Michael Christensen
November 24, 2013
CS 450

Homework 7

Written Exercises:

1. *Suppose that you are sampling a bandlimited signal whose highest frequency is 10 KHz. At what rate would you have to sample this signal assuming perfect reconstruction? At what rate would you have to sample it if the reconstruction filter was a low-pass filter with an imperfect cutoff at 10 KHz but allowed some frequencies up to 15 Khz?*

Assuming perfect reconstruction, the Nyquist rate states that the sampling rate must be twice the maximum/highest frequency. Since the highest frequency here is 10 Khz, the the sampling rate should be **20 Khz**.

If the reconstruction filter was a low-pass filter with an imperfect cutoff at 10 Khz but allowed some frequencies up to 15 Khz, the sampling rate would have to be **30 Khz**. This is because if this sampling rate was lower, it would cause the 15 Khz frequencies to merge with the lower ones and there not to be cleanly separated periods.

2. *Given the following image locations and values, calculate the value of the image at (2.7,4.8) using bilinear interpolation.*

Location	*Value*
(2, 4)	*7*
(2, 5)	*8*
(3, 4)	*8*
(3, 5)	*10*

```
v = ax + by + cxy + d

7 = a2 + b4 + c2*4 + d
8 = a2 + b5 + c2*5 + d
8 = a3 + b4 + c3*4 + d
10 = a3 + b5 + c3*5 + d
```

Using numpy's linalg module, these are the values for a through d: a= -3, b = -1, c = 1, d = 9. For verification:

```
7 = -3*2 + -1*4 + 1*8 + 9 = -6 – 4 + 8 + 9 = 7
8 = -3*2 + -1*5 + 1*10 + 9 = -6 – 5 + 10 + 9 = 8
8 = -3*3 + -1*4 + 1*12 + 9 = -9 – 4 + 12 + 9 = 8
10 = -3*3 + -1*5 +1*15 + 9 = -9 – 5 + 15 + 9 = 10
```

This is the code for this system:

```
a = np.array([[2,4,8,1],[2,5,10,1],[3,4,12,1],[3,5,15,1]])
b = np.array([7,8,8,10])
co = np.linalg.solve(a,b)
```

Thus, the value of the image at (2.7, 4.8) is:

```
v = -3 * 2.7 + -1 * 4.8 + 1 * (2.7 * 4.8) + 9
  = -8.1 – 4.8 + 12.96 + 9 = 9.06
```

3. *Use bilinear warping to produce a mapping from the quadrilateral (0,0), (1,0), (1.1, 1.1), (0,1) to the unit square.*

   ```
   v = ax + by + cxy + d
   ```

   One system of equations for x' and one for y':

   ```
   x':                                      quadr → unitsq
   0 = a*0 + b*0 + c*(0*0) + d              (0,0) → (0,0)
   1 = a*1 + b*0 + c*(1*0) + d              (1,0) → (1,0)
   1 = a*1.1 + b*1.1 + c*(1.1*1.1) + d      (1.1,1.1) → (1,1)
   0 = a*0 + b*1 + c*(0*1) + d              (0,1) → (0,1)

   y':                                      quadr → unitsq
   0 = a*0 + b*0 + c*(0*0) + d              (0,0) → (0,0)
   0 = a*1 + b*0 + c*(1*0) + d              (1,0) → (1,0)
   1 = a*1.1 + b*1.1 + c*(1.1*1.1) + d      (1.1,1.1) → (1,1)
   1 = a*0 + b*1 + c*(0*1) + d              (0,1) → (0,1)
   ```

Using numpy's linalg module,

For x', these are the values for a through d: a= 1, b = 0, c = -0.08264463, d = 0, and this is the code used to solve both systems of equations:

```
a = np.array([[0, 0, 0, 1], [1, 0, 0, 1], [1.1, 1.1, 1.1*1.1, 1], [0, 1, 0, 1]])
b = [0, 1, 1, 0]
co = np.linalg.solve(a,b)
```

For y', these are the values for a through d: a= 0, b = 1, c = -0.08264463, d = 0, and this is the code used to solve both systems of equations:

```
a = np.array([[0, 0, 0, 1], [1, 0, 0, 1], [1.1, 1.1, 1.1*1.1, 1], [0, 1, 0, 1]])
b = [0, 0, 1, 1]
co = np.linalg.solve(a,b)
```

Thus to get the x' mapping use

x' = x – 0.08264463(xy)

And to get the y' mapping use

y' = y – 0.08264463(xy)

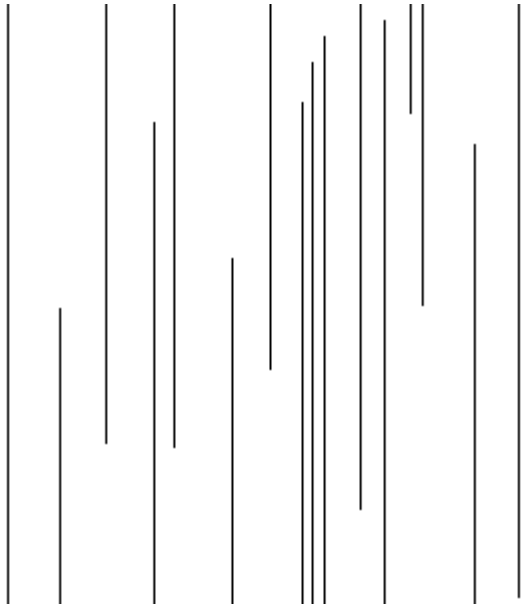
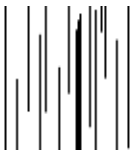
Programming Exercises:
Part A: Image Resizing (Magnification)
Code is found at the end of this document.
Original Parrots.png:

Parrots magnified by 2:

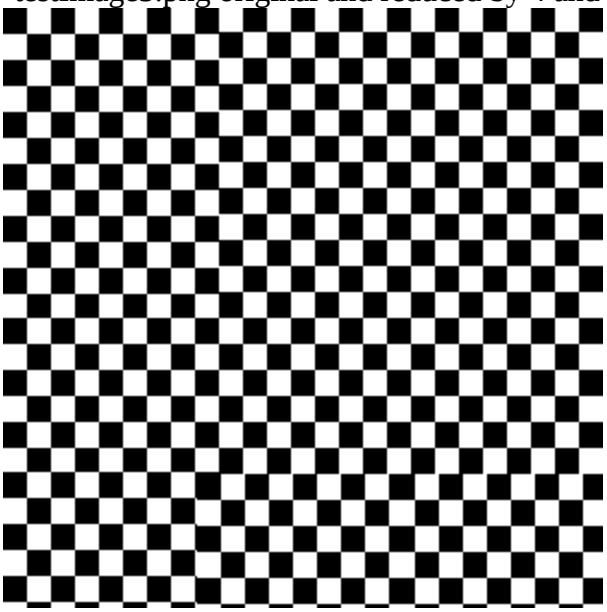
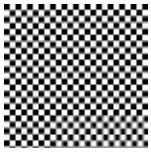
Parrots magnified by 3 (scaled to 87% of image's real size to fit on this page):

Part B: Image Resizing (Reduction);
Parrots reduced by 4 and 8:

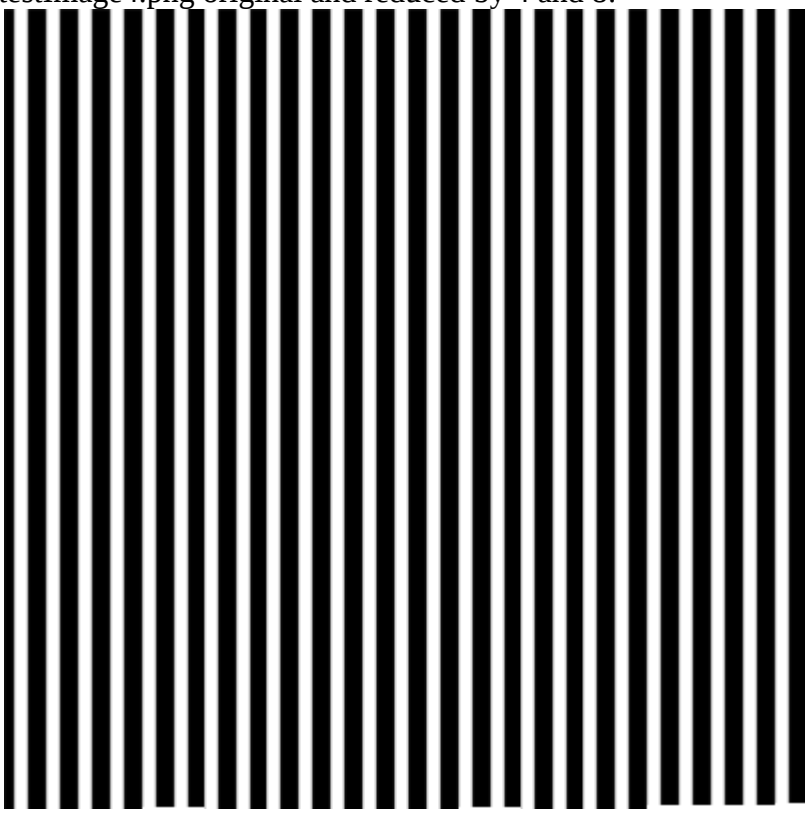
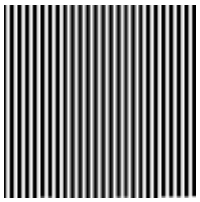
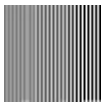

testImage2.png original and reduced by 4 and 8:


testImage3.png original and reduced by 4 and 8:

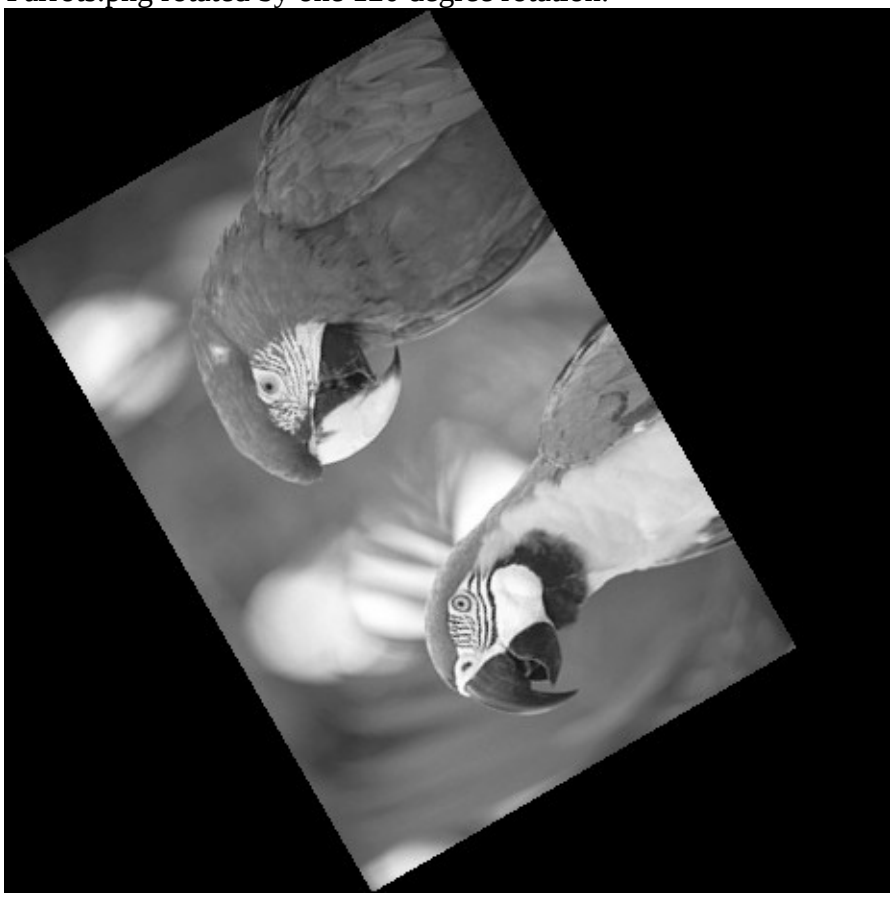




testImage4.png original and reduced by 4 and 8:

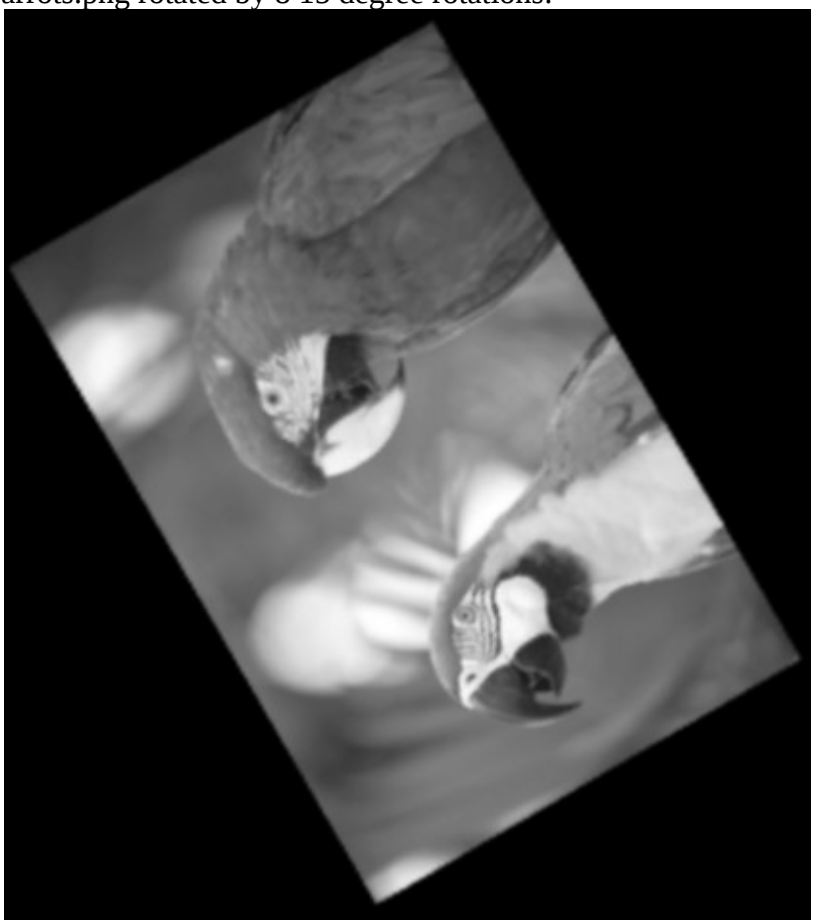




Is there anything you have to do differently from Part A in order to make this work well?
I took an average of the surrounding pixels in the source image, thinking that this would work because the destination image will have less discernible detail.

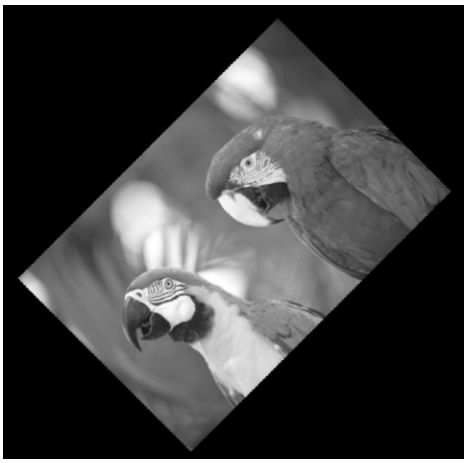
Part C: Image Warping:
Parrots.png rotated by one 120 degree rotation:

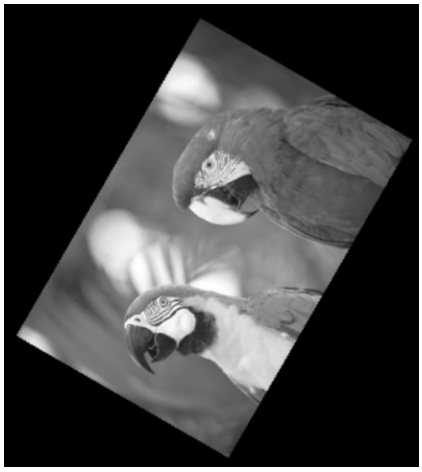


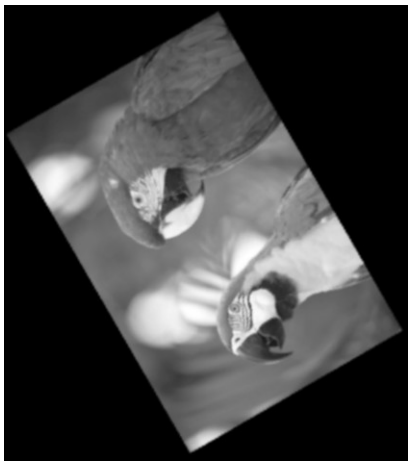
Parrots.png rotated by 8 15 degree rotations:



Here are the images as they looked after each intermediate rotation:







The final image caused by rotating the original by 8 15 degree individual rotations had a significant loss in detail, which I believe was caused by the fact that each rotation involved mapping pixels from the new image to the average of the surrounding pixels in the original image. Therefore, since each rotation involved creating a new image from a previous image created from averages, the averages compounded, until the final image was essentially composed of the average of 7 previous averages. The one 120 degree rotation, on the other hand, only did this backward mapping once, preserving greater detail.

Challenges:
My biggest challenge was with the image warping and figuring out how to do the individual rotations consecutively. The written part took less than an hour, and the programming part took 6 – 8 hours (I can't really remember accurately since I did this over a very wide period of time).

Code:

```
import matplotlib.pyplot as pl
import numpy as np
import math
import matplotlib.cm as cm

#Image resizing (Magnification)
def parta():

      #Read in an image
      img = pl.imread('parrots.png')

      #Magnify it by a specified (integer) factor
      #using bilinear interpolation
      magnifications = (2, 3)

      for mag in magnifications:
            newimg = np.zeros(np.multiply(img.shape, mag), img.dtype)

            for r in range(newimg.shape[0]):
                  for c in range(newimg.shape[1]):
                        #Backward Mapping

                        #Get inverse (where this current (r,c) pixel is
                        #located in the original image, which is probably not
                        #exactly on a pixel)
                        x, y = (r/float(mag), c/float(mag))
                        fracx = x - int(x)
                        fracy = y - int(y)

                        #Get the four closest points (upperleft, upperright,
bottomleft, bottomright)
                        #from the original image to interpolate from
                        p1 = (int(x), int(y))              #upperleft (closer to
zero)
                        p2 = (int(x), int(y) + 1)          #upperright
                        p3 = (int(x) + 1, int(y))          #bottomleft
                        p4 = (int(x) + 1, int(y) + 1) #bottomright


                        # Getting the original images surrounding values, checkign
bounds
                        old1 = img[p1]
                        old2 = None
                        if p2[1] == img.shape[1]:
                              old2 = img[p1]
                        else:
                              old2 = img[p2]
                        old3 = None
                        if p3[0] == img.shape[0]:
                              old3 = img[p1]
                        else:
                              old3 = img[p3]
                        old4 = None
                        if p4[0] == img.shape[0] or p4[1] == img.shape[1]:
                              old4 = img[p1]
                        else:
                              old4 = img[p4]
```

```
                        #Interpolate (using the unit-square approach from wikipedia
and www.cse.unr.edu/~looney/cs674/mgx6/unit6.pdf)
                        val = old1*(1-fracx)*(1-fracy) + old3*fracx*(1-fracy) +
old2*(1-fracx)*fracy + old4*fracx*fracy
                        newimg[r,c] = val

            pl.imsave("ParrotsMagnifiedBy%d"%(mag),newimg, cmap=cm.Greys_r)

# Image Resizing (Reduction)
def partb():
      images = ('parrots.png', 'testImage2.png', 'testImage3.png',
'testImage4.png')

      for imgname in images:
            #Read in an image
            img = pl.imread(imgname)

            #Reduce it by a specified (integer) factor
            reductions = (4, 8)

            for red in reductions:
                  newimg = np.zeros(np.multiply(img.shape, 1/float(red)),
img.dtype)

                  for r in range(newimg.shape[0]):
                        for c in range(newimg.shape[1]):

                              val = get_average((r*red,c*red), red, img)
                              newimg[r,c] = val

                  pl.imsave("%sReducedBy%d.png"%(imgname, red), newimg,
cmap=cm.Greys_r)


def get_average(point, reduction, img):
      #Get average from img in this layout:
      #[(r,c), ........, (r,c+red-1)]
      #[.........................]
      #[(r+red-1),,(r+red-1,c+red-1)]

      mat = np.zeros(img.shape)
      mat[point[0]:point[0]+reduction, point[1]:point[1]+reduction] =
np.ones((reduction,reduction))
      res = np.multiply(mat, img)
      res = res.sum()/float(reduction**2)
      return res


# Image Warping
def partc():
      images = ['parrots.png']

      ######################
      #1 120 degree increment
      #Read in an image
      img = pl.imread('parrots.png')
```

```
      #img.shape[0] --> height
      #img.shape[1] --> length
      origcorners = [(0, 0),
                           (0, img.shape[1]),
                           (img.shape[0], 0),
                           (img.shape[0], img.shape[1])]
      largestdist = np.sqrt(img.shape[0]**2 + img.shape[1]**2)
      newimg = np.zeros((largestdist, largestdist))
      midpoint = np.divide(img.shape,2)
      angle = 120.0
      newcorners = []
      for corn in origcorners:
            x = corn[0]
            y = corn[1]
            newcorner = ((x-midpoint[0])*math.cos(math.radians(angle)) -
(y-midpoint[1])*math.sin(math.radians(angle)) + midpoint[0],
                              (x-midpoint[0])*math.sin(math.radians(angle)) +
(y-midpoint[1])*math.cos(math.radians(angle)) + midpoint[1])
            newcorners.append(newcorner)
      #print newcorners

      xs = map(lambda(x): x[0], newcorners)
      ys = map(lambda(x): x[1], newcorners)
      minx = int(np.min(xs))
      maxx = int(np.max(xs))
      miny = int(np.min(ys))
      maxy = int(np.max(ys))

      for x in range(minx, maxx):
            for y in range(miny, maxy):
                  u = (x-midpoint[0])*math.cos(math.radians(-angle)) -
(y-midpoint[1])*math.sin(math.radians(-angle)) + midpoint[0]
                  v = (x-midpoint[0])*math.sin(math.radians(-angle)) +
(y-midpoint[1])*math.cos(math.radians(-angle)) + midpoint[1]

                  #Get the four closest points
                  #from the original image to interpolate from
                  p1 = (int(u), int(v))              #upperleft (closer to zero)
                  p2 = (int(u), int(v) + 1)          #upperright
                  p3 = (int(u) + 1, int(v))          #bottomleft
                  p4 = (int(u) + 1, int(v) + 1) #bottomright

                  val = 0.0
                  if u < 0 or u >= img.shape[0] - 1 or v < 0 or v >= img.shape[1] -
1:
                        val = 0.0
                  else:
                        # Getting the original images surrounding values
                        old1 = img[p1]
                        old2 = img[p2]
                        old3 = img[p3]
                        old4 = img[p4]

                        #Interpolate
                        b = np.array([old1, old2, old3, old4])
                        a = np.array([[p1[0], p1[1], p1[0]*p1[1], 1], [p2[0],
p2[1], p2[0]*p2[1], 1],
                                                [p3[0], p3[1], p3[0]*p3[1], 1], [p4[0],
```

```
p4[1], p4[0]*p4[1], 1]])
                        coef = np.linalg.solve(a, b)
                        val = coef.dot(np.array([u, v, u*v, 1]))

                  r = x + (0 - minx)
                  c = y + (0 - miny)
                  newimg[r,c] = val

      pl.imsave("ParrotsRotatedOnly120.png", newimg, cmap=cm.Greys_r)


      ########################
      #8 15 degree increments
      #Note: this code isn't perfectly efficient, as it continually
      #increases the size of the image because of the poor way in which I
      #calculate the needed size of the next image that will result after rotation
      #I manually cropped out the surrounding black parts of the huge images that
      #resulted for report.
      img = pl.imread('parrots.png')
      for n in range(8):
            print n
            #img.shape[0] --> height
            #img.shape[1] --> length
            origcorners = [(0, 0),
                           (0, img.shape[1]),
                           (img.shape[0], 0),
                           (img.shape[0], img.shape[1])]
            largestdist = np.sqrt(img.shape[0]**2 + img.shape[1]**2)
            newimg = np.zeros((largestdist, largestdist))
            midpoint = np.divide(img.shape,2)
            angle = 15.0
            newcorners = []
            for corn in origcorners:
                  x = corn[0]
                  y = corn[1]
                  newcorner = ((x-midpoint[0])*math.cos(math.radians(angle)) -
(y-midpoint[1])*math.sin(math.radians(angle)) + midpoint[0],
                               (x-midpoint[0])*math.sin(math.radians(angle)) +
(y-midpoint[1])*math.cos(math.radians(angle)) + midpoint[1])
                  newcorners.append(newcorner)
            #print newcorners

            xs = map(lambda(x): x[0], newcorners)
            ys = map(lambda(x): x[1], newcorners)
            minx = int(np.min(xs))
            maxx = int(np.max(xs))
            miny = int(np.min(ys))
            maxy = int(np.max(ys))

            for x in range(minx, maxx):
                  for y in range(miny, maxy):
                        u = (x-midpoint[0])*math.cos(math.radians(-angle)) -
(y-midpoint[1])*math.sin(math.radians(-angle)) + midpoint[0]
                        v = (x-midpoint[0])*math.sin(math.radians(-angle)) +
(y-midpoint[1])*math.cos(math.radians(-angle)) + midpoint[1]

                        #Get the four closest points
                        #from the original image to interpolate from
```

```
                        p1 = (int(u), int(v))               #upperleft (closer to
zero)
                        p2 = (int(u), int(v) + 1)           #upperright
                        p3 = (int(u) + 1, int(v))           #bottomleft
                        p4 = (int(u) + 1, int(v) + 1)#bottomright

                        val = 0.0
                        if u < 0 or u >= img.shape[0] - 1 or v < 0 or v >=
img.shape[1] - 1:
                              val = 0.0
                        else:
                              # Getting the original images surrounding values
                              old1 = img[p1]
                              old2 = img[p2]
                              old3 = img[p3]
                              old4 = img[p4]

                              #Interpolate
                              b = np.array([old1, old2, old3, old4])
                              a = np.array([[p1[0], p1[1], p1[0]*p1[1], 1], [p2[0],
p2[1], p2[0]*p2[1], 1],
                                                      [p3[0], p3[1], p3[0]*p3[1], 1],
[p4[0], p4[1], p4[0]*p4[1], 1]])
                              coef = np.linalg.solve(a, b)
                              val = coef.dot(np.array([u, v, u*v, 1]))

                        r = x + (0 - minx)
                        c = y + (0 - miny)
                        newimg[r,c] = val
            img = newimg

            #pl.imshow(img, cmap=cm.Greys_r)
            pl.show()
            pl.imsave("ParrotsRotated%d.png"%((n+1)*15), img, cmap=cm.Greys_r)


def run():
      parta()
      partb()
      partc()


if __name__ == "__main__":
      run()
```