```
>> xn = imnoise(x,'gaussian');
```
MATLAB/Octave

or

```
In :  xn = skimage.util.random_noise(x)
```
Python

View your image, and attempt to remove the noise with

(a) Average filtering on each RGB component
(b) Wiener filtering on each RGB component

- 11. Take any color image of your choice and add salt and pepper noise to the intensity component. This can be done with

```
>> ty = rgb2ntsc(tw);
>> tn = imnoise(ty(:,:,1).'salt & pepper');
>> ty(:,:,1) = tn;
```
MATLAB/Octave

or

```
In :  rgb2yiq = np.array
    ([[.299,.587,0.114],[.596,-.275,-.321],[.212,-.528,.311]])
In :  yiq2rgb = np.linalg.inv(rgb2yiq)
In :  ty = ut.img_as_float(tw).dot(rgb2yiq)
In :  tn = ut.random_noise(ty[:,:,0],'s&p')
In :  ty[:,:,0] = tn
In :  t2 = ty.dot(yiq2rgb)
```
Python

Now convert back to RGB for display.

(a) Compare the appearance of this noise with salt and pepper noise applied to each RGB component as shown in Figure 13.21. Is there any observable difference?
(b) Denoise the image by applying a median filter to the intensity component.
(c) Now apply the median filter to each of the RGB components.
(d) Which one gives the best results?
(e) Experiment with larger amounts of noise.
(f) Experiment with Gaussian noise.

# 14 Image Coding and Compression

## 14.1 Lossless and Lossy Compression

We have seen that image files can be very large. It is thus important for reasons both of storage and file transfer to make these file sizes smaller, if possible. In Section 1.9 we touched briefly on the topic of compression; in this section, we investigate some standard compression methods. It will be necessary to distinguish between two different classes of compression methods: *lossless compression*, where all the information is retained, and *lossy compression* where some information is lost.

Lossless compression is preferred for images of legal, scientific, or political significance, where loss of data, even of apparent insignificance, could have considerable consequences. Unfortunately this style tends not to lead to high compression ratios. However, lossless compression is used as part of many standard image formats.

## 14.2 Huffman Coding

The idea of Huffman coding is simple. Rather than using a fixed length code (8 bits) to represent the gray values in an image, we use a variable length code, with smaller length codes corresponding to more probable gray values.

A small example will make this clear. Suppose we have a 2-bit grayscale image with only four gray levels: 0, 1, 2, 3, with the probabilities 0.2, 0.4, 0.3, and 0.1, respectively. That is, 20% of pixels in the image have gray value 50; 40% have gray value 100, and so on. The following table shows fixed length and variable length codes for this image:

| Gray Value | Probability | Fixed Code | Variable Code |
|---|---|---|---|
| 0 | 0.2 | 00 | 000 |
| 1 | 0.4 | 01 | 1 |
| 2 | 0.3 | 10 | 01 |
| 3 | 0.1 | 11 | 001 |

Now consider how this image has been compressed. Each gray value has its own unique identifying code. The average number of bits per pixel can be easily calculated as the expected value (in a probabilistic sense):

$$(0.2 \times 3) + (0.4 \times 1) + (0.3 \times 2) + (0.1 \times 3) = 1.9.$$

Notice that the longest codewords are associated with the lowest probabilities. This average is indeed smaller than 2.

This can be made more precise by the notion of *entropy*, which is a measure of the amount of information. Specifically, the entropy $H$ of an image is the theoretical minimum number of bits per pixel required to encode the image with no loss of information. It is defined by

| Gray Value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Probability | 0.19 | 0.25 | 0.21 | 0.16 | 0.08 | 0.06 | 0.03 | 0.02 |

where the index $i$ is taken over all grayscales of the image, and $p_i$ is the probability of gray level $i$ occurring in the image. Very good accounts of the basics of information theory and entropy are given by Roman [39] and Welsh [55]. In the example given above,

$$H = -\Big(0.2\log_2(0.2) + 0.4\log_2(0.4) + 0.3\log_2(0.3) + 0.1\log_2(0.1)\Big) = 1.8464.$$

This means that no matter what coding scheme is used, it will never use less than 1.8464 bits per pixel. On this basis, the Huffman coding scheme given above, giving an average number of bits per pixel much closer to this theoretical minimum than 2, provides a very good result.

To obtain the Huffman code for a given image we proceed as follows:

1. Determine the probabilities of each gray value in the image.
2. Form a binary tree by adding probabilities two at a time, always taking the two lowest available values.
3. Now assign 0 and 1 arbitrarily to each branch of the tree from its apex.
4. Read the codes from the top down.

To see how this works, consider the example of a 3-bit grayscale image (so the gray values are 0–7) with the following probabilities:

| Gray Value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Probability | 0.19 | 0.25 | 0.21 | 0.16 | 0.08 | 0.06 | 0.03 | 0.02 |

For these probabilities, the entropy can be calculated to be 2.6508. We can now combine probabilities two at a time as shown in Figure 14.1.

**Figure 14.1: Forming the Huffman code tree**

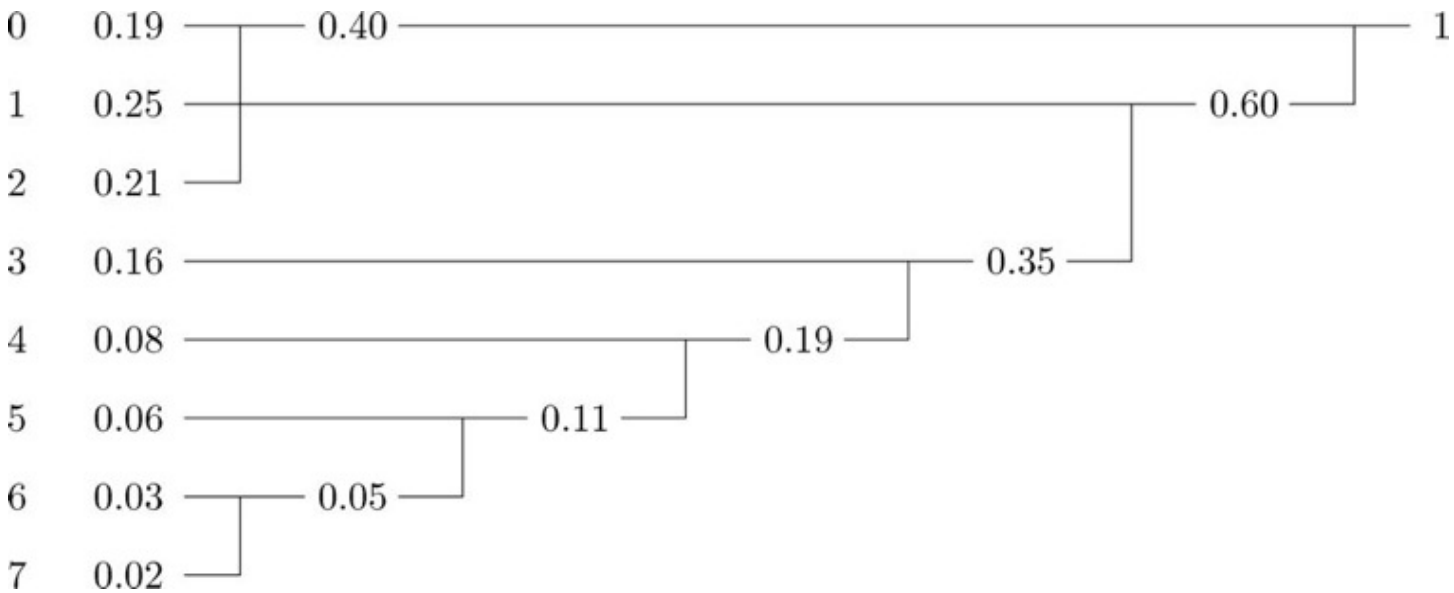
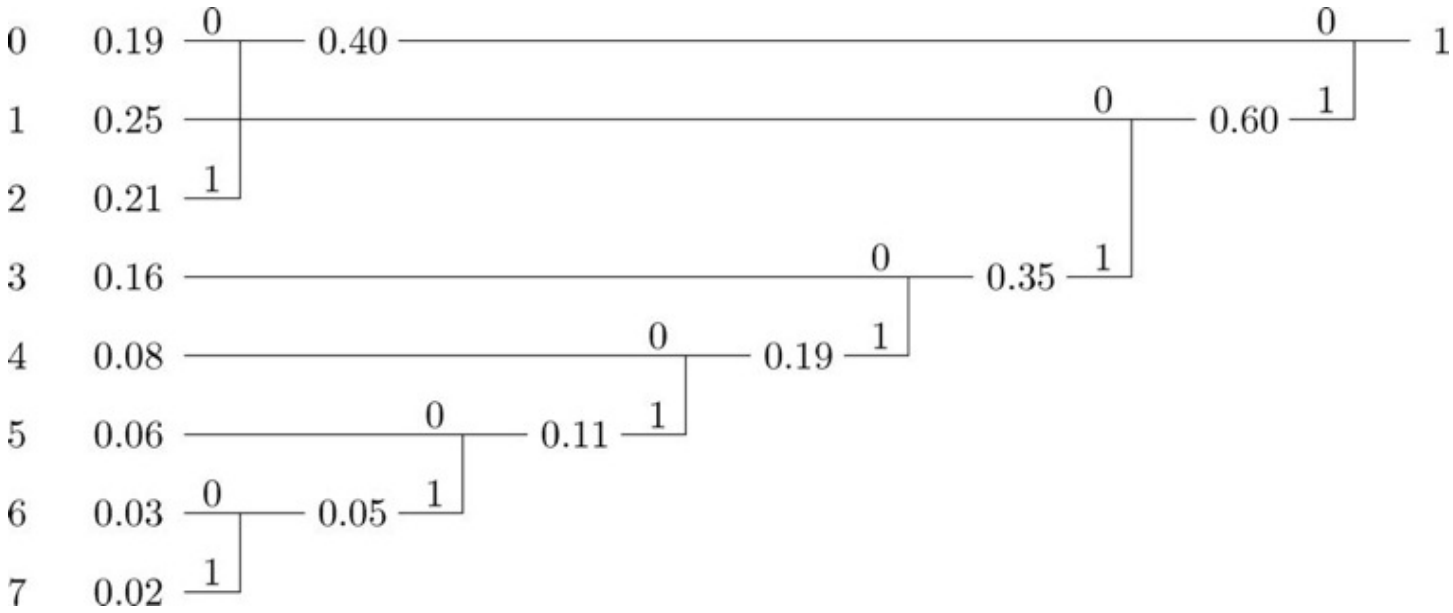


**Figure 14.2: Assigning 0's and 1's to the branches**




Note that if we have a choice of probabilities we choose arbitrarily. The second stage consists of arbitrarily assigning 0's and 1's to each branch of the tree just obtained. This is shown in Figure 14.2.

To obtain the codes for each gray value, start at the 1 on the top right, and work back toward the gray value in question, listing the numbers passed on the way. This produces:

| Gray Value | Huffman Code |
|---|---|
| 0 | 00 |
| 1 | 10 |
| 2 | 01 |
| 3 | 110 |
| 4 | 1110 |
| 5 | 11110 |
| 6 | 111110 |
| h 7 | 111111 |

As above, we can evaluate the average number of bits per pixel as an expected value:

$$(0.19 \times 2) + (0,25 \times 2) + (0.21 \times 2) + (0.16 \times 3) + \\ (0.08 \times 4) + (0.06 \times 5) + (0.03 \times 6) + (0.02 \times 6) = 2.7$$

which is a significant improvement over 3 bits per pixel, and very close to the theoretical minimum of 2.6508 given by the entropy.

Huffman codes are *uniquely decodable*, in that a string can be decoded in only one way. For example, consider the string

$$1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0$$

to be decoded with the Huffman code generated above. There is no code word 1, or 11, so we may take the first three bits 110 as being the code for gray value 3. Notice also that no other code word begins with this string. For the next few bits, 1110 is a code word; no other begins with this string, and no other smaller string is a codeword. So we can decode this string as gray level 4. Continuing in this way we obtain:

$$\underbrace{1 \quad 1 \quad 0}_{3} \quad \underbrace{1 \quad 1 \quad 1 \quad 0}_{4} \quad \underbrace{0 \quad 0}_{0} \quad \underbrace{0 \quad 0}_{0} \quad \underbrace{1 \quad 0}_{1} \quad \underbrace{0 \quad 1}_{2} \quad \underbrace{1 \quad 1 \quad 1 \quad 1 \quad 0}_{5}$$

as the decoding for this string.

For more information about Huffman coding, and its limitations and generalizations, see [13, 37].

## 14.3 Run Length Encoding

Run length encoding (RLE) is based on a simple idea: to encode strings of zeros and ones by the number of repetitions in each string. RLE has become a standard in facsimile transmission. For a binary image, there are many different implementations of RLE; one method is to encode each line separately, starting with the number of 0's. So the following binary image:

```
0 1 1 0 0 0
0 0 1 1 1 0
1 1 1 0 0 1
0 1 1 1 1 0
0 0 0 1 1 1
1 0 0 0 1 1
```

would be encoded as

$$(123)(231)(0321)(141)(33)(0132)$$

Another method [49] is to encode each row as a list of pairs of numbers; the first number in each pair given the starting position of a run of 1's, and the second number its length. So the above binary image would have the encoding

$$(22)(33)(1361)(24)(43)(1152)$$

Grayscale images can be encoded by breaking them up into their *bit planes*; these were discussed in Chapter 3.

To give a simple example, consider the following 4-bit image and its binary representation:

$$\begin{array}{cccc} 10 & 7 & 8 & 9 \\ 11 & 8 & 7 & 6 \\ 9 & 7 & 5 & 4 \\ 10 & 11 & 2 & 1 \end{array} \longrightarrow \begin{array}{cccc} 1010 & 0111 & 1000 & 1001 \\ 1011 & 1000 & 0111 & 0110 \\ 1001 & 0111 & 0101 & 0100 \\ 1010 & 1011 & 0010 & 0001 \end{array}$$

We may break it into bit planes as shown:

$$\begin{array}{cccc} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{array}$$
0th plane

$$\begin{array}{cccc} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{array}$$
1st plane

$$\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{array}$$
2nd plane

$$\begin{array}{cccc} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{array}$$
3rd plane

and then each plane can be encoded separately using our chosen implementation of RLE.

However, there is a problem with bit planes, and that is that small changes of gray value may cause significant changes in bits. For example, the change from value 7 to 8 causes the change of all four bits, since we are changing the binary strings 0111 to 1000. The problem

is of course exacerbated for 8-bit images. For RLE to be effective, we should hope that long runs of very similar gray values would result in very good compression rates for the code. But this may not be the case. A 4-bit image consisting of randomly distributed 7's and 8's would thus result in uncorrelated bit planes, and little effective compression.

To overcome this difficulty, we may encode the gray values with their binary *Gray codes*. A Gray code is an ordering of all binary strings of a given length so that there is only one bit change between a string and the next. So, a 4-bit Gray code is:

| | | | | |
|---|---|---|---|---|
| 15 | 1 | 0 | 0 | 0 |
| 14 | 1 | 0 | 0 | 1 |
| 13 | 1 | 0 | 1 | 1 |
| 12 | 1 | 0 | 1 | 0 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 0 |
| 8 | 1 | 1 | 0 | 0 |
| 7 | 0 | 1 | 0 | 0 |
| 6 | 0 | 1 | 0 | 1 |
| 5 | 0 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 0 |
| 3 | 0 | 0 | 1 | 0 |
| 2 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 |

See [37] for discussion and detail. To see the advantages, consider the following 4-bit image with its binary and Gray code encodings:

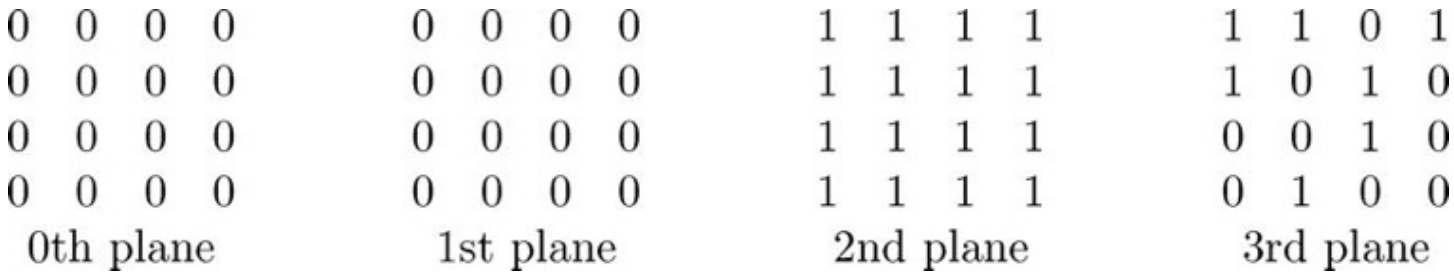
$$\begin{array}{cccc} 8 & 8 & 7 & 8 \\ 8 & 7 & 8 & 7 \\ 7 & 7 & 8 & 7 \\ 7 & 8 & 7 & 7 \end{array} \longrightarrow \begin{array}{cccc} 1000 & 1000 & 0111 & 1000 \\ 1000 & 0111 & 1000 & 0111 \\ 0111 & 0111 & 1000 & 0111 \\ 0111 & 1000 & 0111 & 0111 \end{array} \qquad \begin{array}{cccc} 1100 & 1100 & 0100 & 1100 \\ 1100 & 0100 & 1100 & 0100 \\ 0100 & 0100 & 1100 & 0100 \\ 0100 & 1100 & 0100 & 0100 \end{array}$$

where the first binary array is the standard binary encoding, and the second array the Gray codes. The binary bit planes are:

$$\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{array} \qquad \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{array} \qquad \begin{array}{cccc} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \end{array} \qquad \begin{array}{cccc} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

0th plane 1st plane 2nd plane 3rd plane

and the bit planes corresponding to the Gray codes are:




Notice that the Gray code planes are highly correlated except for one bit plane, whereas all the binary bit planes are uncorrelated.

## Implementing Run Length Coding

We can experiment with run length encoding by writing a simple function to implement it. We shall assume our data is binary, and in one single column. The first step is to find the places at which the data changes values, starting with the value 1, indicating that the first value is indeed different from the previous (non-existent!) value. We also need to append a 1 at the end, so that we can find the length of the final run. This can be done very easily by comparing the data with a shifted version of itself:

```
>> changes = [data ~= circshift(data,1);1];
>> changes(1) = 1;
```

MATLAB/Octave

In Python:

```
In :  changes = np.hstack(((data != np.roll(data,1))*1,[1]))
In :  changes[0] = 1
```

Python

Now the run-length code is simply the difference between the indices of the ones:

```
>> diffs = find(changes==1);
>> rle = diffs - circshift(diffs,1);
```

MATLAB/Octave

```
In :  diffs = np.nonzero(changes)
In :  rle = (diffs-np.roll(diffs,1)).tolist()[0]
```

Python

The only problem here is the first element, which should be changed to zero if the initial value of `data` is 1, and removed if the first element of data is zero: