 は図 4 の 4 倍の大きさである。

考察：今回は 2 倍の拡大だったので、両者に大きな違いは見られなったが、10 倍など、倍率を大きくすると最近傍法は遠くの画素を取ることになり、粗い画像になると思われる。
一方で線形補完法は、線形に欠損画素を補完するという特性を持つため、元画像に忠実な画像が作れる。

課題 4: ガウシアンフィルタによる平滑化

1. ガウシアンフィルタを実装し、σ の値を 20 に固定しフィルタサイズを変えたとき、出力画像はどのように変化するか？ ただし、フィルタサイズは奇数とする。2. また、フィルタ長を 5 に固定しσの値を変えたとき、出力画像はどのように変化するか？ ただし、σは 0 より大きな整数とする。上記について、結果から分かることを報告しその理由も述べよ。また、考察せよ。
プログラム:

```
1     void gaussian_filter(ColorImage *outimage, ColorImage *inimage, float sigma, int win_size) {
2     /* 入力カラー画像inimageに対してガウシアンに基づく平滑化フィルタを計算し，出力用outimageに格
        納する．*/
3     /* 必要であれば、変数宣言を適宜追加すること */
4     /* ここにプログラムを挿入する */
5     int i, j, k, l, L = (win_size - 1) / 2;
6     float sumR = 0.0f, sumG = 0.0f, sumB = 0.0f;
7     float InR,InG,InB; //それぞれ、入力画像の画素値。InR:赤 InG:緑 InB:青
8     float w1 = 0.0f, sumW = 0.0f;
9     for (i=0; i<outimage->height; i++) {
10        for (j=0; j<outimage->width; j++) {
11            sumR = 0.0f; sumG = 0.0f; sumB = 0.0f; sumW = 0.0f;//sumの初期化
12            for (k=-L; k<=L; k++) {
13                for (l=-L; l<=L; l++) {
```

```
                    /* 画像端では、画像の領域外にアクセスしないようにする */
                    if(i+k >= inimage -> width || j+l >= inimage -> height || i+k <= 0 || j+l <=
                        0){
                        //0埋め
                        InR = 0,InG = 0,InB = 0;
                    }
                    else{
                    //画素値の代替変数に画素値を代入。
                        InR = inimage -> r[i+k][j+l];
                        InG = inimage -> g[i+k][j+l];
                        InB = inimage -> b[i+k][j+l];
                    }
                    /* 重みを計算する */
                    w1 = exp(-((pow(k,2)+pow(l,2))/(2*pow(sigma,2))));
                    /* フィルタの係数の乗算 */
                    sumR += InR*w1;
                    sumG += InG*w1;
                    sumB += InB*w1;
                    sumW += w1;
                }
            }
        /* outimage[i][j]にフィルタの出力を格納する */
        outimage -> r[i][j] = sumR/sumW;
        outimage -> g[i][j] = sumG/sumW;
        outimage -> b[i][j] = sumB/sumW;
        }

    }
}
```

問題 1:

実行結果:

フィルタサイズを大きくするにつれて、画像にぼかしがかかった。

元画像　　フィルタサイズ:2×2　　フィルタサイズ:4×4　　フィルタサイズ:6×6

考察:

このフィルタの数式は、

$$g(i,j) = \frac{\sum_{k=-L}^{L}\sum_{l=-L}^{L} f(i+k,j+l)\cdot \exp(-\frac{k^2+l^2}{2\sigma^2})}{\sum_{k=-L}^{L}\sum_{l=-L}^{L} \exp(-\frac{k^2+l^2}{2\sigma^2})} \tag{5}$$

数式 (5) より、二次元ガウス分布 $\exp(-\frac{k^2+l^2}{2\sigma^2})$ で重みづけした周辺画素の平均を出力画素としている。より広い範囲を平均する程画像の重みの差が小さくなり、画像のエッジが鈍る。それゆえに画像が平滑化され、ぼかしがかかったような画像が出力された。

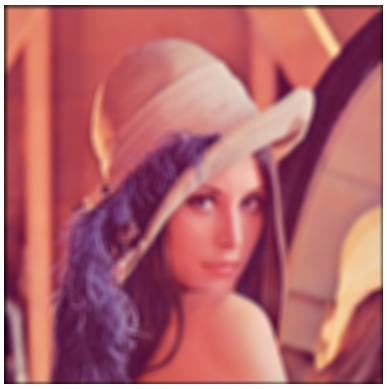
問題 2:

実行結果:

標準偏差の値が大きくなればなるほど画像がぼかされている。

元画像

標準偏差:1



標準偏差:20

考察：

数式 (5) の二次元ガウス分布の部分 $(\exp(-\frac{k^2+l^2}{2\sigma^2}))$ は画像のエッジの重みをあらわす。その分母である $2\sigma^2$ が大きくなればなるほどエッジの重みが全体的に軽くなり、エッジが鈍くなるから標準編差 σ を増やしたとき、画像にぼかしがかかったようになった。

課題 5: ラプラシアンフィルタによる鮮鋭化

1. ラプラシアンフィルタによる鮮鋭化を実行するプログラム（コメント付き）と処理結果画像を貼ること。
2. K の値を変えたときに鮮鋭化結果がどのように変化するか確認し、考察せよ。

プログラム:

```
1     void laplacian_sharpening(ColorImage *outimage, ColorImage *inimage, float K) {
2     /* 入力カラー画像inimageに対してラプラシアンフィルタによる鮮鋭化結果を計算し，出力用outimage
        に格納する．*/
3     /* ここにプログラムを挿入する */
4     int i,j,k,l,L = 1; //参照範囲:winsize^2
5     float sumR = 0.0f ,sumG = 0.0f ,sumB = 0.0f; //分母定義
6     float InR,InG,InB; //inimageの画素地の代替変数
7     int alap[3][3] = {{0,1,0},{1,-4,1},{0,1,0}}; //水平垂直方向のラプラシアンフィルタ
8     for (i=0; i < outimage->height; i++) {
9         for (j=0; j < outimage->width; j++) {
10            sumR = 0.0f,sumG = 0.0f; sumB = 0.0f;
11            for (k = -L; k <= L; k++)
12            {
13                for (l = -L; l <= L; l++)
14                {
15                    //segmentation fault対策
16                    if(i+k >= inimage -> width || j+l >= inimage -> height || i+k <= 0 || j+l <=
                          0){
17                        //0埋め
18                        InR = 0,InG = 0,InB = 0;
```

```
19                      }
20                      else{
21                          //画素値の代替変数に画素値を代入。
22                          InR = inimage -> r[i+k][j+l];
23                          InG = inimage -> g[i+k][j+l];
24                          InB = inimage -> b[i+k][j+l];
25                      }
26                      //合計演算
27                      sumR += alap[k+1][l+1] * InR;
28                      sumG += alap[k+1][l+1] * InG;
29                      sumB += alap[k+1][l+1] * InB;
30                  }
31              }
32              //画像演算 K:定数。
33              outimage -> r[i][j] = inimage -> r[i][j] - K * sumR;
34              outimage -> g[i][j] = inimage -> g[i][j] - K * sumG;
35              outimage -> b[i][j] = inimage -> b[i][j] - K * sumB;
36          }
37      }
38  }
```

実行結果:

元の画像はエッジが鈍く、若干ぼやけた感じだが、エッジを足し合わせたことで船や周辺の景色の輪郭がくっきりするようになっている。また、K が大きくなるほど顕著である。

図 7: 元画像

図 8: K=0.5

図 9: K=2

考察:

ラプラシアンフィルタによる鮮鋭化の計算式は、

$$g(i,j) = f(i,j) - K \sum_{k=-L}^{L} \sum_{l=-L}^{L} a_{k,l}^{Lap} \cdot f(i+k, j+l) \tag{6}$$

$$a_{k,l}^{Lap} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{7}$$

である。(7) はラプラシアンフィルタのカーネルである。(7)× 周辺画素の和は画素値と周辺画素の差、すなわちエッジである。そのエッジを K 回足しこむので、よりエッジが鋭くなり、K の値を大きくするほどくっきりとした画像ができる。

4　考察

課題を総ざらいすると、最近傍法、線形補完には図形と方程式を利用するし、鮮鋭化には線形代数学を利用する。ましてや画像データ自体が行列である。画像処理は数学の応用形といった色合いが強いといったことがわかる。

5　感想

画像処理関係の授業と聞き、始めは機械学習を想像したのだが、骨のある C 言語でプログラミングでき、自らのレベルも上げることができたように思う。また,segmentation fault などのエラーにできるだけ独力で立ち向かったが、これがさらに自分の画像処理に対する理解を深めたと考える。また、レポート執筆中に考察を考えるのに苦労し、それに時間がかかった。この先どんなに難しいプログラミング言語に出会ったとしても日本語より難しい言語は存在しないだろうと考える。

参考文献

[1] 第三回講義スライド

https://lms.kitakyu-u.ac.jp/pluginfile.php/275630/mod_resource/content/3/%E7%AC%AC3%E5%

[2] 第四回講義スライド

https://lms.kitakyu-u.ac.jp/pluginfile.php/276705/mod_resource/content/1/%E7%AC%AC4%E5%

[3] 第七回講義スライド

https://lms.kitakyu-u.ac.jp/pluginfile.php/280118/mod_resource/content/3/%E7%AC%AC7%E5%