数字图像处理
第一次作业

董浩

自动化少 61

2140506111

2019 年 3 月 4 日

摘要：

本次作业内容包括 bmp 文件格式了解，对图像灰度级进行改变，计算图像的均值方差，通过插值的方法放大图像以及对图像进行仿射变换等等。本次作业报告的内容首先对基本原理进行了介绍，然后通过编程的方式实现，并打印出图像处理结果。编程实现的方法分为调用库函数和自己编写函数两种方法。编程环境为 Python3.6.3 和 Opencv-python3.4.2。

一、BMP 图像格式简介，以 7.bmp 为例说明

BMP 文件格式,又称为 Bitmap(位图)或是 DIB(Device-Independent Device)，是 Windows 系统中广泛使用的图像文件格式。由于它可以不作任何变换地保存图像像素域的数据，因此成为我们取得 RAW 数据的重要来源。Windows 的图形用户界面也在它的内建图像子系统 GDI 中对 BMP 格式提供了支持。

下面以 UltraEdit 软件为分析工具，结合 Windows 的位图数据结构对 BMP 文件格式进行一个深度的剖析。BMP 文件的数据按照从文件头开始的先后顺序分为四个部分：**bmp 文件头**(bmp file header)：提供文件的格式、大小等信息；**位图信息头**(bitmap information)：提供图像数据的尺寸、位平面数、压缩方式、颜色索引等信息；**调色板**(color palette)：可选，如使用索引来表示图像，调色板就是索引与其对应的颜色的映射表；**位图数据**(bitmap data)：就是图像数据。下面结合 Windows 结构体的定义，通过一个表来分析这四个部分。

数据段名称	对应的Windows结构体定义	大小(Byte)
bmp文件头	BITMAPFILEHEADER	14
位图信息头	BITMAPINFOHEADER	40
调色板		由颜色索引数决定
位图数据		由图像尺寸决定

我们一般见到的图像以 24 位图像为主，即 R、G、B 三种颜色各用 8 个 bit 来表示，这样的图像我们称为真彩色，这种情况下是不需要调色板的，也就是所位图信息头后面紧跟的就是位图数据了。因此，我们常常见到有这样一种说法：位图文件从文件头开始偏移 54 个字节就是位图数据了，这其实说的是 24 或 32 位图的情况。这也就解释了我们按照这种程序写出来的程序为什么对某些位图文件没用了。

下面针对一幅特定的图像进行分析，来看看在位图文件中这四个数据段的排布以及组成。我们通过 UltraEdit 软件读取 7.bmp 文件得到下图的数据：

```
            0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 6E 04 00 00 00 00 00 00 36 04 00 00 28 00 ; BMn.......6...(.
00000010h: 00 00 07 00 00 00 07 00 00 00 01 00 08 00 00 00 ; ................
00000020h: 00 00 38 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ..8.............
00000030h: 00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02 ; ................
00000040h: 02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06 ; ................
00000050h: 06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A ; ................
```

1.bmp 文件头

变量名	地址偏移	大小	作用
bfType	0000h	2 bytes	说明文件的类型，可取值为： • BM - Windows 3.1x, 95, NT, ... • BA - OS/2 Bitmap Array • CI - OS/2 Color Icon • CP - OS/2 Color Pointer • IC - OS/2 Icon • PT - OS/2 Pointer
bfSize	0002h	4 bytes	说明该位图文件的大小，用字节为单位
bfReserved1	0006h	2 bytes	保留，必须设置为0
bfReserved2	0008h	2 bytes	保留，必须设置为0
bfOffBits	000Ah	4 bytes	说明从文件头开始到实际的图象数据之间的字节的偏移量。 这个参数是非常有用的，因为位图信息头和调色板的长度会根据不同情况而变化， 所以我们可以用这个偏移值迅速的从文件中读取到位图数据。

对照文件数据我们看到：

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 6E 04 00 00 00 00 00 00 36 04 00 00 28 00 ; BMn.......6...(.
00000010h: 00 00 07 00 00 00 07 00 00 00 01 00 08 00 00 00 ; ................
00000020h: 00 00 38 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ..8.............
00000030h: 00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02 ; ................
00000040h: 02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06 ; ................
00000050h: 06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A ; ................
```

1-2：424dh = 'BM',表示这是 Windows 支持的位图格式。有很多声称开头两个字节必须为'BM'才是位图文件，从上表来看应为开头两个字节必须为'BM'才是 Windows 位图文件。

3-5：0000046Eh = 1134B = 1.1kB，通过查询文件属性发现一致。

6-9：这是两个保留段，为 0。

A-D：00000436h = 1078。即从文件头到位图数据需偏移 1078 字节。我们稍后将验证这个数据。

共有 14 个字节。

2.位图信息头

变量名	地址偏移	大小	作用
biSize	000Eh	4 bytes	BITMAPINFOHEADER结构所需要的字数。
biWidth	0012h	4 bytes	说明图像的宽度，用像素为单位
biHeight	0016h	4 bytes	说明图像的高度，以像素为单位。 注：这个值除了用于描述图像的高度之外，它还有另一个用处，就是指明该图像是倒向的位图，还是正向的位图。 如果该值是一个正数，说明图像是倒向的，如果该值是一个负数，则说明图像是正向的。 大多数的BMP文件都是倒向的位图，也就是高度值是一个正数。
biPlanes	001Ah	2 bytes	为目标设备说明颜色平面数， 其值将总是被设为1。
biBitCount	001Ch	2 bytes	说明比特数/像素，其值为1、4、8、16、24或32。
biCompression	001Eh	4 bytes	说明图像数据压缩的类型。取值范围： 0 BI_RGB 不压缩（最常用） 1 BI_RLE8 8比特游程编码（RLE），只用于8位位图 2 BI_RLE4 4比特游程编码（RLE），只用于4位位图 3 BI_BITFIELDS 比特域，用于16/32位位图 4 BI_JPEG JPEG 位图含JPEG图像（仅用于打印机） 5 BI_PNG PNG 位图含PNG图像（仅用于打印机）
biSizeImage	0022h	4 bytes	说明图像的大小， 以字节为单位。当用BI_RGB格式时，可设置为0。
biXPelsPerMeter	0026h	4 bytes	说明水平分辨率，用像素/米表示，有符号整数
biYPelsPerMeter	002Ah	4 bytes	说明垂直分辨率，用像素/米表示，有符号整数
biClrUsed	002Eh	4 bytes	说明位图实际使用的彩色表中的颜色索引数 （设为0的话，则说明使用所有调色板项）
biClrImportant	0032h	4 bytes	说明对图像显示有重要影响的颜色索引的数目 如果是0，表示都重要。

对照数据文件：

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 6E 04 00 00 00 00 00 00 36 04 00 00 28 00 ; BMn.......6...(.
00000010h: 00 00 07 00 00 00 07 00 00 00 01 00 08 00 00 00 ; ................
00000020h: 00 00 38 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ..8.............
00000030h: 00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02 ; ................
00000040h: 02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06 ; ................
00000050h: 06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A ; ................
```

0E-11：00000028h = 40,这就是说我这个位图信息头的大小为 40 个字节。前面我们已经说过位图信息头一般有 40 个字节，既然是这样，为什么这里还要给一个字段来说明呢？这里涉及到一些历史,其实位图信息头原本有很多大小的版本的。我们看一下下表：

大小	文件头	标识	兼容性（被哪些GDI支持）
12	OS/2 V1	BITMAPCOREHEADER	OS/2以及Windows 3.0以上的Windows版本
64	OS/2 V2	BITMAPCOREHEADER2	
40	Windows V3	BITMAPINFOHEADER	Windows 3.0以上的Windows版本
108	Windows V4	BITMAPV4HEADER	Windows 95/NT4以上的Windows版本
124	Windows V5	BITMAPV5HEADER	Windows 98/2000及其新版本

出于兼容性的考虑，大多数应用使用了旧版的位图信息头来保存文件。
而 OS/2 已经过时了,因此现在最常用的格式就仅有 V3 header 了。因此，我们在前面说位图信息头的大小为 40 字节。
12-15：00000007h = 7，图像宽为 7 像素，与文件属性一致。
16-19：00000007h = 7，图像高为 7 像素，与文件属性一致。这是一个正数，说明图像数据是从图像左下角到右上角排列的。
1A-1B：0001h， 该值总为 1。
1C-1D：0008h = 8， 表示每个像素占 8 个比特，即该图像共有 256 种颜色。
1E-21：00000000h，BI_RGB，说明本图像不压缩。
22-25：00000038h，位图全部像素占用的字节数。
26-29：00000000h，水平分辨率，缺省。
2A-2D：00000000h，垂直分辨率，缺省。
2E-31：00000000h,说明使用所有调色板项。
32-35：00000000h,说明颜色索引都重要。

3.调色板

下面的数据就是调色板了。前面也已经提过，调色板其实是一张映射表，标识颜色索引号与其代表的颜色的对应关系。它在**文件中的布局就像一个二维数组 palette[N][4],其中 N 表示总的颜色索引数，每行的四个元素分别表示该索引对应的 B**、G、R 和 Alpha 的值，每个分量占一个字节。如不设透明通道时，Alpha 为 0。因为前面知道，本图有 256 个颜色索引，因此 N = 256。索引号就是所在行的行号，对应的颜色就是所在行的四个元素。这里截取一些数据来说明：

```
          0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00000000h: 42 4D 6E 04 00 00 00 00 00 00 36 04 00 00 28 00 ; BMn.......6...(.
00000010h: 00 00 07 00 00 00 07 00 00 00 01 00 08 00 00 00 ; ................
00000020h: 00 00 38 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ..8.............
00000030h: 00 00 00 00 00 00 00 00 00 00 01 01 01 00 02 02 ; ................
00000040h: 02 00 03 03 03 00 04 04 04 00 05 05 05 00 06 06 ; ................
00000050h: 06 00 07 07 07 00 08 08 08 00 09 09 09 00 0A 0A ; ................
```

索引：(蓝，绿，红，Alpha)
0 号：(00，00，00，00)
1 号：(01，01，01，00)
2 号：(02，02，02，00)
3 号：(03，03，03，00)
4 号：(04，04，04，00)
5 号：(05，05，05，00) 等等。

一共有 256 种颜色，每个颜色占用 4 个字节，就是一共 1024 个字节，再加上前面的文件信息头和位图信息头的 54 个字节加起来一共是 1078 个字节。也就

是说在位图数据出现之前一共有 1078 个字节，与我们在文件信息头得到的信息：文件头到文图数据区的偏移为 1078 个字节一致！

4.位图数据

下面就是位图数据了，每个像素占一个字节，取得这个字节后，以该字节为索引查询相应的颜色，并显示到相应的显示设备上就可以了。注意：由于位图信息头中的图像高度是正数，所以位图数据在文件中的排列顺序是从左下角到右上角，以行为主序排列的。

```
00000430h: FE 00 FF FF FF 00 67 63 64 54 56 62 62 00 62 65 ; þ.ÿÿÿ.gcdTVbb.be
00000440h: 66 56 45 47 5F 00 61 5C 5B 63 48 47 52 00 58 4B ; fVEG_.a\[cHGR.XK
00000450h: 55 65 5A 5B 46 00 68 47 3F 69 5D 4C 2A 00 61 59 ; UeZ[F.hG?i]L*.aY
00000460h: 5A 5F 47 28 45 00 52 52 49 3B 37 50 5A 00       ; Z_G(E.RRI;7PZ.
```

也即我们见到的第一个像素 67 是图像最左下角的数据，第二个像素 63 为图像最后一行第二列的数据，…一直到最后一行的最后一列数据，后面紧接的是倒数第二行的第一列的数据，依此类推。如果图像是 24 位或是 32 位数据的位图的话，位图数据区就不是索引而是实际的像素值了。下面说明一下，此时位图数据区的每个像素的 RGB 颜色阵列排布：

24 位 RGB 按照 BGR 的顺序来存储每个像素的各颜色通道的值，一个像素的所有颜色分量值都存完后才存下一个下一个像素，不进行交织存储。

32 位数据按照 BGRA 的顺序存储，其余与 24 位位图的方式一样。像素的排布规则与前述一致。

二、把 lena 512*512 图像灰度级逐级递减 8-1 显示

图像灰度级数我们见得最多的就是 256 了，如果想调整它的灰度级数，比如一个灰度级 256 的图调整灰度为 4，第一步我们就需要求出每个 block 的大小（也可以理解为 256 应该分为几个区间）。256 / (4-1) = 3 个区间，每个区间 size 是 85。然后我们再求出每个区间的中值，如果原图 pixel 值比这个中间值大，那他就属于这个区间的最大值，否则它就是属于上一个区间的最大值。

实验结果如下：

8 灰度级：

7 灰度级：

6 灰度级：

5 灰度级：

4 灰度级：

3 灰度级：

2 灰度级：

1 灰度级：

结果分析：对一幅 512*512，256 个灰度级的具有较多细节的图像，保持空间分辨率不变，仅将灰度级数依次递减为 128、64、32、16、8、4、2，比较得到的结果就可以发现灰度级数对图像的影响。前四张图灰度级数较高，图像基本看不出什么变化。当灰度级数继续降低，则在灰度缓变区常会出现一些几乎看不出来的非常细的山脊状结构。灰度级数越低，越不能将图像的细节刻画出来。对比实验中处理过的图像，可以发现，虽然都是灰度图，但是灰度范围越大则图像显示出的色彩越丰富。

三、计算 lena 图像的均值方差

均值和方差是验证图像质量的常用指标。其中均值反映了图像的亮度，均值越大说明图像亮度越大，反之越小。方差反映了图像像素值与均值的离散程度，标准差越大说明图像的质量越好。OpenCV 提供了 meanStdDev()函数用于计算一个矩阵的均值和标准差，它的声明如下：mean, stddev=cv.meanStdDev(src[, mean[, stddev[, mask]]])，函数参数：src：输入的源图像或矩阵；mean：输出的均值矩阵；stddev：输出的标准差矩阵；mask：可选的掩码矩阵。计算得到标准差后，平方即可得到方差。计算公式为：

$$N = \sum_{I,\mathrm{mask}(I)\neq 0} 1$$

$$\mathrm{mean}_c = \frac{\sum_{I:\ \mathrm{mask}(I)\neq 0} \mathrm{src}(I)_c}{N}$$

$$\mathrm{stddev}_c = \sqrt{\frac{\sum_{I:\ \mathrm{mask}(I)\neq 0} (\mathrm{src}(I)_c - \mathrm{mean}_c)^2}{N}}$$

实验结果如下：

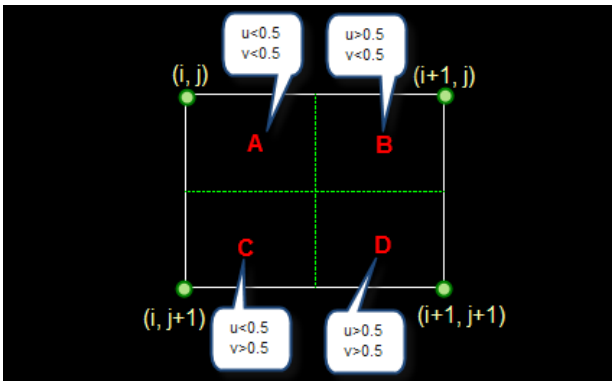
```
均值： [ 99.05121613]
标准差： [ 52.87751733]
方差： [ 2796.03183888]
```

四、把 lena 图像用近邻、双线性和双三次插值法 zoom 到 2048*2048

图像插值就是利用已知邻近像素点的灰度值（或 rgb 图像中的三色值）来产生未知像素点的灰度值，以便由原始图像再生出具有更高分辨率的图像。在图像

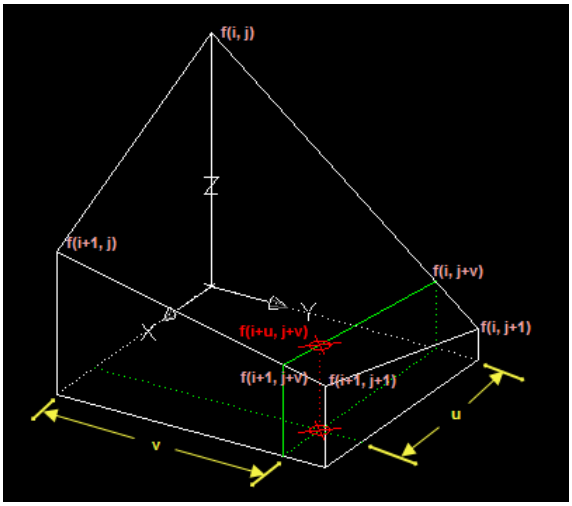
放大过程中，像素也相应地增加，增加的过程就是“插值”发生作用的过程，“插值”程序自动选择信息较好的像素作为增加、弥补空白像素的空间，而并非只使用临近的像素，所以在放大图像时，图像看上去会比较平滑、干净。不过需要说明的是插值并不能增加图像信息，尽管图像尺寸变大，但效果也相对要模糊些，过程可以理解为白酒掺水。插值几乎应用于所有需要进行图像缩放功能的领域内，如数码相机、图像处理软件（如 Photoshop）。常用的插值方法有最近邻、双线性和双三次插值法等等。

最近邻插值是最简单的一种插值方法，不需要计算，在待求象素的四邻象素中，将距离待求象素最近的邻象素灰度赋给待求象素。设 i+u，j+v(i，j 为正整数，u，v 为大于零小于 1 的小数，下同)为待求象素坐标，则待求象素灰度的值 f(i+u，j+v) 如下图所示：



如果(i+u，j+v)落在 A 区，即 u<0.5，v<0.5，则将左上角象素的灰度值赋给待求象素，同理，落在 B 区则赋予右上角的象素灰度值，落在 C 区则赋予左下角象素的灰度值，落在 D 区则赋予右下角象素的灰度值。最邻近元法计算量较小，但可能会造成插值生成的图像灰度上的不连续，在灰度变化的地方可能出现明显的锯齿状。

双线性内插法是利用待求象素四个邻象素的灰度在两个方向上作线性内插，如下图所示：



对于(i，j+v)，f(i，j)到 f(i，j+1)的灰度变化为线性关系，则有：

f(i， j+v) = [f(i， j+1) - f(i， j)] * v + f(i， j)

同理对于(i+1，j+v)则有：

f(i+1，j+v) = [f(i+1，j+1) - f(i+1，j)] * v + f(i+1， j)

从 f(i, j+v)到 f(i+1, j+v)的灰度变化也为线性关系，由此可推导出待求象素灰度的计算式如下：

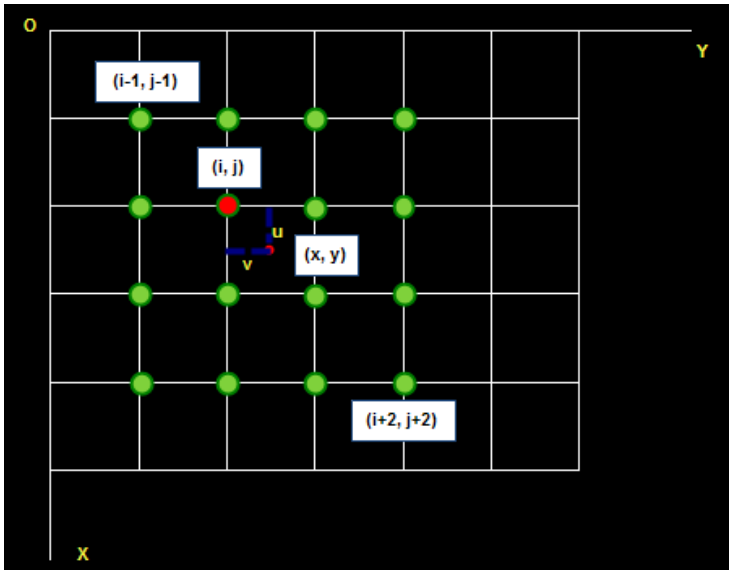
f(i+u, j+v) = (1-u) * (1-v) * f(i, j) + (1-u) * v * f(i, j+1) + u * (1-v) * f(i+1, j) + u * v * f(i+1, j+1)

双线性内插法的计算比最邻近点法复杂，计算量较大，但没有灰度不连续的缺点，结果基本令人满意。它具有低通滤波性质，使高频分量受损，图像轮廓可能会有一点模糊。

双三次内插法利用三次多项式 S(x)求逼近理论上最佳插值函数 sin(x)/x，其数学表达式为：

$$S(x)=\begin{cases}1-2|x|^2+|x|^3 & 0\le|x|<1\\ 4-8|x|+5|x|^2-|x|^3 & 1\le|x|<2\\ 0 & |x|\ge 2\end{cases}$$

待求像素(x, y)的灰度值由其周围 16 个灰度值加权内插得到，如下图：



待求像素的灰度计算式如下：

f(x, y) = f(i+u, j+v) = ABC，其中：

$$A=\begin{pmatrix}S(1+v)\\ S(v)\\ S(1-v)\\ S(2-v)\end{pmatrix}^T$$

$$B=\begin{pmatrix}f(i-1,j-1) & f(i-1,j) & f(i-1,j+1) & f(i-1,j+2)\\ f(i,j-1) & f(i,j) & f(i,j+1) & f(i,j+2)\\ f(i+1,j-1) & f(i+1,j) & f(i+1,j+1) & f(i+1,j+2)\\ f(i+2,j-1) & f(i+2,j) & f(i+2,j+1) & f(i+2,j+2)\end{pmatrix}$$

$$C=\begin{pmatrix}S(1+u)\\ S(u)\\ S(1-u)\\ S(2-u)\end{pmatrix}$$

OpenCV 提供了 resize()函数用于实现图像的插值缩放，它的声明如下：dst=cv.resize(src[, dsize[, dst[, fx[, fy, interpolation]]]])，函数参数：src：输入的源图像或矩阵；dst：输出图像；dsize：输出图像的大小；fx：水平方向的尺度因子；fy：垂直方向的尺度因子；interpolation：插值方法，常用的有 INTER_NEAREST
(最近邻插值)，INTER_LINEAR(双线性插值)和 INTER_CUBIC(双三次插值)。

实验结果如下：

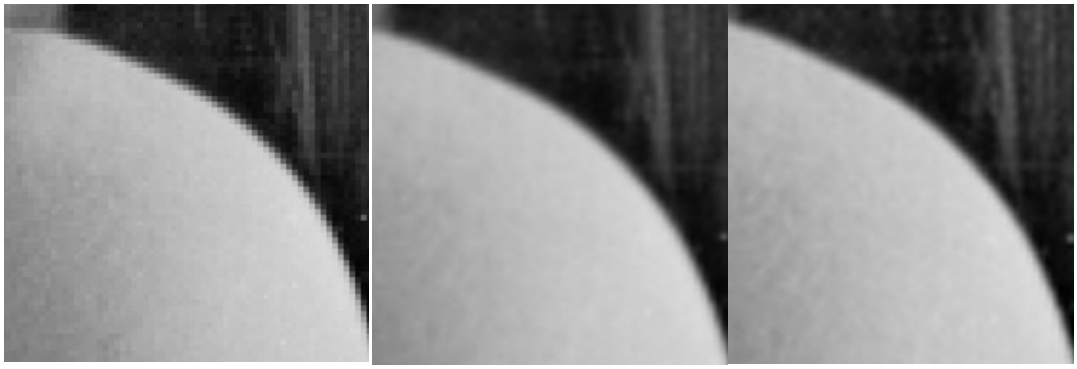
原始图像：

最近邻插值法：

双线性插值法：

双三次插值法：

结果分析：最近邻插值、双线性插值与双三次插值这三种方法之间的区别主要在于点周围像素序列的取法不同。对于最近邻插值，输出像素的值指定为点所属像素的值，不考虑其他像素；对于双线性插值，输出图像的值是最近的 2*2 邻域内像素值得加权平均值；对于双三次差值，输出图像的值是最近的 4*4 邻域内像素值得加权平均值。所以，双线性插值法花费的时间比最近邻法的要长一些，而双三次法花费的时间比双线性法的又要长一些。但　是，参与计算的时间越多，计算结果越精确，通常双线性插值和双三次差值这两种方法比最近邻法得到的效果更好。从得到的图像我们可以看到，最近邻内插产生了最大的锯齿边缘，尤其是在对应图像中的肩膀的边缘部位，产生了很明显的锯齿，而双线性内插得到了明显改进的结果使用双三次内插产生了稍微清晰一些的结果。下图分别是三种插值方法得到的图肩膀部分像局部放大后的结果：



最近邻插值　　双线性插值　　双三次插值

五、把 lena 和 elain 图像分别进行水平 shear(参数可设置为 1.5，或者自行选择)和旋转 30 度，并采用用近邻、双线性和双三次插值法 zoom 到 2048*2048

仿射变换（Affine Transformation 或 Affine Map）是一种二维坐标（x, y）到二维坐标（u, v）的线性变换，其数学表达式形式如下：

$$\begin{cases} u = a_1 x + b_1 y + c_1 \\ v = a_2 x + b_2 y + c_2 \end{cases}$$

对应的齐次坐标矩阵表示形式为：

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

仿射变换保持了二维图形的“平直性”（直线经仿射变换后依然为直线）和“平行性”（直线之间的相对位置关系保持不变，平行线经仿射变换后依然为平行线，且直线上点的位置顺序不会发生变化）。非共线的三对对应点确定一个唯一的仿射变换。

图像处理中，可应用仿射变换对二维图像进行平移、缩放、旋转等操作。实例如下：

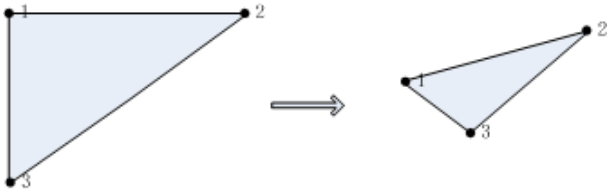


image1　　image2

经仿射变换后，图像关键点依然构成三角形，但三角形形状已经发生变化。

仿射变换通过一系列原子变换复合实现，具体包括：平移(Translation)、缩放(Scale)、旋转(Rotation)、翻转(Flip)和错切(Shear)。

a. 平移

$$R \longrightarrow R$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

b. 缩放

$$R \longrightarrow R$$

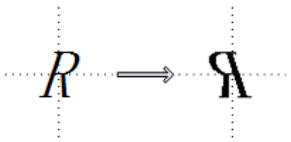
$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

c. 旋转

$$R \longrightarrow R$$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

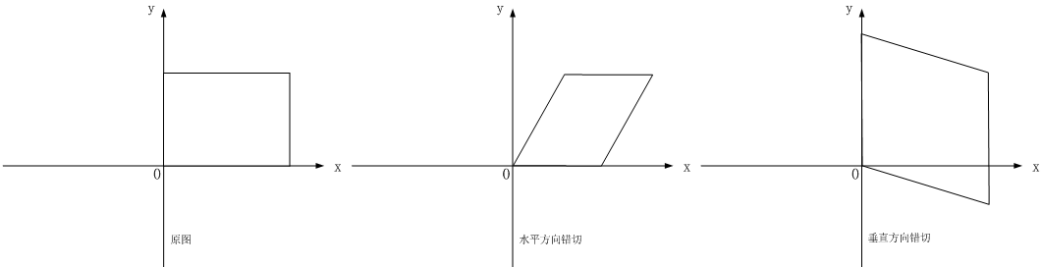
d. 翻转

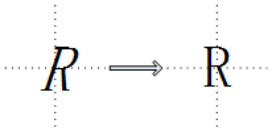


$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & N+1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

e. 错切

错切亦称为剪切或错位变换，包含水平错切和垂直错切，常用于产生弹性物体的变形处理。



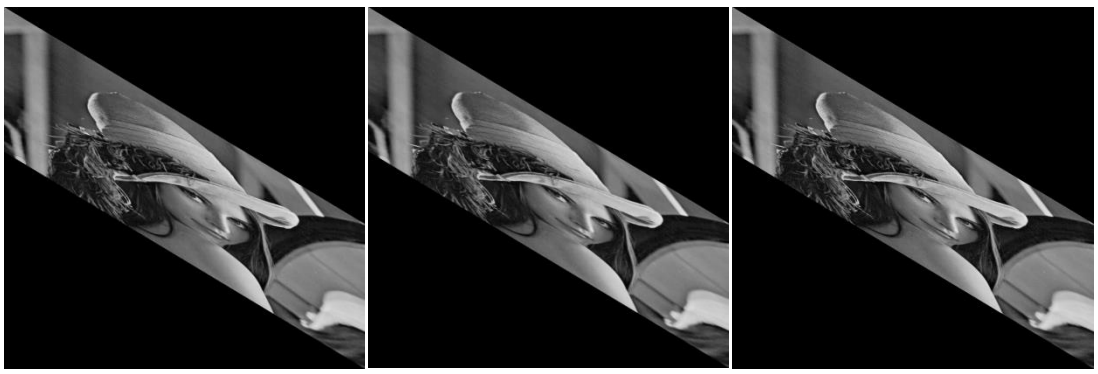


$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & d_x & 0 \\ d_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

OpenCV 提供了 warpAffine()函数用于实现图像的仿射变换，它的声明如下：dst=cv.warpAffine(src[, M, dsize[, dst[, flags[, borderMode[, borderValue]]]]])，函数参数：src：输入的源图像或矩阵；dst：输出图像；M：2×3 变换矩阵；dsize：输出图像的大小；flags：设置插值方法；borderMode：设置外插方法；borderValue：默认为 0。

实验结果如下：

Lena 水平偏移：

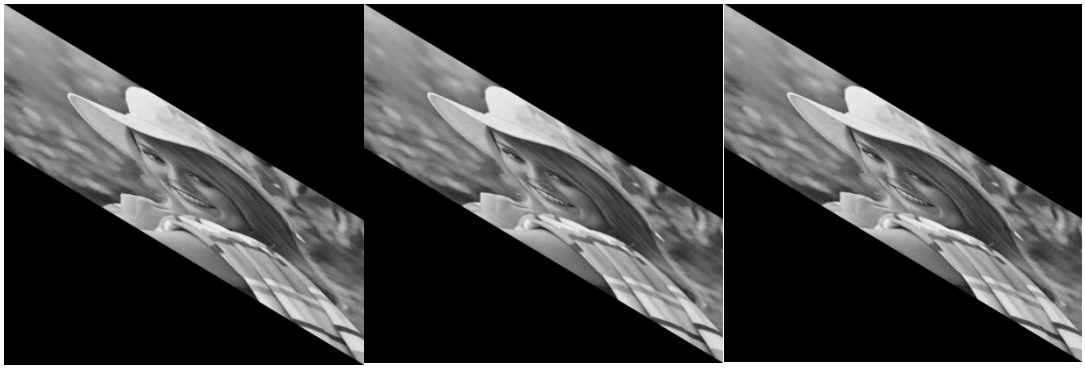


最近邻插值　　双线性插值　　双三次插值

Lena 旋转 30 度

最近邻插值　　双线性插值　　双三次插值

Elain 水平偏移：



最近邻插值　　双线性插值　　双三次插值

Elain 旋转 30 度

最近邻插值　　双线性插值　　双三次插值

结果分析：

在数字图像处理中几何变换由两个基本操作组成：(1)坐标的空间变化；(2)灰度内插，即对空间变换后的像素赋灰度值。最常用的空间坐标变换之一是仿射变换。最近邻内插产生了最大的锯齿边缘，双线性内插得到了明显的改进结果，使用双三次内插产生了稍微清晰一些的结果。

旋转是保持直线特性方面最苛求的几何变换之一，与上一实验结果相同，最近邻内插产生了最大的锯齿边缘，双线性内插得到了明显的改进结果，使用双三次内插产生了稍微清晰一些的结果。

附录 1：参考文献

[1] 冈萨雷斯.数字图像处理(第三版)北京:电子工业出版社,2011

附录 2：源代码

lena1.py

```
# 把 lena 512*512 图像灰度级逐级递减 8-1 显示
import cv2

lena = cv2.imread("./lena.bmp")
img2 = lena.copy()
img4 = lena.copy()
img8 = lena.copy()
img16 = lena.copy()
img32 = lena.copy()
img64 = lena.copy()
img128 = lena.copy()

def getvalue(level, pixel):
    num = level - 1
    size = 256 // num
    for i in range(1, level):
        if pixel > size * i:
            continue
        mid_value = size * (2*i-1) // 2
        left = size * (i-1)
        right = size * i - 1
        if pixel < mid_value:
            return left
        else:
            return right
    return pixel

def quantification(input_img, level):
    for i in range(512):
        for j in range(512):
            for k in range(3):
                input_img[i, j, k] = getvalue(level, input_img[i, j, k])
    return input_img

img2 = quantification(img2, 2)
img4 = quantification(img4, 4)
img8 = quantification(img8, 8)
img16 = quantification(img16, 16)
img32 = quantification(img32, 32)
```

```
img64 = quantification(img64, 64)
img128 = quantification(img128, 128)

cv2.imshow("img2", img2)
cv2.imshow("img4", img4)
cv2.imshow("img8", img8)
cv2.imshow("img16", img16)
cv2.imshow("img32", img32)
cv2.imshow("img64", img64)
cv2.imshow("img128", img128)

cv2.imwrite('./lena_2.bmp', img2)
cv2.imwrite('./lena_4.bmp', img4)
cv2.imwrite('./lena_8.bmp', img8)
cv2.imwrite('./lena_16.bmp', img16)
cv2.imwrite('./lena_32.bmp', img32)
cv2.imwrite('./lena_64.bmp', img64)
cv2.imwrite('./lena_128.bmp', img128)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

lena2.py

```
# 计算 lena 图像的均值方差
# 调用库函数版本
import cv2

lena = cv2.imread("./lena.bmp")
mean, stddv = cv2.meanStdDev(lena)#均值, 标准差
var = stddv * stddv

print("均值: ", mean[0])#均值
print("标准差: ", stddv[0])#标准差
print("方差: ", var[0])#方差
```

lena2_f.py

```
# 计算 lena 图像的均值方差
# 自己编写函数版本
import cv2

lena = cv2.imread("./lena.bmp")
sum = 0
for i in range(512):
    for j in range(512):
```

```
        sum = sum + lena[i, j, 0]
average = sum / 512 / 512
var = 0
for i in range(512):
    for j in range(512):
        var = var + (lena[i, j, 0] - average)*(lena[i, j, 0] - average)
var = var / 512 / 512

print("均值：", average)#均值
print("方差：", var)#方差
```

lena3.py

```
#把 lena 图像用近邻、双线性和双三次插值法 zoom 到 2048*2048
# 调用库函数版本
import cv2

lena = cv2.imread("./lena.bmp")
nearest = cv2.resize(lena, (2048, 2048), interpolation=cv2.INTER_NEAREST)
linear = cv2.resize(lena, (2048, 2048), interpolation=cv2.INTER_LINEAR)
cubic = cv2.resize(lena, (2048, 2048), interpolation=cv2.INTER_CUBIC)

cv2.imshow('nearest', nearest)
cv2.imshow('linear', linear)
cv2.imshow('cubic', cubic)

cv2.imwrite('./lena_nearest.bmp', nearest)
cv2.imwrite('./lena_linear.bmp', linear)
cv2.imwrite('./lena_cubic.bmp', cubic)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

lena3_f.py

```
# 把 lena 图像用近邻、双线性和双三次插值法 zoom 到 2048*2048
# 自己编写函数版本
import cv2
import numpy as np

lena = cv2.imread("./lena.bmp")
height, width, channels = lena.shape
h_scale = 2048 / height
w_scale = 2048 / width

def nearest():
```

```
    emptyImage1 = np.zeros((2048, 2048, 3), np.uint8)
    for i in range(2048):
        for j in range(2048):
            x = int(i/h_scale)
            y = int(j/w_scale)
            emptyImage1[i, j] = lena[x, y]
    return emptyImage1

def linear():
    emptyImage2 = np.zeros((2048, 2048, 3), np.uint8)
    value = [0, 0, 0]
    for i in range(2048):
        for j in range(2048):
            x = int(i/h_scale)
            y = int(j/w_scale)
            u = i/h_scale - x
            v = j/w_scale - y
            x = x - 1
            y = y - 1
            for k in range(3):
                if x >= 0 and y >= 0 and x + 1 <= 511 and y + 1 <= 511:
                    value[k] = int(lena[x, y][k]*(1-u)*(1-v)+lena[x,
y+1][k]*(1-u)*v+lena[x+1, y][k]*u*(1-v)+lena[x+1, y+1][k]*u*v)
            emptyImage2[i, j] = (value[0], value[1], value[2])
    return emptyImage2

def S(x):
    x = np.abs(x)
    if 0 <= x < 1:
        return 1 - 2 * x * x + x * x * x
    if 1 <= x < 2:
        return 4 - 8 * x + 5 * x * x - x * x * x
    else:
        return 0

def cubic():
    emptyImage3 = np.zeros((2048, 2048, 3), np.uint8)
    for i in range(2048):
        for j in range(2048):
            x = int(i / h_scale)
            y = int(j / w_scale)
            p = i / h_scale - x
            q = j / w_scale - y
            x = x - 1
```

```
            y = y - 1
            A = np.array([
                [S(1 + p), S(p), S(1 - p), S(2 - p)]
            ])
            if x-1 >= 0 and y-1 >= 0 and x+2 <= 511 and y+2 <= 511:
                B = np.array([
                    [lena[x - 1, y - 1], lena[x - 1, y],
                     lena[x - 1, y + 1],lena[x - 1, y + 1]],
                    [lena[x, y - 1], lena[x, y],
                     lena[x, y + 1], lena[x, y + 2]],
                    [lena[x + 1, y - 1], lena[x + 1, y],
                     lena[x + 1, y + 1], lena[x + 1, y + 2]],
                    [lena[x + 2, y - 1], lena[x + 2, y],
                     lena[x + 2, y + 1], lena[x + 2, y + 1]]
                ])
                C = np.array([
                        [S(1 + q)],
                        [S(q)],
                        [S(1 - q)],
                        [S(2 - q)]
                    ])
                blue = np.dot(np.dot(A, B[:, :, 0]), C)[0, 0]
                green = np.dot(np.dot(A, B[:, :, 1]), C)[0, 0]
                red = np.dot(np.dot(A, B[:, :, 2]), C)[0, 0]

                def adjust(value):
                    if value > 255:
                        value = 255
                    elif value < 0:
                        value = 0
                    return value

                blue = adjust(blue)
                green = adjust(green)
                red = adjust(red)
                emptyImage3[i, j] = np.array([blue, green, red], dtype=np.uint8)

    return emptyImage3

nearest_result = nearest()
linear_result = linear()
cubic_result = cubic()
cv2.imshow('nearest', nearest_result)
cv2.imwrite('./lena_nearest_f_new.bmp', nearest_result)
```

```
cv2.imshow('linear', linear_result)
cv2.imwrite('./lena_linear_f_new.bmp', linear_result)
cv2.imshow('cubic', cubic_result)
cv2.imwrite('./lena_cubic_f_new.bmp', cubic_result)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

lena_elain.py

```
'''把 lena 和 elain 图像分别进行水平 shear(参数可设置为 1.5，或者自行选择)
和旋转 30 度，并采用用近邻、双线性和双三次插值法 zoom 到 2048*2048'''

import cv2
import numpy as np

lena = cv2.imread("./lena.bmp")
elain = cv2.imread("./elain1.bmp")

M_r = cv2.getRotationMatrix2D((256, 256), 30, 1)
M_s = np.float32([[1, 0, 0], [1.5, 1, 0]])

lena_shear = cv2.warpAffine(lena, M_s, (512, 1280))
lena_r = cv2.warpAffine(lena, M_r, (512, 512))
elain_shear = cv2.warpAffine(elain, M_s, (512, 1280))
elain_r = cv2.warpAffine(elain, M_r, (512, 512))

lena_shear_nearest = cv2.resize(lena_shear, (2048, 2048),
interpolation=cv2.INTER_NEAREST)
lena_shear_linear = cv2.resize(lena_shear, (2048, 2048),
interpolation=cv2.INTER_LINEAR)
lena_shear_cubic = cv2.resize(lena_shear, (2048, 2048),
interpolation=cv2.INTER_CUBIC)
elain_shear_nearest = cv2.resize(elain_shear, (2048, 2048),
interpolation=cv2.INTER_NEAREST)
elain_shear_linear = cv2.resize(elain_shear, (2048, 2048),
interpolation=cv2.INTER_LINEAR)
elain_shear_cubic = cv2.resize(elain_shear, (2048, 2048),
interpolation=cv2.INTER_CUBIC)

lena_r_nearest = cv2.resize(lena_r, (2048, 2048), interpolation=cv2.INTER_NEAREST)
lena_r_linear = cv2.resize(lena_r, (2048, 2048), interpolation=cv2.INTER_LINEAR)
lena_r_cubic = cv2.resize(lena_r, (2048, 2048), interpolation=cv2.INTER_CUBIC)
elain_r_nearest = cv2.resize(elain_r, (2048, 2048),
interpolation=cv2.INTER_NEAREST)
```

```
elain_r_linear = cv2.resize(elain_r, (2048, 2048), interpolation=cv2.INTER_LINEAR)
elain_r_cubic = cv2.resize(elain_r, (2048, 2048), interpolation=cv2.INTER_CUBIC)

cv2.imwrite('./lena_shear_nearest.bmp', lena_shear_nearest)
cv2.imwrite('./lena_shear_linear.bmp', lena_shear_linear)
cv2.imwrite('./lena_shear_cubic.bmp', lena_shear_cubic)
cv2.imwrite('./elain_shear_nearest.bmp', elain_shear_nearest)
cv2.imwrite('./elain_shear_linear.bmp', elain_shear_linear)
cv2.imwrite('./elain_shear_cubic.bmp', elain_shear_cubic)
cv2.imwrite('./lena_rotation_nearest.bmp', lena_r_nearest)
cv2.imwrite('./lena_rotation_linear.bmp', lena_r_linear)
cv2.imwrite('./lena_rotation_cubic.bmp', lena_r_cubic)
cv2.imwrite('./elain_rotation_nearest.bmp', elain_r_nearest)
cv2.imwrite('./elain_rotation_linear.bmp', elain_r_linear)
cv2.imwrite('./elain_rotation_cubic.bmp', elain_r_cubic)
```