第6章
图像几何变换

目录

背景知识

- 图像灰度变换和空间域滤波的共同点是，仅改变图像像素的灰度值，而不会改变像素的位置，因此，图像中的几何结构不会变形。
- 本章所讨论的图像几何变换，其目的是通过**改变像素的位置**来实现图像**几何结构的变化**。
- 图像几何变换广泛应用于图像配准（image registration）、计算机图形学（computer graphics）、电脑动漫（computer animation ）、视觉特效（visual effects）等领域。
- 图像几何变换：改变了像素的位置并重新赋值，这一过程通常包括**空间坐标变换**和**灰度插值**两个基本步骤。

图像几何变换 — 示例

(1)原始图像

(2)平移变换

(3)缩放变换

(4)旋转变换

(5))剪切变换

(6)旋转扭曲变换

(7)球面变换

(8)波浪扭曲变换

6.1 图像几何变换原理

空间坐标变换

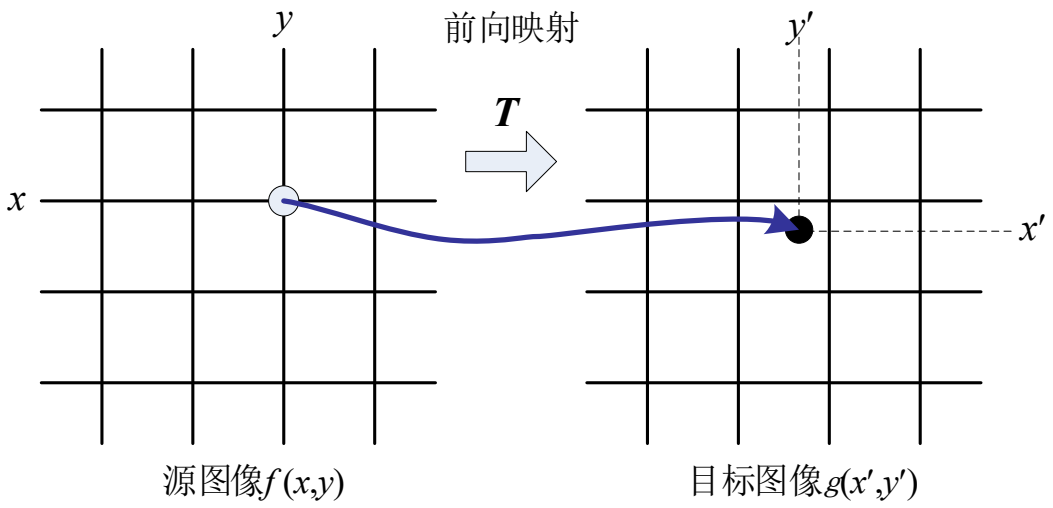
- **空间坐标变换：图像几何变换要先建立描述源图像（输入图像）与目标图像（输出图像）之间坐标对应关系的变换函数（又称坐标映射函数），利用该变换函数将源图像中每个像素坐标(x,y)映射到目标图像中的一个新位置(x',y')。坐标变换函数可表示为：**

$$\begin{cases} x'=T_x(x,y) \\ y'=T_y(x,y) \end{cases}$$

  - 式中，源像素坐标(x,y)值为整数，计算结果坐标(x',y')的值不一定是整数。
  - 从源图像到目标图像的坐标变换，又称前向映射（forward mapping）。

前向映射（源图像→目标图像）

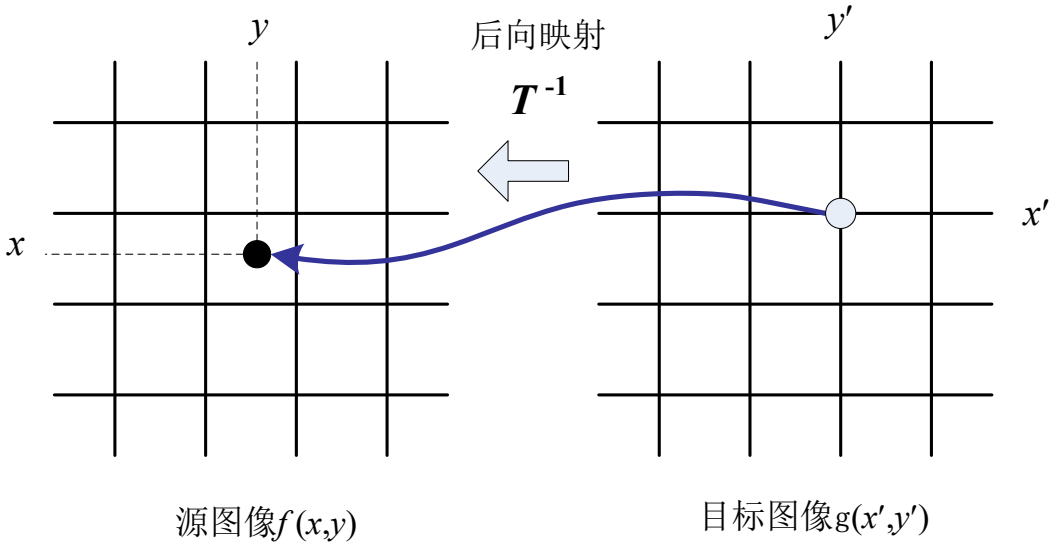
- **前向映射：对源图像的像素坐标(x,y)，通过变换函数T得到其在目标图像中的对应坐标(x',y')。**
  - 如果(x',y')的值是整数，那么，将源像素(x,y)的灰度值或颜色值直接赋给目标像素(x',y')即可。
  - 如果(x',y')为小数，则表明其位于几个像素中间，此时就不能简单地对目标像素直接赋值。



前向映射的一个问题是，目标图像中的一些像素坐标值可能根本没有被赋值，从而产生一些空洞。另外，还可能出现目标像素和多个源像素对应的情况。

后向映射（目标图像→源图像）

- **后向映射：将目标图像中的像素坐标(x',y')映射到源图像中对应位置(x,y)。这一坐标变换过程，称为后向映射（backward mapping），表示为：**



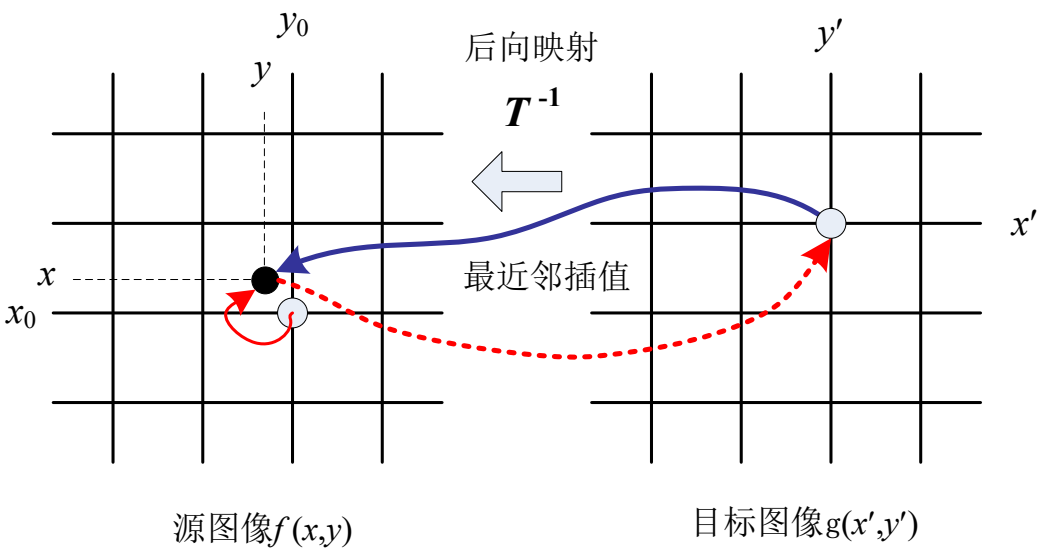
$$\begin{cases} x=T_x^{-1}(x',y') \\ y=T_y^{-1}(x',y') \end{cases}$$

后向映射主要优点是，对目标图像中每一像素都进行一次坐标变换计算和赋值，不会产生空洞和重叠。因此，后向映射被广泛应用于图像几何变换。

后向映射（目标图像→源图像）

- **后向映射目标像素坐标(x',y')值为整数，得到的源坐标(x,y)值不一定是整数。**

  - 若(x,y)是整数值坐标，那么只需简单地把源图像(x,y)处的像素灰度值或颜色值赋给目标像素(x',y')；
  - 若坐标(x,y)不是整数值，就意味着没有一个确定的源像素与之对应，而是位于几个像素之间。这样，就必须利用(x,y)周围像素信息，来估计(x,y)处的灰度值或颜色值，然后把该估计值赋给目标像素(x',y')，这个过程称为灰度插值。



灰度插值示意图

6.2 灰度插值

灰度插值

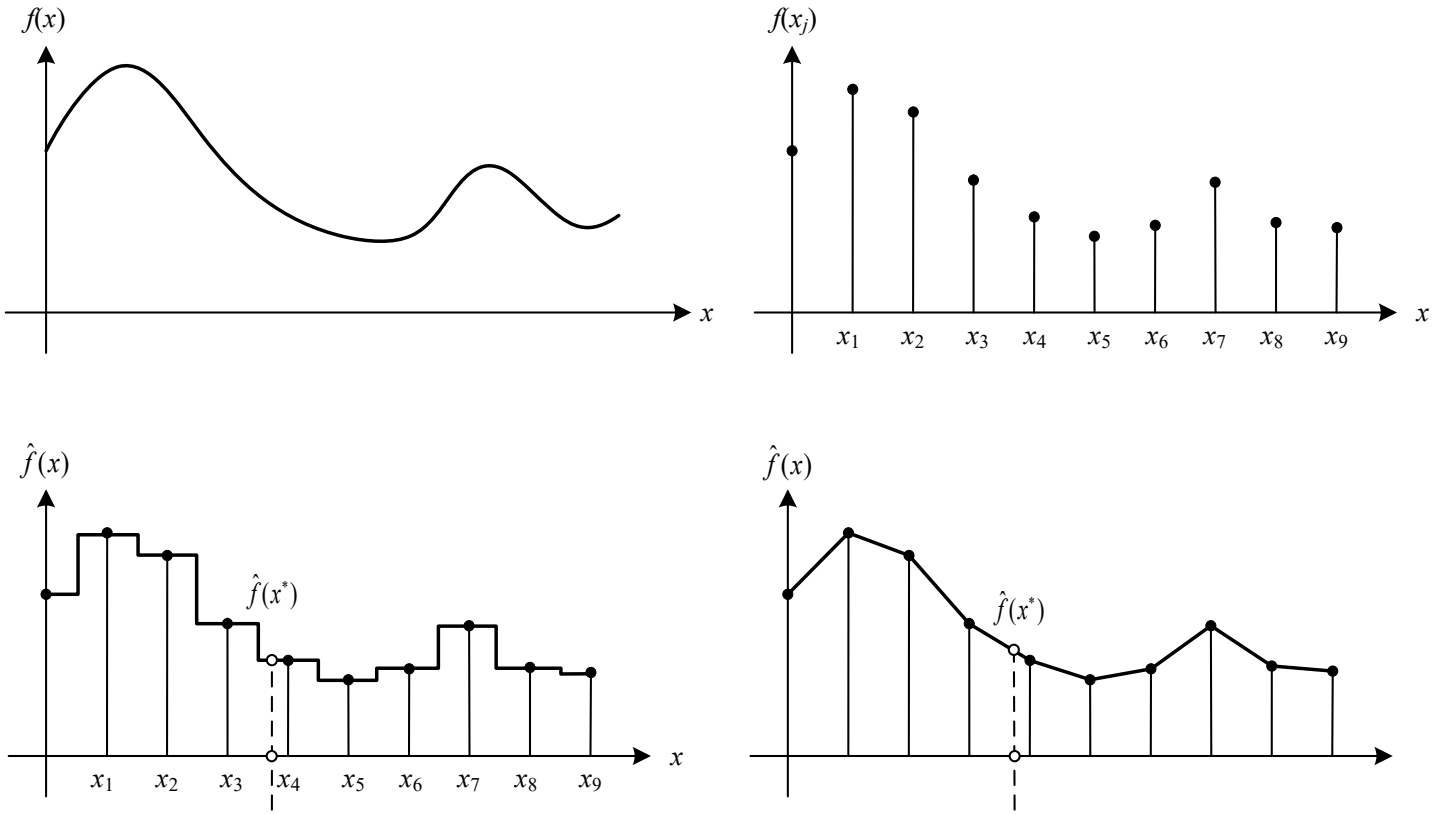
- 如前所述，后向映射坐标变换时，目标图像中像素坐标(x',y')经后向映射得到的坐标(x,y)值可能不是整数，不与源图像中任一像素对应，而是位于几个像素之间。
- 这样，就必须利用点(x,y)周围相邻像素数据，拟合出一个二维连续插值函数，利用该插值函数估计出点(x,y)的灰度值，然后将该估计值赋给目标像素(x',y')。
- 常用的灰度插值方法有：
  - 最近邻插值（nearest neighborhood interpolation）
  - 双线性插值（bilinear interpolation）
  - 双三次插值（bicubic interpolation）

灰度插值也是常用的图像重采样方法，用于增加或减少数字图像像素的数量。例如，数码相机常用灰度插值方法创造出比传感器实际像素更多的图像，或对图像局部放大实现数码变焦。

何谓插值?

- **一维插值**问题可以描述为：假定$f(x)$是定义在区间$[a,b]$上的未知实函数，已知其在区间$[a,b]$内n个不同点$x_0, x_1, x_2, \ldots, x_{n-1}$的函数值$f(x_0), f(x_1), f(x_2), \ldots, f(x_{n-1})$，要求估计区间$[a,b]$内任意一点$x^*$的函数值$f(x^*)$。
- **求解一维插值问题的基本思路：**构造一个连续实函数 $\hat{f}(x)$ 逼近 $f(x)$，满足插值条件 $\hat{f}(x_j)=f(x_j)$，$j=0, 1, 2, \ldots, n-1$。此处，x_j称为插值节点，即样本点；$\hat{f}(x)$称为插值函数。利用插值条件求出插值函数$\hat{f}(x)$的参数，然后用该插值函数估计插值点x^*的函数值 $\hat{f}(x^*)$。当插值点x^*位于包含$x_0, x_1, x_2, \ldots, x_{n-1}$的最小闭区间$[a,b]$内时，相应的插值称为内插，否则称为外插。

一维插值

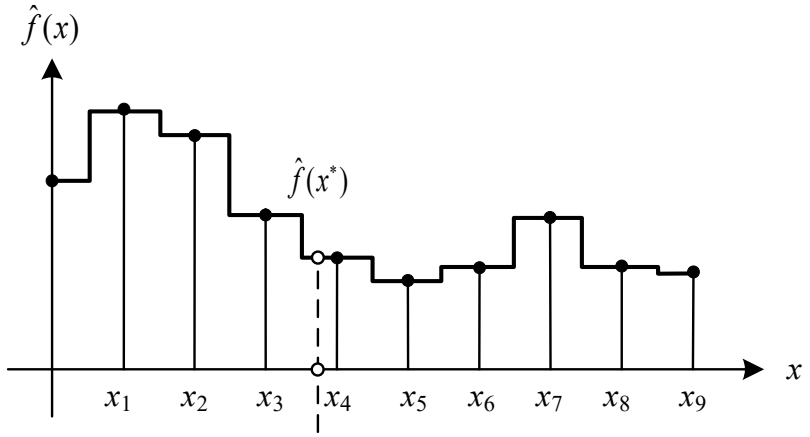


一维离散函数插值示意：目的是估计任意位置x^*的函数值$f(x^*)$。图(1)为一维连续函数$f(x)$；图(2)为对$f(x)$采样得到的离散序列；图(3)为对图(2)的最近邻插值结果，选择距插值点x^*最近的节点x_j的样本值$f(x_j)$作为插值 $\hat{f}\left(x^*\right)$。图(4)为对图(2)的线性插值结果，用一个连接相邻样本值$f(x_j)$和$f(x_{j+1})$的分段线性函数作为插值函数。

一维最近邻插值

- **最近邻插值（Nearest neighborhood interpolation）：最简单的插值方法，它把离插值点x^*最近的节点x_j的函数值$f(x_j)$作为插值函数的估计值，即：**

$$\hat{f}(x^*) = f(x_j)$$



- 其中，x_j是对插值点x^*四舍五入取整的结果，或写成：

$$x_j = \lfloor x^* + 0.5 \rfloor$$

- 符号$\lfloor x \rfloor$的功能是“向下取整”，即取不大于x的最大整数（向下取整是取数轴上最接近x的左边的整数值）。

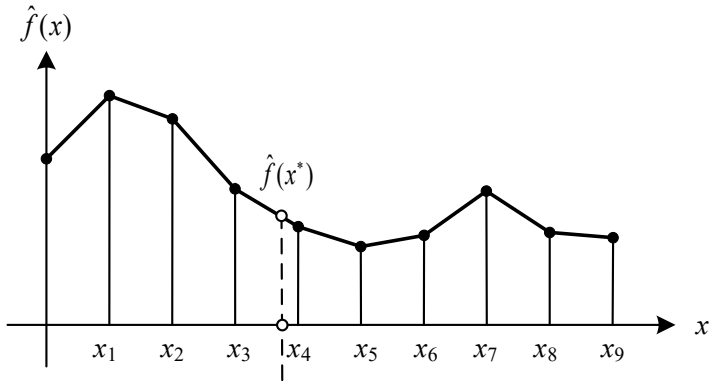
一维线性插值

- 线性插值（linear interpolation）的本质，是用连接两个节点$[x_j, f(x_j)]$和$[x_{j+1}, f(x_{j+1})]$的直线段来近似函数$f(x)$：

$$\hat{f}(x)=\frac{x_{j+1}-x}{x_{j+1}-x_j}\cdot f(x_j)+\frac{x-x_j}{x_{j+1}-x_j}\cdot f(x_{j+1})$$

- 如果采样间隔为1（节点间距为1），即$x_{j+1}=x_j+1$，上式可简化为：

$$\hat{f}(x)=(x_j+1-x)\cdot f(x_j)+(x-x_j)\cdot f(x_j+1)$$



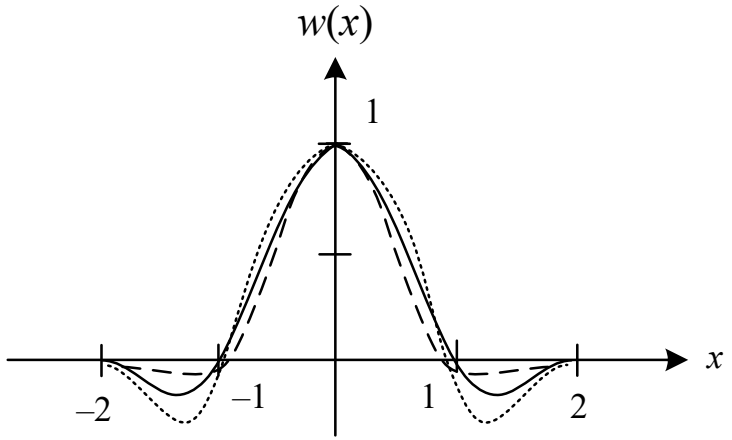
一维三次插值

- 三次插值因采用三次多项式作为插值核函数而得名，又称立方插值（cubic interpolation），离散序列$f(x)$的插值函数，可表示为离散序列$f(x)$与一个连续插值核函数$w(x)$的线性卷积，即：

$$\hat{f}(x)=w(x)*f(x)=\sum_{m=-\infty}^{\infty}w(x-m)f(m)$$

- 常用的三次插值核函数$w(x)$是对Sinc函数的截短近似，定义为一个分段三次多项式：

$$w(x)=\begin{cases}(a+2)|x|^3-(a+3)|x|^2+1, & \text{当}|x|<1\\ a|x|^3-5a|x|^2+8a|x|-4a, & \text{当}1\le|x|<2\\ 0, & \text{其他}\end{cases}$$



一维三次插值

- 三次插值核函数$w(x)$的延展范围较小，当$|x| \geq 2$时，$w(x)=0$，所以对任意位置x^*插值时，运算只需要四个离散值：

$$f(x_0-1),\ f(x_0),\ f(x_0+1),\ f(x_0+2);\quad \text{其中：} x_0=\lfloor x^* \rfloor$$

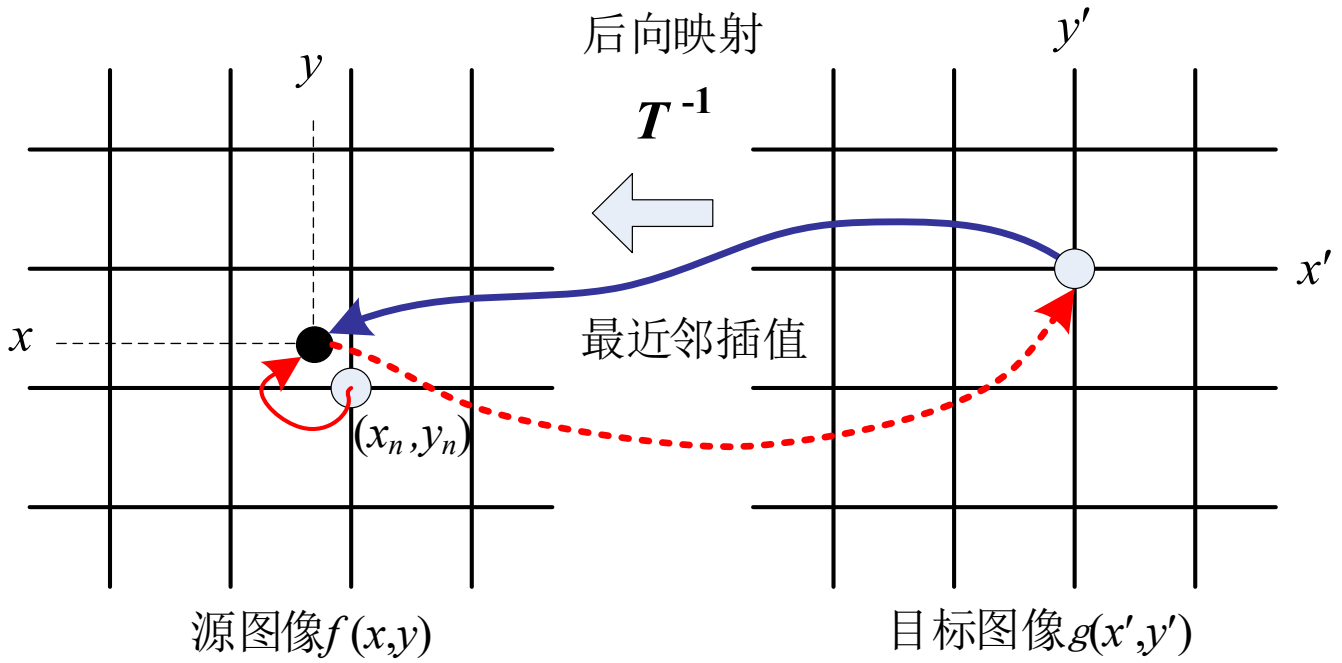
- 这样，就可以上述三次插值函数可简化为：

$$\hat{f}(x^*)=\sum_{u=x_0-1}^{x_0+2} w(x^*-u)\cdot f(u)$$

- 可见，插值核函数$w(x)$相当于一个权值函数，(x^*-u)为插值点x^*到样本节点u的距离。因此，三次插值本质上也是将节点样本值的加权和，作为插值点x^*函数值的估计。

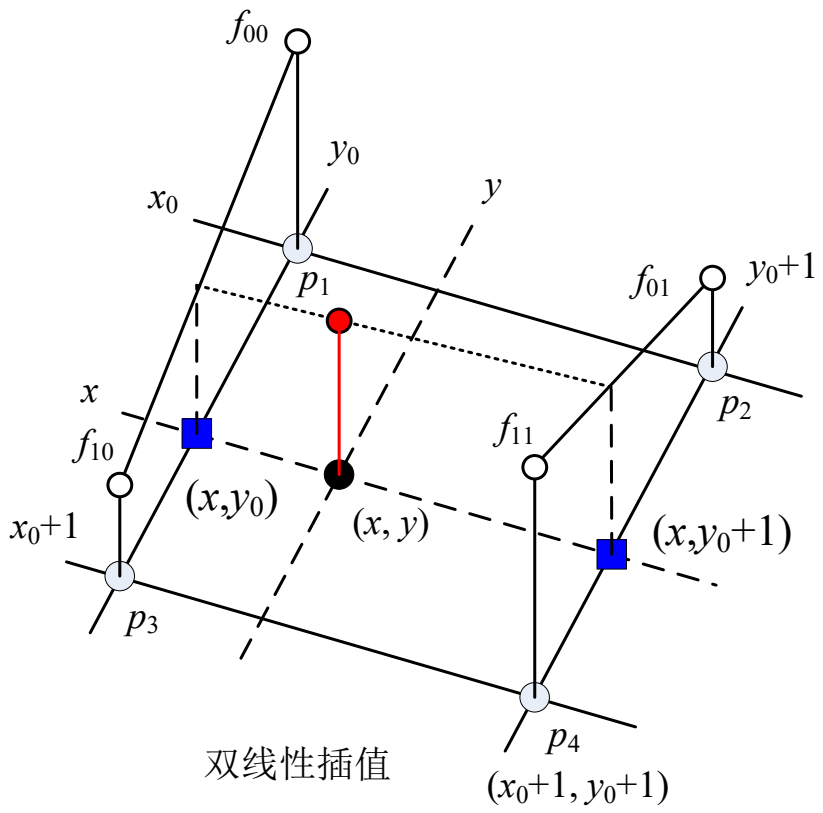
二维最近邻插值

- 把离插值点(x,y)最近的像素(x_0,y_0)的灰度值或颜色值$f(x_0,y_0)$，作为插值点(x,y)的估计值。
- 最近邻插值计算量小，但插值后的图像有很强的块状效应，锯齿波现象明显。



二维双线性插值

- 离插值点(x,y)最近的周围四个像素灰度值的加权和，节点像素离插值点(x,y)距离越近，其权值就越大。



双线性插值

$$\hat{f}(x,y)=(x_0+1-x)(y_0+1-y)\cdot f_{00}+(x-x_0)(y_0+1-y)\cdot f_{10}+(x_0+1-x)(y-y_0)\cdot f_{01}+(x-x_0)(y-y_0)\cdot f_{11}$$

双线性插值把二维插值过程，分解为分别沿x方向和y方向的一维线性插值。

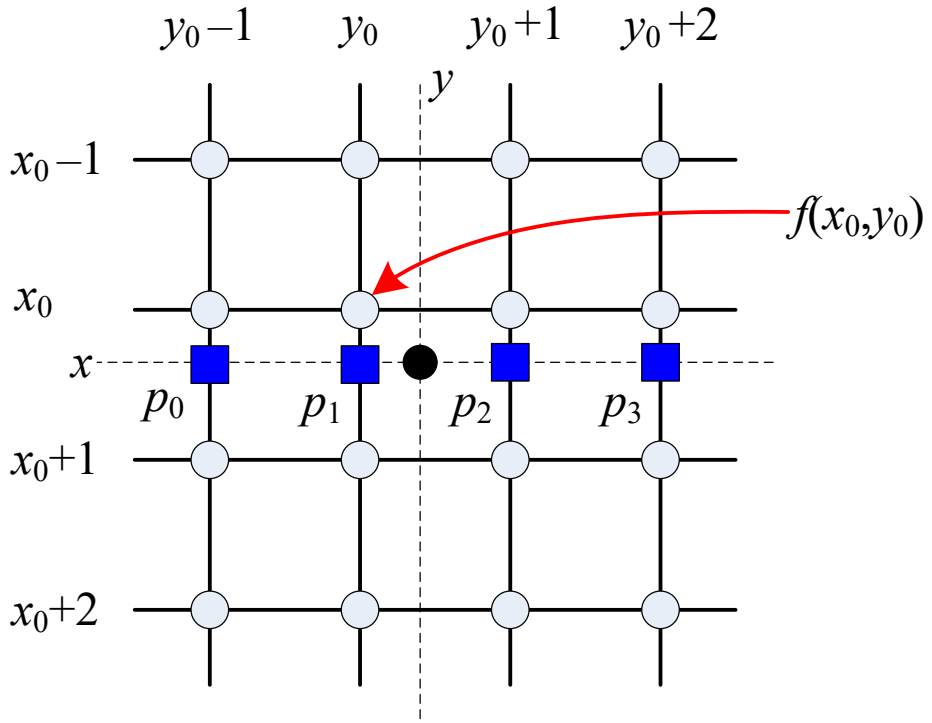
二维双三次插值

- 本质上是对插值点(x,y)周围最近的16个像素灰度值的加权平均。因此，双三次插值是一种更加复杂的插值方式，它能产生比最近邻插值和双线性插值更平滑的图像边缘，同时所需计算量也大。

$$\hat{f}(x,y)=\sum_{n=y_0-1}^{y_0+2}\left[\sum_{m=x_0-1}^{x_0+2}\left[w_{bic}(x-m,y-n)f(m,n)\right]\right]$$

$$=\sum_{n=y_0-1}^{y_0+2}\left[w(y-n)\sum_{m=x_0-1}^{x_0+2}\left[w(x-m)f(m,n)\right]\right]$$

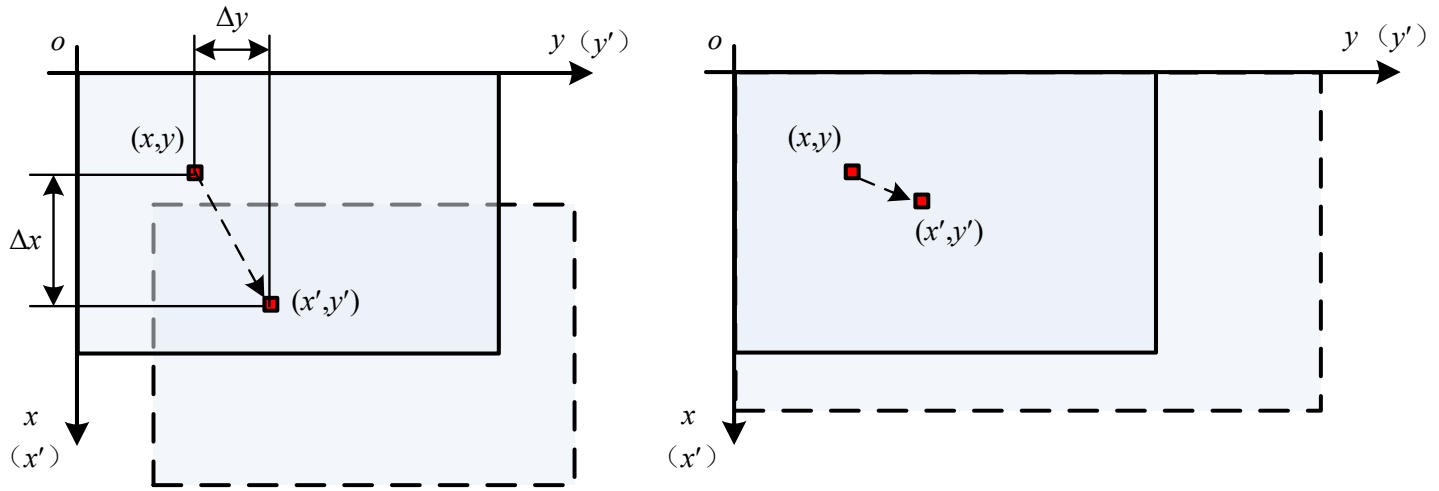
双三次插值把二维三次插值过程，分解为分别沿x方向和y方向的一维三次插值。



6.3 图像的基本几何变换

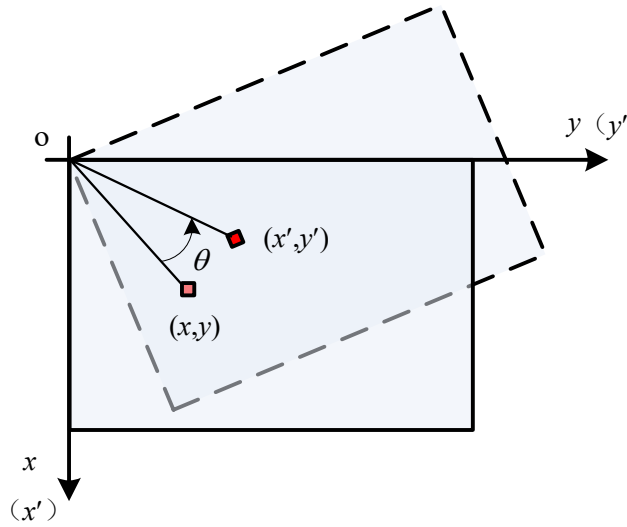
图像的基本几何变换

- 约定图像左上角为坐标系原点，x轴垂直向下、y轴水平向右，图像平面上空间点的坐标向量用列向量表示 。
- **分别给出前向变换函数。**
- **要注意的是：OpenCV、SciPy、Scikit-image等中的图像坐标系约定为：图像左上角为坐标系的原点，x轴水平向右、y轴垂直向下。**

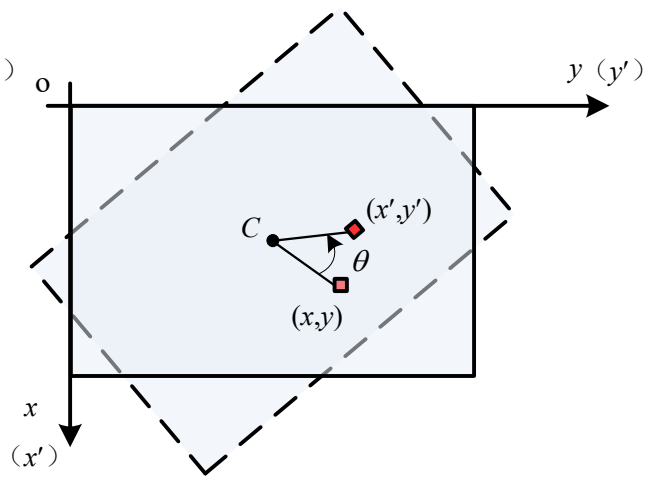


（a）平移变换　　（b）缩放变换

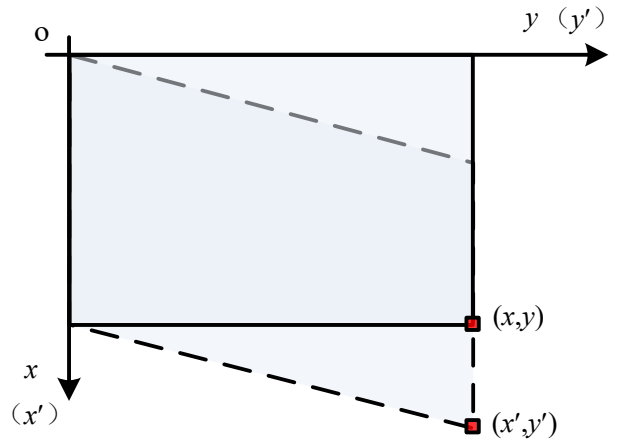
图像的基本几何变换



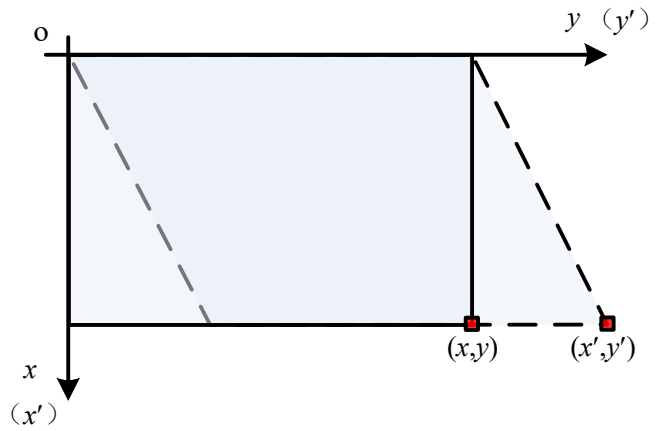
（c）旋转变换（绕图像左上角坐标系原点）



（d）旋转变换（绕图像中心*C*旋转）



（e）剪切变换（沿x方向，bx=0.25）



（f）剪切变换（沿y方向，by=0.25）

齐次坐标

- 齐次坐标（Homogeneous coordinates）提供了用矩阵运算，把二维、三维甚至高维空间中的点，从一个坐标系变换到另一个坐标系的有效运算方法。
- 齐次坐标在笛卡尔直角坐标向量的基础上额外增加一个分量h，用 $n+1$个分量来描述笛卡尔直角坐标系中的n 维坐标向量。例如，当$h\neq 0$时，二维平面上的点(x, y)、三维空间中的点(x, y, z)对应的齐次坐标为：

$$\begin{bmatrix} x \\ y \end{bmatrix} \underset{\text{笛卡尔坐标}}{\overset{\text{齐次坐标}}{\rightleftarrows}} \begin{bmatrix} h\cdot x \\ h\cdot y \\ h \end{bmatrix} \qquad \begin{bmatrix} x \\ y \\ z \end{bmatrix} \underset{\text{笛卡尔坐标}}{\overset{\text{齐次坐标}}{\rightleftarrows}} \begin{bmatrix} h\cdot x \\ h\cdot y \\ h\cdot z \\ h \end{bmatrix}$$

齐次坐标

- 一个笛卡尔坐标向量的齐次坐标不是唯一的，h取不同值时表示的都是笛卡尔坐标系中同一个点。例如，笛卡尔坐标(1,2)，对应的齐次坐标可以是(1,2,1)或(2,4,2)。
- 如果一个点的齐次坐标向量(x, y, h)，相应的笛卡尔坐标可以用齐次坐标前n 个分量除以最后一个分量来获得，即$(x/h, y/h)$。当h取1时，称规范化齐次坐标：

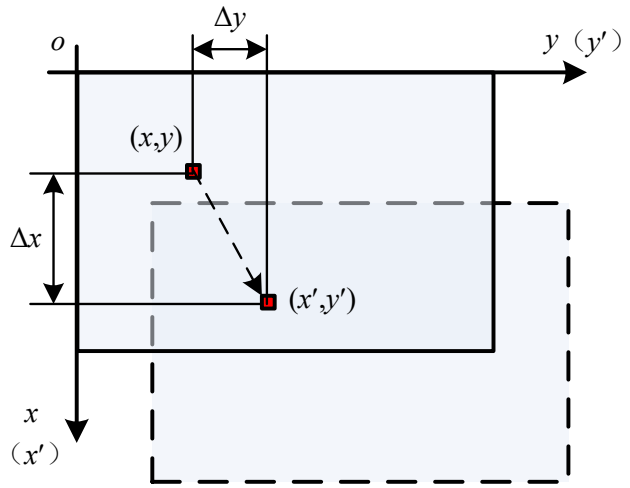
$$\begin{bmatrix} x \\ y \end{bmatrix} \underset{\text{笛卡尔坐标}}{\overset{\text{规范化齐次坐标}}{\rightleftarrows}} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \begin{bmatrix} x \\ y \\ z \end{bmatrix} \underset{\text{笛卡尔坐标}}{\overset{\text{规范化齐次坐标}}{\rightleftarrows}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

采用规范化齐次坐标，就可以将平移、旋转、缩放和剪切等坐标变换函数，统一为矩阵和向量相乘的简洁表达式

平移变换（translation）

- **图像平移把图像沿x方向、y方向移动给定的偏移量Δx、Δy，变换函数定义为：**

$$\begin{cases} x' = x + \Delta x \\ y' = y + \Delta y \end{cases}$$



向量形式：

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

齐次坐标形式：

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

采用规范化齐次坐标，就可以将平移、旋转、缩放和剪切等坐标变换函数，统一为矩阵和向量相乘的简洁表达式

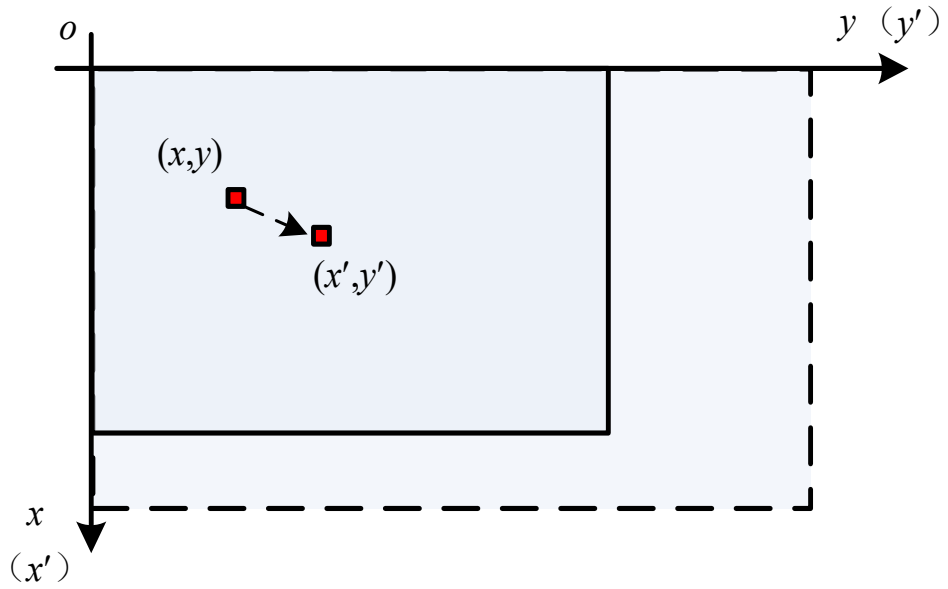
缩放变换（scale）

- 将图像沿x方向、y方向分别缩小或放大s_x和s_y倍，变换函数定义为：

$$\begin{cases} x' = s_x \cdot x \\ y' = s_y \cdot y \end{cases}$$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

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



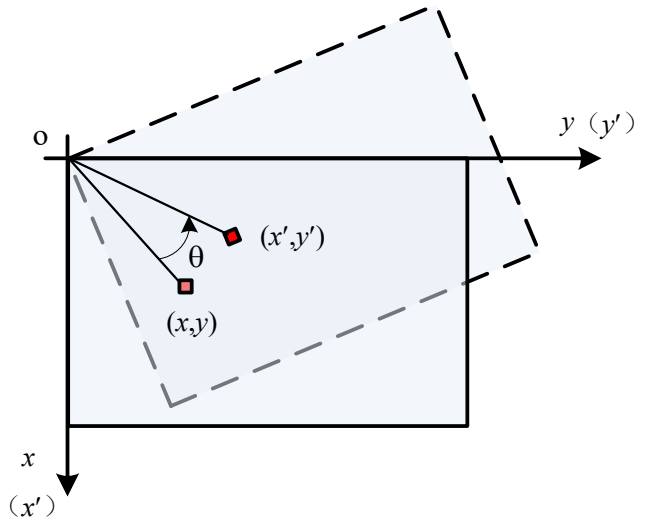
旋转变换（rotation）

- 将图像绕坐标原点旋转角度θ，逆时针方向旋转时θ值取正；顺时针方向旋转时θ值取负。变换函数定义为：

$$\begin{cases} x' = x \cdot \cos\theta - y \cdot \sin\theta \\ y' = x \cdot \sin\theta + y \cdot \cos\theta \end{cases}$$

$$或, \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$或, \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



（c）旋转变换（绕图像左上角坐标系原点）

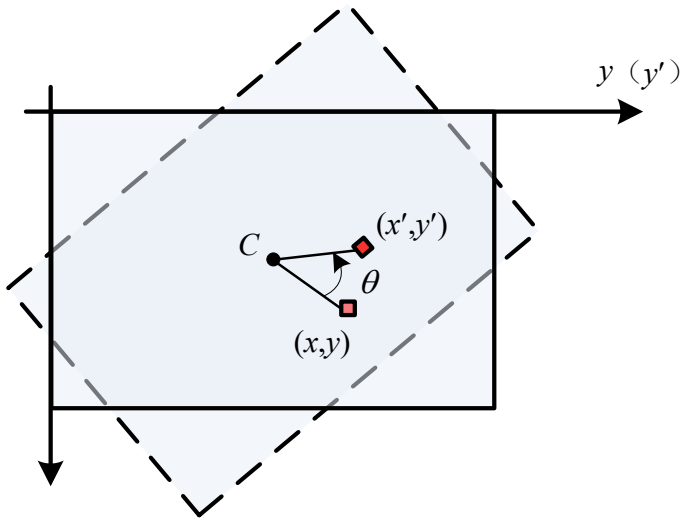
旋转变换（rotation）

- 如果将图像绕图像中心旋转角度θ，并假定原图像的高、宽分别为H、W，相应的变换函数为：

$$\begin{cases} x' = \left(x - \frac{H}{2}\right) \cdot \cos\theta - \left(y - \frac{W}{2}\right) \cdot \sin\theta + \frac{H}{2} \\ y' = \left(x - \frac{H}{2}\right) \cdot \sin\theta + \left(y - \frac{W}{2}\right) \cdot \cos\theta + \frac{W}{2} \end{cases}$$

或，

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \frac{H}{2} \\ 0 & 1 & \frac{W}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -\frac{H}{2} \\ 0 & 1 & -\frac{W}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



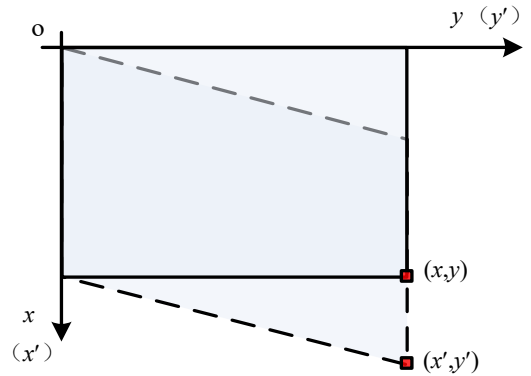
旋转变换（绕图像中心C旋转）

剪切变换（shear）

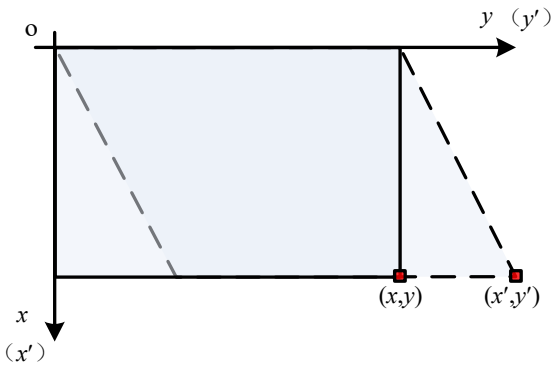
- 又称“**错切变换**”，将图像像素沿x方向或y方向产生不等量移动而引起的图像变形，位移量分别用因子b_x和b_y给出。剪切变换函数定义为：

$$\begin{cases} x' = x + b_x \cdot y \\ y' = y + b_y \cdot x \end{cases} \quad \text{或,} \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & b_x \\ b_y & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\text{或,} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & b_x & 0 \\ b_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



剪切变换（沿x方向，bx=0.25）



剪切变换（沿y方向，by=0.25）

组合变换

- 图像的组合变换，是对给定的图像连续施行若干次平移、比例、旋转或剪切等基本变换所形成的级联变换。假设对图像依次进行基本变换F_1，F_2，…，F_n，相应坐标变换矩阵分别为T_1，T_2，…，T_n，那么对图像实施的组合变换矩阵T，等于各基本变换矩阵按顺序依次左乘的结果，即：

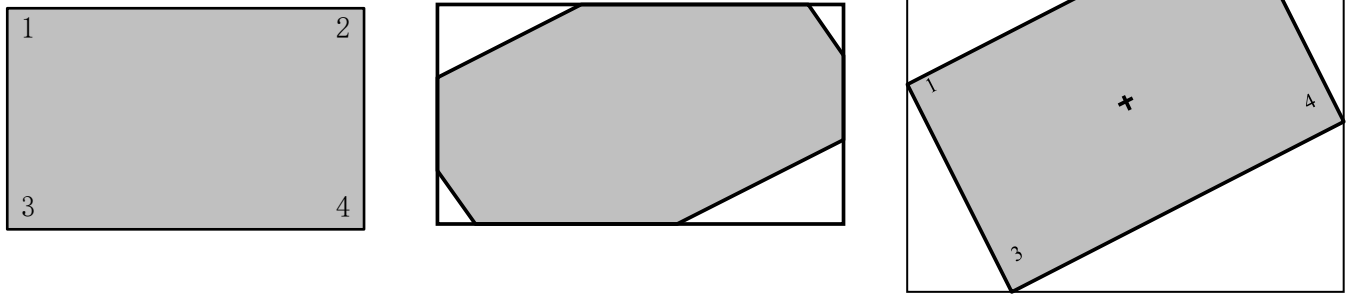
$$T = T_n \cdot T_{n-1} \cdots T_2 \cdot T_1$$

- **绕图像中心的旋转**，实际上可以看作是对图像实施的组合变换，即先做一次平移变换，将图像中心平移到坐标原点，再进行绕图像原点的旋转变换，最后把旋转后的图像再平移变换回去。

几何变换输出图像大小的处理

- **对图像进行空间坐标变换时，图像的大小一般会发生变化，部分像素坐标有可能不在原图像显示范围内，因此需对变换后的输出图像作适当处理。**
  - ‘loose’ 或‘resize’ - 调整输出图像大小，以容纳图像内容。
  - ‘crop’ - 保持与源图像相同大小，裁剪掉超出部分。

画布概念



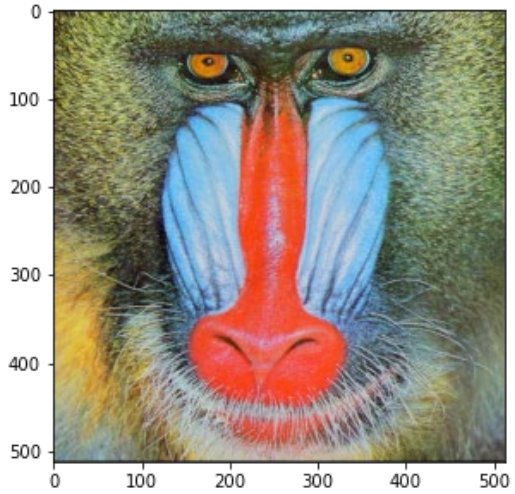
旋转 θ 后的图像（转出部分被截、扩大图像）

图像基本几何变换的编程实现

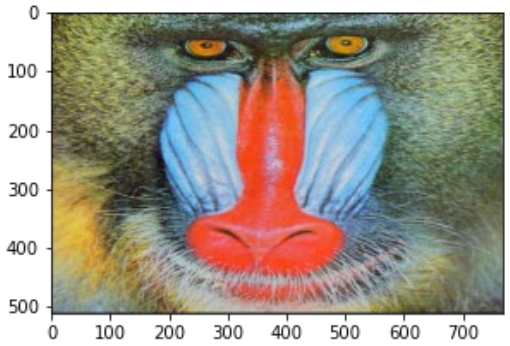
- OpenCV视觉库提供了常用的图像几何变换函数，如缩放cv.resize、计算二维旋转仿射变换矩阵cv.getRotationMatrix2D、利用3组控制点对计算仿射变换矩阵cv.getAffineTransform、利用4组控制点对计算投影变换矩阵cv.getPerspectiveTransform、计算仿射变换矩阵的逆矩阵cv.invertAffineTransform、仿射变换cv.warpAffine、投影变换cv.warpPerspective等。
- Scikit-image扩展库的transform模块，也提供了常用的图像几何变换函数，如图像缩放rescale、resize、图像旋转rotate、图像变形warp、2D仿射变换类AffineTransform、2D投影变换类ProjectiveTransform等。

示例：OpenCV图像缩放

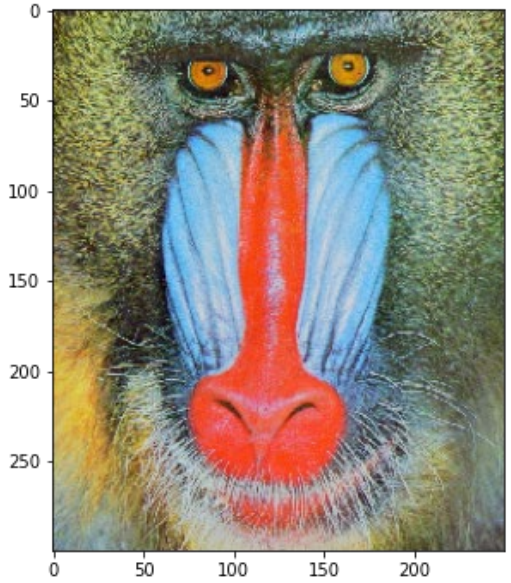
```
#导入本章示例用到的包
import numpy as np
import cv2 as cv
from skimage import color, io, util, transform
from scipy import ndimage
import matplotlib.pyplot as plt
%matplotlib inline
```



(1)原图像



(2)按给定的缩放比例因子



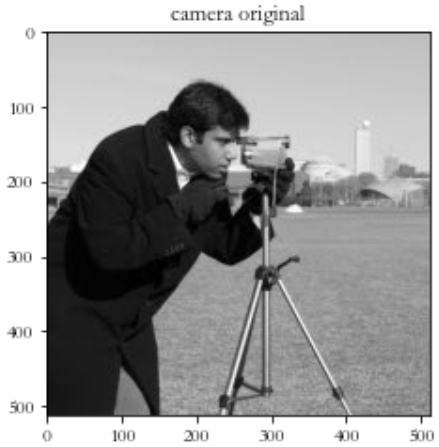
(3)按给定的图像宽/高

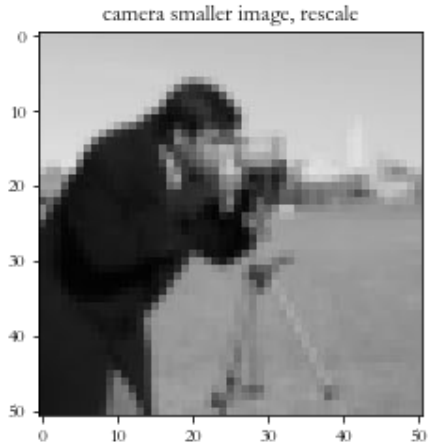
示例：OpenCV图像缩放

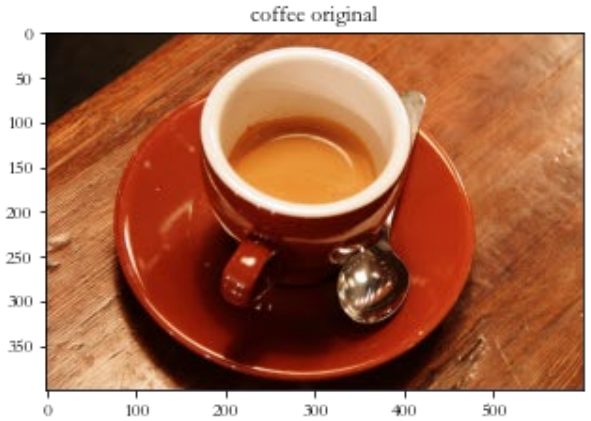
```
#OpenCV: 缩放图像
img = io.imread('./imagedata/baboon.jpg')  #读取一幅彩色图像
#按给定的缩放比例因子调整图像大小，默认双线性插值cv.INTER_LINEAR
imgbig= cv.resize(img,dsize=(0,0),fx=1.5, fy=1)
#按给定的图像尺寸调整图像大小，采用最近邻插值cv.INTER_NEAREST
width = 250; height = 300
imgsmall=cv.resize(img,dsize=(width,height),interpolation=cv.INTER_NEAREST)
#显示结果（略，详见本章Jupyter Notebook可执行笔记本文件）
```

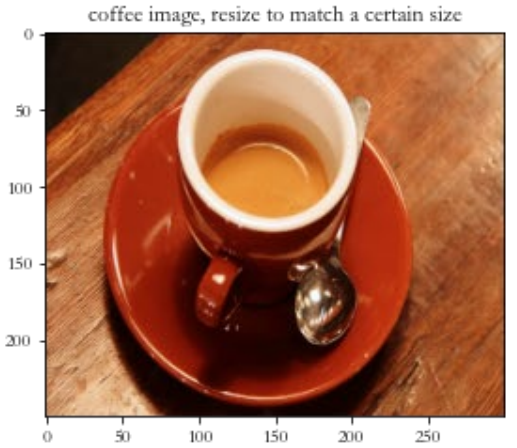
示例：Scikit-image图像缩放

- 在本章可执行笔记本文件“Ch6图像几何变换.ipynb”中。也给出了采用Scikit-image中的函数进行图像缩放的示例。







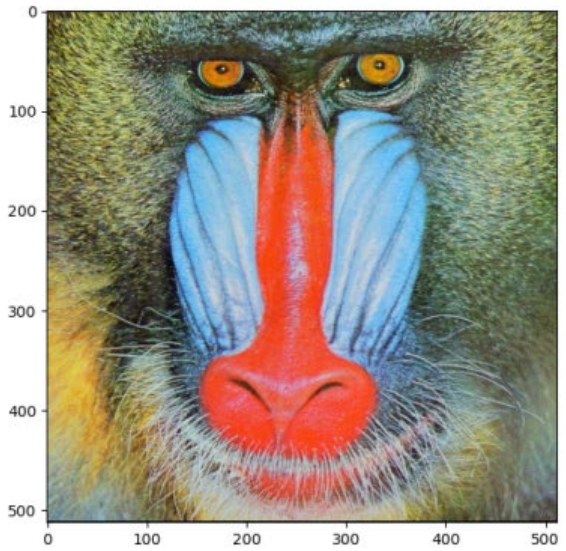


示例：Scikit-image图像缩放

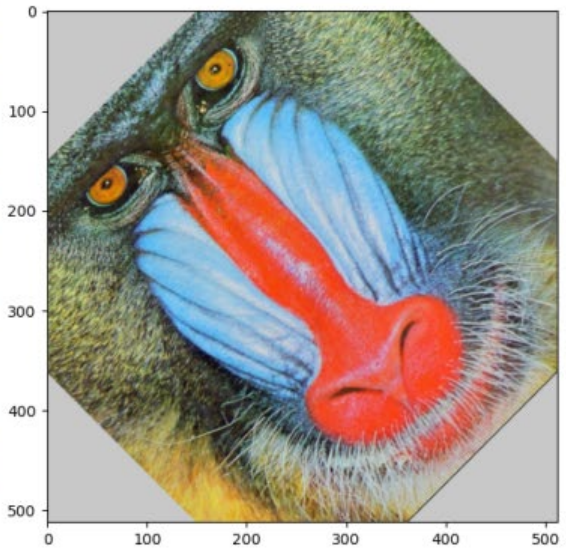
```
img_gray = io.imread('./imagedata/camera.png')  #灰度图像
img_rgb = io.imread('./imagedata/coffee.png')    #彩色图像
#高宽同比例缩小
img_gray_small = transform.rescale(img_gray, 0.1)
#将数据类型由[]0,1]之间的浮点型转换为uint8
img_gray_small = util.img_as_ubyte(img_gray_small)
#宽度放大为1.5倍
img_rgb_big = transform.rescale(img_rgb,(1,1.5),multichannel=True)
#将数据类型由[0,1]之间的浮点型转换为uint8
img_rgb_big = util.img_as_ubyte(img_rgb_big)
#将图像缩放到指定尺寸
img_rgb_size = transform.resize(img_rgb,(250,300))
#将数据类型由[0,1]之间的浮点型转换为uint8
img_rgb_size = util.img_as_ubyte(img_rgb_size)
```

示例：OpenCV图像旋转变换

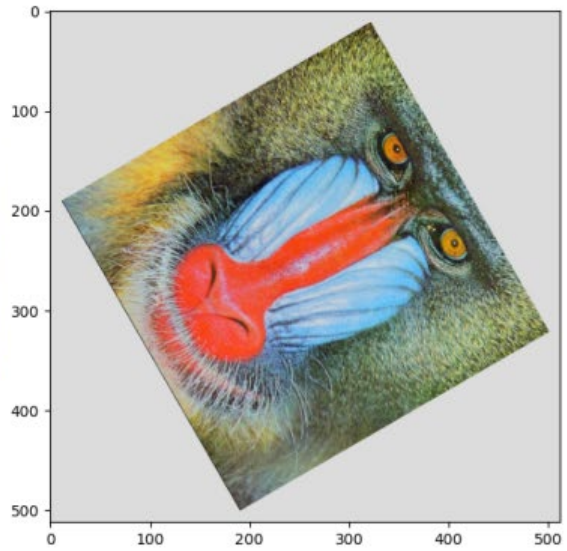
- 先调用OpenCV函数cv.getRotationMatrix2D()构造旋转变换矩阵，再调用函数cv.warpAffine()对图6.10(1)中图像施加仿射变换，实现图像旋转。绕图像中心，将图像逆时针旋转45度，不改变输出图像大小，结果如图(2)所示。绕图像中心，将图像顺时针旋转60度，改变输出图像大小以容纳图像内容，结果如图(3)所示。**注意图像内容大小的变化。**



(1)原图像



(2)逆时针绕图像中心旋转45度



(3)顺时针绕图像中心旋转60度、缩小

示例：OpenCV图像旋转变换

```
#OpenCV: 图像旋转
img = cv.imread('./imagedata/baboon.jpg',cv.IMREAD_COLOR)  #读入一幅彩色图像
img = cv.cvtColor(img,cv.COLOR_BGR2RGB)  #将色序有BGR调整为RGB
rows,cols=img.shape[0:2]  #获取图像的高、宽
#构建旋转变换矩阵，指定旋转中心，旋转角度，旋转后的缩放因子
#绕图像中心，将图像逆时针旋转45度，不改变输出图像大小
M1 = cv.getRotationMatrix2D(center=(cols/2,rows/2), angle=45, scale=1)
#采用仿射变换进行图像旋转，边缘处背景设为亮灰色
imgdst1 = cv.warpAffine(img,M1,dsize=(cols,rows),borderValue =(200,200,200))
#绕图像中心，将图像顺时针旋转60度，改变输出图像大小
M2 = cv.getRotationMatrix2D(center=(cols/2,rows/2), angle=-60, scale=0.7)
#采用仿射变换进行图像旋转，边缘处背景设为亮灰色
imgdst2 = cv.warpAffine(img,M2,dsize=(cols,rows),borderValue =(220,220,220))
```

示例：Scikit-image图像旋转变换

```
img = io.imread('./imagedata/chelsea.png')  #读入图像
#绕图像中心点逆时针旋转45度,放大图像尺寸
imgout1 = transform.rotate(img,45,resize=True)
imgout1 = util.img_as_ubyte(imgout1)   #将数据类型转换为uint8
#绕图像左上角像素(0,0)顺时针旋转30度,图像大小保持不变
imgout2 = transform.rotate(img,-30,center=(0,0))
imgout2 = util.img_as_ubyte(imgout2)   #将数据类型转换为uint8
```

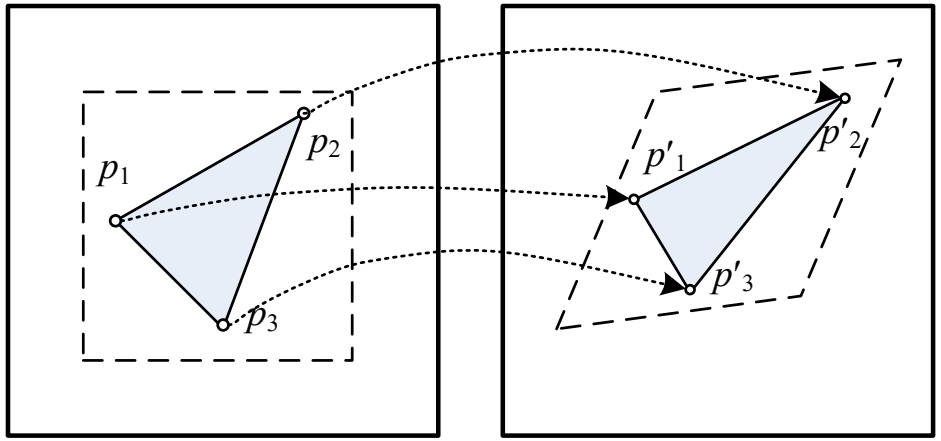
(1)原图像　　(2)逆时针绕图像中心旋转45度　　(3)顺时针绕图像左上角旋转30度

6.4 采用图像控制点的几何变换

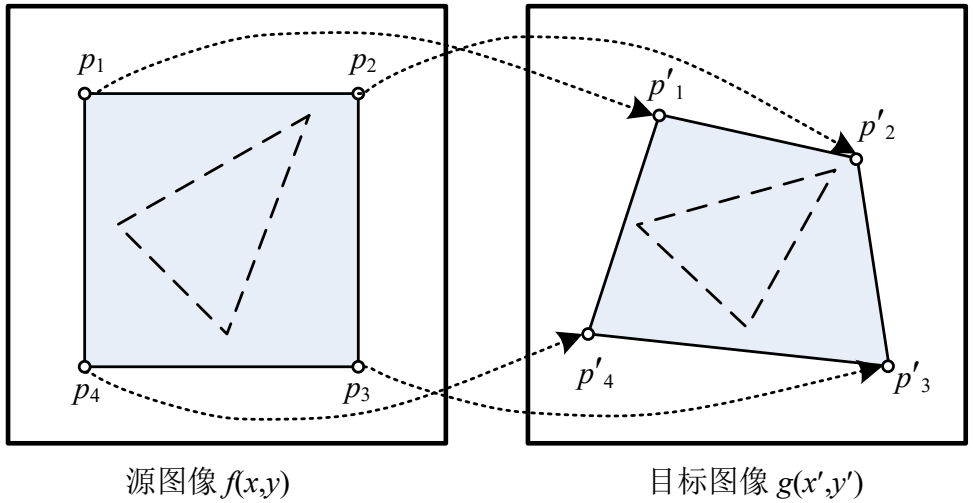
图像控制点

- 图像控制点（image control point），又称约束点，是在图像上选取的用于建立几何变换关系的参考点，即在源图像平面上选取一组点，然后确定它们在目标图像平面上的对应点，因此又称为控制点对。
- 图像控制点是图像几何校正、图像配准和图像变形等几何变换中常用的方法，它可以通过手动交互式，或自动方式在源图像和目标图像上选择一定数量的控制点对，然后利用这些控制点对，来估计图像所实施的几何变换函数。
- 描述控制点对之间映射关系的坐标变换函数，可以采用仿射变换、投影变换、多项式拟合等方法来建立和估计。

图像控制点



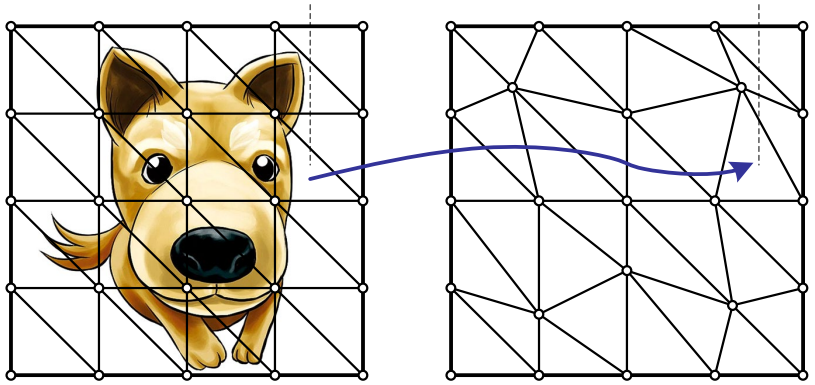
仿射变换图像控制点示意



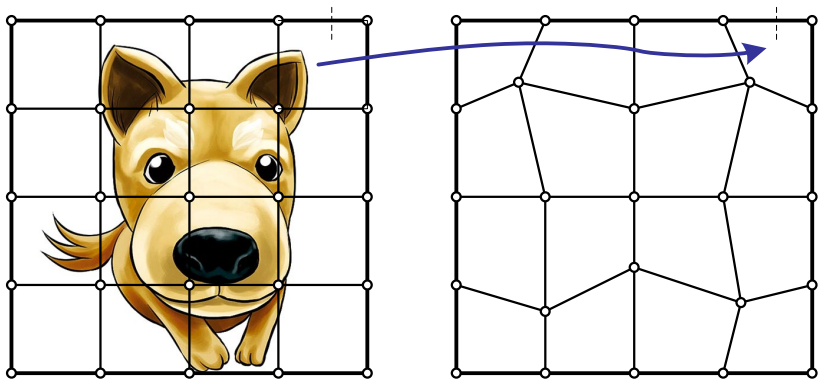
投影变换图像控制点示意

采用图像控制点的几何变换

- 采用图像控制点建立的几何变换函数可以是全局函数，应用于整幅图像中所有像素。
- 也可以是局部函数，对图像实施局部变换或者分段变换，由控制点为顶点构成控制点网格，将图像划分为很多不连续的小块，每一小块都应用各自的变换函数进行独立的变换。实际应用中，通常将图像分成很多三角形或者四边形网格。



控制点网格（仿射变换）

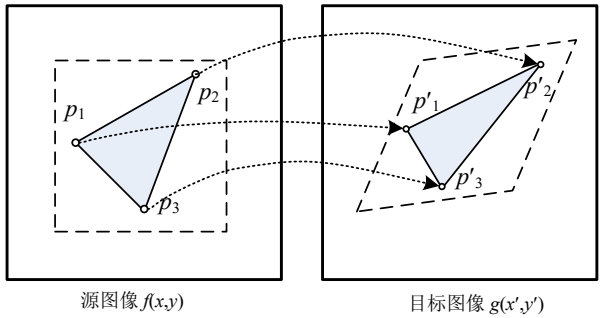


控制点网格（投影变换）

仿射变换

- 把下式描述的二维坐标变换称为“仿射变换”（Affine Transformation），或“仿射映射”（Affine Mapping）。

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



- 经仿射变换后，图像中的直线仍为直线，三角形仍为三角形，矩形则被变换为平行四边形。仿射变换不改变直线上点与点之间距离的比例，能保持直线的平行性，但是会改变两直线间的夹角。
- 仿射变换矩阵共有六个参数，需要至少 3 组非共线控制点对来确定。

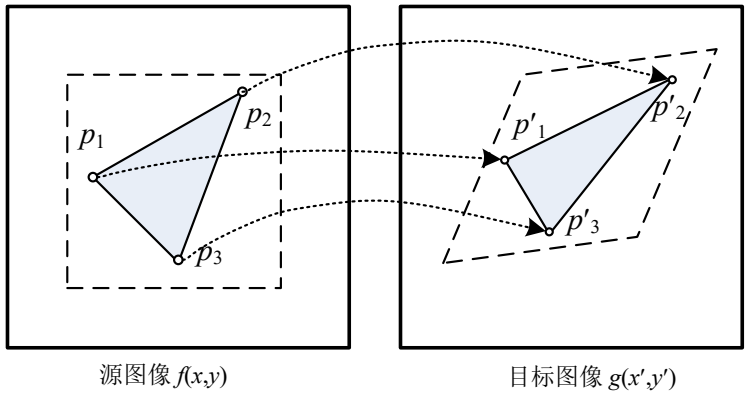
仿射变换矩阵的确定

- 仿射变换矩阵共有六个参数，需要三组非共线控制点对来确定。设$p_1=(x_1,y_1)$、$p_2=(x_2,y_2)$、$p_3=(x_3,y_3)$是位于源图像上三个非共线像素的坐标，它们各自位于目标图像上的对应点为$p'_1=(x'_1,y'_1)$、$p'_2=(x'_2,y'_2)$、$p'_3=(x'_3,y'_3)$。

$$\begin{cases} a_{11}\cdot x_1+a_{12}\cdot y_1+a_{13}=x'_1 \\ a_{11}\cdot x_2+a_{12}\cdot y_2+a_{13}=x'_2 \\ a_{11}\cdot x_3+a_{12}\cdot y_3+a_{13}=x'_3 \\ a_{21}\cdot x_1+a_{22}\cdot y_1+a_{23}=y'_1 \\ a_{21}\cdot x_2+a_{22}\cdot y_2+a_{23}=y'_2 \\ a_{21}\cdot x_3+a_{22}\cdot y_3+a_{23}=y'_3 \end{cases} \Rightarrow \begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 \\ x_2 & y_2 & 1 & 0 & 0 & 0 \\ x_3 & y_3 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & y_1 & 1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 \\ 0 & 0 & 0 & x_3 & y_3 & 1 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ y'_1 \\ y'_2 \\ y'_3 \end{bmatrix} \Rightarrow \begin{aligned} &\mathbf{Ma}=\mathbf{x} \\ &\mathbf{a}=\mathbf{M}^{-1}\mathbf{x} \end{aligned}$$

控制点对多于3对时：$\mathbf{a}=\left(\mathbf{M}^{\mathrm{T}}\mathbf{M}\right)^{-1}\mathbf{M}^{\mathrm{T}}\mathbf{x}$

M称系数矩阵，x为常数向量，a表示未知参数向量。



投影变换

- 投影变换(Projective Transformation)，又称透视变换(Perspective Transformation)，是将图片投影到一个新的视平面(Viewing Plane)的几何变换方法。
- 与仿射变换类似，投影变换也可以用齐次坐标表示为变换矩阵和齐次坐标向量的乘积，只是变换矩阵比仿射变换多了两个参数a_{31}, a_{32}，所以仿射变换可以看作是透视变换的特殊形式。投影变换函数定义为：

$$\begin{bmatrix} h' \cdot x' \\ h' \cdot y' \\ h' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

或，

$$\begin{cases} x' = \dfrac{1}{h'} \cdot \left(a_{11}x + a_{12}y + a_{13} \right) = \dfrac{a_{11}x + a_{12}y + a_{13}}{a_{31}x + a_{32}y + 1} \\ y' = \dfrac{1}{h'} \cdot \left(a_{21}x + a_{22}y + a_{23} \right) = \dfrac{a_{21}x + a_{22}y + a_{23}}{a_{31}x + a_{32}y + 1} \end{cases}$$

投影变换矩阵共有8个参数，需要 4 组非共线控制点对来确定。

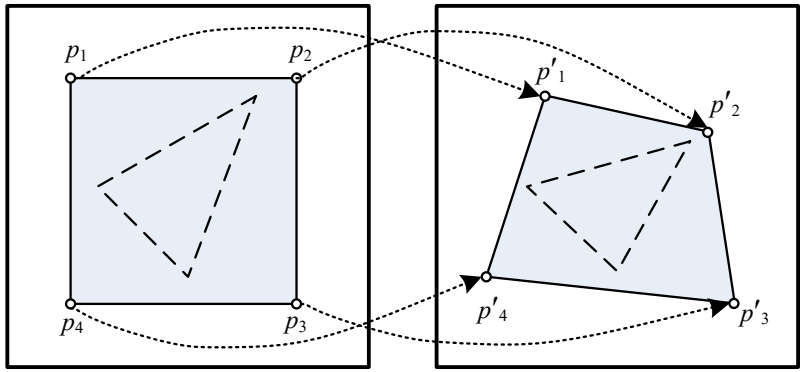
投影变换矩阵的确定

- 投影变换矩阵共有8个参数，需要4组非共线控制点对来确定。设$p_i=(x_i,y_i)$为位于源图像上的像素点（i=14），这4个非共线点构成了一个任意四边形，它们位于目标图像上的对应点为$p'_i=(x'_i,y'_i)$。

$$\begin{bmatrix} x_1 & y_1 & 1 & 0 & 0 & 0 & -x_1x'_1 & -y_1x'_1 \\ x_2 & y_2 & 1 & 0 & 0 & 0 & -x_2x'_2 & -y_2x'_2 \\ x_3 & y_3 & 1 & 0 & 0 & 0 & -x_3x'_3 & -y_3x'_3 \\ x_4 & y_4 & 1 & 0 & 0 & 0 & -x_4x'_4 & -y_4x'_4 \\ 0 & 0 & 0 & x_1 & y_1 & 1 & -x_1y'_1 & -y_1y'_1 \\ 0 & 0 & 0 & x_2 & y_2 & 1 & -x_2y'_2 & -y_2y'_2 \\ 0 & 0 & 0 & x_3 & y_3 & 1 & -x_3y'_3 & -y_3y'_3 \\ 0 & 0 & 0 & x_4 & y_4 & 1 & -x_4y'_4 & -y_4y'_4 \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \end{bmatrix} = \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \\ x'_4 \\ y'_1 \\ y'_2 \\ y'_3 \\ y'_4 \end{bmatrix}$$

$$\Rightarrow \quad \mathbf{Ma}=\mathbf{x} \qquad \mathbf{a}=\mathbf{M}^{-1}\mathbf{x}$$

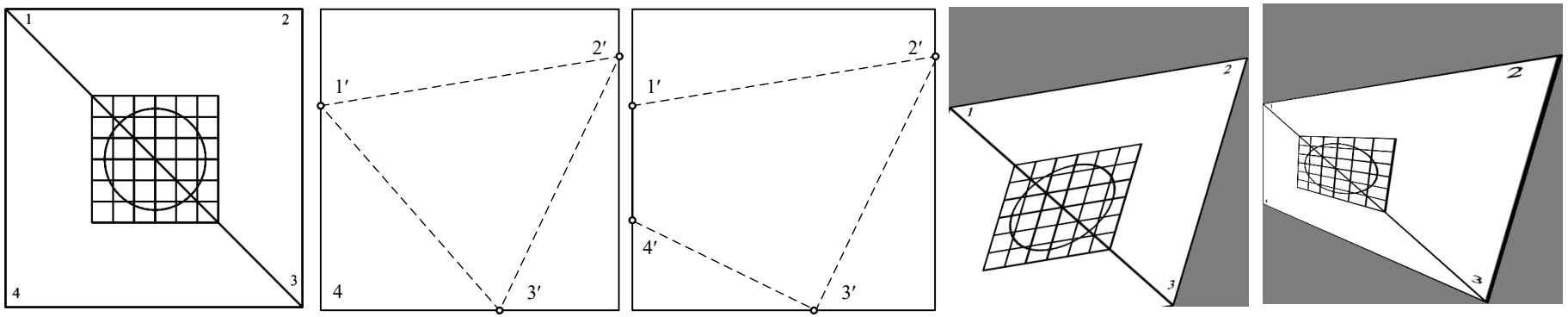
M称系数矩阵，**x**为常数向量，**a**表示未知参数向量。



源图像 $f(x,y)$　　目标图像 $g(x',y')$

示例：采用图像控制点的仿射变换

- 采用图像控制点方法，使用仿射变换、投影变换对一个合成图案进行几何变换。
- 图(1)为源图像，图(2)为仿射变换所需的三个对应点的位置；图(3)为投影变换所需的四个对应点的位置；图(4)是采用图(2)中1-1，2-2，3-3三组控制点对得到的仿射变换结果，图(5)是采用图(3)中1-1，2-2，3-3，4-4四组控制点对得到的投影变换结果。
- 注意，顶点4已被映射到目标图像外部；为方便观察变换前后图像内容的变化，采用灰色填充。



(1)源图像　(2)仿射变换3个对应点位置　(3)投影变换4个对应点位置　(4)仿射变换结果　(5)投影变换结果

示例：采用图像控制点的投影变换

```
#OpenCV: 采用控制点的图像几何变换--仿射变换/投影变换
#读入图像
imgdst_af = cv.imread('./imagedata/square_circle_af.bmp', 0)
imgdst_pe = cv.imread('./imagedata/square_circle_pe.bmp', 0)
img = cv.imread('./imagedata/square_circle.bmp', 0)
rows, cols = img.shape  #获取源图像高/宽
#确定仿射变换的3组控制点对，注意：水平(列)为x,垂直(行)为y
#源图像中标为1,2,3顶点像素的坐标
src1 = np.float32([[0, 0], [cols-1,0], [cols-1, rows-1]])
#源图像中标为1,2,3顶点像素，在期望输出图像中的对应坐标
dst1 = np.float32([[0, 160], [cols-1, 80], [360,rows-1]])
Mat_af = cv.getAffineTransform(src1,dst1)  #估计仿射变换矩阵
#施加仿射变换，采用后向映射
img_af = cv.warpAffine(img, Mat_af,(cols,rows),borderValue=125)
```

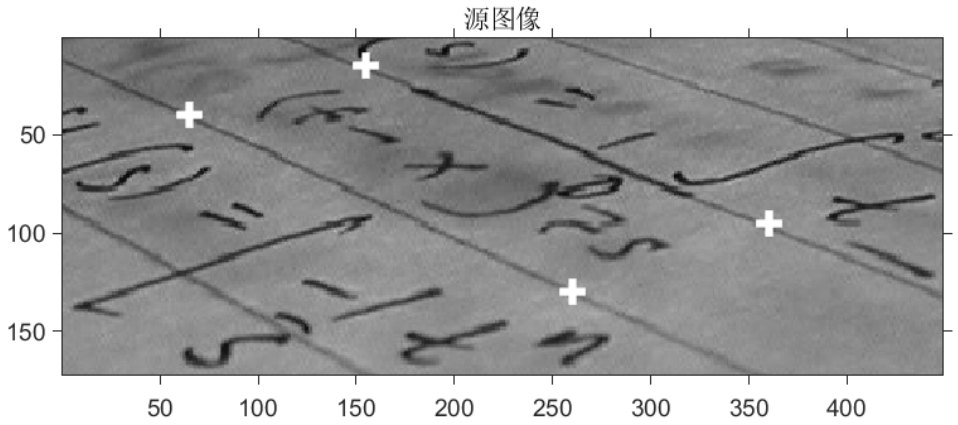
示例：采用图像控制点的投影变换

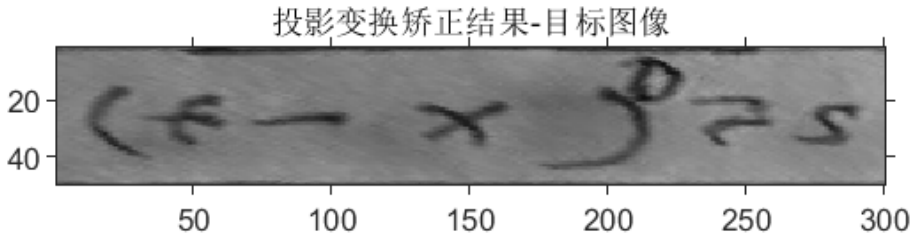
```
#OpenCV: 采用控制点的图像几何变换--仿射变换/投影变换


#确定投影变换的4组控制点对
#源图像中标为1,2,3,4顶点像素的坐标
src2 = np.float32([[0, 0], [cols -1,0], [cols -1,rows -1], [ 0,rows -1]])
#源图像中标为1,2,3,4顶点像素，在期望输出图像中的对应坐标
dst2 = np.float32([[0,160], [cols -1,80], [360,rows -1], [ 0,320]])
Mat_pe = cv.getPerspectiveTransform(src2,dst2)  #估计投影变换矩阵
#施加投影变换，采用后向映射
img_pe = cv.warpPerspective(img, Mat_pe,(cols,rows),borderValue=125)
```

示例：采用投影变换矫正图像畸变

- 假设要识别照片上的字母，但它不是从正面拍摄的，而是以一定角度拍摄，字母产生了投影扭曲变形，识别非常困难。解决这个问题的一种方法是选择一组对应点，对图像进行投影变换以便消除失真。原图像上的叠加4个“+”中心是选择的控制点。





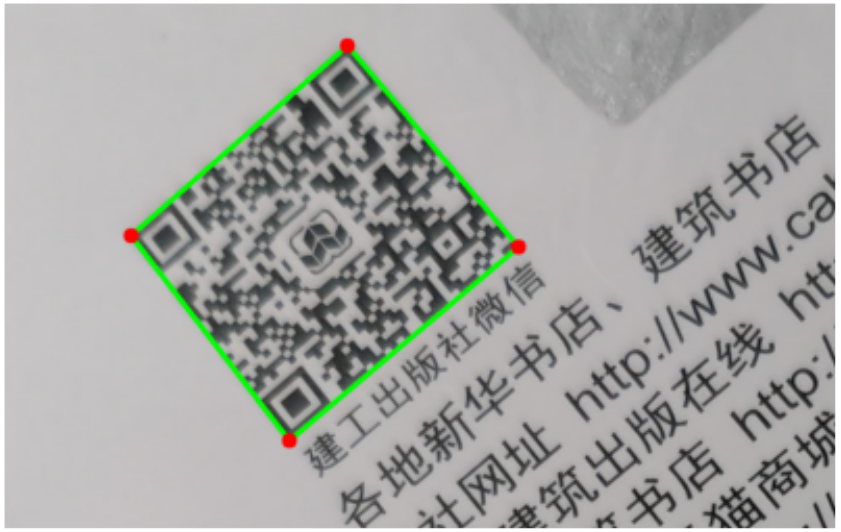
示例：采用投影变换矫正图像畸变

```
#OpenCV: 采用控制点矫正图像的投影畸变
img_text = cv.imread('./imagedata/text.png', 0)
#确定投影变换的4组控制点对，原图像中4个标“X”像素的坐标
src = np.float32([[155, 15], [65, 40], [260, 130], [360, 95]])
#标“X”控制点在期望输出图像中的对应坐标
dst = np.float32([[0, 0], [0, 50], [300, 50], [300, 0]])
#估计投影变换矩阵
Mat_pe = cv.getPerspectiveTransform(src,dst)
#施加投影变换，采用后向映射，输出图像宽300/高50
img_pe = cv.warpPerspective(img_text,Mat_pe,(300,50))
#显示结果（略）
```

示例：QR二维码扫码识别

- QR二维码（Quick Response code）得到广泛应用。OpenCV提供了QR二维码的检测和解码函数。示例程序首先调用函数cv.QRCodeDetector创建QR二维码检测器，然后调用其成员函数detectAndDecode检测指定图像中存在的QR二维码并解码，返回解码信息、检测到的二维码四边形顶点下标以及矫正后的二维码区域二值图像。

解码信息: http://weixin.qq.com/r/bUwfB4nEmNTnrc1k9xl-



矫正后的二维码

示例：QR二维码扫码识别

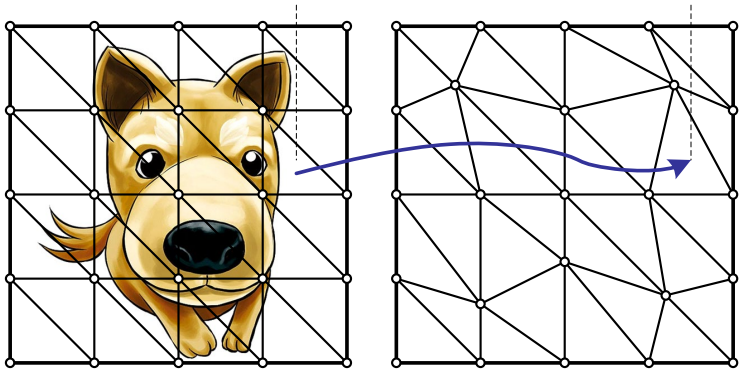
```
#QR二维码扫码识别
img = cv.imread("./imagedata/qrcode_image.jpg")  #读入一幅图像
qrcode_detector = cv.QRCodeDetector()  #创建QR二维码检测器
#检测图像中的二维码并解码
#返回解码信息，检测到的二维码顶点下标以及矫正后的二维码区域二值图像
data, vertices, rectified_qrcode_binarized = qrcode_detector.detectAndDecode(img)
#显示二维码检测及解码结果
if len(data) > 0:
    print("解码信息: '{}'".format(data))  #显示解码信息
    #在输入图像上绘制检测到的二维码四边形
    pts = np.int32(vertices).reshape(-1, 2)
    img = cv.polylines(img, [pts] , True, (0, 255, 0), 5)
    #用圆环标出二维码区域的四个顶点
    for j in range(pts.shape[0]):
        cv.circle(img, (pts[j,0],pts[j,1]), 10, (0, 0, 255), -1)
```

示例：QR二维码扫码识别

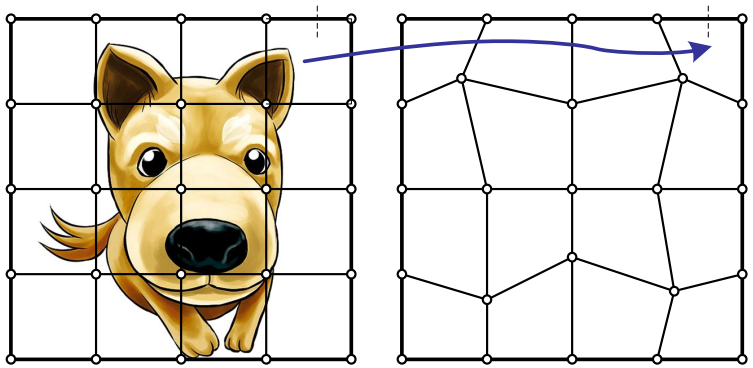
```
    plt.figure(figsize=(12,6))  #创建显示窗口
    plt.suptitle("QR code detection", fontsize=14, fontweight='bold')
    plt.gray()
    #显示检测到的二维码及解码信息
    plt.subplot(1,2,1), plt.imshow(img[:,:,::-1])
    plt.title("decoded data: " + data); plt.axis('off')
    #显示矫正后的二维码区域二值图像
    plt.subplot(1,2,2), plt.imshow(np.uint8(rectified_qrcode_binarized))
    plt.title("Rectified QR code"); plt.axis('off')
    plt.show()
else:
    print("QR Code not detected")
#----------------------------
```

图像分片局部坐标变换

- 用控制点为顶点构成控制点网格，将图像划分为很多线性区域块（piecewise-linear regions），每一小块都应用各自的变换函数进行独立的变换，对图像实施局部变换或者分段变换。实际应用中，通常将图像分成很多错综复杂的三角形或者四边形网格。



(1)局部仿射变换



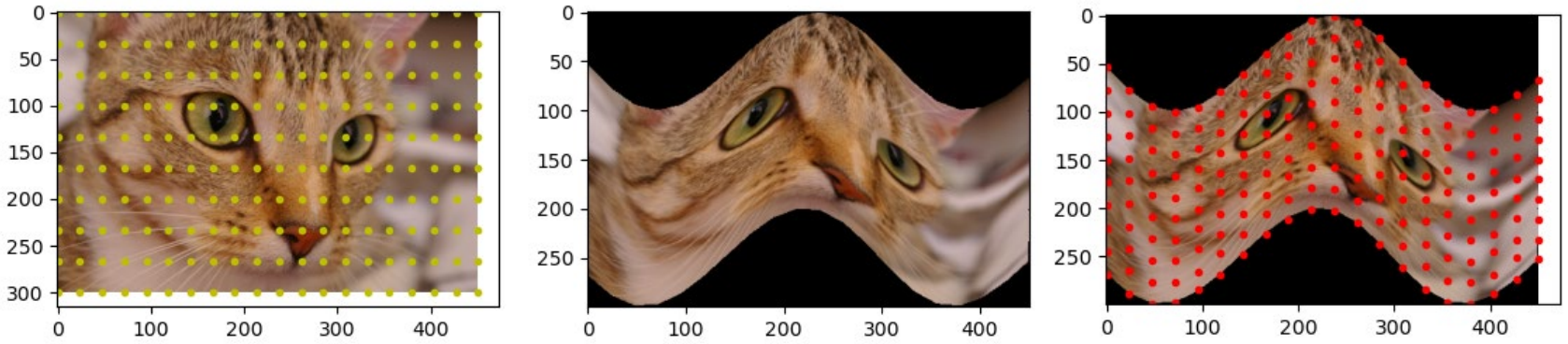
(2)局部投影变换

图像分片局部坐标变换

- 用控制点为顶点构成控制点网格，每一个网格的变换函数都可以采用仿射变换、投影变换或多边形变换，其参数分别由对应的三角形或四边形顶点作为控制点对得到。
- 图像几何局部变换应用广泛，比如航空和卫星图像的配准，或者校正扭曲图像以进行全景拼接。在计算机图形学中，也应用类似的技术，如在绘制2D图像中将纹理图像映射到多边形3D表面。这项技术的另一个广泛的应用是“渐变动画”（morphing），即从一幅图像到另一幅图像逐步进行几何变换，与此同时调节其亮度（或者颜色)值。

示例：采用Scikit-image的分片仿射变换类扭曲图像

```
#采用Scikit-image分片仿射变换函数实现图像变形
img = io.imread('./imagedata/chelsea.png')  #读入图像
rows, cols = img.shape[0:2]  #获取图像的高/宽
#产生原图像三角化网格及其顶坐标，作为控制点
src_cols = np.linspace(0, cols, 20)
src_rows = np.linspace(0, rows, 10)
src_rows, src_cols = np.meshgrid(src_rows, src_cols)
src = np.dstack([src_cols.flat, src_rows.flat])[0]
```



(1)源图像（叠加了网格点）　(2)分片仿射变换正弦波动扭曲结果　(3)正弦波动扭曲结果（叠加扭曲后的网格点）

示例：采用Scikit-image的分片仿射变换类扭曲图像

```
#对行坐标施加正弦波动，扭曲三角化网格，生成对应点
dst_rows = src[:, 1] - np.sin(np.linspace(0, 3 * np.pi, src.shape[0])) * 50
dst_cols = src[:, 0]
dst_rows = 1.5*dst_rows
dst_rows =dst_rows - 1.5 * 50
dst = np.vstack([dst_cols, dst_rows]).T

#定义分片仿射变换类对象
tform = transform.PiecewiseAffineTransform()
#调用类成员函数估计后向映射矩阵
tform.estimate(src,dst)
#对图像施加分片仿射变换
img_out =transform.warp(img, tform)
#显示结果（略）
```

6.5 图像的非线性几何变换

非线性空间坐标变换

- 仿射变换和投影变换都属于线性变换，直线在线性变换的作用下还是直线。非线性空间坐标变换一般会将直线扭曲，以达到某种特殊效果。
- 对于非线性变换，通常很难获取从源图像到目标图像的前向映射变换函数。因此，一般直接给出实现从目标图像坐标到源图像的**后向映射函数**。

(x',y')目标图像像素坐标

(x,y)源图像中对应位置

$$\begin{cases} x=T_x^{-1}(x',y') \\ y=T_y^{-1}(x',y') \end{cases}$$

涡旋变换

- 涡旋变换（Swirl transformation），将图像围绕一个中心点$p_c=(x_c, y_c)$，以一个随位置变化的角度θ进行旋转，使得输出图像呈现漩涡效果。
- 该角度θ在接近中心点p_c处有一个最大值α，并且随着像素离中心点p_c距离的增大而变小。同时，在限定半径r_{max}之外，图像保持不变。实现涡旋变换的后向映射函数定义为：

$$x=\begin{cases} x_c + r\cdot\cos\theta, & r \le r_{\max} \\ x', & r > r_{\max} \end{cases}$$

$$y=\begin{cases} y_c + r\cdot\sin\theta, & r \le r_{\max} \\ y', & r > r_{\max} \end{cases}$$

式中，角度θ和α均采用弧度。

其中：

$$r=\sqrt{d_x^2+d_y^2}$$

$$\theta=\tan^{-1}\left(\frac{d_y}{d_x}\right)+\alpha\cdot\left(\frac{r_{\max}-r}{r_{\max}}\right)$$

$$d_x = x' - x_c$$

$$d_y = y' - y_c$$

涡旋变换

- 或采用以下表达式，让涡旋变换强度约以1/1000速率衰减：

$$x = x_c + r \cdot \cos\theta$$

$$y = y_c + r \cdot \sin\theta$$

其中：

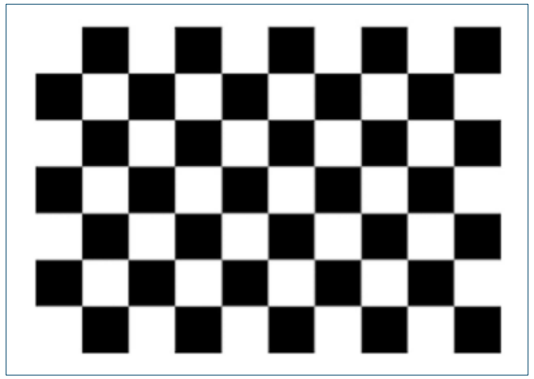
$$r = \sqrt{d_x^2 + d_y^2}, \quad \theta = \tan^{-1}\left(\frac{d_y}{d_x}\right) + \alpha \cdot \exp\left(\frac{-r}{r_a}\right)$$

$$r_a = \ln 2 \cdot r_{\max} / 5$$

$$d_x = x' - x_c, \quad d_y = y' - y_c$$

示例：调用Scikit-image中swirl函数实现涡旋变换

- 图(3)给出了最大旋转角=10弧度、附加逆时针旋转π/6、影响半径350时的涡旋变换结果；图(4)给出了最大旋转角=10弧度、无附加旋转、影响半径350时的涡旋变换结果，涡旋中心均采用默认的图像中心。



(1)源图像checkerboard

(2)源图像coffee

(3)涡旋变换结果（附加旋转）

(4)涡旋变换结果（无附加旋转）

示例：调用Scikit-image中swirl函数实现涡旋变换

```
#旋涡变换swirl transformation,调用swirl函数实现
#读入一幅灰度图像
img_gray = io.imread('./imagedata/chessboard.png')
#旋涡变换
img_gray_swirled = transform.swirl(img_gray,rotation=np.pi/6, strength=10, radius=350)
#将数据类型转换为uint8
img_gray_swirled = util.img_as_ubyte(img_gray_swirled)
#读入一幅RGB彩色图像
img_rgb = io.imread('./imagedata/coffee.png')
#旋涡变换
img_rgb_swirled = transform.swirl(img_rgb,rotation=0, strength=10, radius=350,mode='edge')
#将数据类型转换为uint8
img_rgb_swirled = util.img_as_ubyte(img_rgb_swirled)
 #显示结果（略）
```

球面变换

- 球面变换（Spherical Transformation）模拟通过一个透明的半球看图像或者在图像上放置一个凸透镜时的效果。
- 该变换的参数包括透镜中心位置$p_c=(x_c, y_c)$，透镜半径$r_{\max}$和透镜折射率n。相应的后向映射函数定义如下：

$$x=\begin{cases} x'-t\cdot\tan\alpha_x, & r\le r_{\max} \\ x', & r>r_{\max} \end{cases}$$

$$y=\begin{cases} y'-t\cdot\tan\alpha_y, & r\le r_{\max} \\ y', & r>r_{\max} \end{cases}$$

其中：

$$r=\sqrt{d_x^2+d_y^2}$$

$$t=\sqrt{r_{\max}^2-r^2}$$

$$\alpha_x=\left(1-\frac{1}{n}\right)\cdot\sin^{-1}\left(\frac{d_x}{\sqrt{d_x^2+t^2}}\right)$$

$$\alpha_y=\left(1-\frac{1}{n}\right)\cdot\sin^{-1}\left(\frac{d_y}{\sqrt{d_y^2+t^2}}\right)$$

$$d_x=x'-x_c$$

$$d_y=y'-y_c$$

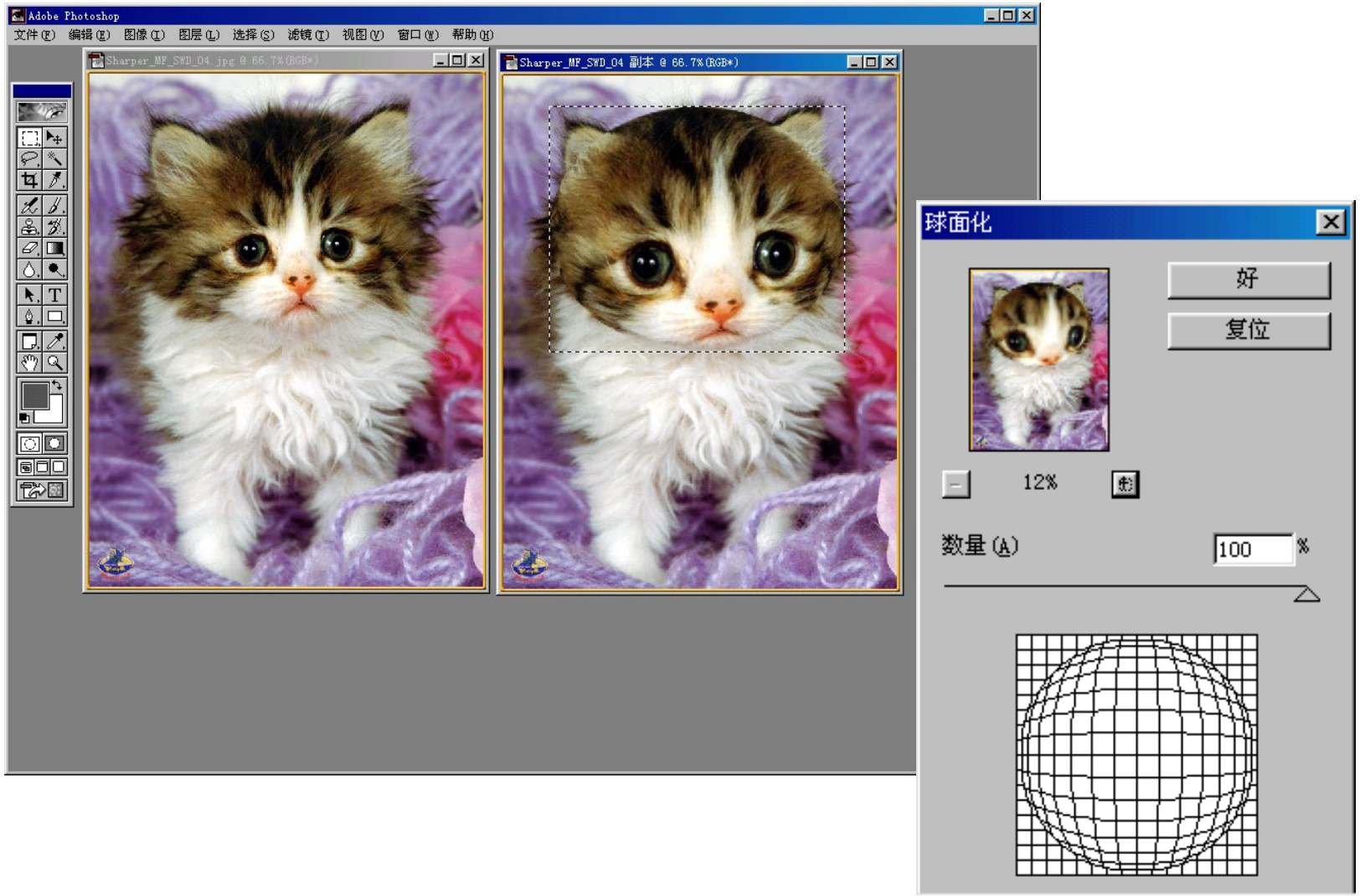
球面变换

(1)源图像

(2)r_{max} =150, n = 2.8

实现程序代码详见本章可执行笔记本文件“Ch6图像几何变换.ipynb”

Photoshop的球面化工具



波浪扭曲变换

- 波浪扭曲变换（Ripple Transformation）用正弦函数对图像进行局部扭曲，使图像在x和y方向局部波浪化。
- 波浪扭曲变换函数的参数包括用于控制x和y方向波浪波动频率的正弦波周期长度t_x、t_y（单位为像素，其值不能为0），和用于控制两个方向上波动幅度的参数a_x、a_y（单位为像素）。
- 波浪扭曲的后向映射函数定义为：

$$x = x' + a_x \sin\left(2\pi \frac{y'}{t_x}\right)$$

$$y = y' + a_y \sin\left(2\pi \frac{x'}{t_y}\right)$$

波浪扭曲变换

(1)源图像

(2)ax=20,ay=30,tx=50,ty=20

实现程序代码详见本章可执行笔记本文件“Ch6图像几何变换.ipynb”

Q&A