第二次课堂作业——仿射变换

学号： 201983160037

姓名： 强盛周

班级： 19信计嵌入1班

邮箱： qshengz@foxmail.com

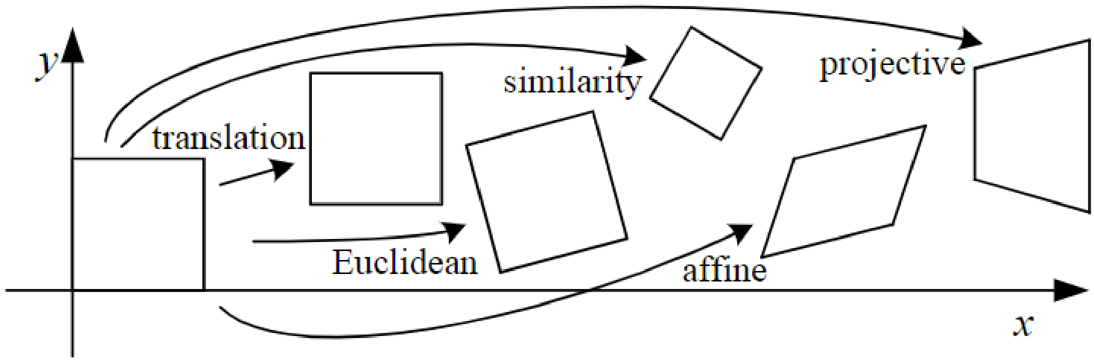
课程名称： 数字图像处理II

授课教师： 陈允杰教授

1. 作业要求

1. 完成恒等、缩入、反射、旋转、平移、垂直剪切、水平剪切等仿射变换
2. 以上步骤在第一次作业的基础上完成

2. 仿射变换介绍



仿射变换（Affine transformation），又称仿射映射，是指在几何中，对一个向量空间进行一次线性变换并接上一个平移，变换为另一个向量空间。

一个对向量$\vec{x}$平移$\vec{b}$，与旋转缩放A的仿射映射为

$\vec{y}=A\vec{x}+\vec{b}$ 上式在齐次坐标上，等价于下面的式子

$\begin{bmatrix}\vec{y}\\1\end{bmatrix}=\begin{bmatrix}A & \vec{b}\\0,\ldots,0 & 1\end{bmatrix}\begin{bmatrix}\vec{x}\\1\end{bmatrix}$ 在分形的研究里，收缩平移仿射映射可以制作具有自相似性的分形。

2.1 性质

仿射变换保留了：

1. 点之间的共线性：在同一条直线上的三个或更多的点（称为共线点）在变换后依然在同一条直线上（共线）；
2. 直线的平行性：两条或以上的平行直线，在变换后依然平行；
3. 集合的凸性：凸集合变换后依然是凸集合。并且，最初的极值点被映射到变换后的极值点集；
4. 平行线段的长度的比例：两条由点 p_1, p_2, p_3, p_4 定义的平行线段，$\overrightarrow{p_1p_2}$ 与 $\overrightarrow{p_3p_4}$的长度的比例等于$\overrightarrow{f(p_1)f(p_2)}$与$\overrightarrow{f(p_3)f(p_4)}$的长度的比例；
5. 不同质量的点组成集合的质心。

仿射变换为可逆的当且仅当A为可逆的。用矩阵表示，其逆元为：

$$\left[\begin{array}{ccc|c} & A^{-1} & & -A^{-1}\vec{b} \\ 0, & \ldots & ,0 & 1 \end{array}\right]$$

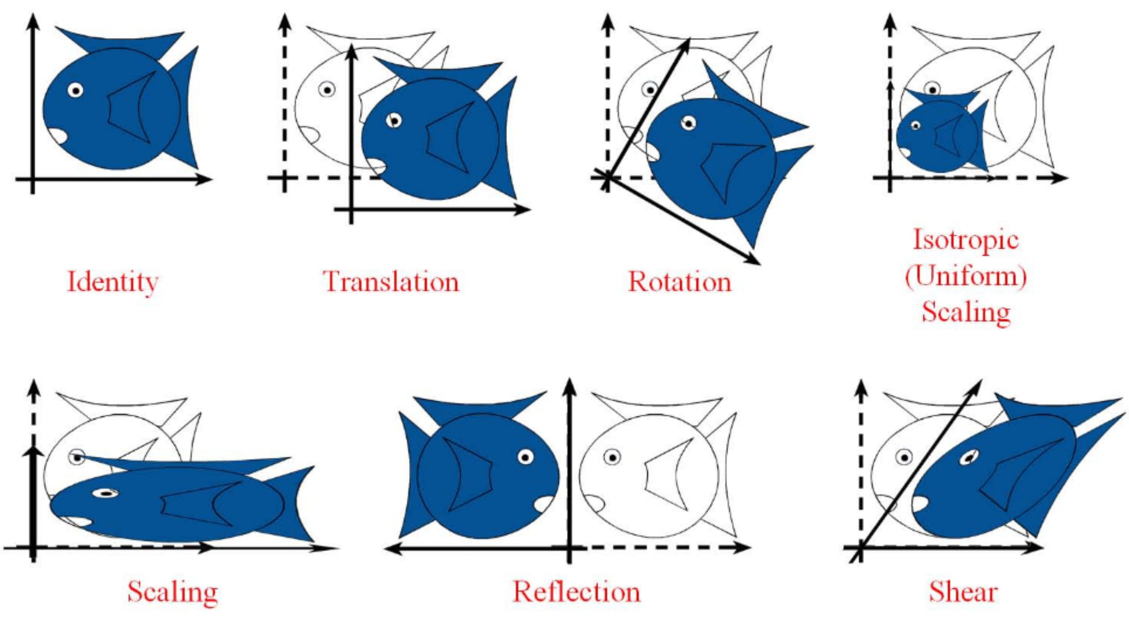
可逆仿射变换组成仿射群，其中包含具n阶的一般线性群为子群，且自身亦为一 $n+1$ 阶的一般线性群之子群。 当A为常数乘以正交矩阵时，此子集合构成一子群，称之为相似变换。举例而言，假如仿射变换于一平面上且假如 A 之行列式为 1 或 -1，那么该变换即为等面积变换。此类变换组成被称为等仿射群的子群。一同时为等面积变换与相似变换的变换，即为一平面上保持欧几里德距离不变的保距映射。

这些群都有一保留了原定向的子群，也就是其对应之 A 的行列式大于零。最后一个例子，即三维空间中刚体的运动组成的群（旋转和平移），刚体的运动在机器人学中尤为常用。

如果有一固定点，我们可以将其当成原点，则仿射变换被缩还到一线性变换。这使得变换更易于分类与理解。举例而言，将一变换叙述为特定轴的旋转，相较于将其形容为平移与旋转的结合，更能提供变换行为清楚的解释。只是，这取决于应用与内容。

2.2 基本仿射变换

仿射变换包括如下所有变换，以及这些变换任意次序次数的组合： 平移、旋转、放缩、剪切、反射

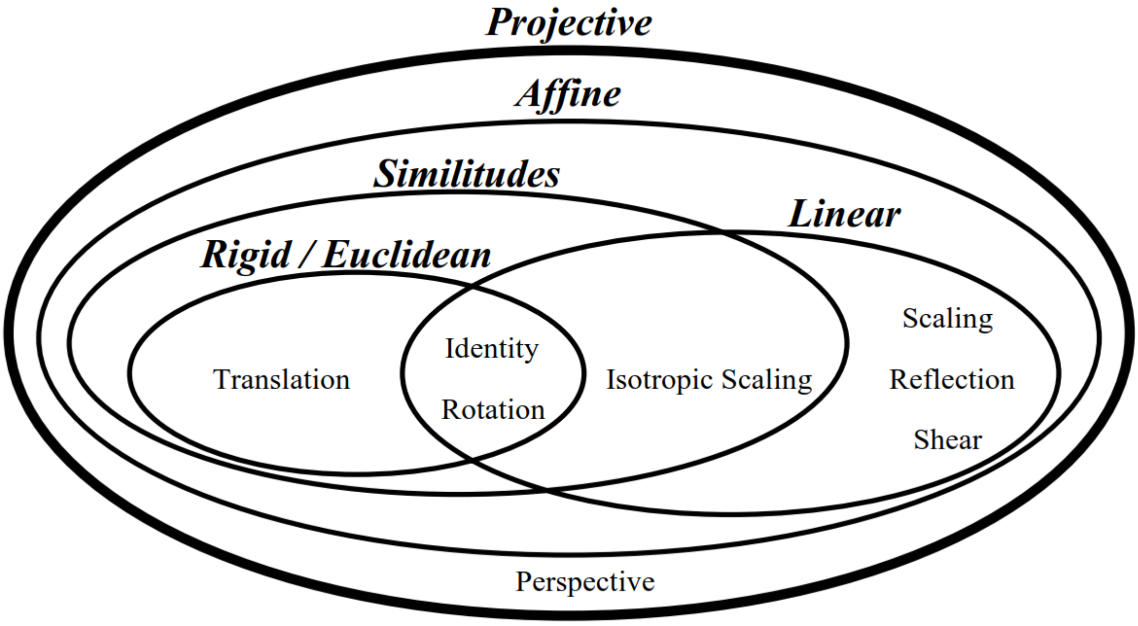


平移（translation）和旋转（rotation）顾名思义，两者的组合称之为欧式变换（Euclidean transformation）或刚体变换（rigid transformation）；

放缩（scaling）可进一步分为uniform scaling和non-uniform scaling，前者每个坐标轴放缩系数相同（各向同性），后者不同；如果放缩系数为负，则会叠加上反射（reflection）——reflection可以看成是特殊的scaling；

刚体变换+uniform scaling 称之为，相似变换（similarity transformation），即平移+旋转+各向同性的放缩； 剪切变换（shear mapping）将所有点沿某一指定方向成比例地平移，语言描述不如上面图示直观。

各种变换间的关系如下面的venn图所示：



2.3 变换矩阵形式

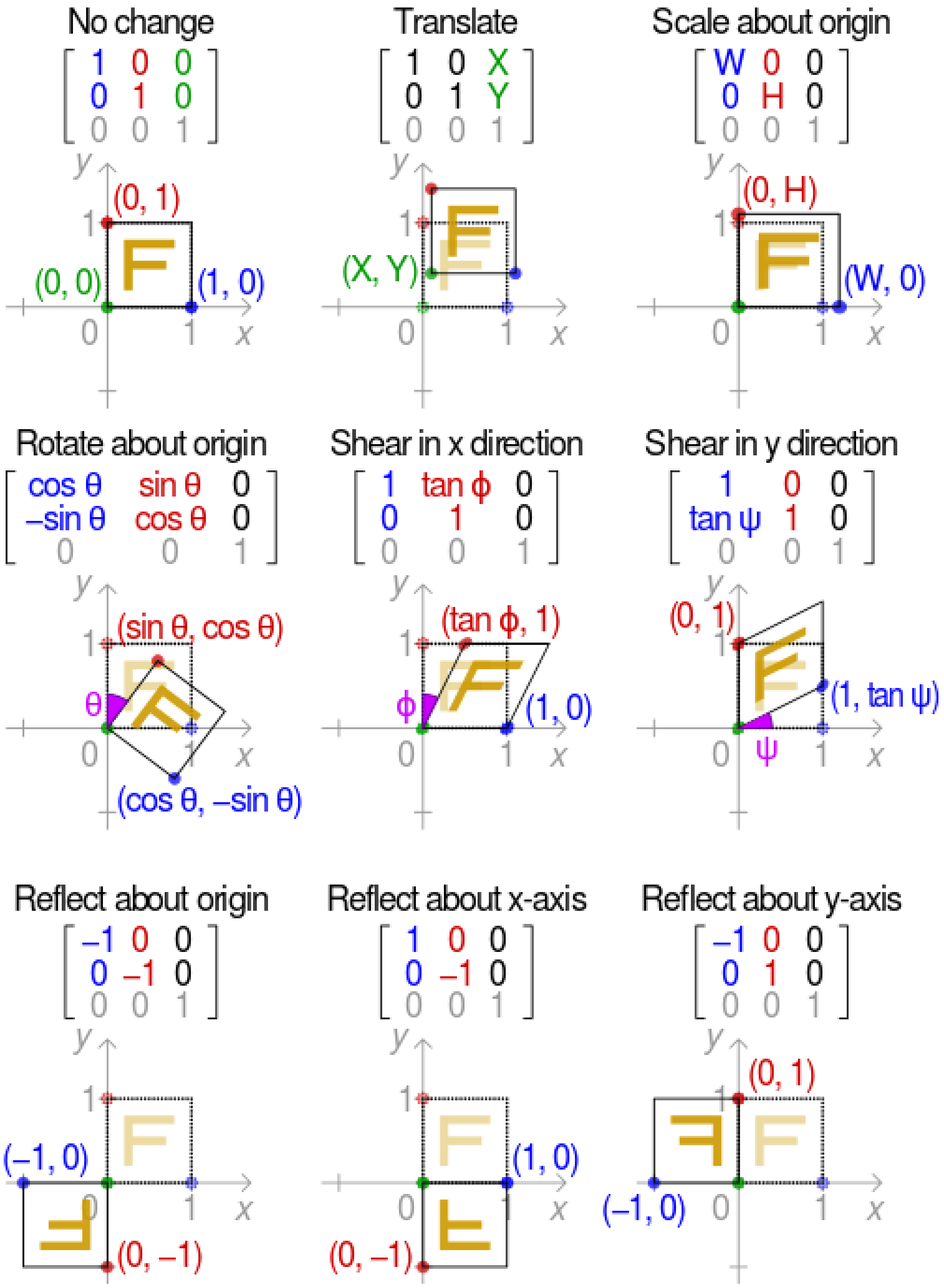
不同变换对应的a, b, c, d约束不同，排除了平移变换的所有仿射变换为线性变换（linear transformation），其涵盖的变换如上面的venn图所示，其特点是原点位置不变，多次线性变换的结果仍是线性变换。

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

为了涵盖平移，引入齐次坐标，在原有2维坐标的基础上，增广 1 个维度，如下所示:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

所以，仿射变换的变换矩阵统一用 $\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$ 来描述，不同基础变换的 a, b, c, d, e, f 约束不同，如下所示:



3. 参考资料

1. 维基百科：仿射变换
   https://zh.wikipedia.org/wiki/%E4%BB%BF%E5%B0%84%E5%8F%98%E6%8D%A2
2. 仿射变换及其变换矩阵的理解 https://www.cnblogs.com/shine-lee/p/10950963.html
3. Python使用OpenCV仿射变换实例 http://t.csdn.cn/9Oa4I

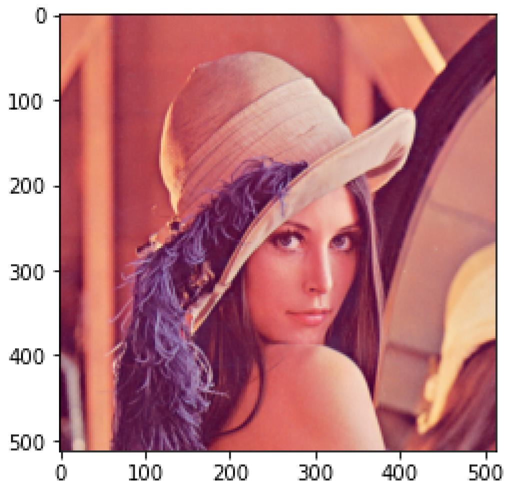
4. 代码以及结果

In [1]:

```
# @Author: Alephant——QSZ
import numpy as np
import matplotlib.pylab as plt
import cv2
from matplotlib import image as mpimg
```

In [2]:
```python
# 读取图片
# 注意cv2读取图片是bgr,mping读取图片是rgb
# img = cv2.imread('../images/lena_small.png')
# img = mpimg.imread('../images/lena_small.png')
# img = mpimg.imread("../images/zyw.png")
img = mpimg.imread("../images/lena.png")
rows, cols, ch = img.shape
plt.imshow(img)
```

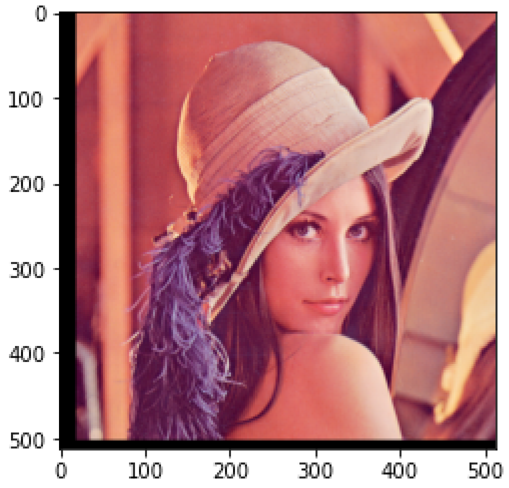
Out[2]: <matplotlib.image.AxesImage at 0x1bd3f340370>



4.1 平移变换

In [3]:
```python
'''
cv2.warpAffine()
仿射变换（从二维坐标到二维坐标之间的线性变换，且保持二维图形的“平直性”和“平行性”。
仿射变换可以通过一系列的原子变换的复合来实现，包括平移，缩放，翻转，旋转和剪切）
    参数:
        img: 图像对象
        M: 2*3 transformation matrix (转变矩阵)
        dsize: 输出矩阵的大小,注意格式为（cols, rows)   即width对应cols, height对应row
        flags: 可选，插值算法标识符，有默认值INTER_LINEAR,
               如果插值算法为WARP_INVERSE_MAP, warpAffine函数使用如下矩阵进行图像转
               dst(x,y)=src(M11*x+M12*y+M13,M21*x+M22*y+M23)
        borderMode: 可选，  边界像素模式，有默认值BORDER_CONSTANT
        borderValue:可选，边界取值，有默认值Scalar()即0
'''
# 创建转变矩阵——平移变换
# 下面的M矩阵表示向x轴正方向（向右）移动20像素，向y轴负方向（向上）移动10像素。
M1 = np.float32([[1, 0, 20], [0, 1, -10]])
# 使用warpAffine()方法执行变换
img1 = cv2.warpAffine(img, M1, dsize=(cols, rows))
plt.imshow(img1)
```

Out[3]: <matplotlib.image.AxesImage at 0x1bd40405b10>

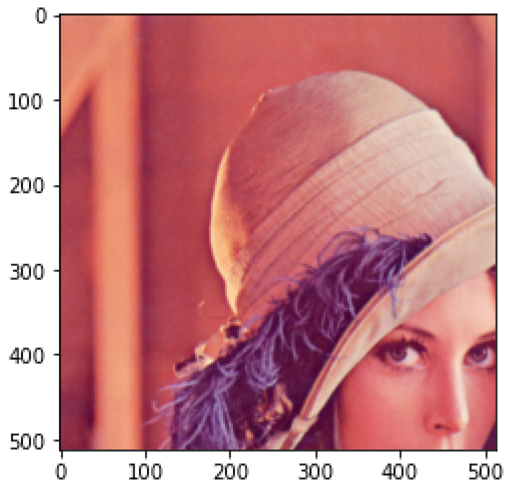


4.2 缩放变换

缩放变换的转变矩阵与平移变换的转变矩阵是一样的。 M是一个2*3矩阵，[[1, 0, 100], [0, 1, 200]]。 从左到右看，第一个“1”代表x轴（长度）是原来的几倍，“100”表示将原图沿x轴如何移动。 第二个“1”表示y轴（宽度）是原来的几倍，“200”表示将原图沿y轴如何移动。 下面的操作表示将原图放大1.5倍。

In [4]:
```
# 创建转变矩阵—缩放变换
M2 = np.float32([[1.5, 0, 0], [0, 1.5, 0]])
img2 = cv2.warpAffine(img, M2, (cols, rows))
plt.imshow(img2)
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x1bd40479390>
```



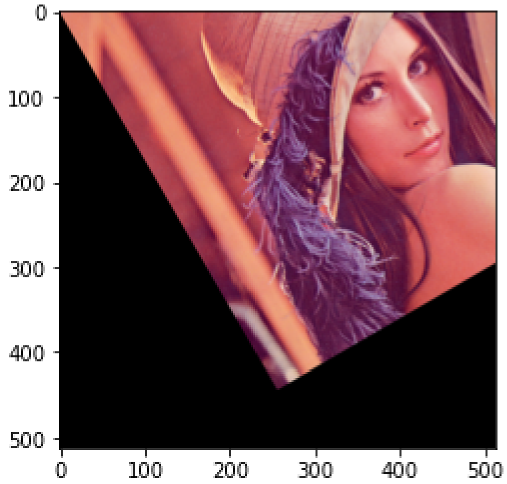
4.3 旋转变换

通过getRotationMatrix2D()能得到转变矩阵M。

In [5]:
```
'''
cv2.getRotationMatrix2D()  返回2*3的转变矩阵（浮点型）
    参数：
        center：旋转的中心点坐标
        angle：旋转角度，单位为度数，证书表示逆时针旋转
        scale：同方向的放大倍数
'''
# 创建转变矩阵—旋转变换
# 以图片的左上角（0，0）原点为旋转中心，旋转30°。
```

```python
M3 = cv2.getRotationMatrix2D((0, 0), 30, 1)
# 执行转变
img3 = cv2.warpAffine(img, M3, (cols, rows))
plt.imshow(img3)
```

Out[5]: <matplotlib.image.AxesImage at 0x1bd404c3e80>



4.4 三点定位，仿射变换矩阵的计算，cv2.getAffineTransform()

通过图片变换前后的三组坐标定位，和cv2.getAffineTransform()方法，可以计算出我们需要的仿射变换矩阵M。

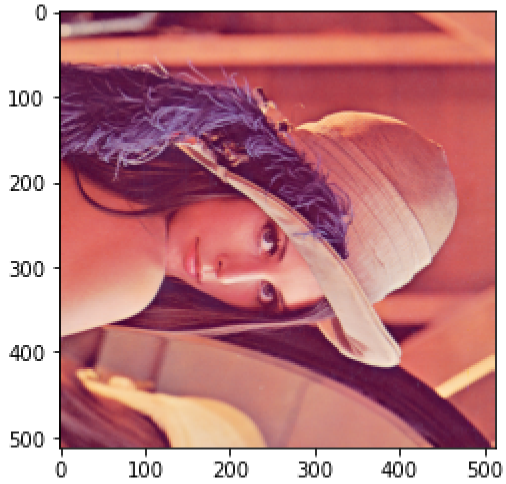
该方法可以一定程度上代替上面三个方法，同时实现旋转、平移和缩放。

getAffineTransform()等同于将平移，旋转和缩放的变换矩阵相乘，最后都会获得仿射变换矩阵。

In [6]:

```python
'''
cv2.getAffineTransform()  返回2*3的转变矩阵
       参数:
            src: 原图像中的三组坐标，如np.float32([[50,50],[200,50],[50,200]])
            dst: 转换后的对应三组坐标，如np.float32([[10,100],[200,50],[100,250]])
'''
# img原图的四个角的坐标为[0, 0] , [cols, 0], [0,rows], [cols, rows]。
#  (此处为[x, y] 坐标形式，不同于shape)
# 原图像中的三组坐标
pts1 = np.float32([[0, 0] , [rows, 0], [rows, cols]])
# 转换后的三组对应坐标
pts2 = np.float32([[rows, 0], [cols, rows], [0, rows]])
# 计算仿射变换矩阵
M4 = cv2.getAffineTransform(pts1, pts2)
# 执行变换
img4 = cv2.warpAffine(img, M4 ,(cols, rows))
plt.imshow(img4)
```

Out[6]: <matplotlib.image.AxesImage at 0x1bd41173280>



4.5 透视变换（三维）

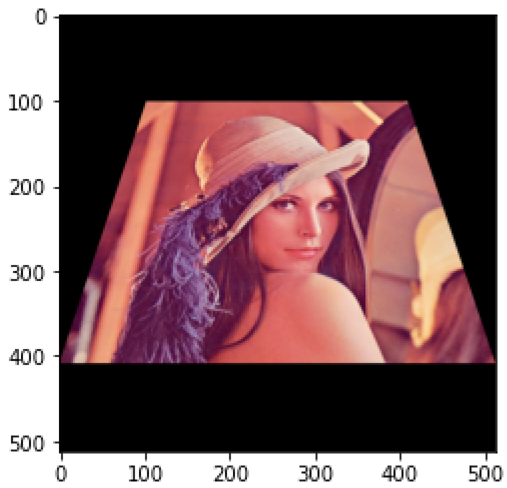
与四、三点定位，仿射变换矩阵的计算类似，三维的仿射变换矩阵需要四组坐标来进行定位。

与之前不同的是，我们需要使用另外两个方法getPerspectiveTransform()和warpPerspective()，仿射变换矩阵M变成了3*3矩阵。

In [7]:
```
'''
cv2.getPerspectiveTransform()   返回3*3的转变矩阵
        参数:
            src: 原图像中的四组坐标,
                如 np.float32([[56,65],[368,52],[28,387],[389,390]])
            dst: 转换后的对应四组坐标,
                如np.float32([[0,0],[300,0],[0,300],[300,300]])

cv2.warpPerspective()
    参数:
        src: 图像对象
        M: 3*3 transformation matrix (转变矩阵)
        dsize: 输出矩阵的大小, 注意格式为 (cols, rows)
                即width对应cols, height对应rows
        flags: 可选, 插值算法标识符, 有默认值INTER_LINEAR,
               如果插值算法为WARP_INVERSE_MAP, warpAffine函数使用如下矩阵进行图像转
               dst(x,y)=src(M11*x+M12*y+M13,M21*x+M22*y+M23)
        borderMode: 可选,  边界像素模式, 有默认值BORDER_CONSTANT
        borderValue:可选, 边界取值, 有默认值Scalar()即0
'''
# 我们将img图片的左上角和右上角往“里”缩一缩, 同时左下角和右下角位置不变。
# 原图的四组顶点坐标
pts3D1 = np.float32([[0, 0], [cols, 0], [0, rows], [cols, rows]])
# 转换后的四组坐标
pts3D2 = np.float32([[int(cols/5), int(rows/5)], [int(cols*4/5), int(rows/5)],
                     [0, int(rows*4/5)], [cols, int(rows*4/5)]])
# pts3D2 = np.float32([[10,10], [40,10], [0,50], [50,50]])
# 计算透视放射矩阵
M5 = cv2.getPerspectiveTransform(pts3D1, pts3D2)
# 执行变换
img5 = cv2.warpPerspective(img, M5, (cols, rows))
plt.imshow(img5)
```

Out[7]: <matplotlib.image.AxesImage at 0x1bd411e6950>


0
100
200
300
400
500
0
100
200
300
400
500