Michael Christensen
December 14, 2013
CS 465

Homework #8 – Compression

Written Exercises:

1. *Use the linked page of English letter frequencies to determine the entropy of this set. What would the entropy be if each letter occurred with the same frequency (probability)?*

Using the formula **H = -** ∑ (from i = 1 to n) $p(a_i)\log_2 p(a_i)$ with the Python script:

```
def written():
    #Entropy for relative frequencies
    relative_frequencies = [0.08167, 0.01492, 0.02782, 0.04253, 0.12702,
                            0.02228, 0.02015, 0.06094, 0.06966, 0.00153,
                            0.00772, 0.04025, 0.02406, 0.06749, 0.07507,
                            0.01929, 0.00095, 0.05987, 0.06327, 0.09056,
                            0.02758, 0.00978, 0.02360, 0.00150, 0.01974,
                            0.00074]

    sigma = 0.00
    for freq in relative_frequencies:
        sigma += freq * np.log2(freq)
    sigma = -sigma
    print "Entropy for English language with relative frequencies: %f" % sigma

    #Entropy if each letter occurred with same frequency
    sigma = 0.00
    freq = 1.0/26.0
    for a in range(26):
        sigma += freq * np.log2(freq)
    sigma = -sigma
    print "Entropy for English language with equal frequencies: %f" % sigma
```

Entropy for English language with relative frequencies: **4.175760 .**
Entropy for English language with equal frequencies: **4.700440**.

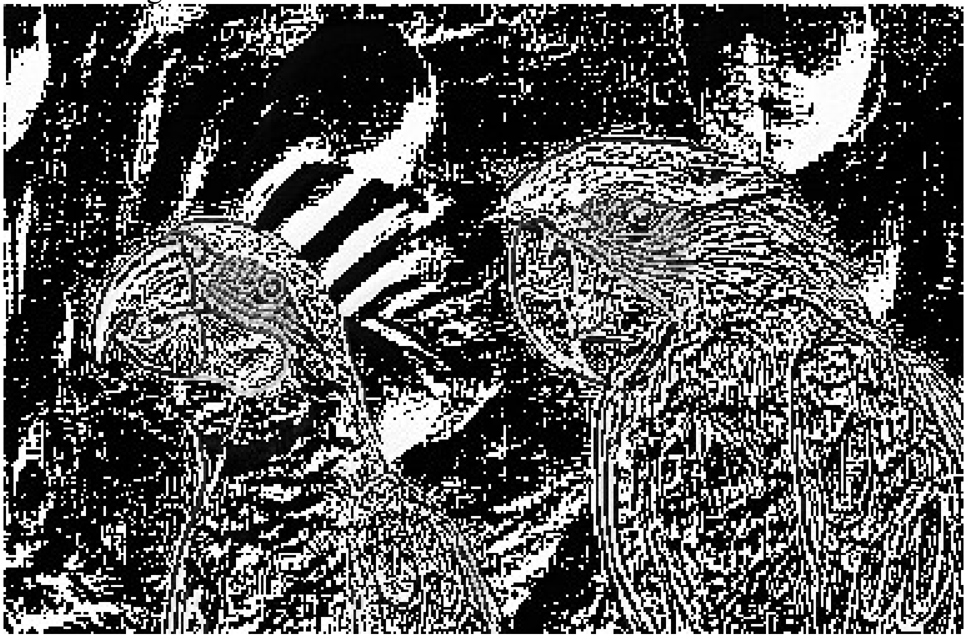
Programming Exercises:

1. Using parrots.tif file, the size of the original file is 295 kB. The size of the original file after being compressed by the unix compact command is 271.4 kB.

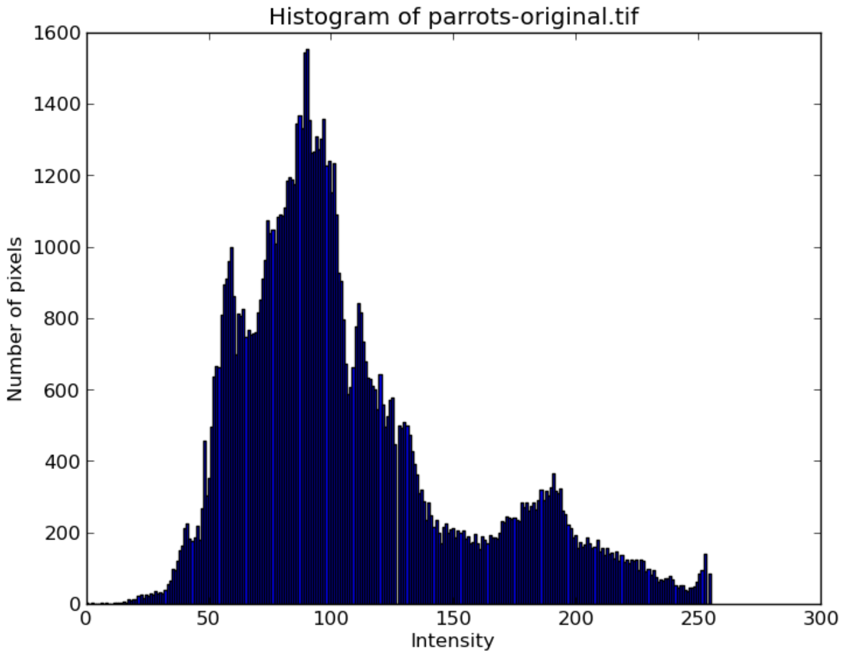
2. *Code is attached at the end of this document*.

Original image:

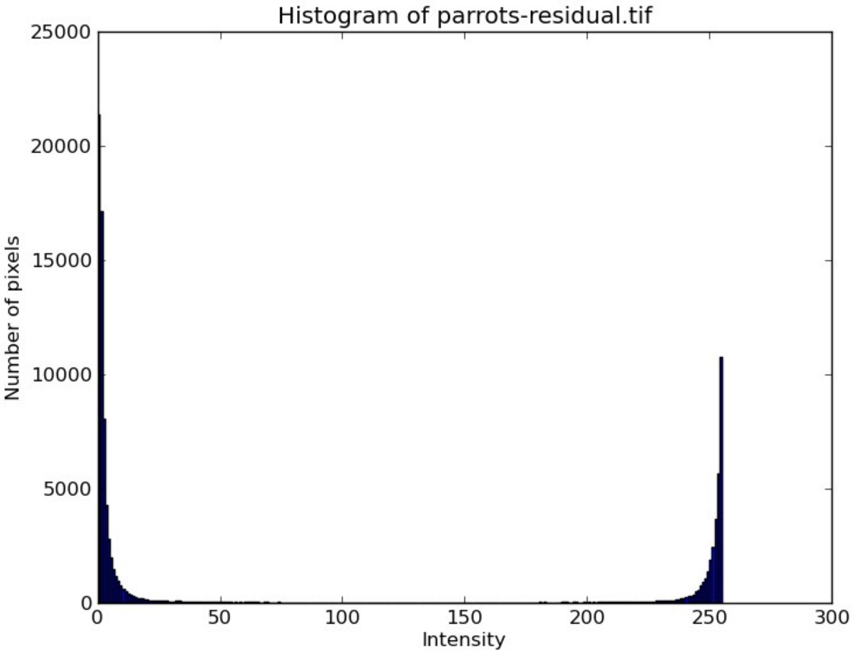
Predictively-encoded image:



3.
Histogram of original image before encoding:



Histogram of predictively-encoded image:



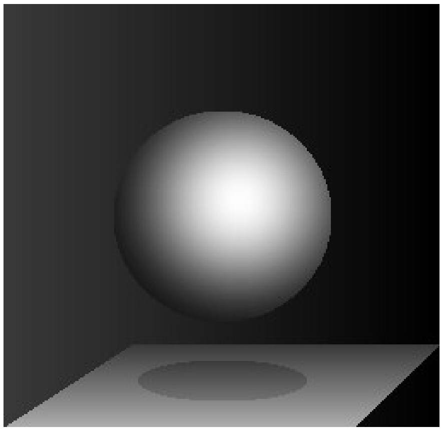
The dynamic range of the original image is larger than the predictively-encoded image. The predictively-encoded image has the majority of the pixels in the lower and upper ends of the spectrum (around 0 and 255, which makes sense since it is encoding change). This residual image is the better candidate for entropy encoding since it needs less data to convey the same amount of information that the more dynamic original image does.

4. The size of the predictively-encoded file is 295 kB, and the size of the compressed predictively-encoded file is 156.2 kB.

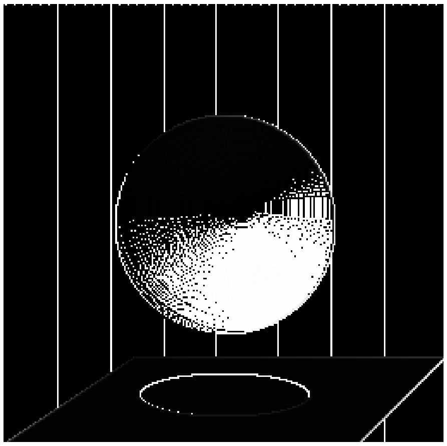
5-6. Image after decompressing it with the unix compact -d command and using my own predictive decoder to reassemble the original image:

7. Five other images:

Original ball.tif

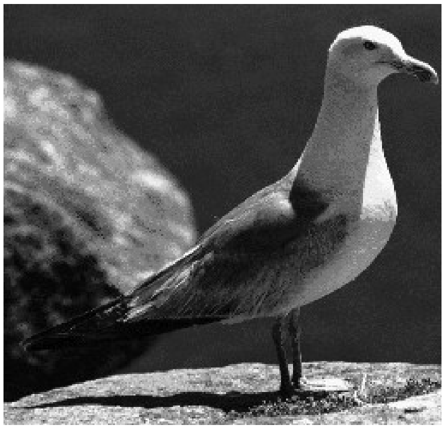


Residual ball.tif

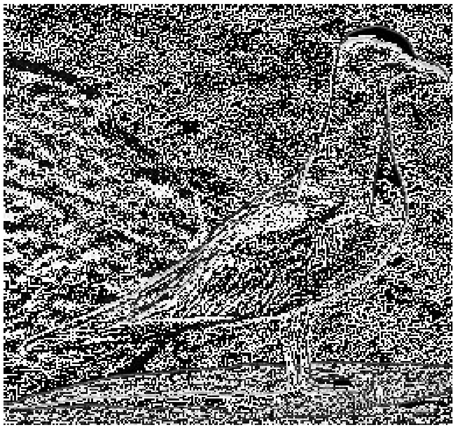


Size of original image: 197 kB
Size of compressed original image: 168.7 kB
Size of compressed residual image: 58 kB

Original gull.tif

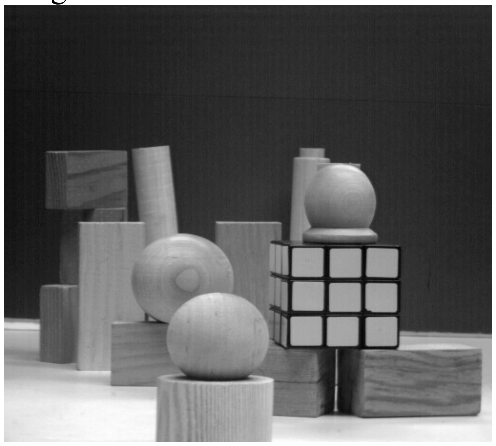


Residual gull.tif

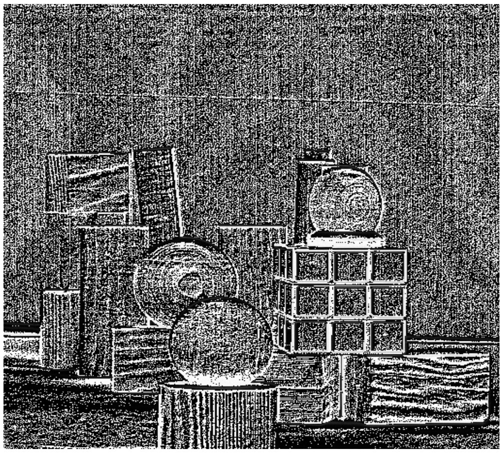


Size of original image: 197 kB
Size of compressed original image: 140.5 kB
Size of compressed residual image; 135.3 kB

Original blocks.tif

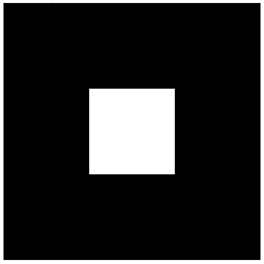


Residual blocks.tif

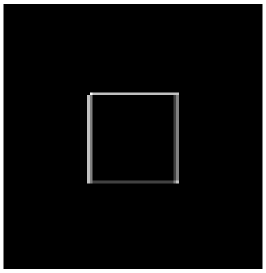


Size of original image: 768.8 kB
Size of compressed original image: 565.9 kB
Size of compressed residual image: 406.8 kB

Original whitebox.tif



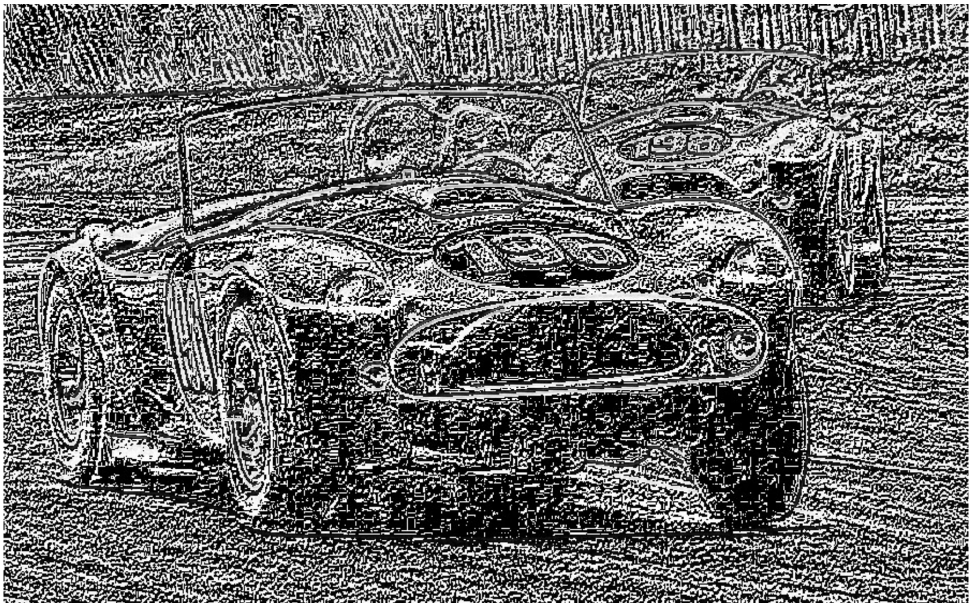
Residual whitebox.tif



Size of the original image: 28 kB
Size of compressed original image: 4.1 kB
Size of compressed residual image: 3.7 kB

Original cobra.tif

Residual cobra.tif



Size of the original image: 1.5 MB
Size of compressed original image: 1.4 MB
Size of compressed residual image: 948.8 kB

Brief explanation of what I did:
I implemented the predictive encoding scheme as described in the book and specified in the assignment, predictiving pixels based on the four nearby pixels preceding them in the image. To handle the case of recording negative residuals, I put converted those values so that they would fit within the [0, 255] range the image required and used the reverse process to understand those values when decoding and convert them back to their negative form.

Challenges;
The largest challenge I had was understanding the algorithm completely. Once I understood it, implementing it was a fairly straightforward process. One bug I did have to fix in the end was that I wasn't properly flooring my predictions, so that it was left to python to determine what to do with predictions that were floating point numbers instead of integers. Once I made sure to consistently floor each prediction, it worked well.

Time for different parts:
Written: 15 minutes
Programming: 5-6 hours

Code:
```
import numpy as np
import matplotlib.pyplot as pl
import matplotlib.cm as cm
import scipy.misc as sp
import math

def written():
    #Entropy for relative frequencies
    relative_frequencies = [0.08167, 0.01492, 0.02782, 0.04253, 0.12702,
                            0.02228, 0.02015, 0.06094, 0.06966, 0.00153,
                            0.00772, 0.04025, 0.02406, 0.06749, 0.07507,
                            0.01929, 0.00095, 0.05987, 0.06327, 0.09056,
                            0.02758, 0.00978, 0.02360, 0.00150, 0.01974,
                            0.00074]

    #For debugging (should equal 1.00 or 0.99)
    #
    # total = 0.0
    # for freq in relative_frequencies:
    #     total += freq
    # print total

    sigma = 0.00
    for freq in relative_frequencies:
        sigma += freq * np.log2(freq)
    sigma = -sigma
    print "Entropy for English language with relative frequencies: %f" % sigma

    #Entropy if each letter occurred with same frequency
    sigma = 0.00
    freq = 1.0/26.0
    for a in range(26):
        sigma += freq * np.log2(freq)
    sigma = -sigma
    print "Entropy for English language with equal frequencies: %f" % sigma


def programming():
    files = [('whitebox-original.tif', 'whitebox-residual.tif')
             ,('cobra-original.tif', 'cobra-residual.tif')
             ,('parrots-original.tif', 'parrots-residual.tif')
             ,('ball-original.tif', 'ball-residual.tif')
             ,('blocks-original.tif', 'blocks-residual.tif')
             ,('gull-original.tif', 'gull-residual.tif')
             ]
    for f in files:
        displayHist(f[0])
        encode(f[0])
        displayHist(f[1])
        decode(f[1])    # to test my decoder works


def encode(filename):
    img = sp.imread(filename)
    img = img[:,:,]
    residimg = np.zeros(img.shape, img.dtype)

    #Use predictive coding to reduce the entropy of the image (input is single
gray-scale
    #and output is a single greyscale image as output)
    #Each pixel is predicted as weighted average of adjacent 4 pixels in raster
order
    #And calculate and encode the residual
    for r in range(img.shape[0]):
        for c in range(img.shape[1]):
            prediction = predictor(r, c, img)
            actual = img[r][c]
            residual = actual - prediction  # prediction error
            residimg[r][c] = convert_enc_residual(residual)
    sp.imsave("%s-residual.tif" % filename.split("-")[0], residimg)


# range that residual will be in is [-p, 255-p] (just shifting [0, 255] over by p)
def convert_enc_residual(residual):
    r = residual % 256 # if residual == -3, r = 253 # store in section above 255-P
and 255
    return r

def decode(residualsfile):
    rimg = sp.imread(residualsfile)
    rimg = rimg[:,:,1]
    origimg = np.zeros(rimg.shape, rimg.dtype)

    for r in range(rimg.shape[0]):
        for c in range(rimg.shape[1]):
            resid = rimg[r][c]
            prediction = predictor(r, c, origimg)
            residual = convert_dec_residual(resid, prediction)
            original = residual + prediction
            origimg[r][c] = original

    sp.imsave("%s-decoded.tif" % residualsfile.split("-")[0], origimg)

def convert_dec_residual(residual, prediction):
    if (255 - prediction) < residual:   # only occurs when residual was negative
in the encoding
        residual = residual - 256
    return residual


def predictor(r, c, img):
    surrounding_pixels = None
    predicted = 0

    # 3 pixels above, 1 directly to left
    # Cover edge cases
    if r == 0:                      # Top row
        if c == 0:                  # First column of top row
            predicted = 0#img[r][c]
        else:
            predicted = img[r][c-1]
    else:                           # Any other row
        if c == 0:                  # First column of any other row (get above two 
pixels)
            surrounding_pixels = [img[r-1][c], img[r-1][c+1]]
        elif c == img.shape[1]-1:   # Last column of any other row (get three 
surrounding pixels)
            surrounding_pixels = [img[r][c-1], img[r-1][c-1], img[r-1][c]]
        else:                       # Any other pixel (get four pixels in raster 
position)
            surrounding_pixels = [img[r][c-1], img[r-1][c-1], img[r-1][c], 
img[r-1][c+1]]
        predicted = math.floor(np.average(surrounding_pixels))

    return predicted


def displayHist(filename):
    img = sp.imread(filename)
    img = img[:,:,1]
    pl.hist(img.flatten(), bins=256)
    pl.title('Histogram of %s' % filename)
    pl.xlabel('Intensity')
    pl.ylabel('Number of pixels')
    pl.show()


def main():
    written()
    programming()

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