

Lab 2 - 图像滤波和傅里叶变换

数学与应用数学 徐圣泽 3190102721

本次实验利用 opencv 进行简单的图像滤波，手工实现各滤波函数并得到图像处理结果。

均值滤波

理论

均值滤波是一种最简单的滤波方式。它以像素邻域内的平均值代替原先的像素：

$$\bar{I}(x, y) = \frac{1}{|N|} \sum_{(u,v) \in N} I(x + u, y + v)$$

均值滤波是线性的，即

$$\overline{I + J} = \bar{I} + \bar{J}$$

因此我们可以引入一个卷积核，用空域卷积来表示它，我们定义：

$$K(u, v) = \begin{cases} \frac{1}{|N|} & \text{if } (u, v) \in N, \\ 0 & \text{else.} \end{cases}$$

那么均值滤波又可以写成

$$\bar{I}(x, y) = \sum_{u,v} K(u, v) I(x + u, y + v)$$

从上面的式子可以看到，这里的卷积核 K 与图像域坐标 (x, y) 无关，因此我们说均值滤波是线性的。

代码

本部分我们要实现形如 `BoxFilter <input-image> <output-image> <w> <h>` 的函数。

首先判断参数个数是否正确，这与其他几个部分中是相同的过程。

```
if (argc != 5) {
    cout << "ERROR: Wrong number of parameters." << endl;
    return -1;
}
```

```
in_path ← argv[1];
out_path ← argv[2];
w ← atoi(argv[3]);
h ← atoi(argv[4]);
```

均值滤波最关键的部分是让我们手工写出一个 `kernel` 并利用 `cv::filter2D` 实现卷积运算。

```

kw ← 2 * w + 1;
kh ← 2 * h + 1;
//w和h是读入的参数，是卷积核中心点到边界的距离，卷积核矩阵的大小应当为 (w*2+1)*(h*2+1)
for i ← 0 to kw
    for j ← 0 to kh
        do kernel.at<double>(i, j) ← 1.0 / (kw * kh);
    end for
end for

```

结果

在本实验中取参数 $w=h=3$ ，得到如下结果。



高斯滤波

理论

通常我们认为图像像素之间的相关性随着距离增加应该不断减弱，但是(1)的均值并没有体现这一性质。在对图像进行均值滤波时，如果图像中有一些很显著的亮点，滤波后它的周围会形成光斑。这正是因为均值滤波无视了距离，对很远处的像素依旧采用同样的权重导致的。一些场合，我们为了美感会需要这种效果。另一些场合，这种结果是不利的，特别是在做图像处理时。

高斯滤波是另一种均匀的线性滤波器，它具有如下形式：

$$\bar{I}(x, y) = \sum_{u,v} G(u, v) I(x + u, y + v)$$

其中 $G(u, v) = \frac{1}{2\pi\sigma^2} \exp -\frac{u^2+v^2}{2\sigma^2}$ 是二维高斯核函数。

代码

我们在本题中最关键的部分就是人工实现一个高斯核，并进行卷积，最开始的部分与之前相同。

```
kernel ← 2*floor(5 * sigma)+1; //卷积核矩阵大小
```

下面是构造高斯核的过程，将数学表达式用代码呈现出来，同时我们需要进行归一化。

```
//构造G(u,v)的第一部分
for i ← 0 to kernel
    for j ← 0 to kernel
        do gauss.at<double>(i, j) ← (1 / (2 * PI*sigma*sigma))*exp(-((i -
center)*(i - center) + (j - center)*(j - center)) / (2 * sigma*sigma));
        sum ← sum + gauss.at<double>(i, j);
    end for
end for
```

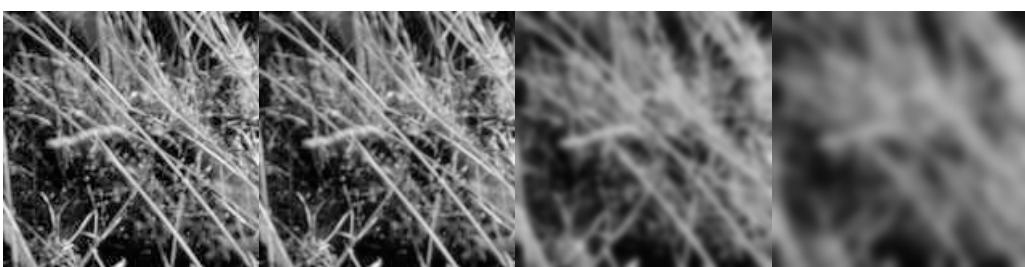
```
//归一化
for i ← 0 to kernel
    for j ← 0 to kernel
        do gauss.at<double>(i, j) ← gauss.at<double>(i, j)/sum;
    end for
end for
```

至此高斯核函数构造完毕，调用 `cv::filter2D` 函数进行卷积。

结果



下面四幅图中，从左至右分别为输入和 $\sigma = 0.5, \sigma = 2, \sigma = 4$ 对应的输出的情况。



我们可以发现，如果 σ 较小，则图像的平滑效果不是很明显，而 σ 较大时平滑效果比较明显。

中值滤波

理论

高斯滤波适用于图像带有高斯噪声情况下的去噪。在图像被非高斯噪声污染的情况下，高斯滤波不一定能得到理想的去噪效果。

中值滤波是一种序统计滤波器（Order-Statistic Filter），序统计滤波器是依据邻域的值在统计上的次序关系来进行过滤的。这里，中值滤波器用邻域内像素亮度的中值来取代原本的像素值，即：

$$\bar{I}(x, y) = \text{Median}\{I(x + u, y + v) | (u, v) \in N\}$$

由定义可见中值滤波器是非线性的。

椒盐噪声（随机的01噪音）不是高斯的，利用前面的卷积滤波方法不能得到很好的结果。中值滤波对于椒盐噪声有比较好的抑制效果。直观上看，在邻域内的样本中，椒盐噪声平均地分布在最大和最小端，通过取中值可以较好地过滤椒盐噪声。

代码

本题只有一个比较关键的部分，那就是人工实现盒装领域的中值滤波，其思想是比较简单的，我们需要获取中值，通过遍历对图实现操作。在这个过程中，我们可以利用 `vector` 进行存储，并且通过 `sort` 函数排序之后用中间下标获得中值。

需要注意的是，我们在遍历过程中会对原图有扩充和还原的操作，以免造成输出时图像大小改变的结果。

```
//对图进行遍历
for i ← 0 to row
    for j ← 0 to col
        vector<vector<int>> color(3);
        Vec3b pixel; //3通道变量用于获取和存储RGB分量
        for x ← i - h + 1 to i + h - 1
            for y ← j - w + 1 to j + w - 1
                //这两个for循环和下面的操作实现了扩充还原，保证输入图与输出图大小相同
                do pixel ← in_img.at<Vec3b>((x + row) % row, (y + col) % col);
                for k ← 0 to 3
                    color[k].push_back(pixel[k]);
                end for
            end for
        end for
    end for
//排序
for k ← 0 to 3
    sort(color[k].begin(), color[k].end());
end for
//取中值
int median ← color[0].size() / 2;
out_img.at<Vec3b>(i, j) ← Vec3b(color[0][median], color[1][median],
color[2][median]);
end for
end for
```

结果

我们对三幅图片进行中值滤波操作，得到下面结果。



我们可以发现，第一幅图经过操作后，改变并不是十分明显，因为其原图被噪声污染的程度较小，而对于第二和第三幅图而言，中值滤波很好地过滤了椒盐噪声，使得图片变得更加平滑。

双边滤波

理论

前面的三种滤波器都会破坏图像的边界，在卷积核很大的时候，均值滤波和高斯滤波都会让边界变得模糊，在邻域很大时，中值滤波会减小边界的曲率。由于物体边界是物体的一个重要特征，很多任务里我们不希望图像边界被破坏。

双边滤波提供了一种降噪同时保持边界的方法。它的思路很简单：如果邻域内像素的亮度差异很大，它在加权平均时的贡献也应当小。我们可以在高斯滤波加权平均的基础上引入一个新的项，反应亮度差带来的加权：

$$\bar{I}(p) = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(|I(p) - I(q)|) I(q)$$

$$W_p = \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(|I(p) - I(q)|)$$

这里 G_{σ_s} 是空间距离上的高斯加权， G_{σ_r} 是灰度距离上的高斯加权。

代码

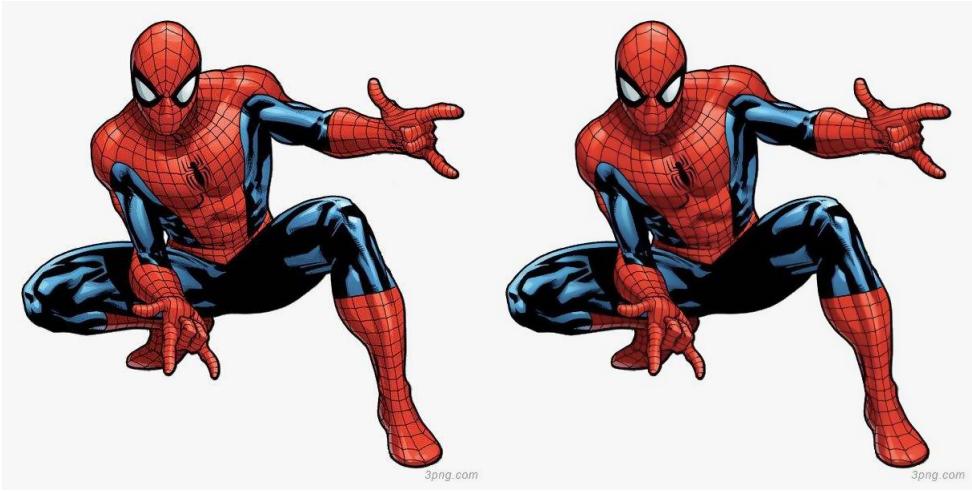
本题的重点在于表示出 W_p 和 $\bar{I}(p)$ ，首先我们需要定义一个函数获取某像素点的 RGB 通道数据。

```
Vec3b color(Mat in_img, int y, int x) {
    return in_img.at<Vec3b>((y + in_img.rows) % in_img.rows, (x + in_img.cols) % in_img.cols);
}
```

接下来只需要将数学表达式转化成代码的形式呈现出来即可。

```
for i ← 0 to row
    for j ← 0 to col
        for k ← 0 to 3
            double wp ← 0;
            double sum ← 0;
            for y ← i - kernel / 2 to i + kernel / 2 + 1
                for x ← j - kernel / 2 to j + kernel / 2 + 1
                    double d ← sqrt(pow(x - j, 2) + pow(y - i, 2));
                    double Id ← abs(color(in_img, i, j)[k] - color(in_img, y, x)
[k]);
                    wp ← wp + Gauss(sigmas, d)*Gauss(sigmar, Id);
                    sum ← sum + Gauss(sigmas, d)*Gauss(sigmar, Id)*color(in_img,
y, x)[k];
                }
            }
            out_img.at<Vec3b>(i, j)[k] ← sum / wp;
        end for
    end for
end for
```

结果



我们可以发现，双边滤波很好地实现了降噪并保持了边界不被破坏。

傅里叶变换

理论

大小为 $W * H$ 的图像 I 的二维傅里叶变换定义为：

$$\hat{I}(\omega_1, \omega_2) = \sum_{x=0}^{W-1} \sum_{y=0}^{H-1} I(x, y) e^{-j2\pi(\frac{x\omega_1}{W} + \frac{y\omega_2}{H})}$$

上式定义在复数域中，因此傅里叶变换的输入和输出都是复数域上的。但我们用实数域表示图像，因此在操作时要额外注意类型：变换到频域时，采用复数表达，变换回图像时，只保留实数部分。

代码

```
#include <opencv2/opencv.hpp>
#include <vector>
#include <math.h>
using namespace std;
using namespace cv;

void fftshift(const Mat &src, Mat &dst) {
    dst.create(src.size(), src.type());

    int rows = src.rows, cols = src.cols;
    Rect roiTopBand, roiBottomBand, roiLeftBand, roiRightBand;

    if (rows % 2 == 0) {
        roiTopBand = Rect(0, 0, cols, rows / 2);
        roiBottomBand = Rect(0, rows / 2, cols, rows / 2);
    }
    else {
        roiTopBand = Rect(0, 0, cols, rows / 2 + 1);
        roiBottomBand = Rect(0, rows / 2 + 1, cols, rows / 2);
    }

    if (cols % 2 == 0) {
        roiLeftBand = Rect(0, 0, cols / 2, rows);
        roiRightBand = Rect(cols / 2, 0, cols / 2, rows);
    }
    else {
        roiLeftBand = Rect(0, 0, cols / 2 + 1, rows);
        roiRightBand = Rect(cols / 2 + 1, 0, cols / 2, rows);
    }

    Mat srcTopBand = src(roiTopBand);
    Mat dstTopBand = dst(roiTopBand);
    Mat srcBottomBand = src(roiBottomBand);
    Mat dstBottomBand = dst(roiBottomBand);
    Mat srcLeftBand = src(roiLeftBand);
    Mat dstLeftBand = dst(roiLeftBand);
    Mat srcRightBand = src(roiRightBand);
    Mat dstRightBand = dst(roiRightBand);
    flip(srcTopBand, dstTopBand, 0);
    flip(srcBottomBand, dstBottomBand, 0);
    flip(dst, dst, 0);
    flip(srcLeftBand, dstLeftBand, 1);
    flip(srcRightBand, dstRightBand, 1);
    flip(dst, dst, 1);
}

int main() {
    Mat I(512, 512, CV_32FC1);
    I = 0;
    I(Rect(256 - 10, 256 - 30, 20, 60)) = 1.0;

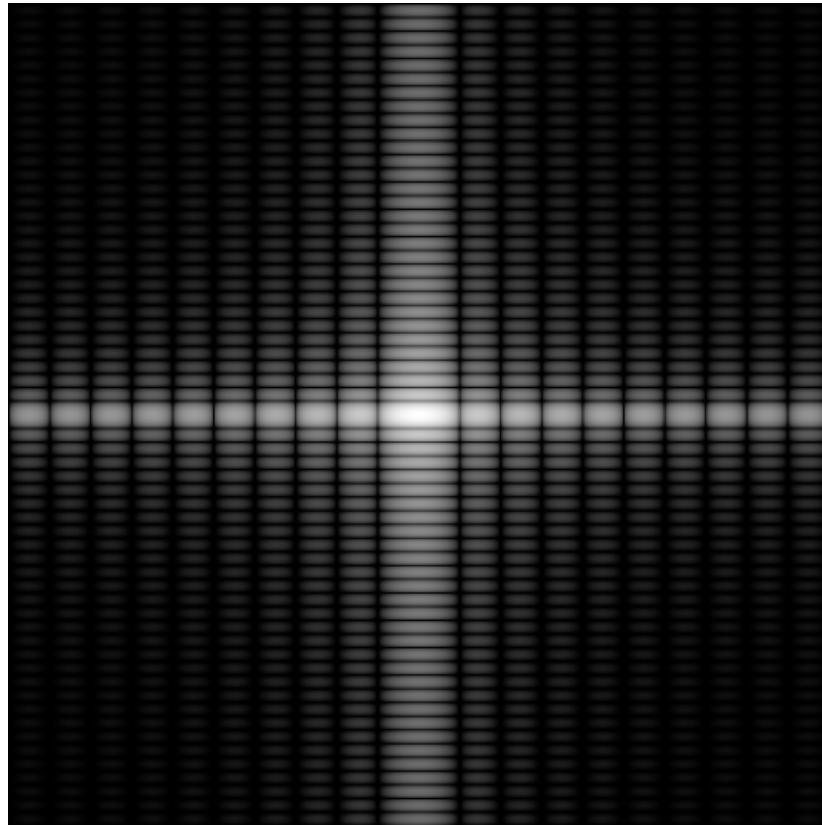
    Mat J(I.size(), CV_32FC2);
    dft(I, J, DFT_COMPLEX_OUTPUT);
    fftshift(J, J);

    Mat Mag;
```

```
vector<Mat> K;
split(J, K); // 将实数和虚数部分分解到 K[0] 和 K[1]
pow(K[0], 2, K[0]); // 计算平方
pow(K[1], 2, K[1]);
Mag = K[0] + K[1]; // 两个分量的平方和

Mat logMag;
Log(Mag + 1, logMag);
normalize(logMag, logMag, 1.0, 0.0, CV_MINMAX);
// ...
imshow("Magnitude", logMag);
waitKey(0);
destroyWindow("in");
destroyWindow("out");
return 0;
}
```

结果



总结

本次实验手工完成了几个常用的库函数，对课程的内容和风格有了较为具体的认识，对于滤波的数学原理理解更加深刻。

作为数学系的学生，本次课程是我第一次接触 C++ 和 OpenCV，在完成作业的过程中，我借鉴学习了网上的资源，虽然刚开始不熟悉课程要求和内容，因此花费的时间和精力较多，但我相信随着课程深入，通过每次作业将理论知识和实践结合起来，应该能跟上班级的进度，希望我也能看到自己的进步。