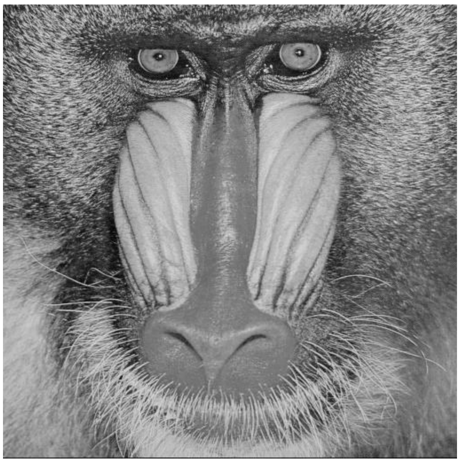
1-1 Image Quantization (binary, gray, index-color)

getGrayScaleImage(原圖) 是一個對圖像灰階化的函式，對原圖做像素遍歷取得每個像素點的 RGB 值，經由公式 Gray = (0.3 * R) + (0.59 * G) + (0.11 * B) 得出灰階化的值並存入輸出圖像。

```
Mat getGrayScaleImage(const Mat& image) {
    Mat grayImage = Mat(image.rows, image.cols, CV_8UC1);
    const uchar* imagePtr;
    uchar* gray;
    for (int row = 0; row < image.rows; row++) {
        imagePtr = image.ptr<uchar>(row);
        gray = grayImage.ptr<uchar>(row);
        for (int col = 0; col < image.cols; col++) {
            uchar blue = *imagePtr++, green = *imagePtr++, red = *imagePtr++;
            *gray++ = (0.3 * red) + (0.59 * green) + (0.11 * blue);
        }
    }
    return grayImage;
}
```



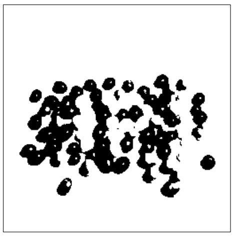
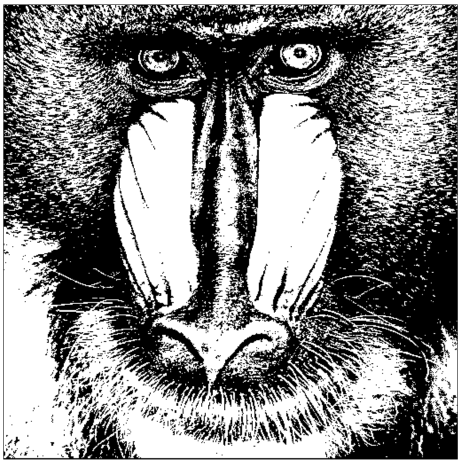



1-2 Convert the grayscale image to a binary image

getBinaryImage(原圖) 是個對圖像二值化的函式，如果原圖是彩色(3 channel)，先做灰階化處理後才做二值化處理，對灰階圖做像素遍歷取得每個像素點的灰階值，0~127 判定為 0、128~255 判定為 255，以此做圖像二值化處理。(下方圖片邊框僅報告內容易判別圖片用)

```
Mat getBinaryImage(const Mat& image) {
    Mat grayImage = image;
    if (image.channels() == 3)
        grayImage = getGrayScaleImage(image);
    Mat binaryImage = Mat(image.rows, image.cols, CV_8UC1);
    const uchar* imagePtr;
    uchar* binary;
    for (int row = 0; row < grayImage.rows; row++) {
        imagePtr = grayImage.ptr<uchar>(row);
        binary = binaryImage.ptr<uchar>(row);
        for (int col = 0; col < grayImage.cols; col++) {
            *binary++ = (*imagePtr++ >= 128) ? 255 : 0;
        }
    }
    return binaryImage;
}
```




1-3 Convert the color image to index-color image

取得 Index-color image 的函式為 getIndexColorImage(原圖, 色彩盤, 色差門檻值)並由 getColorFromMap(色彩盤, 色彩盤大小, 原像素點, 色差門檻值)取得色彩盤的顏色。

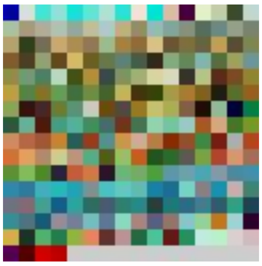
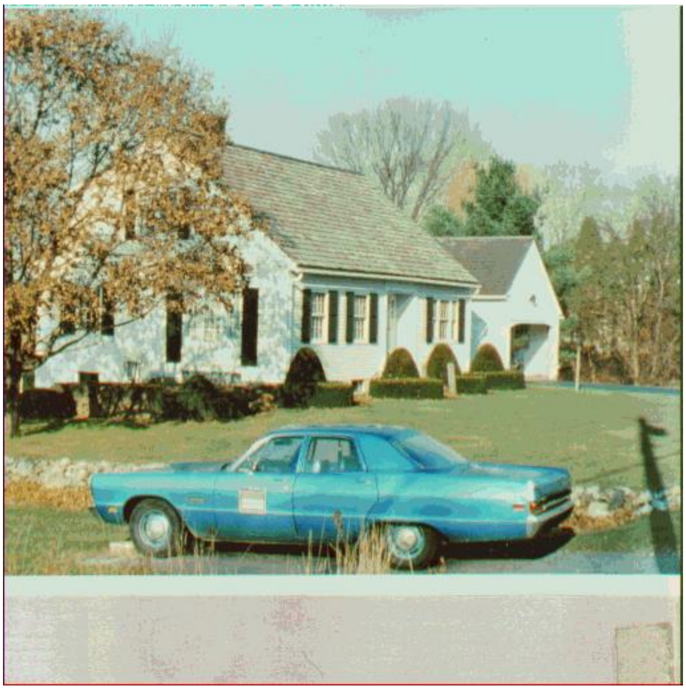
以原圖做掃點，並動態更新色彩盤大小，沒有在色彩盤內的顏色且大於其中的色差門檻值，會被判定為新顏色加入色彩盤；有在色彩盤內的顏色且小於等於其中的色差門檻值，被判定為已有色彩，並將該點顏色設為色彩盤內的顏色。因為每張圖所包含的顏色不盡相同，所以需要設置不一樣的色差門檻值來達到至多 256 色，以此來達到 index-color image 的處理，每張圖像都有不一樣的 color-map，color-map 的圖像經過放大(2-1 的功能)儲存，檢視時較容易。

```
Mat getIndexColorImage(const Mat& image, Mat& colorMap, int threshold) {
    colorMap = Mat(16, 16, CV_8UC3);
    Mat indexColorImage = Mat(image.rows, image.cols, CV_8UC3);
    int colorMapSize = 1;
    colorMap.at<Vec3b>(0, 0) = image.at<Vec3b>(0, 0);
    for (int row = 0; row < image.rows; row++) {
        for (int col = 0; col < image.cols; col++) {
            const Vec3b originPixel = image.at<Vec3b>(row, col);
            indexColorImage.at<Vec3b>(row, col) = getColorFromMap(colorMap, colorMapSize, originPixel, threshold);
            if (colorMapSize > 256) {
                cerr << "[Index color image] Threshold too low, the image process incomplete, please set higher threshold." << endl;
                return indexColorImage;
            }
        }
    }
    return indexColorImage;
}
```

```
Vec3b getColorFromMap(Mat& colorMap, int& colorPixels, const Vec3b bgr, int& threshould) {
    uchar sourceBlue = bgr[0], sourceGreen = bgr[1], sourceRed = bgr[2];
    uchar* colorMapPtr = colorMap.ptr<uchar>(0);
    bool isInColorMap = false;
    for (int i = 0; i < colorPixels; i++) {
        uchar colorMapBlue = *colorMapPtr++, colorMapGreen = *colorMapPtr++, colorMapRed = *colorMapPtr++;
        int colorGap = sqrt(pow((colorMapBlue - sourceBlue), 2) + pow((colorMapGreen - sourceGreen), 2) + pow((colorMapRed - sourceRed), 2));
        if (colorGap <= threshould) {
            isInColorMap = true;
            break;
        }
    }
    if (!isInColorMap) {
        *colorMapPtr++ = sourceBlue;
        *colorMapPtr++ = sourceGreen;
        *colorMapPtr++ = sourceRed;
        colorPixels++;
    }
    return Vec3b(*(colorMapPtr - 3), *(colorMapPtr - 2), *(colorMapPtr - 1));
}
```

Color map Index-color image (by color map)

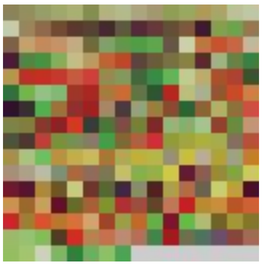

色差門檻值: 14

色差門檻值: 20

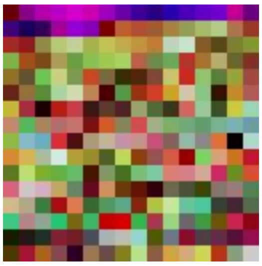

色差門檻值: 15

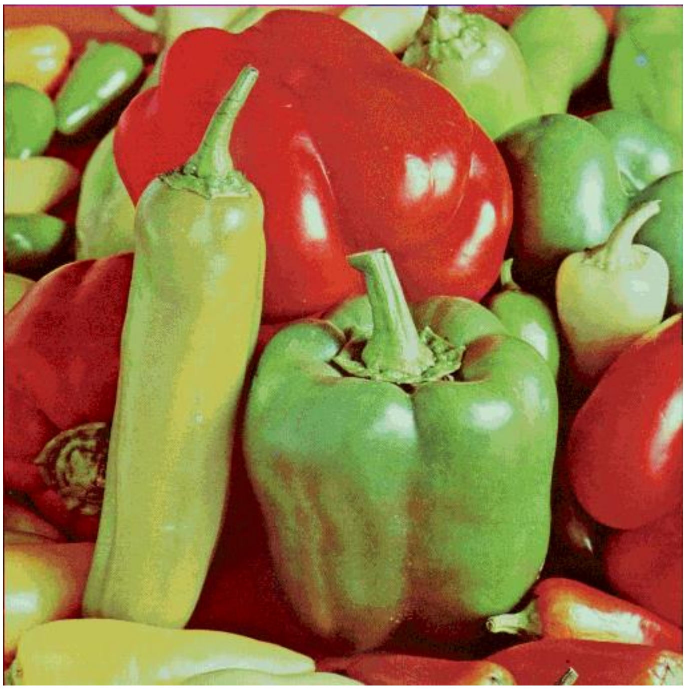
Color map Index-color image (by color map)



色差門檻值: 13

色差門檻值: 25





色差門檻值: 21

2-1 Resizing image to 1/2 and 2 times without interpolation.

getHalfSizeImage(原圖) 負責將原圖縮小 2 倍，原圖每 4 個像素點只取左上角的像素點作為輸出圖像的點。

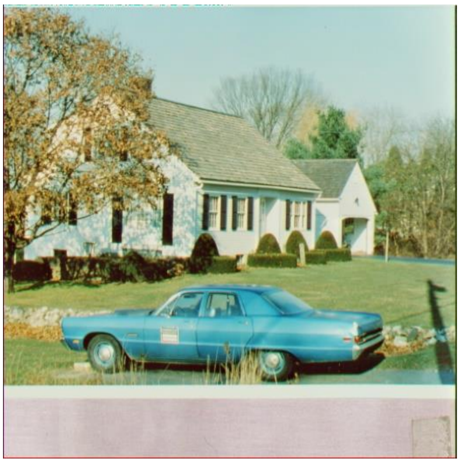
```
Mat getHalfSizeImage(const Mat& image) {
    int halfRow = image.rows / 2, halfCol = image.cols / 2;
    Mat scaledImage = Mat(halfRow, halfCol, CV_8UC3);
    for (int row = 0; row < halfRow; row++) {
        const uchar* imagePtr = image.ptr<uchar>(row * 2);
        uchar* scaledPtr = scaledImage.ptr<uchar>(row);
        for (int col = 0; col < halfCol; col++) {
            uchar imageBlue = *imagePtr++, imageGreen = *imagePtr++, imageRed = *imagePtr++;
            *scaledPtr++ = imageBlue;
            *scaledPtr++ = imageGreen;
            *scaledPtr++ = imageRed;
            imagePtr += 3;
        }
    }
    return scaledImage;
}
```

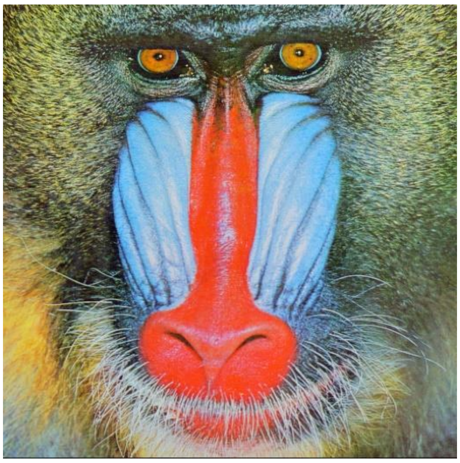





getDoubleSizeImage(原圖) 負責將原圖放大 2 倍，原圖的每個像素點會被填入輸出圖 2x2 的像素點。

```
Mat getDoubleSizeImage(const Mat& image) {
    int doubleRow = image.rows * 2, doubleCol = image.cols * 2;
    Mat scaledImage = Mat(doubleRow, doubleCol, CV_8UC3);
    for (int row = 0; row < doubleRow; row++) {
        const uchar* imagePtr = image.ptr<uchar>(row / 2);
        uchar* scaledPtr = scaledImage.ptr<uchar>(row);
        for (int col = 0; col < image.cols; col++) {
            uchar imageBlue = *imagePtr++, imageGreen = *imagePtr++, imageRed = *imagePtr++;
            for (int pixelColor = 0; pixelColor < 2; pixelColor++) {
                *scaledPtr++ = imageBlue;
                *scaledPtr++ = imageGreen;
                *scaledPtr++ = imageRed;
            }
        }
    }
    return scaledImage;
}
```

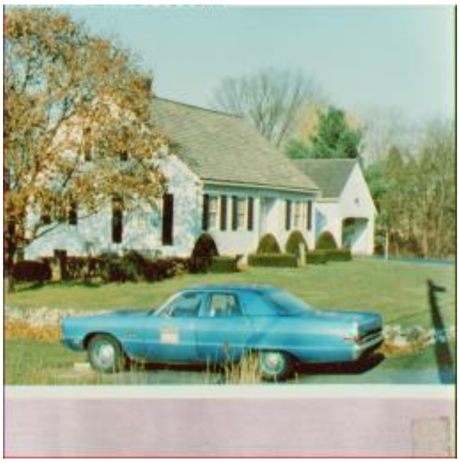



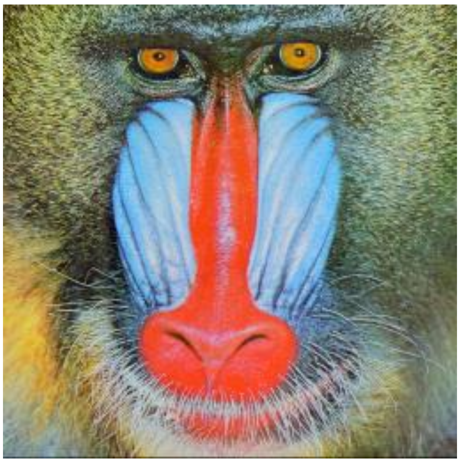




2-2 Resizing image to 1/2 and 2 times with interpolation (round)

getHalfSizeRoundImage(原圖) 是個負責將原圖縮小 2 倍，原圖 2x2 的像素點會經由加總平均後得出新的 1 個像素值並填入輸出圖中。

```
Mat getHalfSizeRoundImage(const Mat& image) {
    int halfRow = image.rows / 2, halfCol = image.cols / 2;
    Mat scaledImage = Mat(halfRow, halfCol, CV_8UC3);
    for (int row = 0; row < halfCol; row++) {
        const uchar* imagePtr1 = image.ptr<uchar>(row * 2);
        const uchar* imagePtr2 = image.ptr<uchar>(row * 2 + 1);
        uchar* scaledPtr = scaledImage.ptr<uchar>(row);
        for (int col = 0; col < halfCol; col++) {
            uchar leftTopBlue = *imagePtr1++, leftTopGreen = *imagePtr1++, leftTopRed = *imagePtr1++;
            uchar rightTopBlue = *imagePtr1++, rightTopGreen = *imagePtr1++, rightTopRed = *imagePtr1++;
            uchar leftBottomBlue = *imagePtr2++, leftBottomGreen = *imagePtr2++, leftBottomRed = *imagePtr2++;
            uchar rightBottomBlue = *imagePtr2++, rightBottomGreen = *imagePtr2++, rightBottomRed = *imagePtr2++;
            *scaledPtr++ = (leftTopBlue + rightTopBlue + leftBottomBlue + rightBottomBlue) / 4;
            *scaledPtr++ = (leftTopGreen + rightTopGreen + leftBottomGreen + rightBottomGreen) / 4;
            *scaledPtr++ = (leftTopRed + rightTopRed + leftBottomRed + rightBottomRed) / 4;
        }
    }
    return scaledImage;
}
```


getDoubleSizeRoundImage(原圖) 是將圖片放大兩倍的函式，其中放大的方式是透過原圖 2x2 的像素點延展成 4x4 的像素點，並經由 Bilinear interpolation 的計算取得放大後的像素點。

getBilinearList(原圖 2x2 像素點) 能得到 Bilinear interpolation 計算出 4x4 的像素值。

getBilinearValue(近點像素值，遠點像素值) 能由兩點距離決定目標像素點的數值。

```cpp
Mat getDoubleSizeRoundImage(const Mat& image) {
    int doubleRow = image.rows * 2, doubleCol = image.cols * 2;
    Mat scaledImage = Mat(doubleRow, doubleCol, CV_8UC3);
    for (int row = 0; row < image.rows / 2; row++) {
        const uchar* imagePtr1 = image.ptr<uchar>(row * 2);
        const uchar* imagePtr2 = image.ptr<uchar>(row * 2 + 1);
        uchar* scaledPtr1 = scaledImage.ptr<uchar>(row * 4);
        uchar* scaledPtr2 = scaledImage.ptr<uchar>(row * 4 + 1);
        uchar* scaledPtr3 = scaledImage.ptr<uchar>(row * 4 + 2);
        uchar* scaledPtr4 = scaledImage.ptr<uchar>(row * 4 + 3);
        for (int col = 0; col < image.cols / 2; col++) {
            uchar leftTopBlue = *imagePtr1++, leftTopGreen = *imagePtr1++, leftTopRed = *imagePtr1++;
            uchar rightTopBlue = *imagePtr1++, rightTopGreen = *imagePtr1++, rightTopRed = *imagePtr1++;
            uchar leftBottomBlue = *imagePtr2++, leftBottomGreen = *imagePtr2++, leftBottomRed = *imagePtr2++;
            uchar rightBottomBlue = *imagePtr2++, rightBottomGreen = *imagePtr2++, rightBottomRed = *imagePtr2++;
            vector<uchar> blueBilinear = getBilinearList(vector<uchar>{ leftTopBlue, rightTopBlue, leftBottomBlue, rightBottomBlue });
            vector<uchar> greenBilinear = getBilinearList(vector<uchar>{ leftTopGreen, rightTopGreen, leftBottomGreen, rightBottomGreen });
            vector<uchar> redBilinear = getBilinearList(vector<uchar>{ leftTopRed, rightTopRed, leftBottomRed, rightBottomRed });
            for (int i = 0; i < 4; i++) {
                *scaledPtr1++ = blueBilinear.at(i);
                *scaledPtr1++ = greenBilinear.at(i);
                *scaledPtr1++ = redBilinear.at(i);
            }
            for (int i = 4; i < 8; i++) {
                *scaledPtr2++ = blueBilinear.at(i);
                *scaledPtr2++ = greenBilinear.at(i);
                *scaledPtr2++ = redBilinear.at(i);
            }
            for (int i = 8; i < 12; i++) {
                *scaledPtr3++ = blueBilinear.at(i);
                *scaledPtr3++ = greenBilinear.at(i);
                *scaledPtr3++ = redBilinear.at(i);
            }
            for (int i = 12; i < 16; i++) {
                *scaledPtr4++ = blueBilinear.at(i);
                *scaledPtr4++ = greenBilinear.at(i);
                *scaledPtr4++ = redBilinear.at(i);
            }
        }
    }
    return scaledImage;
}
```

```cpp
vector<uchar> getBilinearList(vector<uchar> source) {
    vector<uchar> result(16, -1);
    result.at(0) = source.at(0);
    result.at(3) = source.at(1);
    result.at(12) = source.at(2);
    result.at(15) = source.at(3);
    vector<int> sidePixel{ 1, 2, 13, 14 };
    for (int index = 0; index < sidePixel.size(); index++) {
        int pixelIndex = sidePixel.at(index);
        int closeIndex = pixelIndex + 1 * pow(-1, pixelIndex % 2);
        int farIndex = pixelIndex - 2 * pow(-1, pixelIndex % 2);
        result.at(pixelIndex) = getBilinearValue(result.at(closeIndex), result.at(farIndex));
    }
    for (int index = 4; index < 12; index++) {
        int closeIndex = index - 4 * pow(-1, index / 8);
        int farIndex = index + 8 * pow(-1, index / 8);
        result.at(index) = getBilinearValue(result.at(closeIndex), result.at(farIndex));
    }
    return result;
}

uchar getBilinearValue(uchar valueClose, uchar valueFar) {
    return (uchar)(valueClose * (2.0 / 3.0) + valueFar * (1.0 / 3.0));
}
```