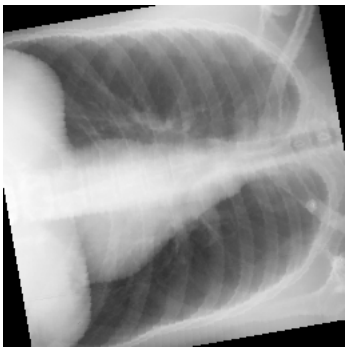
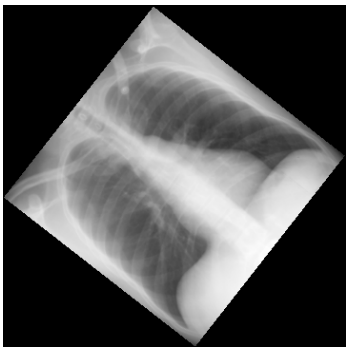
A. Athanassiadis

1 Problem 1

(a) Original



(b) Nearest-Neighbor Interpolation



(c) Linear Interpolation

Figure 1: Image Rotation

In this problem, I created a `rotate()` function to rotate the original image by an arbitrary angle, θ, given in degrees. The rotate function takes an input image, and maps its points to those in a new image by rotating a point in the new image by $-\theta$ and then interpolating the result. I allowed a few options, which included the interpolation scheme, and an image reshaping feature. Figure 1b demonstrates a rotation of $80°$ about the center with nearest-neighbor interpolation. Without the 'reshape' flag set for this rotation, the edges were cropped to preserve the 256×256 original shape. Figure 1c demonstrates a rotation of $-52°$ with the reshaping feature, which calculates the size of the new image based on the rotation angle. This image used linear interpolation, and resulted in a 360×360 image which was scaled above to fit on the page.

Comparing the two rotated images, the advantages of linear interpolation over nearest-neighbor are clear. Nearest-neighbor interpolation introduces pixelation, especially noticeable on diagonal edges, whereas linear interpolation minimizes these effects. Since the difference in computation time for these two is negligible, linear interpolation seems to be the better algorithm to use.

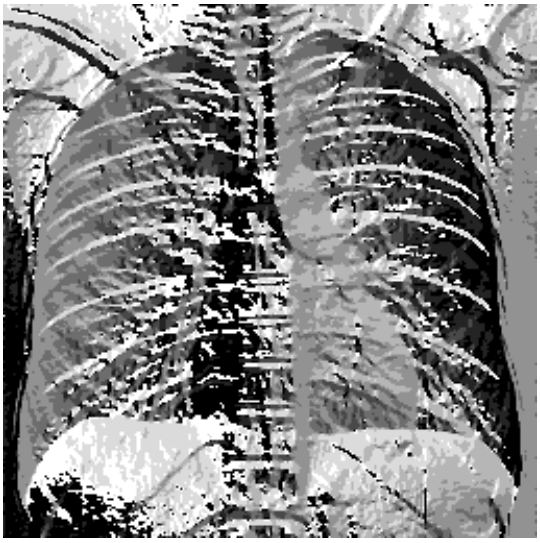
problem1.py

```
#Athanasios Athanassiadis Jan 2012
from preprocessing import *

im = load('figure_problem_set_2')

im_rot = rotate(im, 80)
im_rot2 = rotate(im, -52, interp='linear', reshape=True)

imsave('2-1.png', im)
imsave('2-1a.png',im_rot)
imsave('2-1b.png', im_rot2)
```

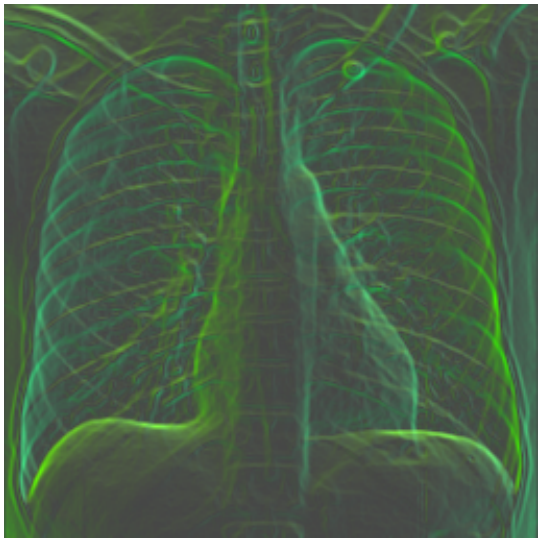
2 Problem 2

(a) Original

(b) Edge Magnitude



(c) 8-bit Edge Direction



(d) Edge Magnitude Contrast

Figure 2: Edge Calculations

Here, I used the Sobel filters, $h_1,\ h_2,\ h_3$ (Sonka, p.146) to perform edge detection on the original image. If x denotes the image response to h_1 and y the response to h_3, then the magnitude image (Figure 2b) was calculated using $I_M = \sqrt{x^2 + y^2}$. Figure 2d shows another rendering of the magnitude image, where the blue shades indicate where the h_1 response was strongest and the yellow shades represent where the h_3 response was strongest. Stronger green represents where both filters had equal response. The direction image (Figure 2c) was calculated using $I_D = \texttt{arctan2}(\frac{x}{y})$, where `arctan2()` is an implementation of the `arctan()` function that preserves quadrant information. The output from `arctan2()` was further processed, binning it by angle. The result is an octary image where each grayscale level represents a direction as seen in Figure 3.

3	2	1
4	×	0
5	6	7

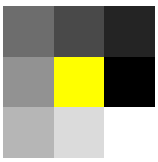


Figure 3: Direction Encoding

It is important to note that there is a loss of information when encoding in this way. At a point where the edge magnitude is zero, the direction image still contains a value between 0 and 8. Because there is no way to encode a pixel in the middle of a constant grey-value region, this information is lost in the encoding, and can only be determined if edge magnitude image is provided with the direction image. Also, the noise in edge direction is apparent in Figure 2d. This noise can be reduce by blurring the input image with a smoothing kernel, and then performing edge detection. However, blurring generally lowers the edge magnitude.

problem2.py

```
#Athanasios Athanassiadis Jan 2012
from preprocessing import *

im = load('figure_problem_set_2')
mags, dirs = edges(im, True)
im1,im2,im3 = edge_sobel(im)

imsave('2-2.png',im)
imsave('2-2a.png',mags)
imsave('2-2b.png',dirs)
imsave('2-2c.png',(im1*255./im1.max(),mags,im3*255./im3.max()))
```

3 Appendix: Common Code

Common functions used for these problems are contained in `preprocessing.py`.

preprocessing.py

```
#Athanasios Athanassiadis Jan 2012
import numpy as np
from scipy.misc import imsave
from scipy.ndimage import convolve
from ball import eight_dist

sin = np.sin
cos = np.cos
atan2 = np.arctan2

#####Image Loading#################################################################
def load(filename, shape=(256,256), d='>i2'):
    im = np.fromfile('figure_problem_set_2',dtype=d) #input file is big endian
        2-byte int
    im = im[64:]   #there seem to be 64*2 extra bytes in this file at the beginning
    im.shape = shape

    return im.astype(np.int16)

#####Interpolation Schemes#########################################################
```

```
20
21 #nearest neighbor interpolation
22 def interp_nearest(im, point):
23     i,j = point
24     l = np.around(i)
25     k = np.around(j)
26
27     #only return points that existed in the original image
28     if (0<=l<im.shape[0])*(0<=k<im.shape[1]):
29         return im[l,k]
30     else:
31         return 0
32
33 #linear interpolation
34 def interp_lin(im, point):
35     i,j = point
36     l,k = np.floor(i),np.floor(j)
37     a = (i-l)
38     b = (j-k)
39
40     #only return points that existed in the original image
41     if (0<=l<im.shape[0]-1)*(0<=k<im.shape[1]-1):
42         return (1-a)*(1-b)*im[l,k] + a*(1-b)*im[l+1,k] + (1-a)*b*im[l,k+1] +
               a*b*im[l+1,k+1]
43     else:
44         return 0
45
46 ischemes = {'nearest':interp_nearest, 'linear':interp_lin}
47
48 #####Linear Transforms##################################################################
49
50 #rotate an image by an angle theta (in degrees) counterclockwise
51 #about center using interp as the interpolation function
52 def rotate(im, theta, interp='nearest', reshape=False):
53     interpolate = ischemes[interp]
54     theta *= np.pi/180 #convert to rad from deg
55     shape = np.array(im.shape)
56     center = shape/2
57
58     # if desired set shape of new image so that no info gets lost in the rotation
59     if reshape:
60         d = eight_dist(shape)   #diameter of rotation
61         newshape = np.ceil(np.abs(shape*cos(theta))+np.abs(d*sin(theta)))
62     else:
63         newshape = shape
64     center2 = newshape/2
65     im_rot = np.zeros(newshape)
66
67     #inverse rotation matrix to map points from new im to old im
68     irotmat = np.array([[cos(theta), -sin(theta)],
69                         [sin(theta), cos(theta)]])
70
71     for i in range(im_rot.shape[0]):
72         for j in range(im_rot.shape[1]):
73             #find point in original image that corresponds to point in new image
74             #adjust for center of rotation and new size
```

```
            point = np.dot(irotmat,np.array([i,j])-center2)+center
            im_rot[i,j] = interpolate(im, point)

    return im_rot

#####Edge Processing#############################################################

#return response of image with 3x3 sobel operators
def edge_sobel(im):
    #define different filters
    h1 = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
    h2 = np.array([[0,1,2],[-1,0,1],[-2,-1,9]])
    h3 = h1[::-1].T

    im1 = 1.0*convolve(im,h1)
    im2 = 1.0*convolve(im,h2)
    im3 = 1.0*convolve(im,h3)

    return im1,im2,im3

#find edge magnitudes and directions using the sobel operator
#optionally encode direction data into octary
def edges(im, encode=False):
    im1,im2,im3 = edge_sobel(im)
    mags = np.sqrt((im1**2) + (im3**2))

    #get dirs by taking inverse tan
    #convert to degrees, and shift so that
    #discontinuity in theta occurs on positive x-axis
    dirs = atan2(im1,im3) * 180/np.pi + 45

    dirs2 = np.zeros(dirs.shape)


    #Create an octary image by separating the angles into directions according to
        the following matrix
    #  3  2  1
    #  4  X  0
    #  5  6  7
    #(split angles at 45deg increments)
    #effectively shift theta=0 by 22.5 degrees
    #so that all the angles pointing in direction 7 are counted together
    dirsplit = np.arange(-157.5,202.5+1, 45)
    dirs[dirs<-157.5] += 360
    for i in range(8):
        dirs2[(dirsplit[i]<=dirs)*(dirs<dirsplit[i+1])] = i

    if encode:
        return mags, dirs2
    else:
        return mags, dirs
```