Part II Image Transformation

For this part you are required to write some functions for interpolation and transformation.

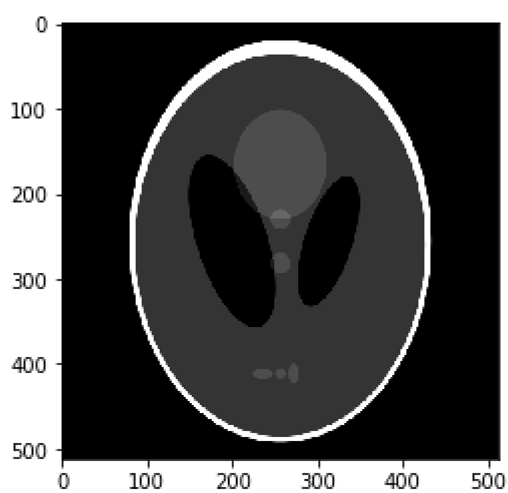
In [2]:
```
import numpy as np
from PIL import Image
from matplotlib import pyplot as plt
import math
```

Please load the example2.png.

In [3]:
```
# if you are using local jupyter notebook, please use the below codes to load im
img = Image.open('example2.png')
```

In [4]:
```
# change the image into a gray image
img = img.convert('L')
h,w = np.shape(img)
print('height:',h,' width: ',w)
plt.figure()
plt.imshow(img, cmap='gray')
plt.show()
```

```
height: 512  width:  512
```



Question1: Bilinear Interpolation

Here you need to implement a function for bilinear interpolation from scratch. Xq and Yq are arrays of coordinates of the points we want to interpolate.

For example, Xq=[0.5, 1.2], Yq=[0.8, 1.9] indicate that we want to interpolate the points (0.5, 0.8) and (1.2, 1.9).

The output should be a list of interpolation result.

In [5]:
```
def interp2(image, Xq, Yq):
  '''
  Write your own code here.
  '''
  interp_points=[]
  for i in range(len(Xq)):
    x=Xq[i]
    y=Yq[i]
    a=int(x//1)
    b=a+1
    c=int(y//1)
    d=c+1
    area=1
    interp_points.append(image[a][c]*(b-x)*(d-y)+image[b][c]*(x-a)*(d-y)+image[a
  return interp_points
# img1=np.array(img)
# print(interp2(img1,[0.5,120.5],[0.8,120.5]))
```

Question 2: Write a function that creates a 2D affine transformation matrix in homogenous and its inverse from a sequence of elementary transformations

The input is a list of operation name and its parameters. The operation is restricted to {rotation, shear, shift, scaling}.

For example, [('scaling', 1.2), ('shift', [10 20]), ('scaling', .2), ('rotation', 90)]

Your return should be the composed affine matrix, and its inverse.

In [6]:
```
from numpy.linalg import inv
import math

def get_affine_matrix(op_list):
  '''
  Write your own code here.
  '''
  affine_matrix=np.zeros([3,3])
  matrixlist=[]
  for element in op_list:
    temp_matrix=np.zeros([3,3])
    if element[0]=='scaling':
      scale=element[1]
      temp_matrix[0][0]=scale
      temp_matrix[1][1]=scale
      temp_matrix[2][2]=1
      matrixlist.append(temp_matrix)
    if element[0]=='shift':
      slice=element[1]
      shiftx=slice[0]
      shifty=slice[1]
      temp_matrix[0][0]=1
      temp_matrix[1][1]=1
      temp_matrix[2][2]=1
      temp_matrix[0][2]=shiftx
      temp_matrix[1][2]=shifty
      matrixlist.append(temp_matrix)
    if element[0]=='shear':
      shearx=element[1]
      sheary=element[2]
      temp_matrix[0][0]=1
      temp_matrix[1][1]=1
      temp_matrix[2][2]=1
      temp_matrix[0][1]=shearx
      temp_matrix[1][0]=sheary
      matrixlist.append(temp_matrix)
    if element[0]=='rotation':
      degree=element[1]
      rotationsin=math.sin(math.radians(degree))
      rotationcos=math.cos(math.radians(degree))
      temp_matrix[0][0]=rotationcos
      temp_matrix[1][1]=rotationcos
      temp_matrix[2][2]=1
      temp_matrix[0][1]=-rotationsin
      temp_matrix[1][0]=rotationsin
      matrixlist.append(temp_matrix)
  affine_matrix=matrixlist[0];
  for i in range(1,len(matrixlist)):
#     global affinematrix
    affine_matrix=np.matmul(matrixlist[i],affine_matrix)
  iaffine_matrix=np.linalg.inv(affine_matrix)
  return affine_matrix, iaffine_matrix
```

Question 3: Based on the below two functions, write a

code to achieve the operation of rotation and scaling.

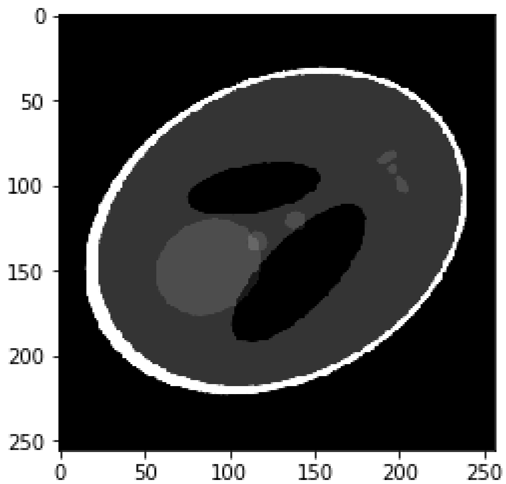
Here you need to write a transformation function which takes the input, affine matrix, iaffine matrix and new shape of your output image. We will compare your transformation result with the functions provided by PIL after a rotation and scaling.

The return should be a 2d matrix of of the result of transforming an image.

In [7]:
```
# write the code for transform function
def transform(img, affine, iaffine, new_shape):
  '''
  Write your own code here.
  '''
  h_new=int(new_shape[0])
  w_new=int(new_shape[1])
  h,w=np.shape(img)
  result=np.zeros([h_new,w_new])
  for i in range(h):
    for j in range(w):
      original=np.zeros([3,1])
      original[0][0]=i
      original[1][0]=j
      original[2][0]=1
      trans=np.matmul(affine,original)
      newx=int(trans[0][0])
      newy=int(trans[1][0])
      if newx<h_new and newy<w_new and newx>0 and newy>0:
        result[newx][newy]=img[i][j]
  return result
```

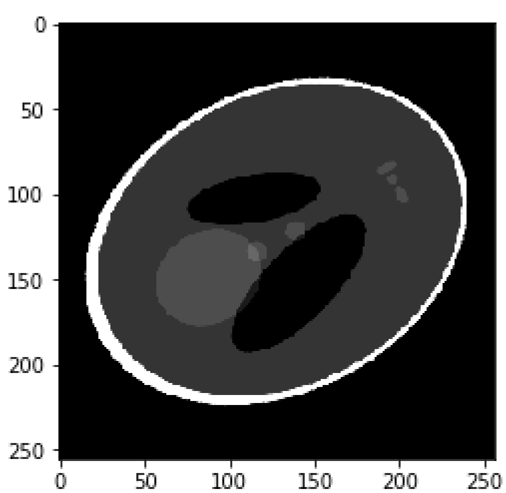
Now we will check if your result compared with the functions from PIL. You will get full credit if you can have a similar output.

In [8]:
```
# show the standard transformation result
theta = 118
scaling_rate = 0.5
standard_img = (img.rotate(theta)).resize((int(h*scaling_rate), int(w*scaling_ra
plt.figure()
plt.imshow(standard_img, cmap='gray')
plt.show()
```



In [9]:
```
# show your transformation result

# get the shape of the output
new_shape = (np.array(np.shape(img))*scaling_rate).astype(int)
h, w = np.shape(img)
# get the related affine matrix
# affine, iaffine = get_affine_matrix([('shift', [-w/2, -h/2]),('rotation',theta,
affine, iaffine = get_affine_matrix([('shift', [-w/2, -h/2]),('rotation',theta),
# transform the image
transfered_img = transform(np.array(img), affine, iaffine, new_shape)
plt.figure()
plt.imshow(transfered_img, cmap='gray')
plt.show()
```



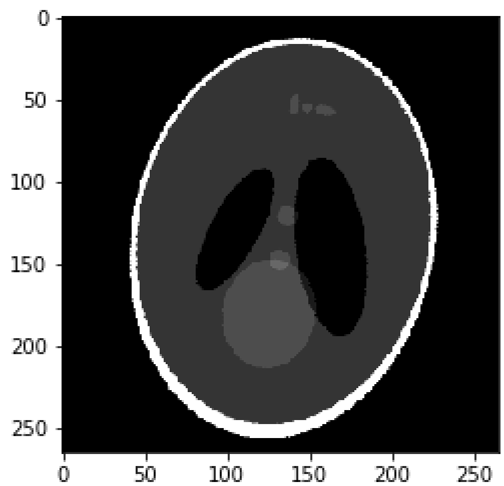
Bonus: Write a solver function that retrieves the affine transformation (in terms of a sequence of elementary transformations) between two provided images (depicting the same object transformed by an affine transformation). Justify your approach and comment on the limitations.

In [10]:
```
# generate the random transformation
random_theta = np.random.random()*180
random_scaling = np.random.random()*0.5 + 0.5
random_shift_x = (np.random.random()-0.5)*w*0.1
random_shift_y = (np.random.random()-0.5)*h*0.1

# get the random transformed image
random_shape = (np.array(np.shape(img))*random_scaling).astype(int)
# affine, iaffine = get_affine_matrix([('rotation', random_theta), ('scaling', r
affine, iaffine = get_affine_matrix([('shift', [-w/2, -h/2]),('rotation',random_
random_transformed_img = transform(np.array(img), affine, iaffine, random_shape)

# show the image
plt.figure()
plt.imshow(random_transformed_img, cmap='gray')
plt.show()

# show the affine matrix
print(affine)
```




```
[[-5.10814251e-01 -9.45373585e-02  2.89514409e+02]
 [ 9.45373585e-02 -5.10814251e-01  2.40612317e+02]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00]]
```

Now write your own code to get the affine matrix based on the original image and random transformed image. And give a description of your method and result.

In [49]:
```
import sys
def solver(img, random_transformed_img):
  w,h=np.shape(img)
  minval=sys.float_info.max
  scale=len(np.array(random_transformed_img))/len(np.array(img))
  for shiftXcount in range(0,10):
    shiftX=-25+shiftXcount*5
    for shiftYcount in range(0,10):
      shiftY=-25+shiftYcount*5
      for thetacount in range(0,6):
        theta=thetacount*30
        affine, iaffine=get_affine_matrix([('shift', [-w/2, -h/2]),('rotation',tl
        generated_img = transform(np.array(img), affine, iaffine, np.shape(randor
        mse = np.mean(((generated_img - random_transformed_img)/255)**2)
        if mse < minval:
          print(mse)
          good_affine=affine
  print(good_affine)
  return good_affine

solver(img,random_transformed_img)
```

```
0.07587787581767781
0.07475057907596128
0.06632656120294395
0.06838866197230327
0.07734314438582307
0.08263850278578123
0.07329669742543637
0.07374115008207403
0.07797818102383876
0.08032325601416548
0.07488372259813536
0.08098900472187423
0.07663482454443885
0.07541908598563221
0.07607599486956916
0.07774281803326458
0.07240532779722014
0.07953487144974912
0.07856747048982585
0.07861554028908836
```

my implementation is a brute force algorithm. We try all combinations of shifting, scaling, and rotation to find the best affine matrix. This affine matrix when multiplied with original image will give the smallest mse value. The limitation of this method is that we are going in steps. So the accuracy of approximation is dependent on the step size and CPU computation speed.