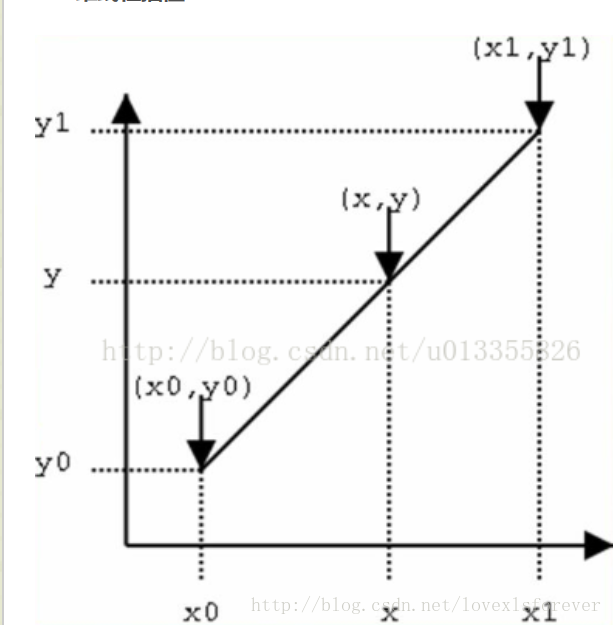
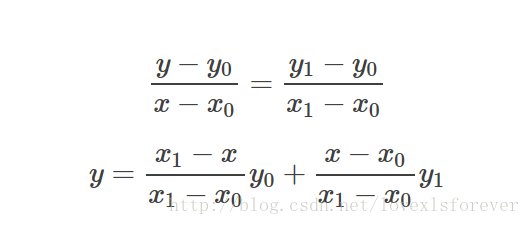
* + [1、线性插值的解释](https://blog.csdn.net/lovexlsforever/article/details/79508602#1线性插值的解释)
    - [单线性插值法](https://blog.csdn.net/lovexlsforever/article/details/79508602#单线性插值法)
    - [双线性插值法](https://blog.csdn.net/lovexlsforever/article/details/79508602#双线性插值法)
  + [2、另一位牛人讲的比较易懂](https://blog.csdn.net/lovexlsforever/article/details/79508602#2另一位牛人讲的比较易懂)
    - [1.双线性插值](https://blog.csdn.net/lovexlsforever/article/details/79508602#1双线性插值)
    - [2.存在的问题](https://blog.csdn.net/lovexlsforever/article/details/79508602#2存在的问题)
  + [3、又是另一位讲的通俗易懂](https://blog.csdn.net/lovexlsforever/article/details/79508602#3又是另一位讲的通俗易懂)
    - [1，原理](https://blog.csdn.net/lovexlsforever/article/details/79508602#1原理)
    - [2，计算方法](https://blog.csdn.net/lovexlsforever/article/details/79508602#2计算方法)
    - [3，加速以及优化策略](https://blog.csdn.net/lovexlsforever/article/details/79508602#3加速以及优化策略)
      * [3.1 源图像和目标图像几何中心的对齐](https://blog.csdn.net/lovexlsforever/article/details/79508602#31-源图像和目标图像几何中心的对齐)
      * [3.2 将浮点运算转换成整数运算](https://blog.csdn.net/lovexlsforever/article/details/79508602#32-将浮点运算转换成整数运算)
    - [4，代码](https://blog.csdn.net/lovexlsforever/article/details/79508602#4代码)

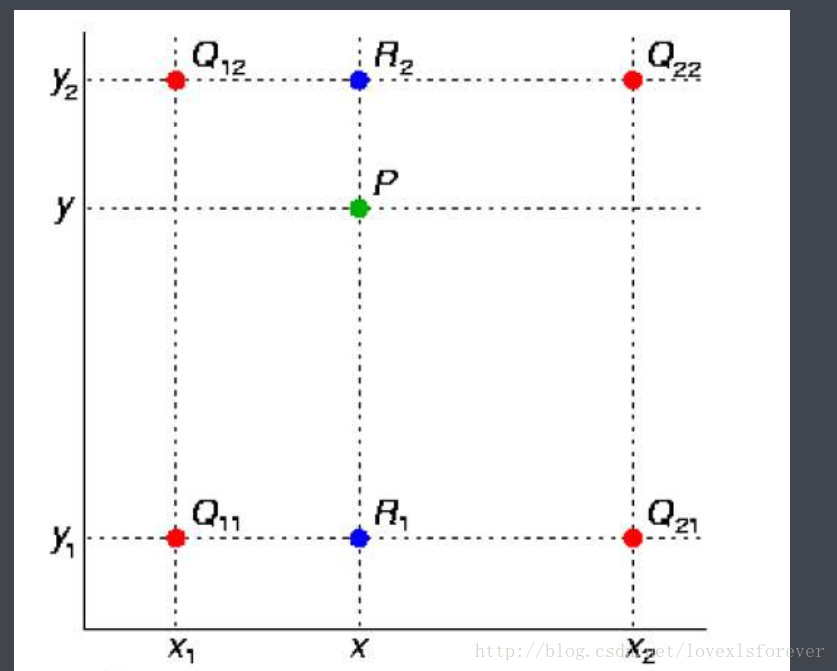
**1、线性插值的解释**

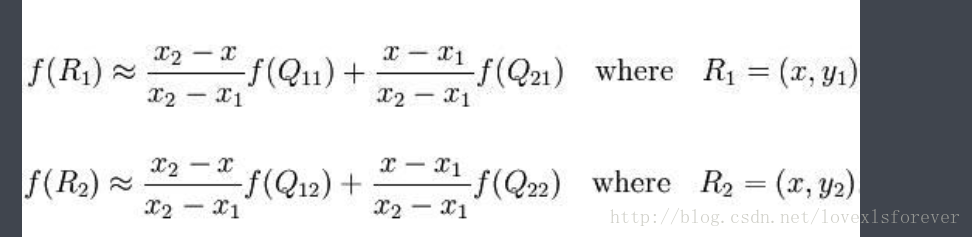
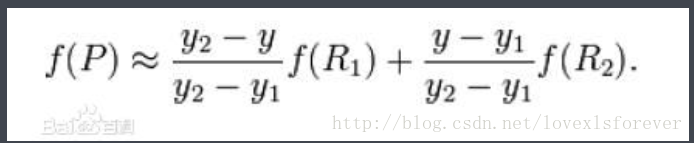
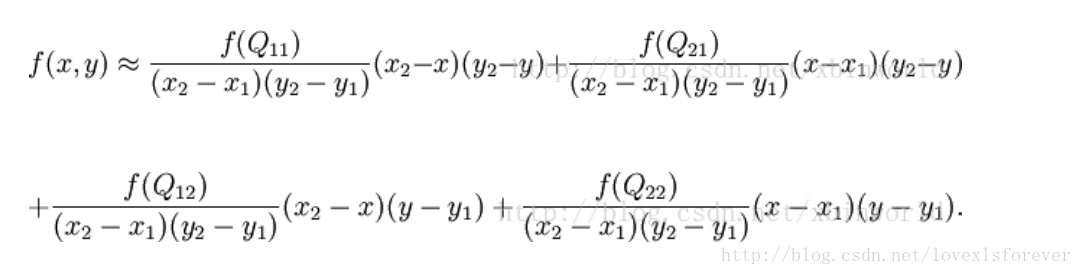
双线性插值，又称为双线性内插。在数学上，双线性插值是有两个变量的插值函数的线性插值扩展，其核心思想是在两个方向分别进行一次线性插值。一下来自于[1]

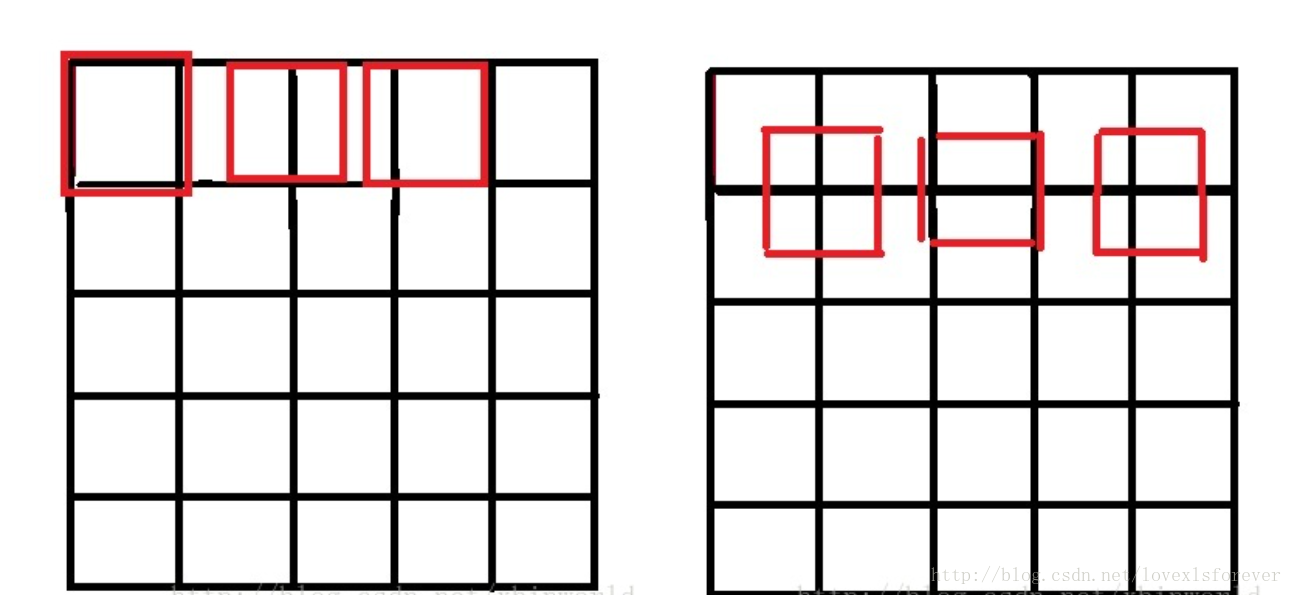
**单线性插值法**

已知数据 (x0, y0) 与 (x1, y1)，要计算 [x0, x1] 区间内某一位置 x 在直线上的y值。   
  
  
上面比较好理解吧，仔细看就是用x和x0，x1的距离作为一个权重，用于y0和y1的加权。双线性插值本质上就是在两个方向上做线性插值。

**双线性插值法**

在数学上，双线性插值是有两个变量的插值函数的线性插值扩展，其核心思想是在两个方向分别进行一次线性插值[1]。见下图：   


假如我们想得到未知函数 f 在点 P = (x, y) 的值，假设我们已知函数 f 在 Q11 = (x1, y1)、Q12 = (x1, y2), Q21 = (x2, y1) 以及 Q22 = (x2, y2) 四个点的值。最常见的情况，f就是一个像素点的像素值。首先在 x 方向进行线性插值，得到   
  
然后在 y 方向进行线性插值，得到   
  
综合起来就是双线性插值最后的结果：   


由于图像双线性插值只会用相邻的4个点，因此上述公式的分母都是1。opencv中的源码如下，用了一些优化手段，比如用整数计算代替float（下面代码中的\*2048就是变11位小数为整数，最后有两个连乘，因此>>22位），以及源图像和目标图像几何中心的对齐   
- SrcX=(dstX+0.5)\* (srcWidth/dstWidth) -0.5   
- SrcY=(dstY+0.5) \* (srcHeight/dstHeight)-0.5，   
这个要重点说一下，源图像和目标图像的原点（0，0）均选择左上角，然后根据插值公式计算目标图像每点像素，假设你需要将一幅5x5的图像缩小成3x3，那么源图像和目标图像各个像素之间的对应关系如下。如果没有这个中心对齐，根据基本公式去算，就会得到左边这样的结果；而用了对齐，就会得到右边的结果：   


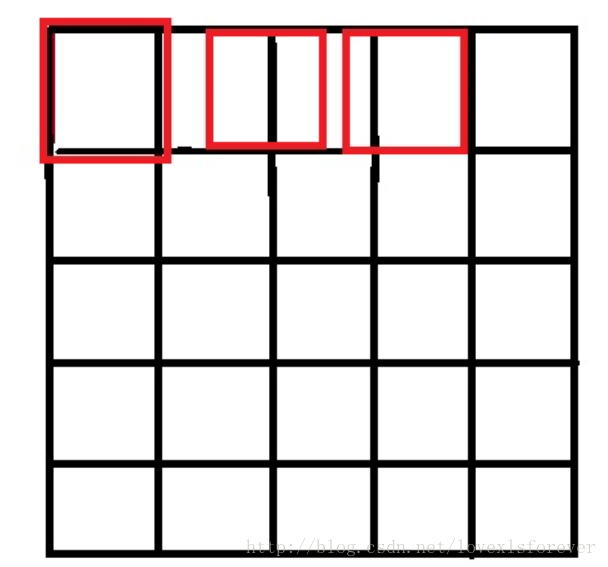
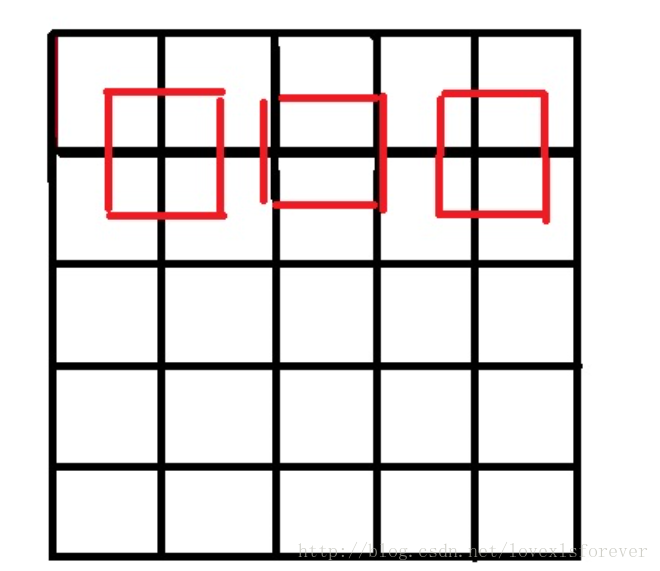
**2、另一位牛人讲的比较易懂**

这博文来自文件[2]

**1.双线性插值**

假设源图像大小为mxn，目标图像为axb。那么两幅图像的边长比分别为：m/a和n/b。注意，通常这个比例不是整数，编程存储的时候要用浮点型。目标图像的第（i,j）个像素点（i行j列）可以通过边长比对应回源图像。其对应坐标为（i\*m/a,j\*n/b）。   
显然，这个对应坐标一般来说不是整数，而非整数的坐标是无法在图像这种离散数据上使用的。双线性插值通过寻找距离这个对应坐标最近的四个像素点，来计算该点的值（灰度值或者RGB值）。如果你的对应坐标是（2.5,4.5），那么最近的四个像素是（2，4）、（2，5）、（3，4），（3，5）。   
若图像为灰度图像，那么（i，j）点的灰度值可以通过一下公式计算：   
f(i,j)=w1\*p1+w2\*p2+w3\*p3+w4\*p4;   
其中，pi(i=1,2,3,4)为最近的四个像素点，wi(i=1,2,3,4)为各点相应权值。关于权值的计算，在维基百科和百度百科上写的很明白。

**2.存在的问题**

这部分的前提是，你已经明白什么是双线性插值并且在给定源图像和目标图像尺寸的情况下，可以用笔计算出目标图像某个像素点的值。当然，最好的情况是你已经用某种语言实现了网上一大堆博客上原创或转载的双线性插值算法，然后发现计算出来的结果和matlab、openCV对应的resize（）函数得到的结果完全不一样。   
那这个究竟是怎么回事呢？   
其实答案很简单，就是坐标系的选择问题，或者说源图像和目标图像之间的对应问题。   
按照网上一些博客上写的，源图像和目标图像的原点（0，0）均选择左上角，然后根据插值公式计算目标图像每点像素，假设你需要将一幅5x5的图像缩小成3x3，那么源图像和目标图像各个像素之间的对应关系如下：   
  
只画了一行，用做示意，从图中可以很明显的看到，如果选择右上角为原点（0，0），那么最右边和最下边的像素实际上并没有参与计算，而且目标图像的每个像素点计算出的灰度值也相对于源图像偏左偏上。   
那么，让坐标加1或者选择右下角为原点怎么样呢？很不幸，还是一样的效果，不过这次得到的图像将偏右偏下。   
最好的方法就是，两个图像的几何中心重合，并且目标图像的每个像素之间都是等间隔的，并且都和两边有一定的边距，这也是matlab和openCV的做法。如下图：   
  
如果你不懂我上面说的什么，没关系，只要在计算对应坐标的时候改为以下公式即可，

int x=(i+0.5)\*m/a-0.5   
int y=(j+0.5)\*n/b-0.5

instead of

int x=i\*m/a   
int y=j\*n/b

利用上述公式，将得到正确的双线性插值结果

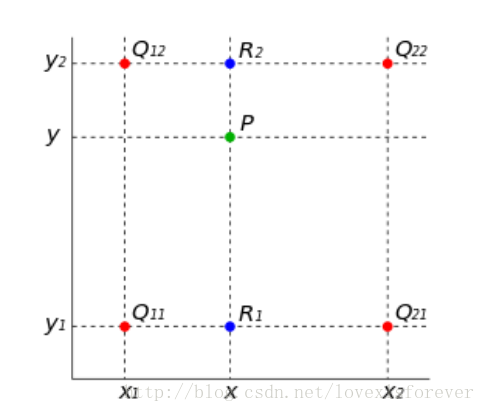
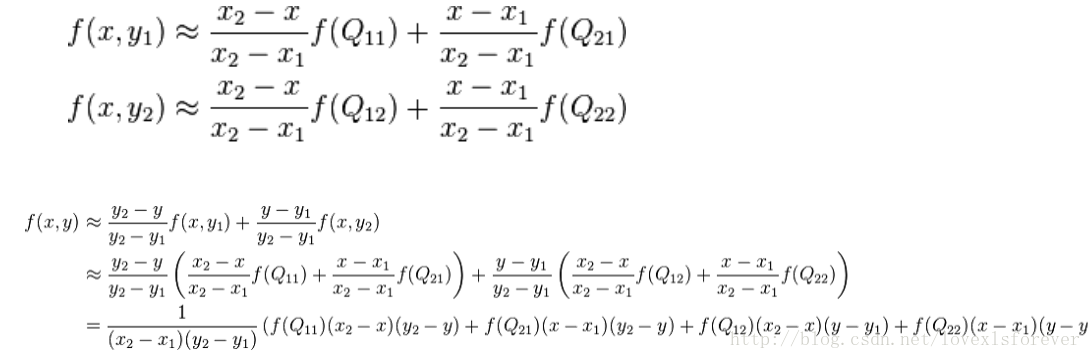
**3、又是另一位讲的通俗易懂**

这位大神讲openCV ——双线性插值（Bilinear interpolation）[3]

**1，原理**

　　在图像的仿射变换中，很多地方需要用到插值运算，常见的插值运算包括最邻近插值，双线性插值，双三次插值，兰索思插值等方法，OpenCV提供了很多方法，其中，双线性插值由于折中的插值效果和运算速度，运用比较广泛。   
　　越是简单的模型越适合用来举例子，我们就举个简单的图像：3\*3 的256级灰度图。假如图像的象素矩阵如下图所示（这个原始图把它叫做源图，Source）：   
234 38 22   
67 44 12   
89 65 63   
　　这 个矩阵中，元素坐标(x,y)是这样确定的，x从左到右，从0开始，y从上到下，也是从零开始，这是图象处理中最常用的坐标系。   
　　如果想把这副图放大为 4\*4大小的图像，那么该怎么做呢？那么第一步肯定想到的是先把4\*4的矩阵先画出来再说，好了矩阵画出来了，如下所示，当然，矩阵的每个像素都是未知数，等待着我们去填充（这个将要被填充的图的叫做目标图,Destination）：   
　　? ? ? ?   
　　? ? ? ?   
　　? ? ? ?   
　　? ? ? ?   
　　然后要往这个空的矩阵里面填值了，要填的值从哪里来来呢？是从源图中来，好，先填写目标图最左上角的象素，坐标为（0，0），那么该坐标对应源图中的坐标可以由如下公式得出srcX=dstX\* (srcWidth/dstWidth) , srcY = dstY \* (srcHeight/dstHeight)   
　　好了，套用公式，就可以找到对应的原图的坐标了(0\*(3/4),0\*(3/4))=>(0\*0.75,0\*0.75)=>(0,0)，找到了源图的对应坐标,就可以把源图中坐标为(0,0)处的234象素值填进去目标图的(0,0)这个位置了。   
　　接下来,如法炮制,寻找目标图中坐标为(1,0)的象素对应源图中的坐标,套用公式:   
(1\*0.75,0\*0.75)=>(0.75,0) 结果发现,得到的坐标里面竟然有小数,这可怎么办?计算机里的图像可是数字图像,象素就是最小单位了,象素的坐标都是整数,从来没有小数坐标。这时候采用的一种策略就是采用四舍五入的方法（也可以采用直接舍掉小数位的方法），把非整数坐标转换成整数，好，那么按照四舍五入的方法就得到坐标（1，0），完整的运算过程就是这样的：(1\*0.75,0\*0.75)=>(0.75,0)=>(1,0) 那么就可以再填一个象素到目标矩阵中了，同样是把源图中坐标为(1,0)处的像素值38填入目标图中的坐标。   
　　依次填完每个象素，一幅放大后的图像就诞生了，像素矩阵如下所示：   
　　234 38 22 22   
　　67 44 12 12   
　　89 65 63 63   
　　89 65 63 63   
　　这种放大图像的方法叫做最临近插值算法，这是一种最基本、最简单的图像缩放算法，效果也是最不好的，放大后的图像有很严重的马赛克，缩小后的图像有很严重的失真；效果不好的根源就是其简单的最临近插值方法引入了严重的图像失真，比如，当由目标图的坐标反推得到的源图的的坐标是一个浮点数的时候，采用了四舍五入的方法，直接采用了和这个浮点数最接近的象素的值，这种方法是很不科学的，当推得坐标值为 0.75的时候，不应该就简单的取为1，既然是0.75，比1要小0.25 ，比0要大0.75 ,那么目标象素值其实应该根据这个源图中虚拟的点四周的四个真实的点来按照一定的规律计算出来的，这样才能达到更好的缩放效果。   
　　双线型内插值算法就是一种比较好的图像缩放算法，它充分的利用了源图中虚拟点四周的四个真实存在的像素值来共同决定目标图中的一个像素值，因此缩放效果比简单的最邻近插值要好很多。   
双线性内插值算法描述如下:   
　　对于一个目的像素，设置坐标通过反向变换得到的浮点坐标为(i+u,j+v) (其中i、j均为浮点坐标的整数部分，u、v为浮点坐标的小数部分，是取值[0,1)区间的浮点数)，则这个像素得值 f(i+u,j+v) 可由原图像中坐标为 (i,j)、(i+1,j)、(i,j+1)、(i+1,j+1)所对应的周围四个像素的值决定，即：f(i+u,j+v) = (1-u)(1-v)f(i,j) + (1-u)vf(i,j+1) + u(1-v)f(i+1,j) + uvf(i+1,j+1)   
其中f(i,j)表示源图像(i,j)处的的像素值，以此类推。   
　　比如，象刚才的例子，现在假如目标图的象素坐标为（1，1），那么反推得到的对应于源图的坐标是（0.75 , 0.75）, 这其实只是一个概念上的虚拟象素,实际在源图中并不存在这样一个象素,那么目标图的象素（1，1）的取值不能够由这个虚拟象素来决定，而只能由源图的这四个象素共同决定：（0，0）（0，1）（1，0）（1，1），而由于（0.75,0.75）离（1，1）要更近一些，那么（1,1）所起的决定作用更大一些，这从公式1中的系数uv=0.75×0.75就可以体现出来，而（0.75,0.75）离（0，0）最远，所以（0，0）所起的决定作用就要小一些，公式中系数为(1-u)(1-v)=0.25×0.25也体现出了这一特点。

**2，计算方法**

  
　首先，在X方向上进行两次线性插值计算，然后在Y方向上进行一次插值计算。   
　   
在图像处理的时候，我们先根据   
　　srcX=dstX\* (srcWidth/dstWidth),   
　　srcY = dstY \* (srcHeight/dstHeight)   
来计算目标像素在源图像中的位置，这里计算的srcX和srcY一般都是浮点数，比如f（1.2, 3.4）这个像素点是虚拟存在的，先找到与它临近的四个实际存在的像素点   
　　（1，3） （2，3）   
　　（1，4） （2，4）   
写成f(i+u,j+v)的形式，则u=0.2,v=0.4, i=1, j=3   
在沿着X方向差插值时，f(R1)=u(f(Q21)-f(Q11))+f(Q11)   
沿着Y方向同理计算。   
或者，直接整理一步计算，   
f(i+u,j+v) = (1-u)(1-v)f(i,j) + (1-u)vf(i,j+1) + u(1-v)f(i+1,j) + uvf(i+1,j+1)

**3，加速以及优化策略**

　单纯按照上文实现的插值算法只能勉强完成插值的功能，速度和效果都不会理想，在具体代码实现的时候有些小技巧。参考OpenCV源码以及网上博客整理如下两点：   
源图像和目标图像几何中心的对齐。   
将浮点运算转换成整数运算

**3.1 源图像和目标图像几何中心的对齐**

　方法：在计算源图像的虚拟浮点坐标的时候，一般情况：   
　　srcX=dstX\* (srcWidth/dstWidth) ,   
　　srcY = dstY \* (srcHeight/dstHeight)   
　　中心对齐(OpenCV也是如此)：   
　　SrcX=(dstX+0.5)\* (srcWidth/dstWidth) -0.5   
　　SrcY=(dstY+0.5) \* (srcHeight/dstHeight)-0.5   
　　**原理：**   
双线性插值算法及需要注意事项这篇博客解释说“如果选择右上角为原点（0，0），那么最右边和最下边的像素实际上并没有参与计算，而且目标图像的每个像素点计算出的灰度值也相对于源图像偏左偏上。”我有点保持疑问。   
　　将公式变形:   
　　srcX=dstX\* (srcWidth/dstWidth)+0.5\*(srcWidth/dstWidth-1)   
　　相当于我们在原始的浮点坐标上加上了0.5\*(srcWidth/dstWidth-1)这样一个控制因子，这项的符号可正可负，与srcWidth/dstWidth的比值也就是当前插值是扩大还是缩小图像有关，有什么作用呢？看一个例子：假设源图像是3\*3，中心点坐标（1，1）目标图像是9\*9，中心点坐标（4，4），我们在进行插值映射的时候，尽可能希望均匀的用到源图像的像素信息，最直观的就是（4,4）映射到（1,1）现在直接计算srcX=4\*3/9=1.3333！=1，也就是我们在插值的时候所利用的像素集中在图像的右下方，而不是均匀分布整个图像。现在考虑中心点对齐，srcX=(4+0.5)\*3/9-0.5=1，刚好满足我们的要求。

**3.2 将浮点运算转换成整数运算**

　　参考图像处理界双线性插值算法的优化   
　　直接进行计算的话，由于计算的srcX和srcY 都是浮点数，后续会进行大量的乘法，而图像数据量又大，速度不会理想，解决思路是：   
　　浮点运算→→整数运算→→”<<左右移按位运算”。   
　　放大的主要对象是u，v这些浮点数，OpenCV选择的放大倍数是2048“如何取这个合适的放大倍数呢，要从三个方面考虑，   
　　第一：精度问题，如果这个数取得过小，那么经过计算后可能会导致结果出现较大的误差。   
　　第二，这个数不能太大，太大会导致计算过程超过长整形所能表达的范围。   
　　第三：速度考虑。假如放大倍数取为12，那么算式在最后的结果中应该需要除以12\*12=144，但是如果取为16，则最后的除数为16\*16=256，这个数字好，我们可以用右移来实现，而右移要比普通的整除快多了。”我们利用左移11位操作就可以达到放大目的。

**4，代码**

uchar\* dataDst = matDst1.data;

int stepDst = matDst1.step;

uchar\* dataSrc = matSrc.data;

int stepSrc = matSrc.step;

int iWidthSrc = matSrc.cols;

int iHiehgtSrc = matSrc.rows;

for (int j = 0; j < matDst1.rows; ++j)

{

float fy = (float)((j + 0.5) \* scale\_y - 0.5);

int sy = cvFloor(fy);

fy -= sy;

sy = std::min(sy, iHiehgtSrc - 2);

sy = std::max(0, sy);

short cbufy[2];

cbufy[0] = cv::saturate\_cast<short>((1.f - fy) \* 2048);

cbufy[1] = 2048 - cbufy[0];

for (int i = 0; i < matDst1.cols; ++i)

{

float fx = (float)((i + 0.5) \* scale\_x - 0.5);

int sx = cvFloor(fx);

fx -= sx;

if (sx < 0) {

fx = 0, sx = 0;

}

if (sx >= iWidthSrc - 1) {

fx = 0, sx = iWidthSrc - 2;

}

short cbufx[2];

cbufx[0] = cv::saturate\_cast<short>((1.f - fx) \* 2048);

cbufx[1] = 2048 - cbufx[0];

for (int k = 0; k < matSrc.channels(); ++k)

{

\*(dataDst+ j\*stepDst + 3\*i + k) = (\*(dataSrc + sy\*stepSrc + 3\*sx + k) \* cbufx[0] \* cbufy[0] +

\*(dataSrc + (sy+1)\*stepSrc + 3\*sx + k) \* cbufx[0] \* cbufy[1] +

\*(dataSrc + sy\*stepSrc + 3\*(sx+1) + k) \* cbufx[1] \* cbufy[0] +

\*(dataSrc + (sy+1)\*stepSrc + 3\*(sx+1) + k) \* cbufx[1] \* cbufy[1]) >> 22;

}

}

}

cv::imwrite("linear\_1.jpg", matDst1);

cv::resize(matSrc, matDst2, matDst1.size(), 0, 0, 1);

cv::imwrite("linear\_2.jpg", matDst2);

参考文件：   
[[1]三十分钟理解：线性插值，双线性插值Bilinear Interpolation算法](http://blog.csdn.net/xbinworld/article/details/65660665)   
[[2] 双线性插值算法及需要注意事项](http://handspeaker.iteye.com/blog/1545126)   
[[3]OpenCV ——双线性插值（Bilinear interpolation）](http://www.cnblogs.com/yssongest/p/5303151.html)   
[[4]FPGA双线性插值算法（图像的缩放](http://blog.csdn.net/angelbosj/article/details/50389423)