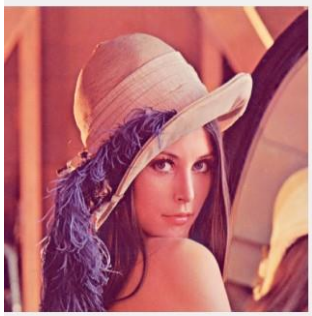
Principle and Applications of Digital Image Processing

Homework 1 Report 林東甫 R12631055

Part 1: (30%) Geometric Transformation

第一題的目標是針對目標影像進行幾何變換，本次作業主要以 Trapezoidal Transformation, Wavy Transformation, Circular Transformation 作為範例

Wavy Transformation



如圖，左圖為原始影像，而右圖為先使用 sin 函數描繪曲線

```
map_x.at<float>(i, j) = j + 10 * sin(i/10.0);
map_y.at<float>(i, j) = i + 10 * sin(j/10.0);
```

再透過 remap 函數製造出波浪效果

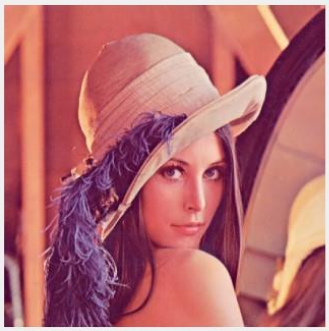
```
remap(img,img2,map_x,map_y,INTER_LINEAR,BORDER_CONSTANT,Scalar(0,0,0));
```

Trapezoidal Transformation

使用透視函數進行幾何轉換，以四邊形四個點作為標定。

```
dstPoints[0] = Point2f(0, 0);
dstPoints[1] = Point2f(src.cols*0.2, src.rows*0.8);
dstPoints[2] = Point2f(src.cols, 0);
dstPoints[3] = Point2f(src.cols*0.8, src.rows*0.8);
Mat M1 = getPerspectiveTransform(srcPoints, dstPoints);
warpPerspective(src, img2, M1, src.size());
```

Circular Transformation

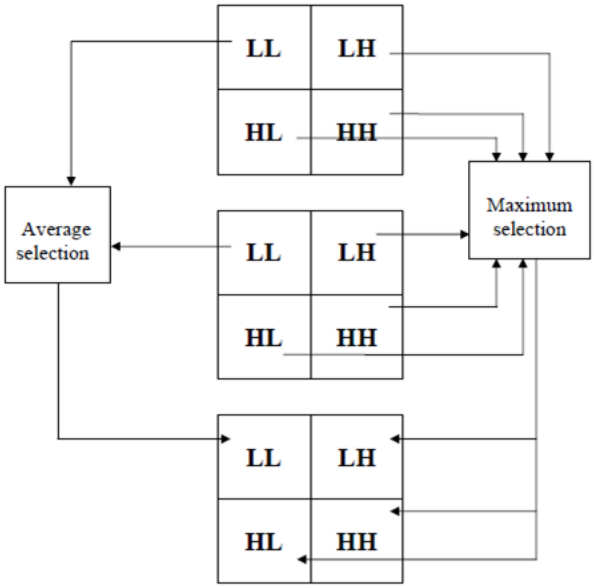


同前，透過三角與反三角函數製造出魚眼效果

```
double xd = jd * 2.0 / dst.cols - 1.0;
double yd = id * 2.0 / dst.cols - 1.0;
double rd = sqrt(xd * xd + yd * yd);
double phid = atan2(yd, xd);
double xs = asin(rd) * 2 / M_PI * cos(phid);
double ys = asin(rd) * 2 / M_PI * sin(phid);
int is = round((ys + 1.0) * dst.rows / 2.0);
int js = round((xs + 1.0) * dst.cols / 2.0);
```

Part 2: (30%) Image Fusion Using Wavelet Transform

1. 本題的目標是透過小波轉換來進行影像的融合，過程如下圖所示：

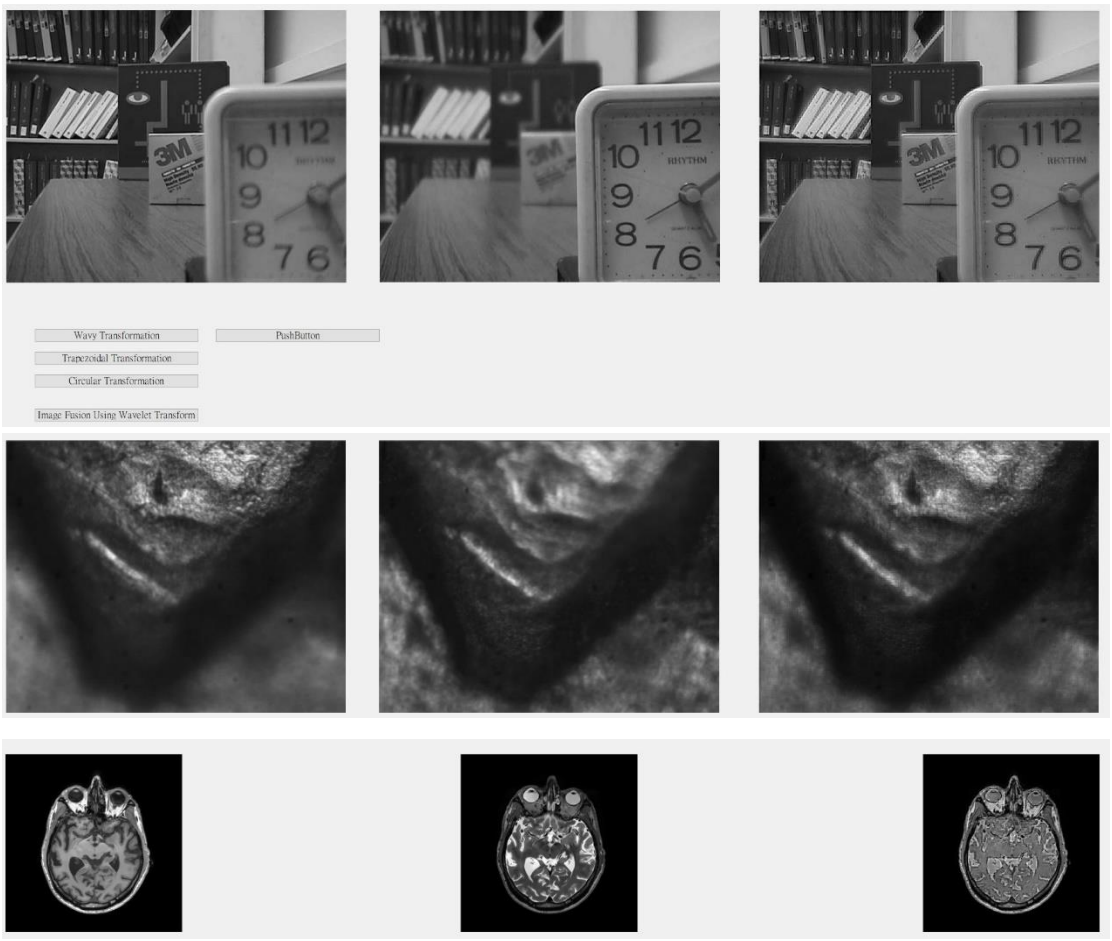


其結果如下：

左圖與右圖為欲融合的原始影像

下圖則為經小波轉換後融合的影像

放在一起比較：

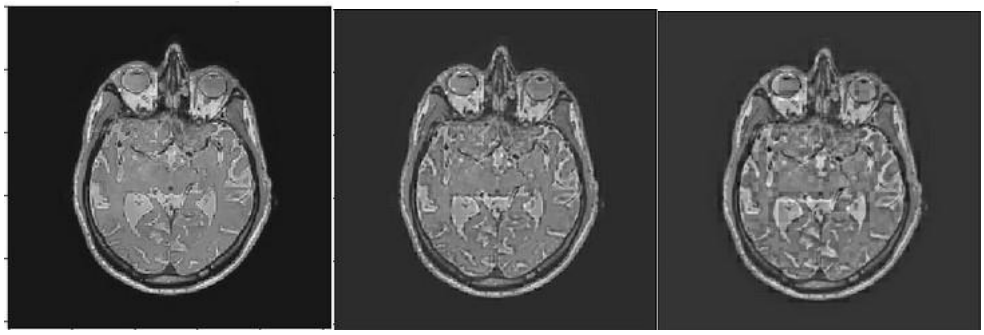


2.

下圖由左至右，分別是 depth=1, depth=2, depth=3

感覺最適合的大概是 2~3，然而其他影像則不見得如此：

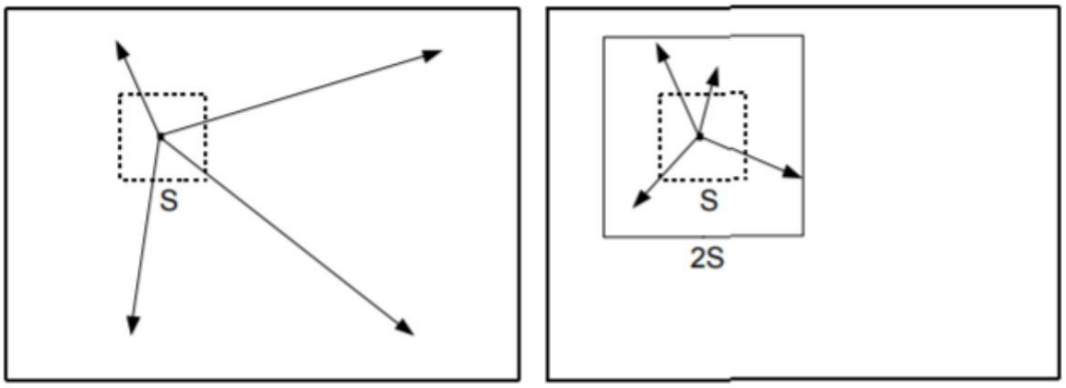
下圖由左至右，分別是 depth=1, depth=2, depth=3



感覺 1 的時候的效果最好，所以最佳的 depth 恐怕是因影像本身而定。

Part 3: (40%) Superpixel-based Regional Segmentation

參照課本以及 SLIC 論文(2011)，其實本質上跟 k-means 相似，但其搜索空間比 k-means 小很多，因此計算量也通常會比較少。如下圖：



左圖為 k-means 搜索長度，右圖為 SLIC 搜索空間，受限於 2S 以內。

而其中 S=sqrt(N/k) N=影像總像素數。

而算法過程如下圖：

Algorithm 1 SLIC superpixel segmentation

/ Initialization */*
Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
Move cluster centers to the lowest gradient position in a 3×3 neighborhood.
Set label $l(i) = -1$ for each pixel i.
Set distance $d(i) = \infty$ for each pixel i.

repeat
  / Assignment */*
  for each cluster center C_k **do**
    for each pixel i in a $2S \times 2S$ region around C_k **do**
      Compute the distance D between C_k and i.
      if $D < d(i)$ **then**
        set $d(i) = D$
        set $l(i) = k$
      end if
    end for
  end for
  / Update */*
  Compute new cluster centers.
  Compute residual error E.
until $E \leq$ threshold

Simple Linear Iterative Clustering (*SLIC*) 2011.

首先按原文中聚類數 k 來等間隔地初始化聚類中心，假設總數為 N，則每隔 N/k 個樣本放置一個中心。在圖片上等間隔地放置 k 個初始聚類中心，也就是把圖片等分成格子，在格子的固定位置放置初始聚類中心。
另外為了避免初始的聚類中心落在物體邊緣上，還要對每個聚類中心都進行一下微調，具體來說就是計算初始聚類中心點周圍鄰域的梯度，把中心點移到梯度最小的點上。

```
int initilizeCenters(cv::Mat &imageLAB, std::vector<center0> &centers, int len)
{
    if (imageLAB.empty())
    {
        std::cout << "In itilizeCenters:     image is empty!\n";
        return -1;
    }

    uchar *ptr = NULL;
    center0 cent;
    int num = 0;
    for (int i = 0; i < imageLAB.rows; i += len)
    {
        cent.y = i + len / 2;
        if (cent.y >= imageLAB.rows) continue;
        ptr = imageLAB.ptr<uchar>(cent.y);
        for (int j = 0; j < imageLAB.cols; j += len)
        {
            cent.x = j + len / 2;
            if ((cent.x >= imageLAB.cols)) continue;
            cent.L = *(ptr + cent.x * 3);
            cent.A = *(ptr + cent.x * 3 + 1);
            cent.B = *(ptr + cent.x * 3 + 2);
            cent.label = ++num;
            centers.push_back(cent);
        }
    }
    return 0;
}
```

把中心移到到周圍八個鄰域中梯度最小的地方,梯度用 Sobel 計算：

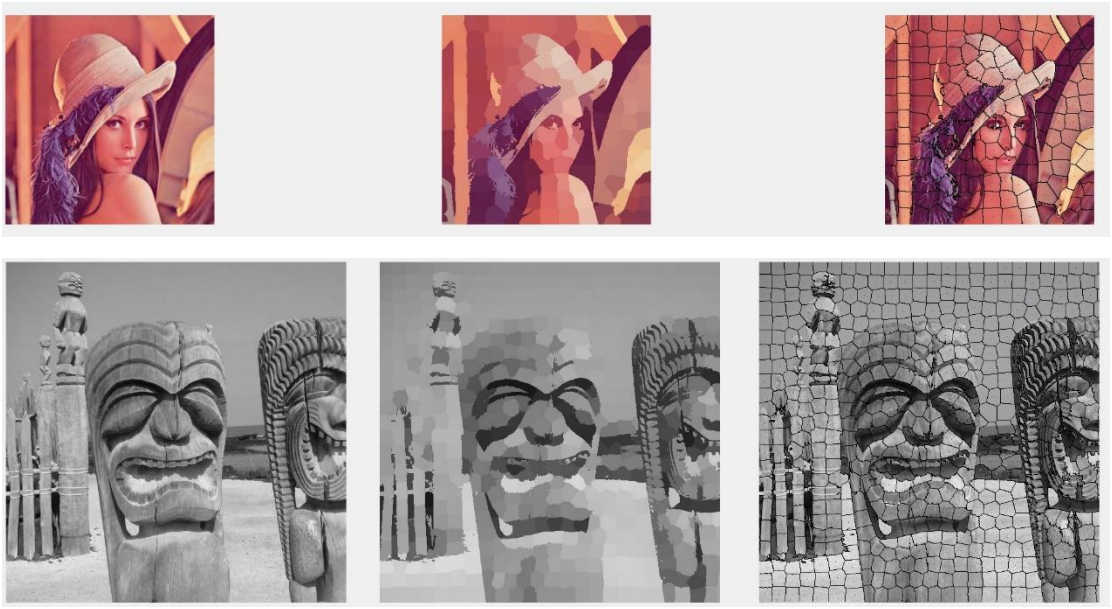
```
int fituneCenter(cv::Mat &imageLAB, cv::Mat &sobelGradient, std::vector<center0> &centers)
{
    if (sobelGradient.empty()) return -1;

    center0 cent;
    double *sobPtr = sobelGradient.ptr<double>(0);
    uchar *imgPtr = imageLAB.ptr<uchar>(0);
    int w = sobelGradient.cols;
    for (int ck = 0; ck < centers.size(); ck++)
    {
        cent = centers[ck];
        if (cent.x - 1 < 0 || cent.x + 1 >= sobelGradient.cols || cent.y - 1 < 0 || cent.y + 1 >= sobelGradient.rows)
        {
            continue;
        }//end if
        double minGradient = 9999999;
        int tempx = 0, tempy = 0;
        for (int m = -1; m < 2; m++)
        {
            sobPtr = sobelGradient.ptr<double>(cent.y + m);
            for (int n = -1; n < 2; n++)
            {
                double gradient = pow(*(sobPtr + (cent.x + n) * 3), 2)
                                + pow(*(sobPtr + (cent.x + n) * 3 + 1), 2)
                                + pow(*(sobPtr + (cent.x + n) * 3 + 2), 2);
                if (gradient < minGradient)
                {
                    minGradient = gradient;
                    tempy = m;//row
                    tempx = n;//column
                }//end if
            }
        }
```

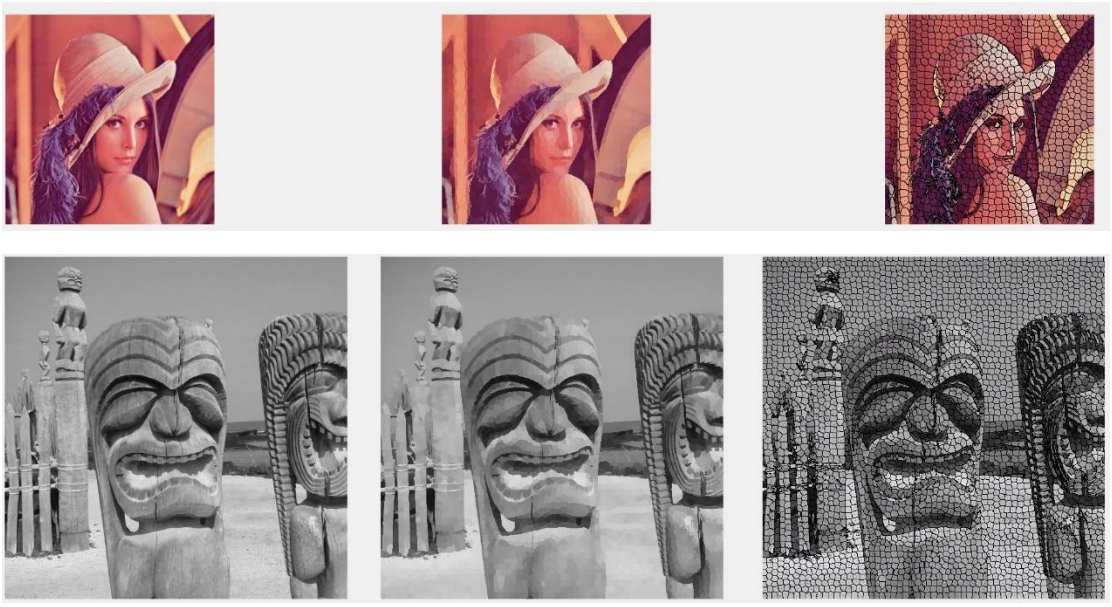
對每一個聚類中心 center_k，計算它到周圍 2len*2len 區域的所有點的距離，如果新計算的距離比原來的距離更小，則把該點歸入 center_k 這一類。

距離的計算方法則近似課本的 $D = \left[(\frac{d_c}{c})^2+(\frac{d_s}{s})^2\right]^{1/2}$ ，使用 $\mathrm{D} = \sqrt{\mathrm{d_c^2} + \mathrm{md_s^2}}$

在 lenˆ2=625, m=10 時



在 lenˆ2=100, m=10 時



在我自己的測試中，大約 lenˆ2=100, m=10 的時候效果最佳。

光就我自己測的感覺，計算速度感覺比上一次作業要求的 k-means 快上許多，特別是當 k 值很高的時候，k-means 真的蠻耗費計算量，而 SLIC 則在同一張影像上無論其值相對都不會差太多。
至於其分割的效果，SLIC 通常分割的形狀比較緊湊，比較能表現出局部的細節構造，在物體的輪廓邊界上也比較容易保留，而 k-means 則通常在輪廓邊界上都很破碎，並且不太能呈現細節的構造，分割的形狀也比較不穩定；另外雖然沒有嘗試過在 SLIC 加入白噪，但 k-means 的抗噪能力很差，預估 SLIC 應該會好很多。