

# 计算可视媒体课程作业

---

张嘉诚 21215029

计算可视媒体课程作业

摘要

算法原理介绍

Bilateral Filtering

Adaptive Bilateral Filtering

Fast Adaptive Bilateral Filtering

问题的重定义

加速的核心思想

常数级别的运算量

代码实现

滤波核的生成以及图像滤波

值域滤波核的宽度 $\sigma(i)$ 和中心 $\theta(i)$ 的生成

图片中的滤波窗口中的像素值的统计最大值和最小值

积分的递归计算

近似函数 $p_i(t)$ 的求解

Fast Adaptive Bilateral Filtering

Brute Force Adaptive Bilateral Filtering

图片去噪

纹理移除

实验总结

## 摘要

---

边缘感知滤波是计算可视媒体技术中的一项关键技术，由于高斯滤波在去除图像噪声的同时不可避免的会对图像带来一定程度的模糊，从而无法保持图像中物体的边缘，C. Tomasi 等人于1998年提出了双边滤波，Bilateral Filtering。双边滤波通过在空域（spatial domain）高斯滤波的基础上加上了一个考虑邻近位置像素值相似度的值域（range domain）高斯滤波，同时考虑了空域信息和邻接像素范围内的灰度值信息，达到了保边去噪，边缘感知的效果。但是传统的双边滤波，对于每个像素值，它们对应的值域高斯滤波的高斯核是固定的，这对于一些诸如纹理移除等应用不是很友好，所以本作业基于这一点，调查并实现了Adaptive Bilateral Filtering及其加速算法。在Adaptive Bilateral Filtering中，对于每个图像中的每个像素值，对应的值域的高斯核的中心和范围是可以变化的，因此，能够实现在移除原图中出现的一些不必要的纹理的同时，保持图片的边缘信息。原始的Adaptive Bilateral Filtering利用暴力法实现，时间复杂度比较高，在该报告中，利用多项式拟合的方法，实现了一个Fast Adaptive Bilateral Filtering的算法。整个算法的代码利用Python实现，主要参考的论文为：“**Fast Adaptive Bilateral Filtering**,” IEEE Transactions on Image Processing, vol. 28, no. 2, pp. 779-790, Feb. 2019  
<https://arxiv.org/abs/1811.02308v1>

## 算法原理介绍

---

本报告主要介绍的Adaptive Bilateral Filtering即自适应双边滤波及其加速算法的实现。所以首先简单介绍一下这几个算法的基本原理。

### Bilateral Filtering

双边滤波是一种在计算机视觉和图像处理中被广泛运用的边缘保持的滤波方法，在对图像进行滤波平滑的同时能够实现对图像边缘信息的保持。双边滤波主要用到了两个高斯核：range kernel和spatial kernel，其中range kernel负责实现对边缘信息的感知，算法的形式化定义如下：

$$g(i) = \eta(i)^{-1} \sum_{j \in \Omega} \omega(j) \phi_i(f(i-j) - f(i)) f(i-j)$$

$$\text{其中, } \eta(i) = \sum_{j \in \Omega} \omega(j) \phi_i(f(i-j) - f(i))$$

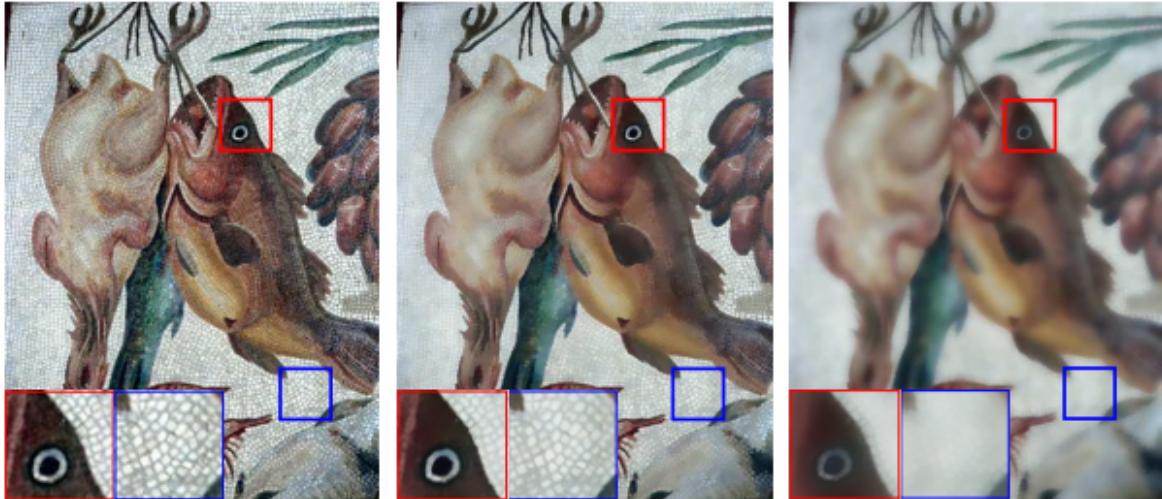
这里的 $f(i)$ 就是图片中在位置*i*处的像素值， $\Omega$ 是以像素*i*为中心的滤波窗口，一般设置为 $[-3\rho, 3\rho]$ ， $\omega$ 和 $\phi$ 分别是空域和值域的高斯核，定义如下：

$$\phi_i(t) = \exp\left(-\frac{t^2}{2\sigma^2}\right)$$

$$\omega(j) = \exp\left(-\frac{\|j\|^2}{2\rho^2}\right)$$

其中 $\sigma$ 和 $\rho$ 分别是值域和空域的高斯滤波核的宽度，这里的值域滤波的中心就是位置*i*处的像素值。从公式中可以看出，**双边滤波的边缘感知的原理**如下：当一个像素的领域范围内的像素值差异很大时，即这个像素的领域中的像素来自边缘的两侧，那么，通过 $\phi$ 这个高斯核，在滤波时赋予邻域范围中的像素的权重就会减少，从而很大程度上被排除在滤波的加权平均的计算中，避免了存在很大像素值差异的像素进行滤波的混合，保证了边缘部分的信息最大程度的被保留。

双边滤波虽然能够在边缘保持方面起到不错的效果，但是，从公式中，我们可以看到，对于每一个进行滤波的像素，它们的值域高斯滤波核的宽度，即 $\sigma$ 都是一样的。高斯滤波核的宽度决定了滤波时的平滑效果， $\sigma$ 越大，平滑效果越好，对噪声的去除更加明显，反之，则平滑效果减弱，但是同时 $\sigma$ 如果越大，对边缘的保持效果也就越差，在边缘区域就会出现模糊的现象，所以这其实是存在着一对矛盾的。例如，在纹理移除的应用中，我们的目标是移除图片中存在的背景纹理，但是尽可能地保留图片的边缘信息，这在很多情景下都是非常有用的，比如在图像美化时，去除照片中人脸上的雀斑等。这样应用对于双边滤波而言，其实并不是能够做得很好，可以看到下面的图中所示：



(a) Input ( $600 \times 450$ ). (b) Bilateral,  $\sigma = 30$ . (c) Bilateral,  $\sigma = 120$ .

我们输入一张带有纹理的照片，当我们把双边滤波中的值域高斯核的 $\sigma$ 设置地稍微小一点的时候，图片中的背景纹理并没有被很好的去除，而当我们把 $\sigma$ 设置地很大的时候，虽然纹理被移除了，但同时，图片的边缘以及整体都变得更加模糊了。因此传统的双边滤波在这样的情景下还需要进一步改进。

## Adaptive Bilateral Filtering

针对传统的双边滤波存在的问题，自适应的双边滤波被提出来对其进行相应的改进。具体来说，自适应的双边滤波针对传统的双边滤波中，去掉了值域高斯滤波的滤波核的中心和宽度固定这一限制，对于每个像素，在进行双边滤波时，值域的高斯核的中心和宽度可以根据该像素的位置进行相应的变化，即值域高斯核的中心和宽度是当前待滤波像素的函数，形式化的定义如下：

$$g(i) = \eta(i)^{-1} \sum_{j \in \Omega} \omega(j) \phi_i(f(i-j) - \theta(i)) f(i-j)$$

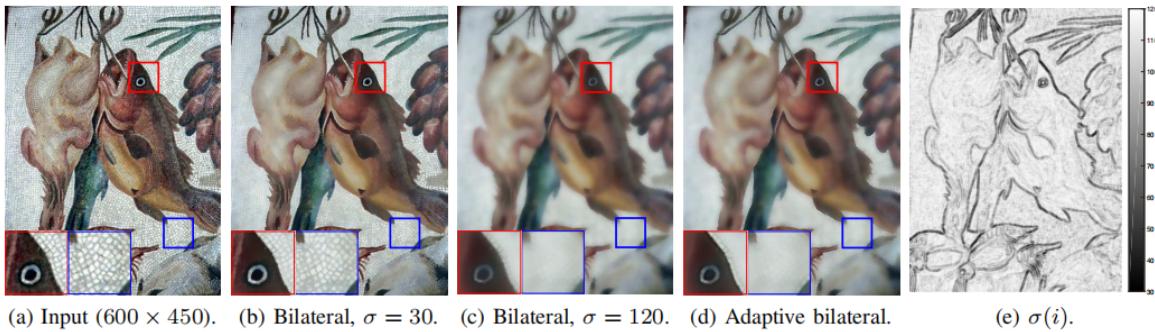
$$\text{其中, } \eta(i) = \sum_{j \in \Omega} \omega(j) \phi_i(f(i-j) - \theta(i))$$

同样的, 这里的 $f(i)$ 就是图片中在位置*i*处的像素值,  $\Omega$ 是以像素*i*为中心的滤波窗口,  $\omega$ 和 $\phi$ 分别是空域和值域的高斯核, 定义如下:

$$\phi_i(t) = \exp\left(-\frac{t^2}{2\sigma(i)^2}\right)$$

$$\omega(j) = \exp\left(-\frac{\|j\|^2}{2\rho^2}\right)$$

可以看到, 自适应双边滤波中值域的高斯滤波核的中心和宽度都和滤波像素*i*有关, 一般来说,  $\theta(i) = f(i) + \zeta(i)$ , 其中 $\zeta(i)$ 是像素*i*的值域高斯滤波核中心的偏移。 $\sigma(i)$ 一般来说, 在自适应的双边滤波中, 在图像的边缘位置的值会比较小, 而在非边缘位置的值会比较大, 从而起到移除图片中的纹理和细节, 同时保持图片的边缘信息。一个例子如下图所示:



可以明显的看到, 对于传统的双边滤波, 自适应双边滤波在移除图片中的纹理信息的同时, 更好的保持了图片中的边缘信息。但是, (自适应) 双边滤波有一个很大的缺点就是计算速度慢。从公式中, 我们可以看到, 我们滤波的窗口大小是 $\Omega = [-3\rho, 3\rho]$ , 我们在这样的一个窗口内对其中的像素进行聚合, 这就导致我们的计算复杂度是 $O(\rho^2)$ 级别的, 这对于一些实时的图片处理应用就非常不友好, 因此, 我们需要一个对上面的(自适应)双边滤波进行加速的算法。

## Fast Adaptive Bilateral Filtering

在“Global color sparseness and a local statistics prior for fast bilateral filtering, TIP 2015”这篇文章中, 作者提出, 可以使用滤波像素领域范围内像素值的统计直方图来近似任意的空域滤波核, 而且这种近似只涉及到对局部的像素的统计直方图的近似, 完全没有涉及到对原始的滤波核的近似计算。基于这种想法, 这里提出了Fast Adaptive Bilateral Filtering, 来加速自适应双边滤波的计算。

接下来就介绍一下该算法的形式化的推导和定义的过程:

### 问题的重定义

首先, 双边滤波本质上是局部像素值的一个加权平均, 因此, 我们可以将Adaptive Bilateral Filtering进行另一种形式的表达:

(1) 定义像素*i*领域内的像素值的集合为:

$$\Lambda_i = \{f(i-j) : j \in \Omega\},$$

并且, 我们定义这样的像素值的集合的最大值和最小值:

$$\alpha_i = \min \{t : t \in \Lambda_i\} \quad \text{and} \quad \beta_i = \max \{t : t \in \Lambda_i\}$$

然后我们统计这个集合中的像素值, 获得邻域范围内的加权后的像素值 (因为已经经过了空域的滤波了) 的统计直方图:

$$h_i(t) = \sum_{j \in \Omega} \omega(j) \delta(f(i-j) - t),$$

其中， $\delta$ 是一个冲击函数，即 $\delta(0) = 1, \delta(x) = 0, x \neq 0$ 。

(2) 原有自适应双边滤波可以用上面定义的领域范围内的统计直方图重新表达：

$$g(i) = \eta(i)^{-1} \sum_{t \in \Lambda_i} t h_i(t) \phi_i(t - \theta(i))$$

$$\eta(i) = \sum_{t \in \Lambda_i} h_i(t) \phi_i(t - \theta(i))$$

这就实现了把空域滤波这一部分转换为局部范围内的像素值的统计直方图。如果在这样的邻域中， $\alpha_i = \beta_i$  即，整个邻域中都是相同的像素值，我们就简单的不考虑在这个邻域中做滤波，因此，我们只考虑 $\alpha_i \neq \beta_i$  的邻域。

### 加速的核心思想

这里的加速算法的核心：我们找到一个简单的连续函数 $p_i(t)$ 来近似离散的 $h_i(t)$ ，然后把上面对领域内加权像素值的统计直方图的离散求和变成一个积分来计算，即：

$$\hat{g}(i) = \hat{\eta}(i)^{-1} \int_{\alpha_i}^{\beta_i} t p_i(t) \phi_i(t - \theta(i)) dt$$

$$\hat{\eta}(i) = \int_{\alpha_i}^{\beta_i} p_i(t) \phi_i(t - \theta(i)) dt$$

而这里的简单，是指保证我们选出来的这样的近似函数 $p_i(t)$ 能够使得上面的积分计算存在一个高效的计算方法。

为了使得上面的计算高效，我们这里假设用多项式函数来近似 $h_i(t)$ ，即：

$$p_i(t) = c_0 + c_1 t + \cdots + c_N t^N$$

所以原来的双边滤波可以表达为：

$$\hat{g}(i) = \hat{\eta}(i)^{-1} \sum_{n=0}^N c_n \int_{\alpha_i}^{\beta_i} t^{n+1} \phi_i(t - \theta(i)) dt$$

$$\hat{\eta}(i) = \sum_{n=0}^N c_n \int_{\alpha_i}^{\beta_i} t^n \phi_i(t - \theta(i)) dt$$

而这里涉及到的积分只有如下这种形式：

$$\int_a^b t^n \exp(-\lambda_i(t - s_i)^2) dt$$

这种积分的计算存在递归关系，因此可以在常数时间内计算得到，假如我们能够选择一个合适的 $N$ 对 $h_i(t)$ 能够有足够的精度的近似，那我们这里整个算法的计算量就是一个常数级别的，这就是整个算法的核心思想。

### 常数级别的运算量

首先，我们要保证我们选择出的 $p_i(t)$ 能够对 $h_i(t)$ 有较好的近似，这里的较好是通过函数的矩来定义的，即我们要求 $p_i(t)$ 和 $h_i(t)$ 的各阶矩都相同，这样就能够保证上述计算的连续积分和原始的离散求和相等。即：

$$m_k = \int_{\alpha_i}^{\beta_i} t^k p_i(t) dt = \sum_{t \in \Lambda_i} t^k h_i(t)$$

通过上面这个等式，我们就能构建关于 $p_i(t)$ 中的系数 $c_1, c_2, \dots, c_N$ 的方程组，从而进行求解：

$$\mathbf{A}c = m$$

其中， $m = (m_0, \dots, m_N)$ ,  $c = (c_0, \dots, c_N)$ ,  $\mathbf{A} = \{\mathbf{A}_{m,n} : 1 \leq m, n \leq N + 1\}$ ,  
 $\mathbf{A}_{m,n} = \frac{1}{m+n-1} (\beta_i^{m+n-1} - \alpha_i^{m+n-1})$

矩阵 $A$ 里的元素是通过求一个形如： $\int_{\alpha_i}^{\beta_i} t^{m+n}$ 的积分得到的。

很显然， $m, c, A$ 都是关于像素*i*的函数。

到这里，我们似乎发现，这里需要计算整张图片每个位置*i*的邻域范围内的最大值 $\beta_i$ ，最小值 $\alpha_i$ ，还需要计算*N*个 $h_i(t)$ 的矩，还需要计算矩阵*A*的逆，这些都是时间复杂度较高，似乎并没有降低时间复杂度，但是其实，这些计算都有线性时间复杂度的算法。

(1) 矩阵每个元素的邻域最值问题可以通过MAX-FILTER算法通过 $O(1)$ 时间复杂度计算得到

(2)  $h_i(t)$ 的矩可以通过 $O(1)$ 计算复杂度通过递归的方式记得得到

(3) 矩阵*A*的计算有点技巧，考虑如果将积分区间从 $\alpha_i$ 到 $\beta_i$ 映射到0到1，即

$\Gamma_i = \left\{ \frac{t - \alpha_i}{\beta_i - \alpha_i} : t \in \Lambda_i \right\} \subseteq [0, 1]$ ，我们就有： $\mathbf{A}_{m,n} = \frac{1}{m+n-1}$ ，即矩阵*A*是一个希尔伯特矩阵，而希尔伯特矩阵的逆可以直接算出来：

$$(\mathbf{A}^{-1})_{m,n} = (-1)^{m+n}(m+n-1) \binom{N+m}{N+1-n} \binom{N+n}{N+1-m} \binom{m+n-2}{m-1}^2$$

在这个时候，原来的 $h_i(t)$ 就可以表达为：

$$H_i(t) = h_i(\alpha_i + t(\beta_i - \alpha_i))$$

原来的函数矩就可以表达为：

$$\mu_k = \sum_{t \in \Gamma_i} t^k H_i(t)$$

原来的矩阵求解就可以表达为：

$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \mathbf{A}^{-1} \begin{bmatrix} \mu_0 \\ \mu_1 \\ \vdots \\ \mu_N \end{bmatrix}$$

所以可以看到，上面三个核心的步骤的计算复杂度都是 $O(1)$ ，因此，我们的算法最后是一个非常高效的算法，这种高效体现在我们的算法的计算时间并不会随着空域高斯核 $\rho$ 的增加而增加。

## 高效的算法实现

(1) 积分区间转换到[0,1]之后，我们的原函数 $h_i(t)$ 的矩和转换后的函数 $H_i(t)$ 的矩存在着一定的关系，可以通过这个关系较为简单获得转换后的 $H_i(t)$ 的矩 $\mu_k$ ，即：

$\mu_0 = m_0$ ，当 $1 \leq k \leq N$ 时，有：

$$\mu_k = (\beta_i - \alpha_i)^{-k} \sum_{r=0}^k \binom{k}{r} (-\alpha_i)^{k-r} m_r$$

通过这样我们就能很快的计算出上面求解 $p_i(t)$ 的系数。

(2) 由于积分区间变换到了[0,1]，我们把原来的双边滤波的公式里的变量也做同样的映射：

首先是 $\theta(i)$ ，将其值映射到 $\theta_0(i) \in [0, 1]$

$$\theta_0(i) = (\beta_i - \alpha_i)^{-1} (\theta(i) - \alpha_i)$$

然后是值域卷积核，原本的 $\phi_i(t) = \exp\left(-\frac{t^2}{2\sigma(i)^2}\right)$ 中的 $t \in [\alpha_i - \beta_i, \beta_i - \alpha_i]$ ，这里通过把 $(\beta_i - \alpha_i)^2$ 放到括号外面，使得 $t \in [-1, 1]$

$$\lambda_i = (2\sigma(i)^2)^{-1} (\beta_i - \alpha_i)^2$$

$$\psi_i(t) = \exp(-\lambda_i t^2)$$

最后，在上面的积分区间的转换后，我们最后的滤波形式化定义可以改写为：

$$F(i) = \frac{\sum_{t \in \Gamma_i} t H_i(t) \psi_i(t - \theta_0(i))}{\sum_{t \in \Gamma_i} H_i(t) \psi_i(t - \theta_0(i))}$$

$$g(i) = \alpha_i + (\beta_i - \alpha_i) F(i)$$

上面  $F(i)$  的分子分母都是一些积分，因为  $H_i(t)$  已经用多项式函数表示了，所以将它分子分母展开就是一些积分的线性组合，积分如下：

$$I_k = \int_0^1 t^k \exp(-\lambda(t - t_0)^2) dt$$

**最后的滤波器的形式：**

$$\hat{g}(i) = \alpha_i + (\beta_i - \alpha_i) \frac{c_0 I_1 + \dots + c_N I_{N+1}}{c_0 I_0 + \dots + c_N I_N}$$

(3) 滤波器中的积分计算。这里的积分存在着递归计算方式，需要推导一下积分过程，这里直接给出最后的递归式：

$$\begin{aligned} I_0 &= \frac{1}{2} \sqrt{\frac{\pi}{\lambda}} \left[ \operatorname{erf}\left(\sqrt{\lambda}(1-t_0)\right) - \operatorname{erf}\left(-\sqrt{\lambda}t_0\right) \right] \\ I_1 &= \int_0^1 t \exp(-\lambda(t - t_0)^2) dt = t_0 I_0 + \frac{1}{2\lambda} \left( \exp(-\lambda t_0^2) - \exp(-\lambda(1-t_0)^2) \right) \\ I_k &= t_0 I_{k-1} + \frac{k-1}{2\lambda} I_{k-2} - \frac{1}{2\lambda} \exp(-\lambda(1-t_0)^2) \end{aligned}$$

## 代码实现

本次实验我使用的是Python来实现，由于Python语言本身执行速度稍慢，所以最后的速度结果和原文有一些出入。这里的应用，我主要是展示两个：**图像去噪，纹理移除**

### 滤波核的生成以及图像滤波

```
# 参考: https://stackoverflow.com/questions/23471083/create-2d-log-kernel-in-opencv-like-fspecial-in-matlab
# 获得LOG滤波器的卷积核
def get_log_kernel(siz, std):
    assert siz%2 == 1, "kernel size must be odd!"
    siz = (siz-1)/2
    x = y = np.linspace(-siz, siz, 2*siz+1)
    x, y = np.meshgrid(x, y)
    arg = -(x**2 + y**2) / (2*std**2)
    h = np.exp(arg)
    h[h < sys.float_info.epsilon * h.max()] = 0
    h = h/h.sum() if h.sum() != 0 else h
    h1 = h*(x**2 + y**2 - 2*std**2) / (std**4)
    return h1 - h1.mean()

# 参考: https://blog.csdn.net/ckghostwj/article/details/12177273
# 获得Gauss滤波器的卷积核
def get_gaussian_kernel(siz, std):
    assert siz%2 == 1, "kernel size must be odd!"
    siz = (siz-1)/2
    x = y = np.linspace(-siz, siz, 2*siz+1)
    x, y = np.meshgrid(x, y)
    arg = -(x**2 + y**2) / (2*std**2)
    h = np.exp(arg)
    h1 = h/np.sum(h)
```

```

    return h1

# 获得正方形均值滤波器的卷积核
def get_square_kernel(siz, val):
    assert siz%2 == 1, "kernel size must be odd!"
    h = np.ones(siz)
    h1 = h*val
    return h1

def imfilter(img, kernel, mode):

    assert len(img.shape) == 2, "only support gray image!"
    h,w = img.shape
    k_size = kernel.shape[0]
    assert k_size%2 == 1, "kernel size must be odd!"

    # padding
    # 边界对称填充
    # 参考: https://blog.csdn.net/it\_flying625/article/details/104592233
    padding = k_size//2
    if mode == "constant":
        dst = np.pad(img, (padding, padding), "constant")
    elif mode == "symmetric":
        dst = np.pad(img, (padding, padding), "symmetric")

    # ps: need convert to float precession
    dst = dst.astype(np.float32)
    # filtering
    for y in range(h):
        for x in range(w):
            dst[y, x] = np.sum(kernel * dst[y: y + k_size, x: x + k_size])

    # 剪切
    dst = dst[: h, :w]

    return dst

```

上面实现了三个滤波核，分别是LoG滤波核，Gaussian滤波核和Square的方形滤波核，生成的方式都是给定滤波核的尺寸和相关参数，按照参数生成对应大小的滤波核。实现图像滤波的时候就是直接拿这些滤波核在进行padding后的图像上进行滑动窗口匹配，然后计算窗口内的滤波值。这里主要提一下这里的LoG滤波，这是高斯拉普拉斯滤波器，即先进行高斯平滑过滤高频信息，然后再进行拉普拉斯算子的计算，以保证后面拉普拉斯算子计算的稳定性。计算的公式如下：

$$LoG(x, y) = \frac{1}{\pi\sigma^4} \left(1 - \frac{x^2+y^2}{\sigma^2}\right) e^{-\frac{x^2+y^2}{2\sigma^2}}$$

## 值域滤波核的宽度 $\sigma(i)$ 和中心 $\theta(i)$ 的生成

$\sigma(i)$ 反应了整个图像的一些边缘信息，或者更准确来说是反应了需要我们的Adaptive Bilateral Filtering滤波的要求，所以这个的生成需要根据不同的应用来变化，但是大致还是要翻译出边缘的位置，因为我们的Adaptive Bilateral Filtering就是希望在边缘部分能够 $\sigma(i)$ 尽量小，能够尽量保持边缘信息。所以这里的生成过程主要是一个高通滤波器 LoG (Laplacian-of-Gaussian filter)，通过这个滤波器就能过滤出图像的边缘信息，然后通过一个线性映射，将边缘部分的高响应值映射为低的 $\sigma(i)$ 值。同时，这里我们还通过一个低通滤波器对原图进行过滤，然后拿原图减去这个模糊后的图像，得到那些被模糊掉的信息作为我们值域滤波器的位移，相当于 $\theta(i) = f(i) + \zeta(i)$ ，即我们希望我们的滤波器在值域滤波的时候能够在这些纹理或者细节部分能够尽可能的过滤掉这些信息。

```

def logClassifier(f, rho, sigma_interval):
    h_log = get_log_kernel(9,9/6)
    f_log = imfilter(f,h_log, 'symmetric')

    L = np.zeros(f.shape)
    L[f_log>60] = 60
    L[f_log<-60] = -60
    mask = (f_log>-60)& (f_log<60)
    L[mask] = f_log[mask]
    L = np.round(L**2)/2

    sigma_r = linearMap(np.abs(L),[0,np.max(np.abs(L))],
    [sigma_interval[1],sigma_interval[0]])

    h_square = get_square_kernel(6*rho+1,(1./(6*rho+1)**2))
    fbar = imfilter(f,h_square, 'symmetric')
    zeta = f - fbar
    return zeta, sigma_r

```

## 图片中的滤波窗口中的像素值的统计最大值和最小值

前面提到，我们在每个像素位置进行滤波时，需要统计这个像素的邻域范围内的像素值的最大值和最小值，以便进行后面的积分求解，而我们的滤波的窗口一般是一个 $w \times w$ 的矩形窗口，相当于是求数组中每个元素的在以自身为中心的 $w \times w$ 的窗口中的最大值和最小值，这其实暴力法实现的话是一个二次方的计算复杂度，但是我们可以通过动态规划的方法，分别进行行和列的最值扫描，通过按照 $w$ 为周期，依次保存最值，可以实现一个高效的算法。但是，需要注意，这里是用Python实现的，Python中的for loop操作比较耗时，所以这里可能并不能体现出这个算法的高效性。

```

def MinMaxFilter(F,w):
    """计算数组F中每个元素的k x k的window中的最大值和最小值

    Args:
        F ([numpy]): [the array that needs to calculate the local min and max
    element]
        w ([int]): [windows of each element in F to calculate the min and max]
    Return:
        Min ([numpy]): each element is the min element of w x w window in F
        Max ([numpy]): each element is the max element of w x w window in F
    """
    minput, ninput = F.shape

    sym = (w-1)/2
    sym = int(sym)

    rowpad = (minput/w + 1 if minput%w != 0 else 0)*w - minput
    columnpad = (ninput/w + 1 if ninput%w != 0 else 0)*w - ninput
    rowpad = int(rowpad)
    columnpad = int(columnpad)

    fmin = np.zeros((minput,ninput),dtype=np.float32)
    fmax = np.zeros((minput,ninput),dtype=np.float32)

    templateMax = np.pad(F,((0,rowpad),(0,columnpad)), 'edge').astype(np.float32)

```

```

templateMin = np.pad(F,((0,rowpad),(0,columnpad)), 'edge').astype(np.float32)

m = minput+rowpad
n = ninput+columnpad

rmax,lmax,rmin,lmin = None, None, None, None
tminptr, tmaxptr = None, None

#! Scan along rows
Lmax = np.zeros((n,))
Lmin = np.zeros((n,))
Rmax = np.zeros((n,))
Rmin = np.zeros((n,))
for ii in range(1,minput+1):
    # 取每一行进行操作
    tmaxptr = templateMax[ii-1]
    tminptr = templateMin[ii-1]
    Lmax[0] = tmaxptr[0]
    Lmin[0] = tminptr[0]

    Rmax[n-1] = tmaxptr[n-1]
    Rmin[n-1] = tminptr[n-1]

    for k in range(2,n+1):
        if (k-1)%w==0:
            Lmax[k-1] = tmaxptr[k-1]
            Rmax[n-k] = tmaxptr[n-k]
            Lmin[k-1] = tminptr[k-1]
            Rmin[n-k] = tminptr[n-k]
        else:
            Lmax[k-1] = Lmax[k-2] if Lmax[k-2]>tmaxptr[k-1] else tmaxptr[k-1]
            Rmax[n-k] = Rmax[n-k+1] if Rmax[n-k+1]>tmaxptr[n-k] else tmaxptr[n-k]
            Lmin[k-1] = Lmin[k-2] if Lmin[k-2]<tminptr[k-1] else tminptr[k-1]
            Rmin[n-k] = Rmin[n-k+1] if Rmin[n-k+1]<tminptr[n-k] else tminptr[n-k]

        for k in range(1,n+1):
            p = k - sym
            q = k + sym
            rmax = -1 if p<1 else Rmax[p-1]
            rmin = np.inf if p<1 else Rmin[p-1]
            lmax = -1 if q>n else Lmax[q-1]
            lmin = np.inf if q>n else Lmin[q-1]

            tmaxptr[k-1] = rmax if rmax>lmax else lmax
            tminptr[k-1] = rmin if rmin<lmin else lmin

#! Scan along columns
Lmax = np.zeros((m,))
Lmin = np.zeros((m,))
Rmax = np.zeros((m,))
Rmin = np.zeros((m,))

for jj in range(1,ninput+1):

```

```

tmaxptr = templateMax[:,jj-1]
tminptr = templateMin[:,jj-1]
Lmax[0] = tmaxptr[0]
Lmin[0] = tminptr[0]
Rmax[m-1] = tmaxptr[m-1]
Rmin[m-1] = tminptr[m-1]
for k in range(2,m+1):
    if (k-1)%w==0:
        Lmax[k-1] = tmaxptr[k-1]
        Rmax[m-k] = tmaxptr[m-k]
        Lmin[k-1] = tminptr[k-1]
        Rmin[m-k] = tminptr[m-k]

    else:
        Lmax[k-1] = Lmax[k-2] if Lmax[k-2]>tmaxptr[k-1] else tmaxptr[k-1]
        Rmax[m-k] = Rmax[m-k+1] if Rmax[m-k+1]>tmaxptr[m-k] else tmaxptr[m-k]
        Lmin[k-1] = Lmin[k-2] if Lmin[k-2]<tminptr[k-1] else tminptr[k-1]
        Rmin[m-k] = Rmin[m-k+1] if Rmin[m-k+1]<tminptr[m-k] else tminptr[m-k]

for k in range(1, m+1):
    p = k - sym
    q = k + sym
    rmax = -1 if p<1 else Rmax[p-1]
    rmin = np.inf if p<1 else Rmin[p-1]
    lmax = -1 if q>m else Lmax[q-1]
    lmin = np.inf if q>m else Lmin[q-1]
    if k<=minput:
        fmax[k-1,jj-1] = rmax if rmax>lmax else lmax
        fmin[k-1,jj-1] = rmin if rmin<lmin else lmin
return fmin, fmax

```

## 积分的递归计算

这里积分的计算是直接按照公式计算的，先算出前面2项，之后就直接按照递推关系进行求解即可。

```

# 计算积分
from scipy import special
def compInt(N,lbda,t0 ):
    I = np.zeros((lbda.shape[0],N+1))
    zero = 0
    rootL = np.sqrt(lbda)
    ulim = lbda*(1-t0)*(1-t0)

    expU = np.exp(-ulim)

    # Compute I_0 and I_1 directly
    I[:,zero] = 0.5 * np.sqrt(np.pi/lbda) * (special.erf(rootL*(1-t0)) -
    special.erf(-rootL*t0))

    I[:,zero+1] = t0*I[:,zero] - (expU - np.exp(-lbda*t0*t0))/(2*lbda)

    # Use recurrence relation for k>1

```

```

for k in range(2,N+1):
    I[:,zero+k] = t0*I[:,zero+k-1] + ((k-1)*I[:,zero+k-2] - expu)/(2*lbda)

return I

```

## 近似函数 $p_i(t)$ 的求解

近似函数 $p_i(t)$ 的求解部分是该算法的核心，也是比较复杂的地方，主要的步骤就是先计算 $p_i(t)$ 的各阶矩 $m_k$ ，然后通过 $m_k$ 和 $\mu_k$ 之间的关系，获得转换积分区间之后的函数的矩，最后计算希尔伯特矩阵的逆，然后进行矩阵相乘，获得拟合函数 $p_i(t)$ 的系数向量。

先通过公式：

$$h_i(t) = \sum_{j \in \Omega} \omega(j) \delta(f(i-j) - t),$$

$$m_k = \int_{\alpha_i}^{\beta_i} t^k p_i(t) dt = \sum_{t \in \Lambda_i} t^k h_i(t)$$

算得原来函数 $h_i(t)$ 的各阶矩，然后通过 $\mu_k$ 和 $m_k$ 的关系：

$$\mu_k = (\beta_i - \alpha_i)^{-k} \sum_{r=0}^k \binom{k}{r} (-\alpha_i)^{k-r} m_r$$

来计算 $H_i(t)$ 的各阶矩，也是我们用来近似的多项式函数的各阶矩（因为我们找这个多项式函数就是通过保证近似前和近似后的函数的各阶矩相同来实现的），有了这个 $\mu$ 的向量之后，就可以通过求解方程

组：
$$\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{bmatrix} = \mathbf{A}^{-1} \begin{bmatrix} \mu_0 \\ \mu_1 \\ \vdots \\ \mu_N \end{bmatrix}$$
 来获得近似多项式函数的系数了。需要注意的是，这里的代码实现上，采用

了递推的方式实现，即在对每个 $k$ 计算它的矩 $\mu_k$ 的时候是通过把求和号展开，利用里面的相邻两项的递推关系来计算的，后面每一项都是前面一项额外乘以新的一项内容得到的，所以这样就不用每一项都从头开始计算，节省了计算量。

```

def fitPolynomial( f, rho, N, Alpha, Beta, filtotype ):
    if filtotype == 'gaussian':
        spker = get_gaussian_kernel(6*rho+1, rho)
    else:
        spker = get_square_kernel(2*rho+1, 1. / (2*rho+1)**2)

    # Find pixels where Alpha=Beta or Alpha=0
    mask = (Alpha != Beta)
    num_pixels = np.count_nonzero(mask)
    Alpha = Alpha[mask]
    Beta = Beta[mask]
    Alpha0_mask = (Alpha==0)
    Alpha_non0 = Alpha[~Alpha0_mask]
    Beta_non0 = Beta[~Alpha0_mask]

    # Filter powers of f
    fpow_filt = np.zeros((np.count_nonzero(mask), N+1))
    fpow = np.ones(f.shape)
    zero = 0
    fpow_filt[:, zero] = 1
    for k in range(1, N+1):
        fpow = fpow*f
        fbar = imfilter(fpow, spker, 'symmetric')

```

```

fpow_filt[:,zero+k] = fbar[mask]
# Compute moments of the shifted histograms (using numerically stable
recursion)
zero = 0
M[:,zero] = 1; # 0th moment is always 1
multiplier = np.ones((np.count_nonzero(~Alpha0_mask),))
Beta_k = np.ones((np.count_nonzero(Alpha0_mask),))
for k in range(1,N+1):
    Beta_k = Beta_k*Beta[Alpha0_mask]
    multiplier = multiplier*(-Alpha_non0/(Beta_non0-Alpha_non0)) #! need to
squeeze the multiplier to 1-dim
    prevTerm = multiplier
    non0_mom = prevTerm
    for r in range(1,k+1):
        temp1 = fpow_filt[:,zero+r]
        temp2 = fpow_filt[:,zero+r-1]
        nextTerm = ((r-k-1)/r) *
temp1[~Alpha0_mask]/(Alpha_non0*temp2[~Alpha0_mask]) *prevTerm
        non0_mom = non0_mom + nextTerm
        prevTerm = nextTerm

    mom_k = np.zeros((num_pixels,))
    mom_k[~Alpha0_mask] = non0_mom #! need to unsqueeze the non0_mom at the
last dim
    temp = fpow_filt[:,zero+k]

    mom_k[Alpha0_mask] = temp[Alpha0_mask]/(Beta_k)
    M[:,zero+k] = mom_k

M = M.T

# Compute polynomial coefficients
Hinv = invhilb(N+1)
C = np.dot(Hinv,M)
C = C.T

return C

```

## Fast Adaptive Bilateral Filtering

最后是快速自适应双边滤波的实现。计算过程就是上面我们描述的算法流程，输入空域滤波的卷积核的宽度 $\rho$ 和计算好的各个位置的值域滤波的卷积核的宽度 $\sigma_r$ ，先根据滤波窗口大小计算出每个窗口内的最值，然后求解每个位置拟合的多项式函数的系数，之后再通过递推式求解滤波器表达式中分子分母上的多项式系数对应的项的积分，最后按照公式用多项式系数和这些积分进行线性组合得到最后的滤波之后的结果。

```

# 快速Adaptive Bilater Filtering
def fastABF( f,rho,sigma_r,theta=None,N=None,filtertype=None ):
    if theta is None:
        theta = f
    if N is None:
        N = 5
    if filtertype is None:
        filtertype = "gaussian"

    [rr,cc] = f.shape

```

```

f = f/255
theta = theta/255
sigma_r = sigma_r/255

# Compute local histogram range
if filtertype == 'gaussian':

    #! 统计MinMaxFilter的用时
    t1 = time.time()
    [Alpha,Beta] = MinMaxFilter(f,6*rho+1)
    t2 = time.time()
    print("Time of MinMaxFilter: ",t2-t1)

elif filtertype == 'box':
    [Alpha,Beta] = MinMaxFilter(f,2*rho+1)
else:
    raise('Invalid filter type')

mask = (Alpha!=Beta);    # Mask for pixels with Alpha~=Beta
a = np.zeros((rr,cc))
a[mask] = 1. / (Beta[mask]-Alpha[mask])

# Compute polynomial coefficients at every pixel
#! 统计fitPolynomial的用时
t1 = time.time()
C = fitPolynomial(f,rho,N,Alpha,Beta,filtertype)
t2 = time.time()
print("Time of FitPolynomial: ",t2-t1)

# Pre-compute integrals at every pixel
zero = 0
t0 = (theta[mask]-Alpha[mask])/(Beta[mask]-Alpha[mask])
lbda = 1/(2*sigma_r[mask]*sigma_r[mask]*a[mask]*a[mask])
#! 统计compInt的用时
t1 = time.time()
I = compInt(N+1,lbda,t0)
t2 = time.time()
print("Time of compInt: ",t2-t1)

# Compute numerator and denominator
Num = np.zeros((np.count_nonzero(mask),))
Den = Num.copy()
for k in range(0,N+1):
    Ck = C[:,zero+k]
    Num = Num + Ck*I[:,zero+k+1]
    Den = Den + Ck*I[:,zero+k]

# Undo shifting & scaling to get output (eq. 29 in paper)
g_hat = np.zeros((rr,cc))
g_hat[mask] = Alpha[mask] + (Beta[mask]-Alpha[mask])*Num/Den
g_hat[~mask] = f[~mask]
g_hat = 255*g_hat

return g_hat

```

## Brute Force Adaptive Bilateral Filtering

为了对比，这里我也实现了一个暴力法实现自适应双边滤波的算法。实现起来也很简单，就是对图片中每个位置进行空域和值域的卷积滤波。循环执行即可：

```
# 定义暴力法的Adaptive Bilateral filtering
def abf_bruteforce(f, rho, sigma_r, theta=None, filtertype=None):
    if theta is None:
        theta = f
    if filtertype is None:
        filtertype = "gaussian"

    [fr, fc] = f.shape

    if filtertype == 'gaussian':
        rad = 3*rho
    elif filtertype == 'box':
        rad = rho

    f = np.pad(f, (rad, rad), 'symmetric')
    f = f.astype(np.float32)

    if filtertype == 'gaussian':
        omega = get_gaussian_kernel(2*rad+1, rho)
    elif filtertype == 'box':
        omega = get_square_kernel(2*rad+1, 1)

    w = np.zeros((fr,fc))
    z = np.zeros((fr,fc))
    for j1 in range(rad+1,rad+fr):
        for j2 in range(rad+1,rad+fc):
            nb = f[j1-rad:j1+rad+1,j2-rad:j2+rad+1]
            r_arg = (nb - theta[j1-rad,j2-rad])**2
            rker = np.exp(-0.5*r_arg/(sigma_r[j1-rad,j2-rad]**2))
            w[j1-rad,j2-rad] = np.sum(np.sum(omega * rker * nb))
            z[j1-rad,j2-rad] = np.sum(np.sum(omega * rker))

    bf = w / (z+1e-6)

    return bf
```

## 图片去噪

基于上面实现的算法，这里首先对实现了一个图像去噪的应用。这里的实验图片如下所示：



可以看到，上面的图片中存在很多的噪点，整个图片看起来不是很清晰，我们可以对其执行自适应的双边滤波和暴力法的自适应双边滤波，实现代码如下：

```
# demo1: sharpening
# 加载图片
sharpening_file = "E:\Code\DataMining\images\peppers_degraded.tif"
f = Image.open(sharpening_file)
plt.figure("original image")
plt.imshow(f,cmap=plt.cm.gray)
f = np.array(f)

rho = 5
N = 5

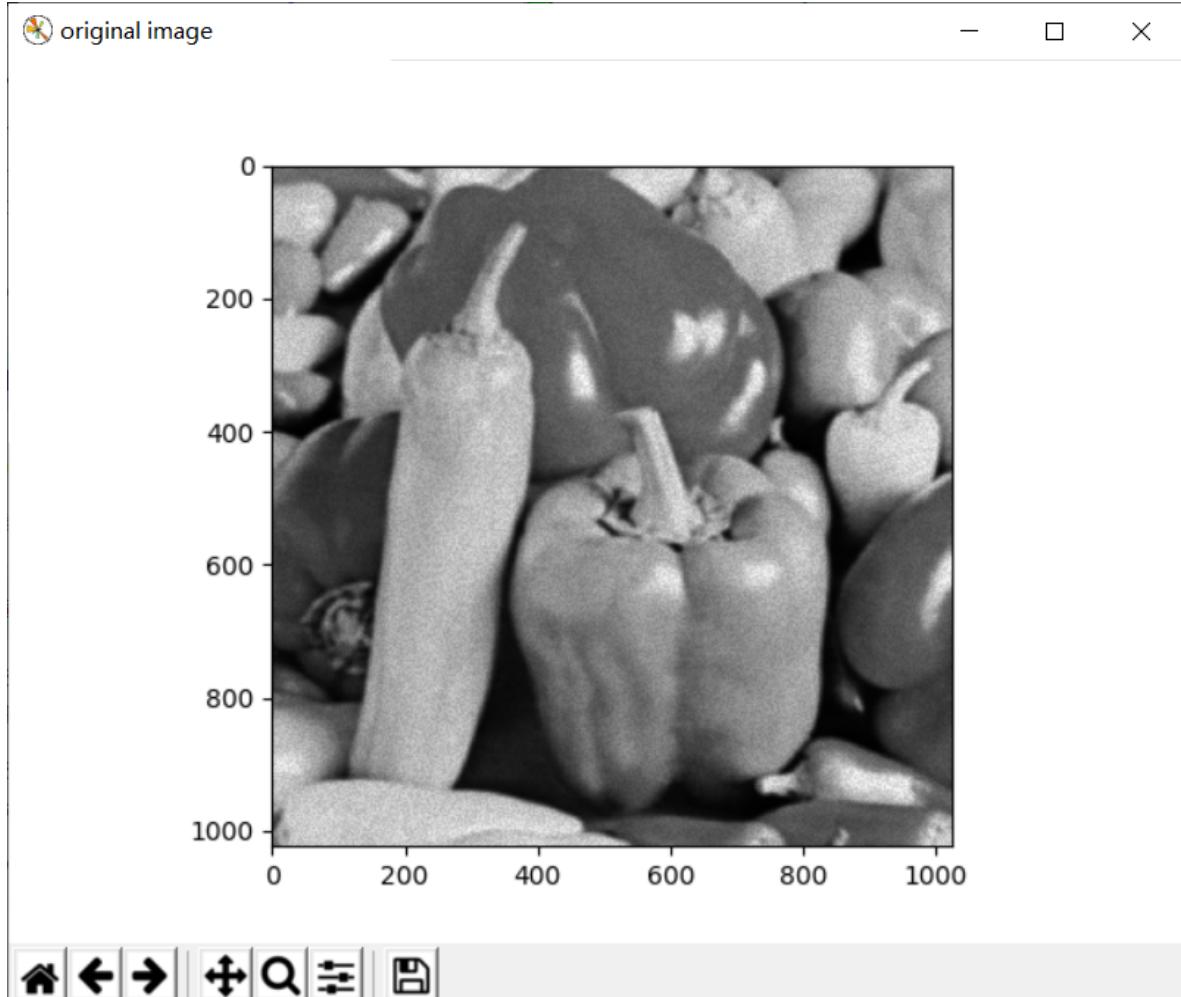
#! 统计计算zeta和sigma_r的用时
t1 = time.time()
[zeta,sigma_r] = logClassifier(f,rho,[23,33])
t2 = time.time()
print("Time of zeta and sigma_r: ",t2-t1)

#! 统计暴力法的用时
t1 = time.time()
g = abf_bruteforce(f,rho,sigma_r,f+zeta)
t2 = time.time()
print("Time of brute force: ",t2-t1)
plt.figure("brute force")
plt.imshow(g,cmap=plt.cm.gray)

#! 统计整个fastABF的用时
```

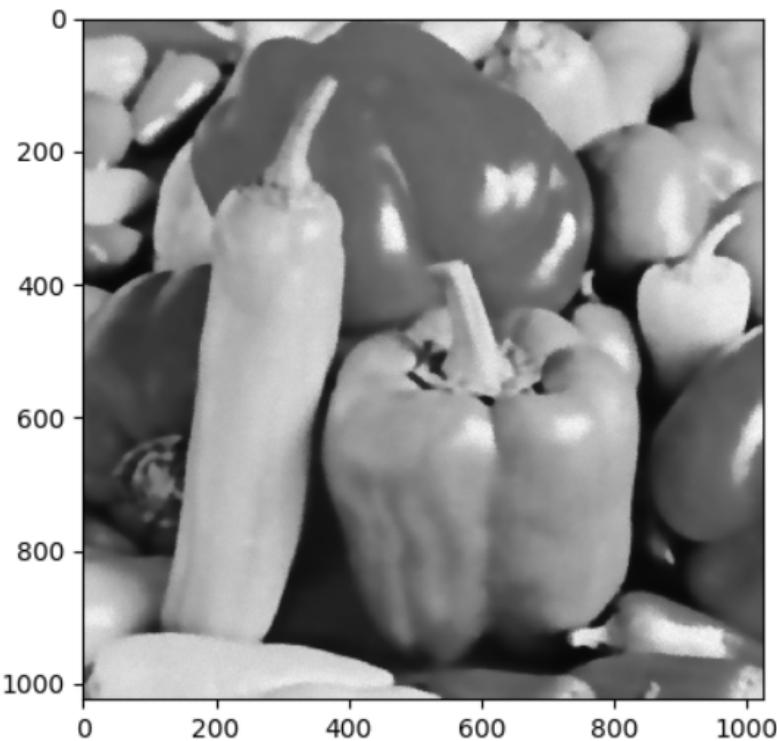
```
t1 = time.time()
g_hat = fastABF(f,rho,sigma_r,f+zeta,N)
t2 = time.time()
print("Total time of fastABF: ",t2-t1)
g_hat[g_hat>255] = 255
g_hat[g_hat<0] = 0
plt.figure("fastABF")
plt.imshow(g_hat,cmap=plt.cm.gray)
plt.show()
```

最终的运行效果如下图所示：



 brute force

- □ ×

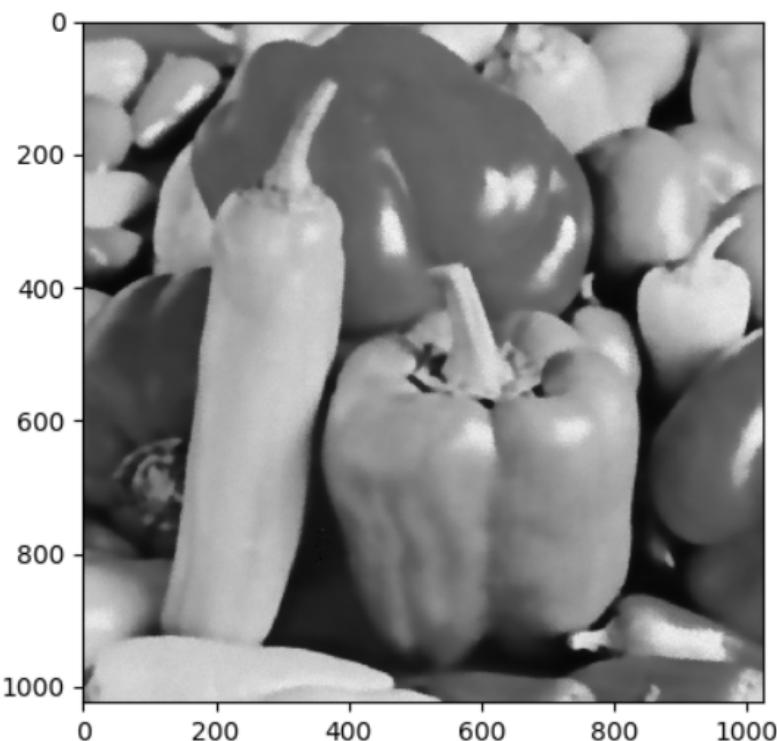


x=40.5043 y=524.245 [62.7]



 fastABF

- □ ×



x=807.95 y=823.465 [147]

可以看到，经过自适应双边滤波之后，有效的去除了原图片中的噪声，原图变得更加清晰，看起来物体表面也变得更加光滑了，视觉效果得到了明显的改善。同时，暴力法和快速自适应双边滤波算法两者的计算用时的统计如下图所示：

```
PS E:\Code\DataMining> & D:/Python3.6.5/python.exe e:/Code/DataMining/fastABF.py
TIFFSetField: tempfile.tif: Unknown pseudo-tag 65538.
Time of zeta and sigma_r: 16.851632595062256
Time of brute force: 51.12547516822815
Time of MinMaxFilter: 7.053017854690552
Time of FitPolynomial: 41.5799925327301
Time of compInt: 0.38796234130859375
Total time of fastABF: 49.27331233024597
PS E:\Code\DataMining>
```

快速自适应双边滤波在计算用时上还是要比暴力法的自适应双边滤波要快的。需要注意的是，这里看上去加速效果并不明显，但其实这主要是以为你这里采用了Python实现，很多的for loop循环导致运行速度变慢，没能展现出加速效果，比如这里的MinMaxFilter函数，这个函数的高效计算在使用了Python实现后并不明显，完全可以采用C++实现。同时这里的多项式拟合部分，也是因为存在多个for loop，导致严重拖慢了运行速度，因此，这个加速效果还远不止这里展示的一样。

## 纹理移除

为了体现自适应双边滤波的优势，这里还实现了一个纹理移除的应用。正如前文所提到的那样，传统的双边滤波，由于值域卷积核在每个像素上都是一样的，所以在纹理移除上并不能够满足我们的要求，这里我们采用的自适应双边滤波，我们的实验图片如下图所示：



可以看到，原图的中存在着很多细小的纹理，这严重影响了图片的视觉效果，我们要做的就是移除这些细小的纹理，同时尽可能的保持图片中的边缘信息不被丢失。这里我们采用自适应双边滤波，实现的代码如下：

```
# demo2: texture removal
# 加载图片
texture_removal_file = "E:\\Code\\DataMining\\images\\fish.jpg"

f = cv2.imread('E:\\Code\\DataMining\\images\\fish.jpg')
f = f[:, :, ::-1] # ndarray

plt.figure("original image")
plt.imshow(f)

print("Image shape:", f.shape)
# 参数
rho_smooth = 2.      # Spatial kernel parameter for smoothing step
rho_sharp = 4.        # Spatial kernel parameter for sharpening step

# Set pixelwise sigma (range kernel parameters) for smoothing

M = mrtv(f, 5)
```

```

sigma_smooth = linearMap(1-M,[0,1],[30,70])

smooth_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,(5,5))
sigma_smooth = cv2.dilate(sigma_smooth, smooth_kernel) # Clean up the fine
noise

g = f.copy()
g = g.astype(np.float32)

for it in range(0,2):
    out = np.zeros(f.shape,dtype=np.float32)
    for k in range(0,f.shape[2]):

        out[:, :, k] = fastABF(g[:, :, k],rho_smooth,sigma_smooth,None,4)

        out[out>255] = 255.
        out[out<0] = 0.
        g = out
        sigma_smooth = sigma_smooth*0.8

g_gray = cv2.cvtColor(g.astype(np.uint8),cv2.COLOR_RGB2GRAY)
g_gray = g_gray.astype(np.float32)

[zeta,sigma_sharp] = logClassifier(g_gray,rho_sharp,[30,31])
for it in range(0,1):      # Run more iterations for greater sharpening
    for k in range(0,f.shape[2]):
        g[:, :, k] = fastABF(g[:, :, k],rho_sharp,sigma_sharp,g[:, :, k]+zeta,5)
        g[g>255] = 255.
        g[g<0] = 0.

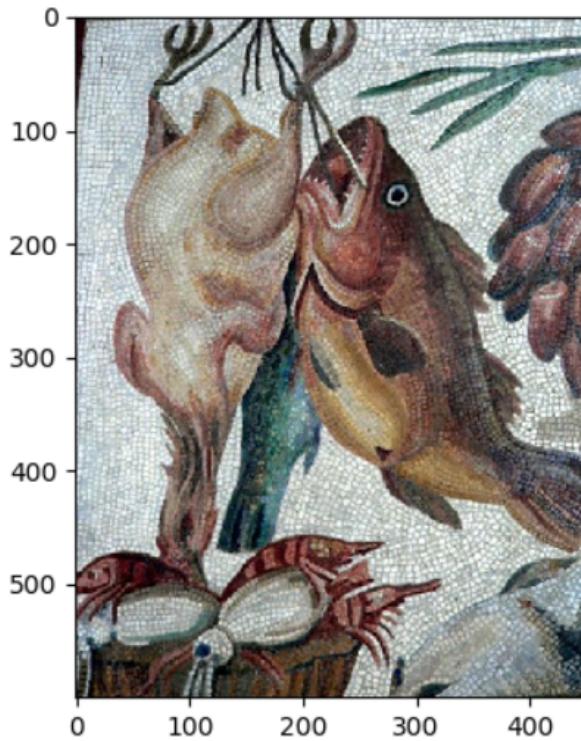
plt.figure("after texture removal")
plt.imshow(g.astype(np.uint8))
plt.show()

```

需要注意的是，我们的自适应双边滤波由于值域卷积核的宽度是可以变化的，我们不仅可以用来做去噪平滑，我们还可以用来做锐化，所以这里我们的纹理移除是先通过mrtv函数（这是一个图像处理里面用的一个区分纹理和边缘的一个计算量）来计算出滤波时去除纹理所需要的各个位置的值域滤波核的宽度，即 `sigma_smooth`（经过了线性映射），并通过腐蚀操作，细化这个 `sigma_smooth`，以求更好的保持边缘信息。之后我们对原来的RGB图三个通道都做2次的双边滤波平滑，移除纹理，同时为了更好的强调边缘，我们还利用了自适应双边滤波进行了一次锐化操作，此时由于纹理大部分已经被移除，我们的滤波窗口 `rho_sharp` 可以设置得大一点，同时，利用原图的灰度图，计算出此时的值域滤波核的宽度分布，要求边缘部分更小，从而起到锐化边缘的效果。

 original image

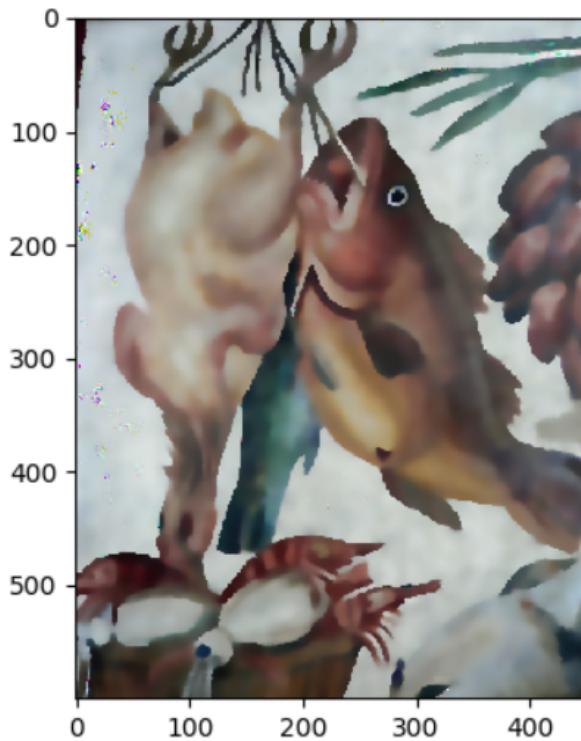
- □ ×



x=221.253 y=230.669 [175, 139, 117]

 after texture removal

- □ ×



可以看到，经过自适应双边滤波后，原图像中的细小的纹理被移除了，同时图像中原本的原因信息也得到了较好的保持，这对于传统的双边滤波是比较困难的（可以参看前文的图）。计算时间的统计如下图所示：

```
PS E:\Code\DataMining> & D:/Python3.6.5/python.exe e:/Code/DataMining/texture.py
Image shape: (600, 450, 3)
Time of MinMaxFilter: 1.9787027835845947
Time of FitPolynomial: 6.527859210968018
Time of compInt: 0.09577012062072754
Time of MinMaxFilter: 1.7602415084838867
Time of FitPolynomial: 6.548432350158691
Time of compInt: 0.09374880790710449
Time of MinMaxFilter: 1.7553291320800781
Time of FitPolynomial: 6.394888639450073
Time of compInt: 0.09275317192077637
Time of MinMaxFilter: 1.7452826499938965
Time of FitPolynomial: 6.741918563842773
Time of compInt: 0.07978677749633789
Time of MinMaxFilter: 1.7901830673217773
Time of FitPolynomial: 6.526902437210083
Time of compInt: 0.08077287673950195
Time of MinMaxFilter: 1.9039053916931152
Time of FitPolynomial: 6.549476861953735
Time of compInt: 0.09378981590270996
g_gray shape: (600, 450)
Time of MinMaxFilter: 1.71114182472229
Time of FitPolynomial: 10.064823389053345
Time of compInt: 0.10666966438293457
Time of MinMaxFilter: 1.7692663669586182
Time of FitPolynomial: 10.079931259155273
Time of compInt: 0.10072994232177734
Time of MinMaxFilter: 1.7932219505310059
Time of FitPolynomial: 10.379193544387817
Time of compInt: 0.11768531799316406
```

平滑过程

锐化过程

整体的计算速度还是比较快的。

## 实验总结

本次课程实验，主要调查并实现了一种对传统双边滤波算法的拓展算法，即自适应双边滤波算法，及其加速算法。自适应双边滤波通过去掉双边滤波中值域滤波核固定的限制，使得每个位置的像素在滤波时可以自适应的选择值域滤波核的中心位置和滤波核的宽度 $\sigma(i)$ ，在诸如纹理移除这一类应用中取得了相较于传统双边滤波核更好的效果。同时，本次实验还实现了自适应双边滤波算法的加速算法，这里的加速的核心是利用滤波像素局部的像素值统计直方图来近似空域滤波，通过一个连续的多项式函数来近似这样的统计直方图，同时将离散的求和项转换为存在递推高效计算的积分求和，使得原本 $O(\rho^2)$ 的计算复杂度降低为和 $\rho$ 无关的计算量，因为我们总是可以选择一个合理的多项式的阶数，使得其对局部的像素值统计直方图的近似达到让我们满意的结果进而进行高效计算。

本次课程实验从阅读论文开始，整体的了解了双边滤波的原理及其相关的发展，接触到了诸如利用傅里叶近似等各种双边滤波算法的改进工作，通过对这些论文的深入了解才有了这次的实验报告，最后再编程实现这些算法，总体上收获非常大。但是这次实验还是存在着一些问题，正如前文所说，我这里并没有使得这个加速算法的效果更加明显的展示出来，主要是因为Python语言对for loop这类循环非常不友好，因此在实验的时候，我还尝试了一些诸如Python调用C++函数来进行运算，以及使用numba库对Python代码进行加速等方法，但是最后都遇到了一些问题，没有成功的加速，所以这也是值得改进的地方。