TSBK38 Image and Audio Compression
Quick Reference

Harald Nautsch

March 4, 2025

Contents

1 Foreword **5**

2 Introduction **5**
2.1 What is data compression? 5
2.2 Example: JPEG still image coding 5
2.3 Performance measures 6
2.4 What is the original? 7
2.5 Properties to use 7

3 Random source models **7**
3.1 Discrete sources 7
3.2 Random variables and processes 7
3.3 Memory sources 8
3.4 Markov source example 9
3.5 Random modeling 10

4 Lossless compression **11**
4.1 Source coding 11
4.2 Properties of codes 12
4.3 Tree codes 12
4.4 Decoding a tree code 13
4.5 Code performance 13
4.6 Kraft's inequality, mean codeword length 13
4.7 Entropy as a lower bound 14
4.8 Optimal codes 14
4.9 Extended codes 15

5 Information theory **15**
5.1 Information measure 15
5.2 Entropy 16
5.3 Entropy of sources 17

6 Practical coding methods **17**
6.1 Huffman coding 17
6.2 The unary code 18
6.3 Golomb codes 18
6.4 Arithmetic coding 19
6.4.1 Interval division 20
6.4.2 Generating the codeword 21
6.4.3 Average codeword length and rate 21
6.4.4 Memory sources 22
6.4.5 Arithmetic decoding 22
6.4.6 Coding example 22
6.4.7 Decoding example 23

6.4.8 Practical problems 24
6.5 Lempel-Ziv coding 25
6.5.1 LZ77 25
6.5.2 LZSS 25
6.5.3 Other improvements 26
6.5.4 Buffer sizes 26
6.5.5 LZ78 27
6.5.6 LZW 27

7 Coding with distortion **27**
7.1 Continuous alphabet sources 27
7.2 Distortion measure 28
7.3 Random signal models 29
7.3.1 Examples of probability distributions 29
7.3.2 Dependence and correlation 30
7.3.3 Auto correlation function 30
7.4 Distortion for random signals 31
7.5 Theoretical limit 31
7.6 Quantization 32
7.7 Uniform quantization 33
7.8 Lloyd-Max quantization 35
7.9 Quantization followed by source coding 36
7.10 Fine quantization 36
7.11 Vector quantization 39
7.12 The LBG algorithm 41

8 Linear predictive coding **42**
8.1 Optimization of predictor coefficients 43
8.2 Prediction gain 44
8.3 Signals with nonzero mean 45
8.4 Multidimensional predictors 45
8.5 Lossless predictive coding 45

9 Colour images **46**
9.1 Converting from RGB to YCbCr 46

10 Transform coding **46**
10.1 Main idea 47
10.2 Linear transforms 48
10.3 Orthonormal transforms 48
10.4 The transform as a basis change 49
10.5 Transform properties 49
10.6 The Karhunen-Loève-transform (KLT) 49
10.7 The discrete cosine transform (DCT) 50
10.8 The discrete Walsh-Hadamard transform 51
10.9 Comparison between DCT and DWHT 51

10.10Comparison between DCT and KLT 51
10.11Twodimensional signals 52
10.12Block size 53
10.13Distortion 53
10.14Zonal coding 54
10.15Transform coding gain 55
10.16Threshold coding 55
10.17JPEG 56

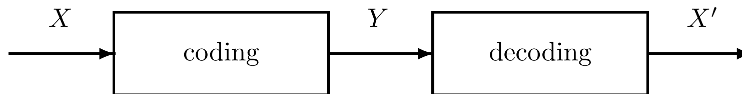
11 Subband coding 57
11.1 Subband coder (M bands) 57
11.2 Recursive filtering 58
11.3 Filter properties 60
11.4 Twodimensional signals 61
11.5 Quantization and source coding 62
11.6 JPEG 2000 62

1 Foreword

This document is not meant to be the sole source for learning about data compression. The contents are based on the course slides and examples shown on lectures and/or lessons and give brief descriptions of theory and coding methods. For more details I refer you to other literature, in particular the 4th edition of "*Introduction to Data Compression*" by Khalid Sayood. This book is available in electronical form through the university library.

2 Introduction

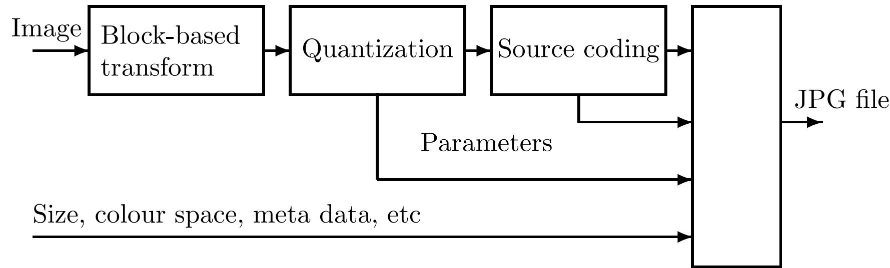
2.1 What is data compression?



We have some data X that we want to compress. This can be almost anything (text, audio, still images, video streams, etc.). The date is compressed using *source coding*. The result of this coding is a new signal Y that is "smaller" than the original X, ie it requires less bits to represent. The signal Y is what we are sending or storing. The reverse operation is *source decoding*, giving us the decoded signal X'.
We can differentiate between two classes of coding; *lossless* coding or *lossy* coding. In lossless coding, the decoded data X' is exactly the same as the original data X, while in lossy coding the decoded data is different from the original (ie we introduce some form of *distortion*). Depending on what kind of data we have, distortion might or might not be tolerated. For example, if the data is written text we usually do not want any distortion. If we instead have still images, we can usually tolerate that there might be small changes to the pixel values. The advantages of doing lossy compression is that this will give us the ability to get much more compression compared to lossless compression.

2.2 Example: JPEG still image coding

This is a simplified block diagram description of a JPEG still image coder:



JPEG is an example of a lossy coding method. The source coding box by itself is a lossless coding methods. The quantizer is the part where information is removed and where the distortion is introduced.
A JPEG decoder does everything in reverse order.

2.3 Performance measures

We need to measure how good our coder is. The obvious way is of course to measure how much *compression* we get from our method. This can be done in several (but equivalent) ways.
The first way is to measure the *compression ratio*, ie the ratio between the size of the original data and the size of the compressed data. For example, if we start with a file of size 25000 bytes and compress it down to a file of size 10000 bytes then the compression ratio is 2.5.
Another way of measuring is to look at the *average data rate* in bits/symbol (or bits/pixel, bits/sample). If we again look at the same example, assuming that the original symbols in the data consisted of single 8-bit bytes, then we get a rate of $10000 \cdot 8/25000 = 3.2$ bits/symbol, compared to the uncompressed version which has a rate of 8 bits/symbol.
The two ways of measuring are of course related, since $8/3.2 = 2.5$. In this course we mostly are going to use the rate as a measure, since it is easy to compare to theoretical limits (entropy).
For video and audio signals and other time signals, the rate is often given as bits/s (kbits/s, Mbits/s).
If we are doing lossless compression, then the rate (or compression ratio) is the only important part. However, when doing lossy compression we also have to measure how much *distortion* we have introduced into the decoded signal.
For media signals (audio, images, video) the most “correct” way would be to use human evaluation of the compressed data. Ie, let a test panel grade the quality of the decoded signal compared to the original signal, on a scale from 1-5.
While human evaluation is in some way the proper way of measuring distortion, it is very impractical. Instead, we would like to have some mathematical measure that is easy to calculate. The most common way of measuring distortion is the *mean square error (mse)* which is usually given as the *signal to noise ratio (SNR)* in dB.

For audio signals (music, speech) there are good models for how the human hearing works, and then we can adapt the mathematical models to take into account more subjective (qualitative) experiences.

2.4 What is the original?

What is the original data that we are compressing? For audio and images the original signals are *sampled* and *finely quantized* amplitude signals, ie we have digital signals.
The signal can be either scalar (mono sound, grayscale images) or vector valued (RGB, CMYK, multispectral images, stereo sound, surround sound).
Even though the original signal is almost always already quantized we will still often use amplitude continuous models for it

2.5 Properties to use

There are a few properties of the signal can we use to achieve compression:

- All symbols are not equally common. For instance in a music signal small amplitude values are more common than large amplitude values. A good code will have short descriptions for common values and longer descriptions for uncommon values.
- Dependence between symbols (samples, pixels). For instance in an image two pixels next to each other usually have almost the same value. A good coding algorithm will take advantage of this dependence.
- Properties of the human vision or hearing system. We can remove information that a human can not see or hear anyway.

3 Random source models

3.1 Discrete sources

A *source* is something that produces a sequence of *symbols*. The symbols are elements in a discrete *alphabet* $\mathcal{A} = \{a_1, a_2, \ldots, a_L\}$ of size L. We will mostly deal with finite alphabets, but infinite alphabets can also be used.
In most cases we only have access to a symbol sequence generated by the source and we will have to model the source from the given sequence.

3.2 Random variables and processes

The source models we will focus on are *random models*, where we assume that the symbols are generated by random variables or random processes.
The simplest random model for a source is a discrete random variable X. We have the probability that the random variable takes a certain value from the alphabet:

$$Pr(X = a_i) = P_X(a_i) = P(a_i) = p_i$$

$$P(a_i) \geq 0 \ , \quad \forall a_i$$

$$\sum_{i=1}^{L} P(a_i) = 1$$

Random variables can be useful as models if we have sources where there is no dependence between different symbols in a sequence (ie white noise). However, most real world signals have some kind of dependence between symbols in the sequence (text, images and audio don't look or sound like white noise).
A better source models where we can model the dependence is a discrete stationary *random process*.
A random process X_t can be viewed as a sequence of random variables, where we get an outcome in each time instance t. (We are using "time" as a general term for sequence number here, for instance in a text or an image we don't have an explicit time, but we do have a position in the sequence.)
Joint and conditional probabilities given the output of the source at two different times t and s:

$$P(x_t, x_s) = Pr(X_t = x_t, X_s = x_s)$$

$$P(x_s|x_t) = \frac{P(x_t, x_s)}{P(x_t)} \ , \text{ when } P(x_t) > 0$$

$$P(x_t, x_s) = P(x_t) \cdot P(x_s|x_t)$$

3.3 Memory sources

Dependence between the signal at different times is called *memory*. If X_t and X_{t+k} are independent for all $k \neq 0$ the source is *memoryless*.
For a memoryless source we have:

$$P(x_t, x_{t+k}) = P(x_t) \cdot P(x_{t+k})$$

$$P(x_{t+k}|x_t) = P(x_{t+k})$$

A *Markov source* of order k is a memory source with limited memory k steps back in the sequence.

$$P(x_n|x_{n-1}x_{n-2}\ldots) = P(x_n|x_{n-1}\ldots x_{n-k})$$

If the alphabet is $\mathcal{A} = \{a_1, a_2, \ldots, a_L\}$, the Markov source can be described as a state model with L^k states $(x_{n-1}\ldots x_{n-k})$ where we at time n move from state $(x_{n-1}\ldots x_{n-k})$ to state $(x_n\ldots x_{n-k+1})$ with probability $P(x_n|x_{n-1}\ldots x_{n-k})$. These probabilities are called *transition probabilities*

The sequence of states is a random process $S_n = (X_n, X_{n-1} \dots, X_{n-k+1})$ with alphabet $\{s_1, s_2, \dots, s_{L^k}\}$ of size L^k.
The Markov source can be described using its starting state and its *transition matrix* $\mathbf{P}$. This quadratic matrix has in row r and column k the transition probability from state s_r to s_k.
If it is possible to move, with positive probability, from every state to every other state in a finite number of steps, the Markov source is called *irreducible*.
If we at time n are in state s_i with the probability p_i^n, we can calculate the probabilities for time $n+1$ as

$$(p_1^{n+1}\ p_2^{n+1}\ \dots\ p_{L^k}^{n+1}) = (p_1^n\ p_2^n\ \dots\ p_{L^k}^n) \cdot \mathbf{P}$$

A distribution over the states such that the distribution at time $n+1$ is the same as at time n is called a *stationary distribution*. If the probabilities for the starting state of a Markov source is a stationary distribution, then the source is a stationary process.
If the Markov source is irreducible and aperiodic the stationary distribution is unique and regardless of the starting distribution, the distribution over the states will approach the stationary distribution as time goes to infinity.
We denote the stationary probabilities w_i and define the row vector

$$\bar{w} = (w_1, w_2, \dots, w_{L^k})$$

If the stationary distribution exists, it can be found as the solution of the equation system

$$\bar{w} = \bar{w} \cdot \mathbf{P}$$

or

$$\bar{w} \cdot (\mathbf{P} - \mathbf{I}) = \bar{0}$$

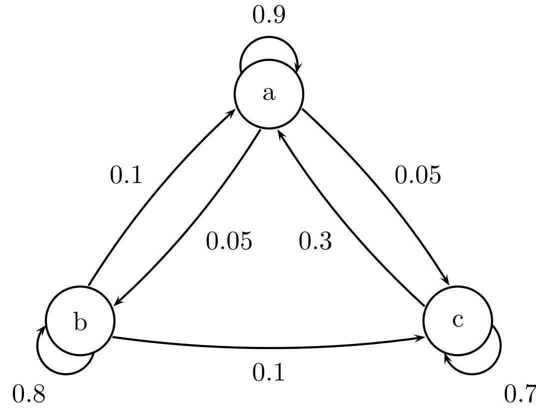
This equation system is under-determined (if $\bar{w}$ is a solution then $c \cdot \bar{w}$ is also a solution). To find the correct solution we add the equation $\sum_{j=1}^{L^k} w_j = 1$ (w_j are probabilities and therefore their sum is 1).
(If you prefer equation systems with column vectors, you can just transpose the entire expression and solve $\bar{w}^T = \mathbf{P}^T \cdot \bar{w}^T$ instead.)

3.4 Markov source example

Suppose we have a stationary Markov source X_n of order 1 with the alphabet $\mathcal{A} = \{a, b, c\}$ and the transition probabilities $P(x_n|x_{n-1})$ given by

$$\begin{array}{lll} P(a|a) = 0.9 & P(b|a) = 0.05 & P(c|a) = 0.05 \\ P(a|b) = 0.1 & P(b|b) = 0.8 & P(c|b) = 0.1 \\ P(a|c) = 0.3 & P(b|c) = 0 & P(c|c) = 0.7 \end{array}$$

We can draw the Markov source as the state graph below:



The stationary distribution of the Markov source are the probabilities $P(x_n)$ for being in the states at any given time. For a source of order 1 this is the same as the probabilities for the different symbols. For the example source this can be gotten from the equation system

$$\begin{cases} P(a) &= 0.9 \cdot P(a) + 0.1 \cdot P(b) + 0.3 \cdot P(c) \\ P(b) &= 0.05 \cdot P(a) + 0.8 \cdot P(b) \\ P(c) &= 0.05 \cdot P(a) + 0.1 \cdot P(b) + 0.7 \cdot P(c) \end{cases}$$

This is an underdetermined system (ie one of the equations is always a linear combination of the other equations) with a parametric solution. By adding the equation

$$P(a) + P(b) + P(c) = 1$$

we can find the solution

$$P(a) = \frac{4}{6} \ ; \ P(b) = \frac{1}{6} \ ; \ P(c) = \frac{1}{6}$$

Given this we can now also calculate any other joint or conditional probabilities of the source. For instance, to cacluate the probability distribution $P(x_n, x_{n+1})$ for pairs of symbols from the source, we use Bayes' rule $P(x_n, x_{n+1}) = P(x_n) \cdot P(x_{n+1}|x_n)$ which gives us

$$\begin{array}{lll} P(a,a) = 36/60 & P(a,b) = 2/60 & P(a,c) = 2/60 \\ P(b,a) = 1/60 & P(b,b) = 8/60 & P(b,c) = 1/60 \\ P(c,a) = 3/60 & P(c,b) = 0 & P(c,c) = 7/60 \end{array}$$

Similarily, we can calculate the probabilities of triples as $P(x_n, x_{n+1}, x_{n+2}) = P(x_n) \cdot P(x_{n+1}|x_n) \cdot P(x_{n+2}|x_{n+1})$.

3.5 Random modeling

Give a long symbol sequence from a source, how do we make a random model for it?

Use relative frequencies: To estimate the probability for a symbol, count the number of times that symbol appears and divide by the total number of symbols in the sequence. In the same way this can be done for pair probabilities, triple probabilities, conditional probabilities et c.
Example: We have a source with alphabet $\{a, b\}$. A sequence from the source looks like: *bbbbaabbbaaaaabbbbbabaaabbbb*.
To estimate the symbol probabilities we count how often each symbol appears: a appears 11 times, b 17 times. The estimated probabilities $P(x_t)$ are then:

$$P(a) = \frac{11}{28}, \quad P(b) = \frac{17}{28}$$

For pair probabilities and conditional probabilities we instead count how often the different symbol pairs appear. aa appears 7 times, ab 4 times, ba 4 times and bb 12 times. The estimated probabilities $P(x_t, x_{t+1})$ and $P(x_{t+1}|x_t)$ are:

$$P(aa) = \frac{7}{27}, \quad P(ab) = \frac{4}{27}, \quad P(ba) = \frac{4}{27}, \quad P(bb) = \frac{12}{27}$$

$$P(a|a) = \frac{7}{11}, \quad P(b|a) = \frac{4}{11}, \quad P(a|b) = \frac{4}{16}, \quad P(b|b) = \frac{12}{16}$$

4 Lossless compression

4.1 Source coding

Source coding means mapping sequences of symbols from a source alphabet onto binary sequences (called *codewords*).
The set of all codewords is called a *code*.
A code where all the codewords have the same length (number of bits) is called a *fixed-length code*, otherwise we have a *variable length code*.
Example: $\mathcal{A} = \{a, b, c, d\}$

Symbol	Code 1	Code 2	Code 3	Code 4	Code 5
a	00	0	0	0	0
b	01	0	1	10	01
c	10	1	00	110	011
d	11	10	11	111	111

Code the sequence *abbacddcd* using our five codes

Code 1: 000101001011111011

Code 2: 000011010110

Code 3: 01100011110011

Code 4: 01010011011111110111

Code 5: 00101001111111011111

4.2 Properties of codes

If you from any sequence of codewords can recreate the original symbol sequence, the code is called *uniquely decodable*.
If you can recognize the codewords directly while decoding, the code is called *instantaneous*.
If no codeword is a prefix to another codeword, the code is called a *prefix code* (in some literature they are called *prefix free* codes). These codes are *tree codes*, ie each codeword can be described as the path from the root to a leaf in a binary tree.
All prefix codes are instantaneous and all instantaneous codes are prefix codes, ie they are the same class .
Example, $\mathcal{A} = \{a, b, c, d\}$

Symbol	Code 1	Code 2	Code 3	Code 4	Code 5
a	00	0	0	0	0
b	01	0	1	10	01
c	10	1	00	110	011
d	11	10	11	111	111

Code 1 Uniquely decodable, instantaneous (tree code)

Code 2 Not uniquely decodable

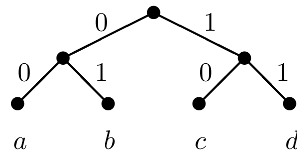
Code 3 Not uniquely decodable

Code 4 Uniquely decodable, instantaneous (tree code)

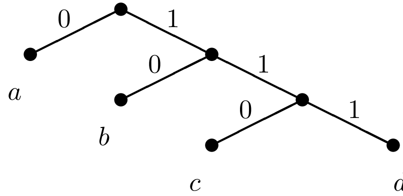
Code 5 Uniquely decodable, not instantaneous

4.3 Tree codes

An instantaneous code can always be described as a binary tree, ie the codewords are given by the paths from the root to the leaves in a binary tree. Code 1 and code 4 from the previous examples are instantaneous codes.
Code 1 is described by this binary tree:



Code 4 is described by this binary tree:



4.4 Decoding a tree code

When decoding an instantaneous code, we can use the tree describtion for decoding. Start at the root node of the tree. Read a bit from the bit sequence and go down the corresponding branch. Keep doing this until we reach a leaf node, then output the corresponding symbol to the symbol sequence and start over at the root node. Repeat until all the bits have been decoded.
For example, if we have coded a symbol sequence using code 4 above into the following bit sequence

$$1011000101110010$$

then using the above algorithm we can easily decode it into the symbol sequence

$$bcaabdaab$$

This algorithm could of course also be used for decoding data coded with code 1. However, since this is a fixed length code (all codewords have length 2), we can just split the bit sequence into chunks of 2 bits and look up the corresponding symbols in the codeword table.

4.5 Code performance

How good a code is is determined by its *mean data rate* R (usually just referred to as *rate* or *data rate*) in bits/symbol.

$$R = \frac{\text{average number of bits per codeword}}{\text{average number of symbols per codeword}}$$

Since we're doing compression we want R to be as small as possible.
For a random source, there is a theoretical lower bound on the rate.
Note that R is a measure of how good the code is *on average* over all possible sequences from the source. It tells us nothing of how good the code is for a particular sequence. Some symbol sequences will give short bit sequences, while other symbol sequences give long bit sequences

4.6 Kraft's inequality, mean codeword length

An instantaneous code (prefix code, tree code) with the codeword lengths $l_1, \ldots, l_L$ exists if and only if

$$\sum_{i=1}^{L} 2^{-l_i} \leq 1$$

The inequality also holds for all uniquely decodable codes. It is then called Kraft-McMillan's inequality.
Mean codeword length:

$$\bar{l} = \sum_{i=1}^{L} p_i \cdot l_i \quad \text{[bits/codeword]}$$

if we code one symbol with each codeword we have

$$R = \bar{l}$$

4.7 Entropy as a lower bound

There is a lower bound on the mean codeword length of a uniquely decodable code:

$$\bar{l} \geq -\sum_{i=1}^{L} p_i \cdot \log_2 p_i = H(X_t)$$

$H(X_t)$ is the *entropy* of the source (more on entropy in section 5).
Proof of $\bar{l} \geq H(X_t)$

$$\begin{aligned}
H(X_t) - \bar{l} &= -\sum_{i=1}^{L} p_i \cdot \log_2 p_i - \sum_{i=1}^{L} p_i \cdot l_i = \sum_{i=1}^{L} p_i \cdot (\log_2 \frac{1}{p_i} - l_i) \\
&= \sum_{i=1}^{L} p_i \cdot (\log_2 \frac{1}{p_i} - \log_2 2^{l_i}) = \sum_{i=1}^{L} p_i \cdot \log_2 \frac{2^{-l_i}}{p_i} \\
&\leq \frac{1}{\ln 2} \sum_{i=1}^{L} p_i \cdot (\frac{2^{-l_i}}{p_i} - 1) = \frac{1}{\ln 2}(\sum_{i=1}^{L} 2^{-l_i} - \sum_{i=1}^{L} p_i) \\
&\leq \frac{1}{\ln 2}(1 - 1) = 0
\end{aligned}$$

where we used the inequality $\ln x \leq x - 1$ and Kraft-McMillan's inequality.

4.8 Optimal codes

A code is called *optimal* if no other code exists (for the same probability distribution) that has a lower mean codeword length.
There are of course several codes with the same mean codeword length. The simplest example is to just switch all ones to zeros and all zeros to ones in the codewords.
Even codes with different sets of codeword lengths can have the same mean codeword length.
Given that we code one symbol at a time, an optimal code satisfies $\bar{l} < H(X_t) + 1$

Let $l_i = \lceil -\log p_i \rceil$. We have that $-\log p_i \leq \lceil -\log p_i \rceil < -\log p_i + 1$.

$$\begin{aligned}\sum_{i=1}^{L} 2^{-l_i} &= \sum_{i=1}^{L} 2^{-\lceil -\log p_i \rceil} \\ &\leq \sum_{i=1}^{L} 2^{\log p_i} \\ &= \sum_{i=1}^{L} p_i \;=\; 1\end{aligned}$$

Kraft's inequality is satisfied, therefore a tree code with the given codeword lengths exists.
What's the mean codeword length of this code?

$$\begin{aligned}\bar{l} &= \sum_{i=1}^{L} p_i \cdot l_i = \sum_{i=1}^{L} p_i \cdot \lceil -\log p_i \rceil \\ &< \sum_{i=1}^{L} p_i \cdot (-\log p_i + 1) \\ &= -\sum_{i=1}^{L} p_i \cdot \log p_i + \sum_{i=1}^{L} p_i = H(X_t) + 1\end{aligned}$$

An optimal code can't be worse than this code, then it wouldn't be optimal. Thus, the mean codeword length for an optimal code also satisfies $\bar{l} < H(X_t)+1$.

NOTE: If $p_i = 2^{-k_i}, \forall i$ for integers k_i, we can construct a code with codeword lengths k_i and $\bar{l} = H(X_t)$.

4.9 Extended codes

For small alphabets with skewed distributions, or sources with memory, a Huffman code can be relatively far from the entropy bound. This can be solved by *extending* the source, ie by coding multiple symbols with each codeword.
If we code n symbols with each codeword, and the code has the mean codeword length $\bar{l}$ the rate will be

$$R = \frac{\bar{l}}{n}$$

The maximal redundancy of an optimal code (the difference between the rate and the entropy) is $\frac{1}{n}$ when we code n symbols at a time.

5 Information theory

5.1 Information measure

Given a discrete randoms variable X with distribution

$$p_i = P(a_i) = Pr(X = a_i)$$

The self information of the outcomes is

$$i(a_i) = -\log p_i$$

The logarithm can be taken in any base. For practical reasons, base 2 is usually used. The unit is then called *bits*. We can then easily compare the entropy with the rate of a binary code.
The lower the probability of an outcome is, the larger the information of the outcome is

$$p_i \to 0 \implies i(a_i) \to \infty$$
$$p_i = 1 \implies i(a_i) = 0$$

5.2 Entropy

The mean value of the information is called *entropy*.

$$H(X) = \sum_{i=1}^{L} p_i \cdot i(a_i) = -\sum_{i=1}^{L} p_i \cdot \log\ p_i$$

The entropy can be seen as a measure of the average information in X, or a measure of the uncertainty of X.
The entropy is bounded by

$$0 \leq H(X) \leq \log\ L$$

The entropy is maximized when all outcomes are equally probable, ie a uniform distribution.
If any one of the outcomes has probability 1 (and thus all the other outcomes have probability 0) the entropy is 0, ie there is no uncertainty.
Given two random variables X and Y with alphabets $\mathcal{A}$ and $\mathcal{B}$ (these alphabets can be the same) and joint distribution

$$P_{XY}(a_i, b_j) = P_X(a_i) \cdot P_{Y|X}(b_j|a_i) = P_Y(b_j) \cdot P_{X|Y}(a_i|b_j)$$

The *joint entropy* is defined by

$$H(X, Y) = -\sum_{i,j} P_{XY}(a_i, b_j) \cdot \log P_{XY}(a_i, b_j)$$

The *conditional entropy* is defined by

$$H(Y|X) = -\sum_{i,j} P_{XY}(a_i, b_j) \cdot \log P_{Y|X}(b_j|a_i)$$

$H(Y|X) \leq H(Y)$ with equality if X and Y are independent.
The joint entropy can be written as a sum of conditional entropies

$$H(X,Y) = H(X) + H(Y|X) = H(Y) + H(X|Y)$$

This can be generalized to any number of random variables

$$H(X,Y,Z,W) = H(X) + H(Y|X) + H(Z|X,Y) + H(W|X,Y,Z)$$

5.3 Entropy of sources

Given a stationary random process X_t
Similar to a random variable we have

$$H(X_t) = \sum_{i=1}^{L} p_i \cdot i(a_i) = -\sum_{i=1}^{L} p_i \cdot \log\ p_i$$

Conditional entropy

$$H(X_t|X_{t-1}) \leq H(X_t)$$

with equality if X_t is memoryless.
Joint entropy

$$H(X_{t-1}, X_t) = H(X_{t-1}) + H(X_t|X_{t-1}) \leq 2 \cdot H(X_t)$$

$$H(X_1, \ldots, X_n) = H(X_1) + H(X_2|X_1) + \ldots + H(X_n|X_1 \ldots X_{n-1}) \leq n \cdot H(X_t)$$

The *entropy rate* of the source (usually just called entropy) is given

$$\lim_{n\to\infty} \frac{1}{n} H(X_1 \ldots X_n) = \lim_{n\to\infty}\ H(X_n|X_1 \ldots X_{n-1})$$

ie the entropy of one symbol at any point in time, conditioned on everything that happened before that time.
For a memoryless source the entropy rate is just $H(X_t)$.
For a Markov source of order k the entropy rate is $H(X_t|X_{t-1} \ldots X_{t-k})$
The entropy rate gives a lower bound on the data rate of a uniquely decodable code for the source, ie if we want to do lossless coding of the output of a random source, the rate can never be less than the entropy rate of the source.

6 Practical coding methods

6.1 Huffman coding

Huffman codes are a method for constructing optimal tree codes. They are given by a simple algorithm, where we build the code tree from the leaves to the root.

Start with symbols as leaves.
In each step connect the two least probable nodes to an inner node. The probability for the new node is the sum of the probabilities of the two original nodes. If there are several nodes with the same probability to choose from it doesn't matter which ones we choose.
When we have constructed the whole code tree, we create the codewords by setting 0 and 1 on the branches in each node. Which branch that is set to 0 and which that is set to 1 doesn't matter.

6.2 The unary code

Huffman coding gives optimal tree codes and can be adapted to any probability distribution. However, this means that we need to transmit more extra information about the code and that the decoding algorithm might be slow. In many real world applications we will encounter distributions that are monotonously decreasing. Assuming that we have a the alphabet $\mathcal{A} = \{0, 1, 2, \ldots\}$ we will have

$$P(0) > P(1) > P(2) > \ldots$$

An example of a simple code for this type of distribution is the *unary code*. The unary codeword for a non-negative integer n consists of n ones followed by a zero.

Symbol	codeword
0	0
1	10
2	110
3	1110
4	11110
⋮	⋮

The unary code will give a rate that is equal to the entropy for the dyadic distribution $P(i) = 2^{-(i+1)}$
Note that we can actually define unary codes in two ways. In some applications long sequences of ones are not desirable. Then we can use the other definition, where the codeword is n zeros followed by a one.

6.3 Golomb codes

The unary code is perfect for the dyadic distribution. However we might have a distribution where the probabilities decrease slower than that. We would like to be able to adapt the code to any monotonously decreasing distribution. This leads us to *Golomb codes*.
Golomb codes are a class of codess that are suitable for these types of distributions. A Golomb code is specified by a single parameter m. This can be any positive integer. In practical applications we usually restrict ourselves to integer powers of two, because then the coding and decoding is especially easy.

Represent the integer n that we want to code as $q = \lfloor \frac{n}{m} \rfloor$ and $r = n - qm$, ie we do integer division of n with m and get a quotient q and a remainder r.
First we send the codeword for the quotient q, using a unary code.
If m is an integer power of two, code r using a fixed length code with length $\log m$ bits.
For example, if $n = 21$ and we use the Golomb code with $m = 8 = 2^3$, we get $q = 2$ and $r = 5$. The codeword for q is **110** (two ones and a zero) and the codeword for r is **101** (the number 5 written using 3 bits), giving us the complete codeword **110101**.
When decoding, we first find q by reading bits and counting how many ones we get until we reach a zero. Then we read the next $\log m$ bits and interpret that as a binary number, giving us r. Finally we get the decoded value as $n = qm + r$.
If we want a parameter m that is not an integer power of two, the coding algorithm is a little bit more complicated. q is still coded with a unary code, but the codeword for r will use either $\lfloor \log m \rfloor$ bits or $\lceil \log m \rceil$ bits according to:

If $0 \le r < 2^{\lceil \log m \rceil} - m$	Code r binary with $\lfloor \log m \rfloor$ bits
If $2^{\lceil \log m \rceil} - m \le r \le m - 1$	Code $r + 2^{\lceil \log m \rceil} - m$ binary with $\lceil \log m \rceil$ bits

The decoding of q is the same. For decoding r, start by reading $\lfloor \log m \rfloor$ bits and interpret this as a binary number. If this value r is less than $2^{\lceil \log m \rceil} - m$ we are finished, otherwise read one more bit, interpret the bits read as a binary number and then subtract $2^{\lceil \log m \rceil} - m$ from this value.
Note that the whole Golomb code is specified by the single parameter m, so there is very little information needed to describe the code.
Examples of Golomb codes:

Symbol	$m = 1$	$m = 2$	$m = 3$	$m = 4$
0	0	0 0	0 0	0 00
1	10	0 1	0 10	0 01
2	110	10 0	0 11	0 10
3	1110	10 1	10 0	0 11
4	11110	110 0	10 10	10 00
5	111110	110 1	10 11	10 01
6	1111110	1110 0	110 0	10 10
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Golomb codes are optimal for geometric distributions:

$$P(i) = s^i \cdot (1 - s) \quad ; \quad 0 < s < 1$$

if we choose $m = \lceil -\frac{1}{\log s} \rceil$

6.4 Arithmetic coding

Huffman coding is optimal in theory, but it can be impractical to use for skewed distributions and/or when extending the source.

Exampe: $\mathcal{A} = \{a, b, c\}$, $P(a) = 0.95, P(b) = 0.02, P(c) = 0.03$
The entropy of the source is approximately 0.3349. The mean codeword length of a Huffman code is 1.05 bits/symbol, ie more than 3 times the entropy. If we want the rate to be no more than 5% larger than the entropy we have to extend the source and code 8 symbols at a time. This gives a Huffman code with $3^8 = 6561$ codewords.
We would like to have coding method where we can directly find the codeword for a given sequence, without having to determine the codewords for *all* possible sequences. One way of doing this is *arithmetic coding.*
Suppose that we have a source X_t taking values in the alphabet $\{1, 2, \ldots, L\}$. Assume that the probabilities for all symbols are strictly positive: $P(i) > 0$, $\forall i$. The cumulative distribution function $F(i)$ is defined as

$$F(i) = \sum_{k \leq i} P(k)$$

$F(i)$ is a step function where the step in k has the height $P(k)$.
Example:
$\mathcal{A} = \{1, 2, 3\}$
$P(1) = 0.5,\ P(2) = 0.3,\ P(3) = 0.2$
$F(0) = 0,\ F(1) = 0.5,\ F(2) = 0.8,\ F(3) = 1$

6.4.1 Interval division

The main idea behind arithmetic doing is to associate each possible symbol sequence of length n with an interval somewhere inside the whole probability interval $[0, 1)$.
The size of each interval will be equal to the probability of the corresponding sequence. There is no overlap between the intervals and there are no parts of the whole probability interval that do not belong to any sequence interval (this follows from the fact that the sum of the probabilities of all possible sequences is 1).
Suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$.
Start with the whole probability interval $[0, 1)$. In each step j divide the interval proportional to the cumulative distribution $F(i)$ and choose the subinterval corresponding to the symbol x_j that is to be coded.
If we have a memory source the intervals are divided according to the conditional cumulative distribution function.
An iterative algorithm for finding the interval for a given sequence is as follows: Again, suppose that we want to code a sequence $\mathbf{x} = x_1, x_2, \ldots, x_n$. We denote the lower limit in the corresponding interval by $l^{(n)}$ and the upper limit by $u^{(n)}$. The interval generation is the given iteratively by

$$\begin{cases} l^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j - 1) \\ u^{(j)} = l^{(j-1)} + (u^{(j-1)} - l^{(j-1)}) \cdot F(x_j) \end{cases}$$

Starting values of the limits are $l^{(0)} = 0$ and $u^{(0)} = 1$.

The interval size is equal to the probability of the sequence, so

$$u^{(n)} - l^{(n)} = P(\mathbf{x})$$

6.4.2 Generating the codeword

Each symbol sequence of length n uniquely identifies a subinterval. The codeword for the sequence is a number in the interval. The number of bits needed in the codeword depends on the interval size, so that a large interval (ie a sequence with high probability) gets a short codeword, while a small interval gives a longer codeword.
The codeword for an interval is given by the shortest bit sequence $b_1 b_2 \ldots b_k$ such that the binary number $0.b_1 b_2 \ldots b_k$ is in the interval and that all other numbers staring with the same k bits are also in the interval.
Given a binary number a in the interval $[0, 1)$ with k bits $0.b_1 b_2 \ldots b_k$. All numbers that have the same k first bits as a are in the interval $[a, a + \frac{1}{2^k})$.
A necessary condition for all of this interval to be inside the interval belonging to the symbol sequence is that it is less than or equal in size to the symbol sequence interval, ie

$$P(\mathbf{x}) \geq \frac{1}{2^k} \quad \Rightarrow \quad k \geq \lceil -\log P(\mathbf{x}) \rceil$$

We can't be sure that it is enough with $\lceil -\log P(\mathbf{x}) \rceil$ bits, since we can't place these intervals arbitrarily (the intervals specified by k bits have fixed positions, while the intervals given by the symbol sequences can be placed anywhere). We can however be sure that we need at most one extra bit.
The codeword length $l(\mathbf{x})$ for a sequence $\mathbf{x}$ is thus given by

$$l(\mathbf{x}) = \lceil -\log P(\mathbf{x}) \rceil \quad \text{or} \quad l(\mathbf{x}) = \lceil -\log P(\mathbf{x}) \rceil + 1$$

6.4.3 Average codeword length and rate

The average codeword length when doing arithmetic coding is given by

$$\begin{aligned} \bar{l} &= \sum_{\mathbf{x}} P(\mathbf{x}) \cdot l(\mathbf{x}) \leq \sum_{\mathbf{x}} P(\mathbf{x}) \cdot (\lceil -\log P(\mathbf{x}) \rceil + 1) \\ &< \sum_{\mathbf{x}} P(\mathbf{x}) \cdot (-\log P(\mathbf{x}) + 2) = -\sum_{\mathbf{x}} P(\mathbf{x}) \cdot \log P(\mathbf{x}) + 2 \cdot \sum_{\mathbf{x}} P(\mathbf{x}) \\ &= H(X_1 X_2 \ldots X_n) + 2 \end{aligned}$$

The resulting data rate is thus bounded by

$$R = \frac{\bar{l}}{n} < \frac{1}{n} H(X_1 X_2 \ldots X_n) + \frac{2}{n}$$

This is a little worse than the rate for an extended Huffman code, but extended Huffman codes are not practical for large n. The complexity of an arithmetic coder, on the other hand, is independent of how many symbols n that are coded. In arithmetic coding we only have to find the codeword for a particular sequence and not for all possible sequences.

6.4.4 Memory sources

When doing arithmetic coding of memory sources, we let the interval division depend on earlier symbols, ie we use different F in each step depending on the value of earlier symbols.
For example, if we have a binary Markov source X_t of order 1 with alphabet $\{1,\ 2\}$ and transition probabilities $P(x_t|x_{t-1})$

$$P(1|1) = 0.8, \quad P(2|1) = 0.2, \quad P(1|2) = 0.1, \quad P(2|2) = 0.9$$

we will use two conditional cumulative distribution functions $F(x_t|x_{t-1})$

$$F(0|1) = 0, \quad F(1|1) = 0.8, \quad F(2|1) = 1$$

$$F(0|2) = 0, \quad F(1|2) = 0.1, \quad F(2|2) = 1$$

For the first symbol in the sequence we can either choose one of the two distributions (ie we assume that the Markov source is starting in a particular state) or use a third cumulative distribution function based on the stationary probabilities.

6.4.5 Arithmetic decoding

The decoder receives a bit stream of coded data. In order to decode, the decoder needs to know F and the length n of the original sequence.
The decoder will keep track of the same upper and lower limits u and l as the coder used, starting with $l^{(0)} = 0$ and $u^{(0)} = 1$. The decoder reads bits from the bitstream, interpreting the bits as a number. When enough bits have been read that the number is certain to be in one of the intervals given
When decoding the decoder will read bits from the bitstream one at a time and interpret as a binary number. The read bits will specify an interval. This bit interval will be compared to the symbol intervals and as soon as all of the bit interval is wholly inside a symbol interval we can decode that corresponding symbol. The symbol interval is then split into its subintervals. Keep reading bits until we have deocded n symbols.

6.4.6 Coding example

Assume a source with alphabet $\mathcal{A} = \{1, 2, 3\}$ and symbol probabilities

$$P(1) = 0.6,\ P(2) = 0.3,\ P(3) = 0.1$$

The cumulative distribution function is then

$$F(0) = 0,\ F(1) = 0.6,\ F(2) = 0.9,\ F(3) = 1$$

We want to code the sequence 1,3,2,1. The number of symbols to code $n = 4$.

$$l^{(0)} \quad = \quad 0$$

$$
\begin{aligned}
u^{(0)} &= 1
\end{aligned}
$$

$x_1 = 1$

$$
\begin{aligned}
l^{(1)} &= 0 + (1-0) \cdot 0 = 0 \\
u^{(1)} &= 0 + (1-0) \cdot 0.6 = 0.6
\end{aligned}
$$

$x_2 = 3$

$$
\begin{aligned}
l^{(2)} &= 0 + (0.6-0) \cdot 0.9 = 0.54 \\
u^{(2)} &= 0 + (0.6-0) \cdot 1 = 0.6
\end{aligned}
$$

$x_3 = 2$

$$
\begin{aligned}
l^{(3)} &= 0.54 + (0.6-0.54) \cdot 0.6 = 0.576 \\
u^{(3)} &= 0.54 + (0.6-0.54) \cdot 0.9 = 0.594
\end{aligned}
$$

$x_4 = 1$

$$
\begin{aligned}
l^{(4)} &= 0.576 + (0.594-0.576) \cdot 0 = 0.576 \\
u^{(4)} &= 0.576 + (0.594-0.576) \cdot 0.6 = 0.5868
\end{aligned}
$$

The sequence corresponds to the interval $[0.576 \quad 0.5868)$. The interval size is 0.0108 and thus we will need at least $\lceil -\log_2 0.0108 \rceil = 7$ bits in our codeword, maybe one more. Write the two interval limits as binary numbers:

$$
\begin{aligned}
0.576 &= 0.10010011011\ldots \\
0.5868 &= 0.10010110001\ldots
\end{aligned}
$$

The smallest seven bit number inside the interval is 0.1001010, and all numbers starting with these bits are also inside the interval (ie smaller than the upper interval limit). Thus, seven bits are enough. The codeword is **1001010**.

6.4.7 Decoding example

The decoder has to know the number of symbols coded ($n = 4$) and the cumulative distribution function:

$$F(0) = 0,\ F(1) = 0.6,\ F(2) = 0.9,\ F(3) = 1$$

Decode the first codeword in the bitstream starting 1001010001011...

Read one bit at a time. The bits read sofar $b_1 b_2 \ldots b_k$ specify an interval $[0.b_1 b_2 \ldots b_k \quad 0.b_1 b_2 \ldots b_k + 1/2^k)$ As soon as this interval is wholly inside the subinterval for a particular symbol sequence, we can decode one symbol. For

the first symbol, the intervals corresponding to symbols 1, 2 and 3 are [0 0.6), [0.6 0.9) and [0.9 1) respectively.

bits	binary interval	decimal interval
1	[0.1 1)	[0.5 1)
10	[0.10 0.11)	[0.5 0.75)
100	[0.100 0.101)	[0.5 0.625)
1001	[0.1001 0.1010)	[0.5625 0.625)
10010	[0.10010 0.10011)	[0.5625 0.59375)

Since this interval is wholly inside the interval for symbol 1, we can decode the first symbol as 1. We then split this symbol interval into three parts acording to F. This is gives us the intervals corresponding to symbols 1,2 and 3 as [0 0.36), [0.36 0.54) and [0.54 0.6) respectively. We again check our bit interval and see that we are wholly inside the interval for 3. Thus we decode the second symbol as 3. We again split the symbolinterval into three parts acording to F. This is gives us the intervals corresponding to symbols 1,2 and 3 as [0.54 0.576), [0.576 0.594) and [0.594 0.6) respectively. Checking our bit interval again, we see it covers more than one symbol interval. We must thus read more bits.

bits	binary interval	decimal interval
100101	[0.100101 0.10011)	[0.578125 0.59375)

This interval is wholly inside the interval for symbol 2, so we can decode the third symbol as 2. Split this symbol interval into three parts acording to F. This is gives us the intervals corresponding to symbols 1,2 and 3 as [0.576 0.5868), [0.5868 0.5922) and [0.5922 0.594) respectively. Checking our bit interval again, we see it covers more than one symbol interval. We must thus read more bits.

bits	binary interval	decimal interval
1001010	[0.1001010 0.1001011)	[0.578125 0.5859375)

This interval is wholly inside the interval for symbol 1, so we can decode the fourth symbol as 1. Since we have now decoded 4 symbols, we are finished. The following bits in the bitstream belong to the next codeword.
Our decoded symbol sequence is 1,3,2,1 which is exactly the symbol sequence that that we coded.

6.4.8 Practical problems

The way we described arithmetic coding sofar is a theoretical description that assumes that we first find the interval and then find the codeword for the resulting interval. We also assumed that all calculations can be done exactly. and that we have arbitrary precision in values. However, this is not true in the real world.
When implementing arithmetic coding we have limited precision data types available in our software or hardware and can't store interval limits and probabilities with aribtrary resolution.

We also want to start sending bits without having to wait for the whole sequence with n symbols to be coded.
In this course we will not go further into practical implementations of arithmetic coding, but if you are interested you should read the corresponding material in the course book.

6.5 Lempel-Ziv coding

Lempel-Ziv coding is a group of coding algorithms where the code words are referencing previous data in the sequence
There are two main types of Lempel-Ziv coders:

- Use a history buffer, code a partial sequence as a pointer to when that particular sequence last appeared (LZ77).
- Build a dictionary of all unique partial sequences that appears. The codewords are references to earlier words (LZ78).

The coder and decoder don't need to know the statistics of the source. It can be shown that given a stationary random source, the rate will asymptotically reach the entropy rate of the source (assuming good choices of coding parameters). Any coding method that has this property is called *universal*.
Lempel-Ziv coding in all its different variants are very popular methods for general file compression and archiving, eg zip, gzip, ARJ and compress.
The image coding standards GIF and PNG use Lempel-Ziv.

6.5.1 LZ77

View the sequence to be coded through a sliding window. The window is split into two parts, one part containing already coded symbols (search buffer) and one part containing the symbols that are about to coded next (look-ahead buffer).
Find the longest sequence in the search buffer that matches the sequence that starts in the look-ahead buffer. The codeword is a triple $< o, l, c >$ where o is a pointer to the position in the search buffer where we found the match (offset), l is the length of the sequence, and c is the next symbol that doesn't match. This triple is coded using a fixlength codeword. The number of bits required is

$$\lceil \log S \rceil + \lceil \log(W+1) \rceil + \lceil \log L \rceil$$

where S is the size of the search buffer, W is the size of the look-ahead buffer and L is the alphabet size.

6.5.2 LZSS

If we look at the basic LZ77 algorithm, it is obvious that we are wasting bits. It is unnecessary to send a pointer and a length if we don't find a matching sequence. Also, we only need to send a new symbol if we don't find a matching

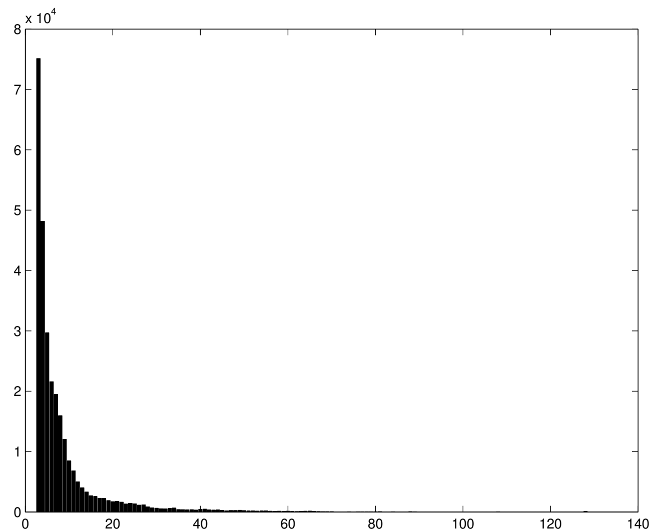
sequence. One way to make the algorithm more efficient is to use an extra flag bit that tells if we found a match or not. We either send $< 1, o, l >$ or $< 0, c >$. This variant of LZ77 is called LZSS.

6.5.3 Other improvements

Depending on buffer sizes and alphabet sizes it can be better to code short sequences as a number of single symbols instead of as a match.
In the beginning of the coding, before we have filled up the search buffer, we can use shorter codewords for o and l.
All o, l and c are not equally probable, so we can get even higher compression by coding them using variable length codes (eg Huffman codes, Golomb codes or arithmetic codes) instead of using fixed length coding.

6.5.4 Buffer sizes

In principle we get higher compression for larger search buffers. For practical reasons, typical search buffer sizes used are around $2^{15} - 2^{16}$, but there are variants of LZ77 that use even larger buffer sizes. We of course never need to use a buffer size that is larger than the actual data that we want to code.
Very long match lengths are usually not very common, so it is often enough to let the maximum match length (ie the look-ahead buffer size) be a couple of hundred symbols.
Example: LZSS coding of a text file, buffer size 32768, match lengths 3-130 (128 possible values). Histogram for match lengths:



6.5.5 LZ78

A dictionary of unique sequences is built. In the beginning the dictionary is empty, apart from index 0 that means "no match".
Every new sequence that is coded is sent as the tuple $< i, c >$ where i is the index in the dictionary for the longest matching sequence we found and c is the next symbol of the data that didn't match. The matching sequence plus the next symbol is added as a new word to the dictionary.
The number of bits required is

$$\lceil \log S \rceil + \lceil \log L \rceil$$

where S is the current size of the dictionary.
The decoder will build an identical dictionary during decoding, so we never have to transmit any explicit information about the dictionary content.
Depending on the application, we might also have a maximum allowed dictionary size, for instance if we have limited storage capacity and we are coding really long sequences.
What to do when the dictionary becomes full? There are a few alternatives:

- Throw away the dictionary and start over.
- Keep coding with the dictionary, but stop adding new words to the dictionary.
- As above, but only as long as the rate stays approximately constant. If the rate starts to increase, throw away the dictionary and start over. In this case we might have to add an extra symbol to the alphabet that informs the decoder to start over.

6.5.6 LZW

LZW is the most commonly encountered variant of LZ78.
Instead of sending a tuple $< i, c >$ we only send index i in the dictionary. For this to work, the starting dictionary must contain words of all single symbols in the alphabet.
Find the longest matching sequence in the dictionary and send the index as a new codeword. The matching sequence plus the next symbol is added as a new word to the dictionary.

7 Coding with distortion

7.1 Continuous alphabet sources

We have a signal $x_n,\ n = 1 \ldots N$ to code. The alphabet is a subset of the real numbers $\mathcal{A} \subseteq \mathbb{R}$. The alphabet can be continuous.
If we don't have the demand that the decoded signal should be exactly the same as the original signal we can get a lower data rate than if we have lossless

coding. Typically the signal is described using a smaller alphabet than the original signal uses (quantization).
In the case where the original alphabet is continuous, in general an infinite number of bits is required to describe the signal losslessly.
The more bits that are used, the closer to the original signal the decoded signal $\hat{x}_n$ will be.

7.2 Distortion measure

We need a measure of how much error we have in the decoded signal, the so called *distortion*.
The most common measure is a quadratic error measure, combined with averaging over the whole sequence

$$D = \frac{1}{N} \sum_{n=1}^{N} (x_n - \hat{x}_n)^2$$

This is the *mean square error* of the decoded sequence.
There are other distortion measure that can be used (such as mean absolute erroer), but in this course we will focus mainly on the men square error.
Often we want to consider the distortion (or noise power) relative to the signal power, the so called *signal to noise ratio* (SNR)

$$\sigma_x^2 = \frac{1}{N} \sum_{n=1}^{N} x_n^2$$

$$\text{SNR} = \frac{\sigma_x^2}{D}$$

SNR is usually expressed in dB

$$\text{SNR} = 10 \cdot \log_{10} \frac{\sigma_x^2}{D}$$

When coding still images and video we usually use the *peak-to-peak signal to noise ratio* (PSNR)

$$\text{PSNR} = 10 \cdot \log_{10} \frac{x_{pp}^2}{D}$$

where x_{pp} is the difference between the maximum and minum values of the signal.

For example, if the data to be coded is a grayscale image quantized to 8 bits, the signal can assume values between 0 and 255. The PSNR is then

$$\text{PSNR} = 10 \cdot \log_{10} \frac{255^2}{D}$$

7.3 Random signal models

A signal can be modelled as an amplitude continuous stationary random process X_n, with distribution function $F_X(x)$ and density function $f_X(x)$.

$$F_X(x) = Pr(X \leq x)$$

$$f_X(x) = \frac{d}{dx} F_X(x)$$

$$f_X(x) \geq 0 \ , \quad \forall x$$

$$\int_{-\infty}^{\infty} f_X(x) dx = 1$$

$$Pr(a \leq X \leq b) = F_X(b) - F_X(a) = \int_a^b f_X(x) dx$$

Mean value

$$m_X = E\{X_n\} = \int_{-\infty}^{\infty} x \cdot f_X(x) dx$$

Quadratic mean value

$$E\{X_n^2\} = \int_{-\infty}^{\infty} x^2 \cdot f_X(x) dx$$

Variance

$$\sigma_X^2 = E\{(X_n - m_x)^2\} = E\{X_n^2\} - m_x^2$$

In most of our cases we will use signal models with mean value 0. In those cases the variance is equal to the quadratic mean value.
The variance (or rather the quadratic mean value) is a measure of the signal power.

7.3.1 Examples of probability distributions

Uniform distribution

$$f_X(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

Mean value $m = \frac{a+b}{2}$, variance $\sigma^2 = \frac{(b-a)^2}{12}$
Gaussian distribution (normal distribution)

$$f_X(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-m)^2}{2\sigma^2}}$$

Laplace distribution

$$f_X(x) = \frac{1}{\sqrt{2}\sigma} e^{-\frac{\sqrt{2}|x-m|}{\sigma}}$$

7.3.2 Dependence and correlation

The dependence of the signal value in two times instances n och m is given by the twodimensional density function $f_{X_n X_m}(x_n, x_m)$.
If we can write this as a product $f_X(x_n) \cdot f_X(x_m)$ we say that the signal in the two time instances are *independent*.
A signal where all time instances are independent of each other is a *memoryless* signal or a *white* signal.
In most cases we will describe the dependence using the *correlation* $E\{X_n \cdot X_m\}$.
If $E\{X_n \cdot X_m\} = E\{X_n\} \cdot E\{X_m\}$ we say that the signal in the two time instances are *uncorrelated*. Independent signals are uncorrelated, but the reverse is not necessarily true.
Similar to the discrete case, we can define a Markov source of order k as a source with limited memory k steps back in time:

$$f(x_n | x_{n-1} x_{n-2} \ldots) = f(x_n | x_{n-1} \ldots x_{n-k})$$

Markov models are a little to complicated, so we often restrict ourselves to linear models, where the signal is modelled as white noise ϵ_n filtered by a linear filter. Depending on the type of filter, we can classify our linear models as AR (auto-regressive), MA (moving average) or ARMA (auto-regressive, moving average).
AR(N)

$$x_n = \sum_{i=1}^{N} a_i \cdot x_{n-i} + \epsilon_n$$

MA(M)

$$x_n = \sum_{j=1}^{M} b_i \cdot \epsilon_{n-j} + \epsilon_n$$

ARMA(N, M)

$$x_n = \sum_{i=1}^{N} a_i \cdot x_{n-i} + \sum_{j=1}^{M} b_i \cdot \epsilon_{n-j} + \epsilon_n$$

All of these linear models are Markov models, but all Markov models can not be modelled using linear models. Gaussian sources are an exception, any gaussian source can always be modelled as an AR model.

7.3.3 Auto correlation function

The correlation properties of the signal is usually expressed using the *auto correlation function (acf)*, which for a stationary process is given by

$$R_{XX}(k) = E\{X_n X_{n+k}\}$$

The auto correlation function is symmetric: $R_{XX}(-k) = R_{XX}(k)$.
We also have: $|R_{XX}(k)| \leq R_{XX}(0) = E\{X_n^2\}$.

For a memoryless (white) process we have

$$R_{XX}(k) = \sigma_X^2 \cdot \delta(k) = \begin{cases} \sigma_X^2 & k = 0 \\ 0 & \text{otherwise} \end{cases}$$

For an AR(1) process we have

$$R_{XX}(k) = a^{|k|} \cdot \sigma_X^2 \qquad (|a| < 1)$$

The auto correlation function can of course also be defined for multidimensional signals. For instance, for a twodimensional stationary random process $X_{i,j}$ the auto correlation function is given by

$$R_{XX}(k,l) = E\{X_{i,j} X_{i+k,j+l}\}$$

The auto correlation function is symmetric: $R_{XX}(-k,-l) = R_{XX}(k,l)$
We also have: $|R_{XX}(k,l)| \leq R_{XX}(0,0) = E\{X_{i,j}^2\}$

7.4 Distortion for random signals

For a random signal X_n that is coded and then decoded to $\hat{X}_n$, the distortion is given by

$$D = E\{(X - \hat{X})^2\} = \int_{-\infty}^{\infty} (x - \hat{x})^2 f_X(x) dx$$

The signal power is (given mean zero)

$$E\{X^2\} = \sigma_X^2 - (E\{X\})^2 = \sigma_X^2$$

and SNR as before

$$\text{SNR} = 10 \cdot \log_{10} \frac{\sigma_X^2}{D}$$

7.5 Theoretical limit

The rate-distortion function $R(D)$ for a source gives the theoretically lowest rate R we can use to code the source, on the condition that the maximum allowed distortion is D. Compare to the entropy rate limit for lossless coding.
For a white gaussian process with variance σ^2 we get

$$R(D) = \begin{cases} \frac{1}{2} \log \frac{\sigma^2}{D} & 0 < D \leq \sigma^2 \\ 0 & \text{otherwise} \end{cases}$$

Ie, if we allow a distortion that is larger than the variance of the process, we don't need to transmit any bits at all. The decoder can just set the decoded signal equal to the mean value at each time instance, which will give a distortion equal to the variance.
We can also se that $R \to \infty$ when $D \to 0$

For gaussian sources with memory, the rate-distortion function can be calculated from the power spectral density.

$$\Phi(\theta) = \mathcal{F}\{R_{XX}(k)\} = \sum_{k=-\infty}^{\infty} R_{XX}(k) \cdot e^{-j2\pi\theta k}$$

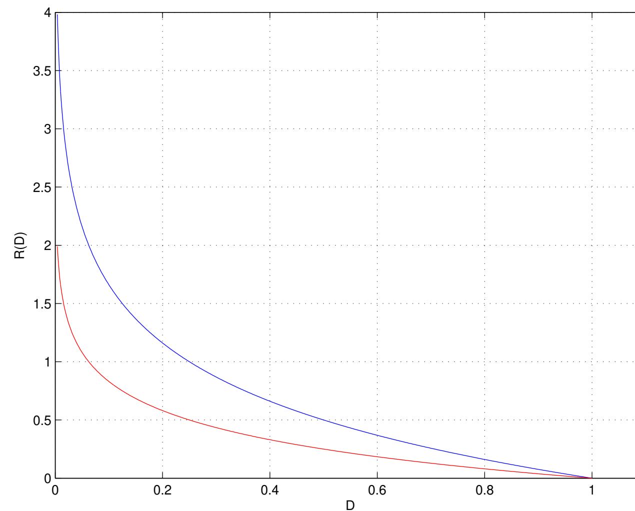
The rate-distortion function is given by

$$R(D) = \int_{-1/2}^{1/2} \max\{\frac{1}{2}\log\frac{\Phi(\theta)}{\lambda}, 0\}\ d\theta$$

where

$$D = \int_{-1/2}^{1/2} \min\{\lambda, \Phi(\theta)\}\ d\theta$$

The integration can of course be done over any interval of size 1, since the power spectral density is a periodic function.



$R(D)$ for an ideally bandlimited gaussian source (red), compared to the $R(D)$ for a memoryless/white gaussian source (blue). Both sources have variance 1. As D tends towards 0, $R(D)$ tends towards infinity for both sources.

7.6 Quantization

Quantization is a mapping from a continuous alphabet to a discrete alphabet (or mapping from a large discrete alphabet to a smaller one). After quantizing a signal we have a discrete signal, on which we can use our source coding methods (Huffman, arithmetic coding, et c.)
A general M level quantizer is specified by $M+1$ *decision borders* b_i ; $i = 0 \ldots M$ and M *reconstruction levels (or reconstruction points)* y_i ; $i = 1 \ldots M$.

The quantization operator $Q(x)$ is given by

$$Q(x) = y_i \text{ if } b_{i-1} < x \leq b_i$$

And the reconstructed signal is thus

$$\hat{x}_n = Q(x_n)$$

Sometimes it can be useful to see quantization and reconstruction as two separate operations instead of just one operation.
Quantization: $x \to j$ such that $b_{j-1} < x \leq b_j$
Reconstruction: $\hat{x} = y_j$
A sequence of x thus gives a sequence of indices j that can then be coded by a source coder
The receiver decodes the index sequence and the maps the indices to the corresponding reconstruction points.
Given a random signal model the distortion is

$$\begin{aligned} D &= E\{(X - \hat{X})^2\} = \\ &= \int_{-\infty}^{\infty} (x - Q(x))^2 f_X(x) dx = \\ &= \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} (x - y_i)^2 f_X(x) dx \end{aligned}$$

If no special source coding is used, ie if we just code the quantized signal using a fixed length code, the rate is

$$R = \lceil \log_2 M \rceil$$

7.7 Uniform quantization

In a uniform quantizer, the distance between two reconstruction points is constant

$$y_j - y_{j-1} = \Delta$$

Δ is the *stepsize* of the quantizer.
The reconstruction points are placed in the middle of their intervals, which means that all decision regions (apart from the end intervals in some cases) also are of the same size

$$b_i - b_{i-1} = \Delta$$

To simplify the calculations we can assume that the number of reconstruction points is given by $M = 2^R$ and that the quantizer is symmetric around the origin. The following results can easily be generalized to arbitrary M.
The reconstruction point belonging to the interval $[(j-1)\Delta, j\Delta]$ is

$$y_j = \frac{2j - 1}{2} \Delta$$

The simplest case is when the input distribution is uniform on the interval $[-A, A]$:

$$\Delta = \frac{2A}{M}$$

The distortion for uniform quantization of a uniform distribution:

$$D = \sum_{i=-M/2+1}^{M/2} \int_{(i-1)\Delta}^{i\Delta} (x - \frac{2i-1}{2}\Delta)^2 \frac{1}{2A} dx = M \cdot \frac{1}{2A} \cdot \frac{\Delta^3}{12} = \frac{\Delta^2}{12}$$

$$\sigma_X^2 = \frac{(2A)^2}{12} = \frac{\Delta^2 M^2}{12}$$

$$\begin{aligned} \text{SNR} &= 10 \cdot \log_{10} \frac{\sigma_X^2}{D} = 10 \cdot \log_{10} M^2 = \\ &= 10 \cdot \log_{10} 2^{2R} = 20 \cdot R \cdot \log_{10} 2 \approx 6.02 \cdot R \end{aligned}$$

For every bit added to the quantizer (ie for every doubling of the number of reconstruction points) we will get approximately 6 dB higher SNR.
For unlimited distributions (eg a gaussian distribution) the two end intervals will be infinitely large (in the calculations below we assume that that M is even and that the quantizer is symmetric around the origin).

$$\begin{aligned} D &= \sum_{i=-M/2+1}^{M/2} \int_{(i-1)\Delta}^{i\Delta} (x - \frac{2i-1}{2}\Delta)^2 f_X(x) dx + \\ &+ \int_{(M/2)\Delta}^{\infty} (x - \frac{M-1}{2}\Delta)^2 f_X(x) dx + \\ &+ \int_{-\infty}^{-(M/2)\Delta} (x - \frac{-M+1}{2}\Delta)^2 f_X(x) dx \end{aligned}$$

The last two terms are called the *overload distortion* of the quantizer.
To find the best choice of Δ (the one that minimizes the distortion) we have to solve

$$\frac{\partial}{\partial \Delta} D = 0$$

which in the general case is a hard problem. Normally we will have to find a numeric solution.
If the number of quantization levels M is large and Δ is chosen such that the overload distortion is small compared to the total distortion, the distortion is approximately

$$D \approx \frac{\Delta^2}{12}$$

7.8 Lloyd-Max quantization

How should we chose decision borders and reconstruction points to minimize the distortion? The answer will of course depend on the distribution of the signal. For a general quantizer we have

$$D = \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} (x - y_i)^2 f_X(x) dx$$

We want to find the the quantizer that minimizes the distortion D.

$$\frac{\partial}{\partial y_j} D = 0 \Rightarrow y_j = \frac{\int_{b_{j-1}}^{b_j} x \cdot f_X(x) dx}{\int_{b_{j-1}}^{b_j} f_X(x) dx}$$

The optimal placement of the reconstruction points is thus in the centroid of the probability mass in each interval.

$$\frac{\partial}{\partial b_j} D = 0 \Rightarrow b_j = \frac{y_{j+1} + y_j}{2}$$

The optimal placement of the decision borders is thus at the midpoints between the reconstruction points, ie we should always quantize to the closest reconstruction point.
Note that these demands are necessary but not sufficient.
Also note that y_j depends on b_{j-1} and b_j and that b_j depends on y_{j+1} and y_j.
Usually we can only find closed solutions for simple distributions and for a small number of reconstruction points.
If we can't find a closed solution, we have to find the solution numerically. One way of doing this is using Lloyd's algorithm:

1. Start with a set of reconstruction points $y_i^{(0)}$, $i = 1 \ldots M$. Set $k = 0$, $D^{(-1)} = \infty$ and choose a threshold ϵ.
2. Calculate optimal decision borders $b_j^{(k)} = \frac{y_{j+1}^{(k)} + y_j^{(k)}}{2}$
3. Calculate the distortion $D^{(k)} = \sum_{i=1}^{M} \int_{b_{i-1}^{(k)}}^{b_i^{(k)}} (x - y_i^{(k)})^2 f(x) dx$
4. If $D^{(k-1)} - D^{(k)} < \epsilon$ stop, otherwise continue
5. $k = k+1$. Calculate new optimal reconstruction points $y_j^{(k)} = \frac{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} x \cdot f(x) dx}{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} f(x) dx}$
6. Repeat from 2

Other stopping criteria (step 4 in the algorithm) can be used. For instance, stop when $D^{(k)}/D^{(k-1)} > 1 - \epsilon$.
Another simple variant would be to not have a stopping criteria and just run the algorithm for a fixed number of iterations.

7.9 Quantization followed by source coding

The probability $P(j)$ of being in interval j is

$$P(j) = \int_{b_{j-1}}^{b_j} f_X(x)dx$$

In the general case these probabilities are different for different intervals. We could thus get a lower rate than $\log M$ by using some form of source coding.
Finding the optimal quantizer given an allowed rate R after source coding is a hard problem. However, it can be shown that for sufficiently large R (fine quantization) the optimal quantizer is a uniform quantizer. Thus, if we are using some form of source coding, it is enough to use the simplest form of quantization.

7.10 Fine quantization

When we have *fine quantization*, ie when the number of quantization levels is large, the distortion is approximatively given by

$$D \approx c \cdot \sigma_X^2 \cdot 2^{-2R}$$

where σ_X^2 is the signal variance, R is the rate and c is a constant depending on the type of quantization and the distribution of the signal.
Gaussian distribution, Lloyd-Max quantization:

$$c = \frac{\pi\sqrt{3}}{2}$$

Gaussian distribution, uniform quantization, perfect source coding ($R = H(\hat{X})$):

$$c = \frac{\pi e}{6}$$

Uniform distribution, uniform quantization:

$$c = 1$$

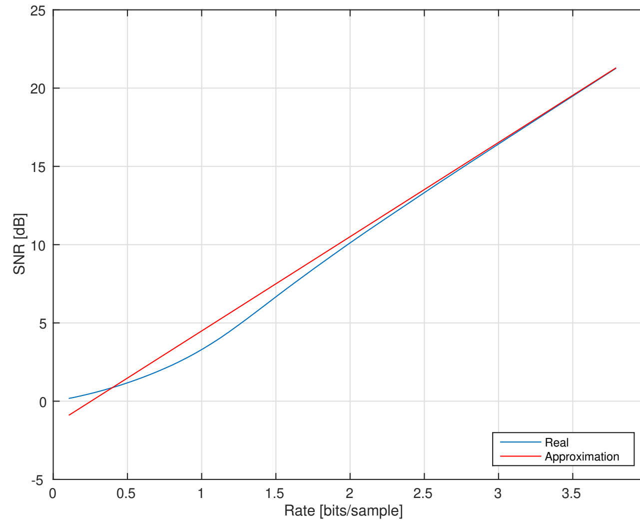
For fine quantization we have the approximations

$$\begin{cases} D \approx \frac{1}{12}\int_{-\infty}^{\infty} \Delta^2(x) f(x) dx \\ M \approx \int_{-\infty}^{\infty} \frac{1}{\Delta(x)} dx \end{cases}$$

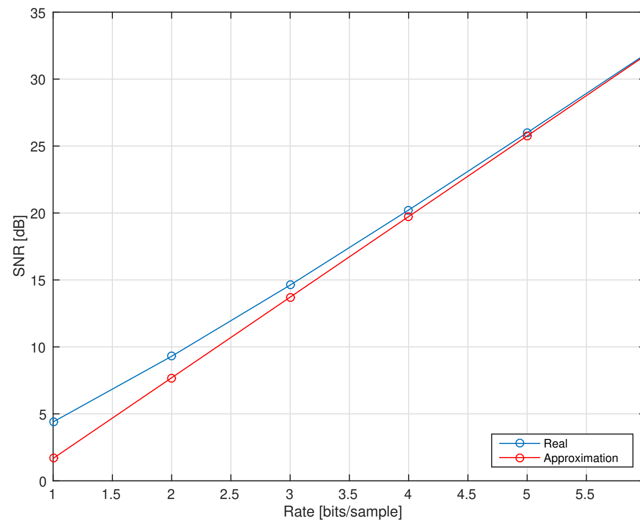
where $\Delta(x)$ is a function describing the size of the quantization interval at x and M is the resulting number of reconstruction points.
For fine Lloyd-Max quantization, we should choose

$$\Delta(x) = k \cdot (f(x))^{-\frac{1}{3}}$$

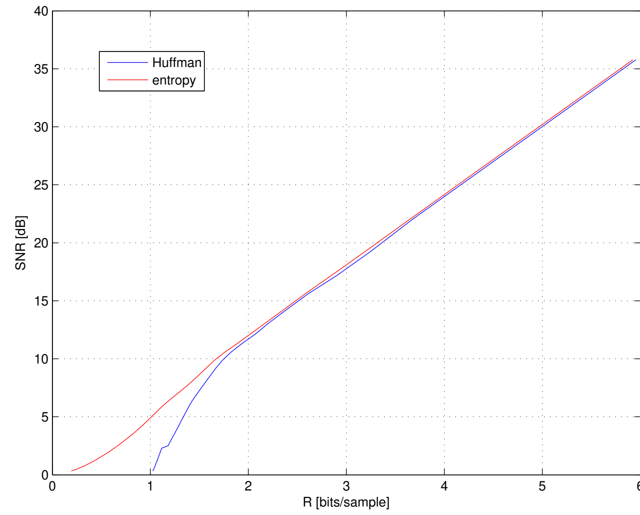
and the resulting rate will be $R = \log_2 M$.
Uniform quantization of Gaussian signal, followed by perfect source coding ($R = H(\hat{X})$). Real values compared to the approximation $D \approx \frac{\pi e}{6} \cdot \sigma_X^2 \cdot 2^{-2R}$.



Lloyd-Max quantization of Gaussian signal. Real values compared to the approximation $D \approx \frac{\pi\sqrt{3}}{2} \cdot \sigma_X^2 \cdot 2^{-2R}$.

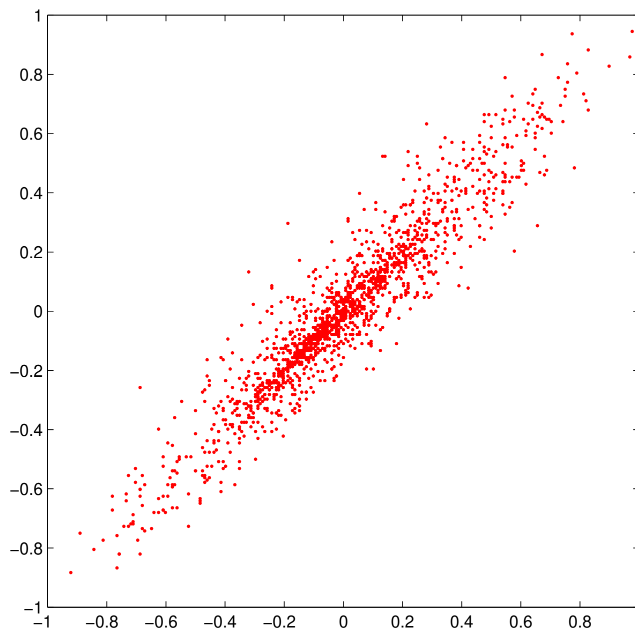


Real world signal example: A mono music file is coded using a uniform quantizer (midtread), followed by Huffman coding. The rate is varied by varying the quantizer stepsize. No limitation of the number of levels is done. For comparison, we have also estimated the entropy of the quantized signal.



7.11 Vector quantization

Consecutive samples in a signal are often strongly correlated.
Example: 4000 samples from a speech signal, plot $[x_i, x_{i+1}]$

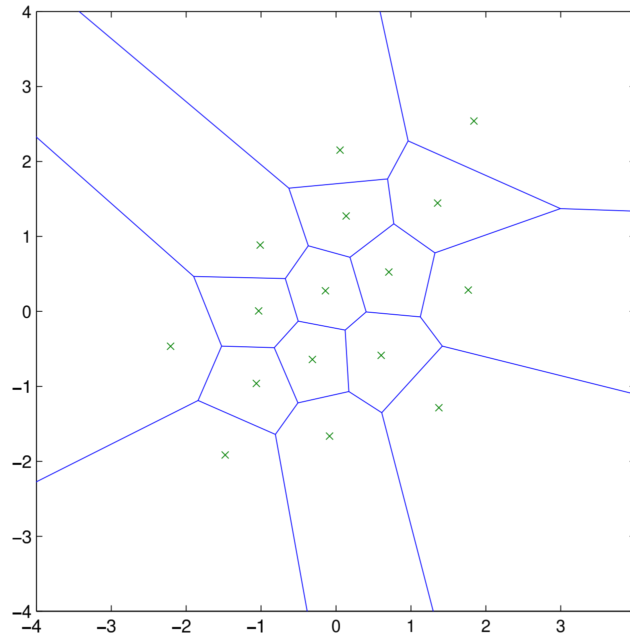


In order to utilize the dependence between samples we can use some form of source coding that uses the memory of the signal, eg extended Huffman coding or arithmetic coding where we let the interval division depend on previous symbols. We can also use the correlation between samples directly in the quantizer, by quantizing several samples at once, *vector quantization*.
View L samples from the source as a L-dimensional vector.

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_L \end{pmatrix}$$

The quantizer is defined by its reconstruction vectors $\mathbf{y}_i$ and decision regions V_i.

Example: In two dimensions it can look like



The set of reconstruction points $\{\mathbf{y}_i\}, i = 1 \dots M$ is usually called the *codebook*.
Distortion

$$D = \frac{1}{L} \sum_{i=1}^{M} \int_{V_i} |\mathbf{x} - \mathbf{y}_i|^2 \cdot f(\mathbf{x}) \; d\mathbf{x}$$

Compare this to the distortion when doing scalar quantization

$$D = \sum_{i=1}^{M} \int_{b_{i-1}}^{b_i} (x - y_i)^2 f(x) dx$$

A scalar quantizer is a onedimensional vector quantizer.
To code the M reconstruction points using a fixed length code we need $\lceil \log M \rceil$ bits. Since we're coding L samples at a time, the resulting rate is

$$R = \frac{\lceil \log M \rceil}{L} \quad \text{[bits/sample]}$$

Alternatively, given dimension L and rate R

$$M = 2^{RL}$$

It is of course possible to use some kind of source coding method when doing vector quantization, but that's usually not done.
We need to store the codebook both on the coder and the decoder side. It might also have to be transmitted as side information. If we are using L dimensions,

have a rate of R bits per sample and each element of the vectors is stored using b bits, we need

$$2^{RL} \cdot L \cdot b$$

bits to store the whole codebook. The required storage space thus grows very quickly when we increase the dimension.
If there is no structure in the codebook we have to compare each vector that we quantize with *all* the vectors in the codebook in order to find the closest one. When the dimension L is large this will take a lot of time.

7.12 The LBG algorithm

Usually we don't know the probability density function $f(\mathbf{x})$ of the source. One way would be to estimate the density function from a long sequence from the source (training data).
Instead of doing this, Lloyd's algorithm can be modified to use the training data directly.
This variant of the algorithm is usually called the *LBG algorithm* or *K-means*.

1. Begin with a starting codebook $\mathbf{y}_i^{(0)},\ i = 1 \ldots M$ and a set of training vectors $\mathbf{x}_n,\ n = 1 \ldots N$. Let $k = 0$, $D^{(-1)} = \infty$ and choose a threshold ϵ.

2. Determine optimal decision regions

$$V_i^{(k)} = \{\mathbf{x}_n : |\mathbf{x_n} - \mathbf{y}_i|^2 < |\mathbf{x_n} - \mathbf{y}_j|^2\ \forall j \neq i\}$$

3. Calculate the distortion $D^{(k)} = \sum_{i=1}^{M} \sum_{\mathbf{x}_n \in V_i^{(k)}} |\mathbf{x_n} - \mathbf{y}_i^{(k)}|^2$

4. If $D^{(k-1)} - D^{(k)} < \epsilon$ stop, otherwise continue.

5. $k = k + 1$. Determine new optimal reconstruction points as the average of all vectors in each $V_i^{(k-1)}$.

6. Repeat from 2.

Just as with Lloyd's algorithm, we can use alternative stopping conditions.
Depending on how we choose the starting codebook we can get different resulting codebooks. A few variants:

- Choose M arbitrary vectors.
- Choose M vectors from the training data.
- Generate several random starting codebooks and choose the one that gives the lowest distortion.

- PNN (Pairwise Nearest Neighbour).
  Start with each training vector as a reconstruction point. In each step remove the two vectors that are closest to each other and replace them with the average of the two vectors. Repeat until we have M vectors.

What do we do if a region becomes empty during a step in the LBG algorithm? Replace the reconstruction vector that has an empty region with a new vector. A few variants:

1. Choose the new reconstruction point randomly from the region that has the highest number of training data.

2. Choose the new reconstruction point randomly from the region that has the largest distortion.

3. Optimize a two level quantizer in the region that has the largest distortion.

Method 3 is more computationally intensive, but it doesn't give any benefits compared to the other methods.

8 Linear predictive coding

Samples close to each other in a signal are often strongly correlated. To get an efficient coding it's always a good idea to utilize the correlation (memory).
A general *predictive coder* utilizes the signal samples up to N steps back in time to make a prediction (guess) of what the signal sample at the present time should be and then codes the difference between the prediction and the real value.
We will concentrate on *linear predictors*, where the prediction is a linear combination of old sample values. This corresponds to having an AR model of the source.
Idea: We guess (predict) the signal value at time n as a linear combination of the previous N sample values.

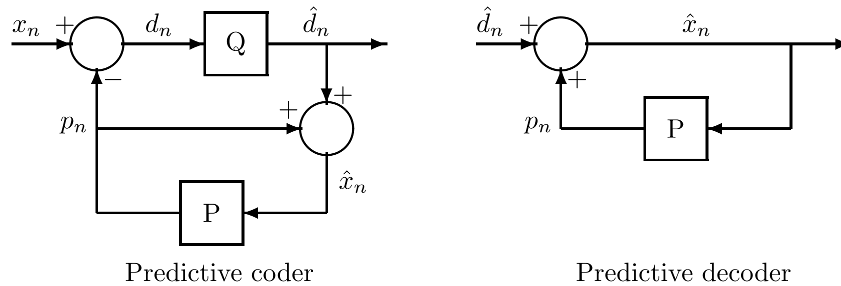
$$\begin{aligned} p_n &= a_1 x_{n-1} + a_2 x_{n-2} + \ldots + a_N x_{n-N} = \\ &= \sum_{i=1}^{N} a_i x_{n-i} \end{aligned}$$

The difference between the real value and the predicted value, the *prediction error*, $d_n = x_n - p_n$ is quantized, possibly source coded and then sent to the receiver. The receiver reconstructs d_n, calculates p_n and can then recreate x_n.
Unfortunately, this will not work in practice!
The problem is that the receiver can only recreate a distorted version $\hat{d}_n$ of the prediction error and therefore only a distorted version $\hat{x}_n$ of the signal.
In order for the predictive coder to work, the coder must perform the same prediction that the decoder can perform.

The prediction must be done from the reconstructed signal $\hat{x}_n$ instead of from the original signal.

$$\begin{aligned} p_n &= a_1\hat{x}_{n-1} + a_2\hat{x}_{n-2} + \ldots + a_N\hat{x}_{n-N} = \\ &= \sum_{i=1}^{N} a_i\hat{x}_{n-i} \end{aligned}$$

The prediction error d_n is quantized and transmitted. Both coder and decoder recreate $\hat{d}_n$ and $\hat{x}_n = p_n + \hat{d}_n$.
Block schedule of a linear predictive coder and decoder:



Predictive coder   Predictive decoder

8.1 Optimization of predictor coefficients

How should we choose the predictor coefficients a_i?
Given a rate R we want to minimize the distortion

$$D = E\{(x_n - \hat{x}_n)^2\} = E\{((p_n + d_n) - (p_n + \hat{d}_n))^2\} = E\{(d_n - \hat{d}_n)^2\}$$

The quantization makes it hard to calculate optimal a_i exactly. If we assume *fine quantization*, ie that the number of quantization levels is large, we can do the approximation

$$\hat{x}_n \approx x_n$$

ie we will do our calculations as if we predicted from the original signal.
Using fine quantization we also have the approximation

$$D \approx c \cdot \sigma_d^2 \cdot 2^{-2R}$$

where σ_d^2 is the variance of the prediction error and c depends on what type of quantization we're doing and the distribution of d_n. We can thus minimize the distortion by minimizing the variance of the prediction error.
Variance of the prediction error:

$$\begin{aligned} \sigma_d^2 &= E\{d_n^2\} = E\{(x_n - p_n)^2\} = \\ &= E\{(x_n - \sum_{i=1}^{N} a_i\hat{x}_{n-i})^2\} \approx \\ &\approx E\{(x_n - \sum_{i=1}^{N} a_i x_{n-i})^2\} \end{aligned}$$

Differentiate with respect to each a_j and set equal to 0, which gives us N equations

$$\frac{\partial}{\partial a_j}\sigma_d^2 = -2 \cdot E\{(x_n - \sum_{i=1}^{N} a_i x_{n-i}) \cdot x_{n-j}\} = 0$$

This can be written in the form of a matrix equation

$$\mathbf{RA} = \mathbf{P}$$

where

$$\mathbf{R} = \begin{bmatrix} R_{XX}(0) & R_{XX}(1) & \cdots & R_{XX}(N-1) \\ R_{XX}(1) & R_{XX}(0) & \cdots & R_{XX}(N-2) \\ \vdots & \vdots & \ddots & \cdots \\ R_{XX}(N-1) & R_{XX}(N-2) & \cdots & R_{XX}(0) \end{bmatrix}$$

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix}, \quad \mathbf{P} = \begin{bmatrix} R_{XX}(1) \\ R_{XX}(2) \\ \vdots \\ R_{XX}(N) \end{bmatrix}$$

and where $R_{XX}(k) = E\{x_n \cdot x_{n+k}\}$ is the auto correlation function of x_n.
The solution can be found as

$$\mathbf{A} = \mathbf{R^{-1}P}$$

For the optimal predictor $\mathbf{A}$ we get the prediction error variance

$$\sigma_d^2 = R_{XX}(0) - \mathbf{A^T P}$$

NOTE: This formula can not be used for other choices of prediction coefficients.

8.2 Prediction gain

With fine quantization the distortion and signal to noise ratio are given approximately as

$$D_p \approx c \cdot \sigma_d^2 \cdot 2^{-2R}, \quad \mathrm{SNR}_p = 10 \cdot \log_{10} \frac{\sigma_x^2}{D_p}$$

where σ_x^2 is the variance of the original signal.
If we had quantized the original signal directly we would have gotten

$$D_o \approx c \cdot \sigma_x^2 \cdot 2^{-2R}, \quad \mathrm{SNR}_o = 10 \cdot \log_{10} \frac{\sigma_x^2}{D_o}$$

The difference is referred to as the *prediction gain*

$$\mathrm{SNR}_p - \mathrm{SNR}_o = 10 \cdot \log_{10} \frac{D_o}{D_p} \approx 10 \cdot \log_{10} \frac{\sigma_x^2}{\sigma_d^2}$$

8.3 Signals with nonzero mean

What do we do if we have a signal with a mean value $m \neq 0$?

1. If the signal mean value is small compared to the variance we can just use linear prediction.

2. If not, we can create a new signal $y_n = x_n - m$, construct a linear predictor for y_n and send m as side information.

3. Alternatively we can construct an *affine predictor*

$$p_n = \sum_{i=1}^{N} a_i x_{n-i} + a_0$$

   Disregarding the quantization this will give the same result as alternative 2.

8.4 Multidimensional predictors

We can of course generalize linear prediction to work with multidimensional signals, like images.
For example, if we have an image signal x_{ij} and want to do prediction from the pixel to the left of and the pixel above the current pixel

$$p_{ij} = a_1 \cdot x_{i,j-1} + a_2 \cdot x_{i-1,j}$$

The optimal predictor is then given by the solution to the equation system

$$\begin{bmatrix} E\{x_{i,j-1}^2\} & E\{x_{i,j-1} \cdot x_{i-1,j}\} \\ E\{x_{i,j-1} \cdot x_{i-1,j}\} & E\{x_{i-1,j}^2\} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} E\{x_{i,j} \cdot x_{i,j-1}\} \\ E\{x_{ij} \cdot x_{i-1,j}\} \end{bmatrix}$$

or, expressed using the auto correlation function

$$\begin{bmatrix} R_{XX}(0,0) & R_{XX}(1,-1) \\ R_{XX}(1,-1) & R_{XX}(0,0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} R_{XX}(0,1) \\ R_{XX}(1,0) \end{bmatrix}$$

8.5 Lossless predictive coding

Linear predictive coding can also be used for lossless coding, by removing the quantization.
Assuming that we have an integer input signal, we must make sure that the predictor also produces integers, by using some form of rounding when the predictor coefficients are not integers.
As an example we have lossless JPEG, which can use the predictors

1. $p_{ij} = I_{i-1,j}$

2. $p_{ij} = I_{i,j-1}$

3. $p_{ij} = I_{i-1,j-1}$
4. $p_{ij} = I_{i,j-1} + I_{i-1,j} - I_{i-1,j-1}$
5. $p_{ij} = \lfloor I_{i,j-1} + (I_{i-1,j} - I_{i-1,j-1})/2 \rfloor$
6. $p_{ij} = \lfloor I_{i-1,j} + (I_{i,j-1} - I_{i-1,j-1})/2 \rfloor$
7. $p_{ij} = \lfloor (I_{i,j-1} + I_{i-1,j})/2 \rfloor$

9 Colour images

When coding images we usually don't use the RGB colour space. Instead the image is described in another colour space, where the pixel values are given using a *luminance* (or *luma*) component (called Y), that tells us how bright the pixel is (ie basically a grayscale signal) and two *chrominance* (or *chroma*) components (called Cb and Cr) that tells the actual colour of the pixel.
The chrominance components can often be downsampled to a lower resolution, without a human observer noticing any reduction in image quality.
There are many variants of luminance-chrominance colour spaces, but they are rather similar to each other.

9.1 Converting from RGB to YCbCr

Suppose that E_R, E_G and E_B are analog values between 0 and 1 that describe how much red, green and blue there is in a pixel (given eight bit quantization we have $E_R = R/255$, $E_G = G/255$ and $E_B = B/255$). A typical conversion to luminance-chrominance is then given by

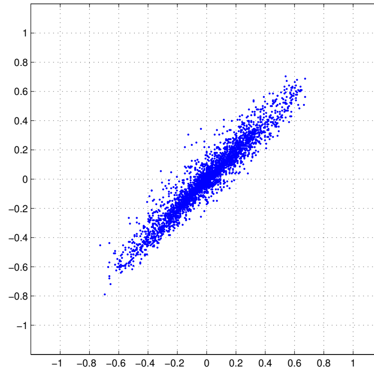
$$\begin{cases} E_Y &= 0.299 \cdot E_R + 0.587 \cdot E_G + 0.114 \cdot E_B \\ E_{Cb} &= -0.169 \cdot E_R - 0.331 \cdot E_G + 0.500 \cdot E_B \\ E_{Cr} &= 0.500 \cdot E_R - 0.419 \cdot E_G - 0.081 \cdot E_B \end{cases}$$

where E_Y is between 0 and 1 and E_{Cb} and E_{Cr} are between -0.5 and 0.5. There are several different variantions of this where the coefficients are different. It is outside the scope of this course to delve further into colour image theory.
Conversion to 8-bit integer values can be done by

$$\begin{cases} Y &= \lfloor 219 \cdot E_Y \rfloor + 16 \\ Cb &= \lfloor 224 \cdot E_{Cb} \rfloor + 128 \\ Cr &= \lfloor 224 \cdot E_{Cr} \rfloor + 128 \end{cases}$$

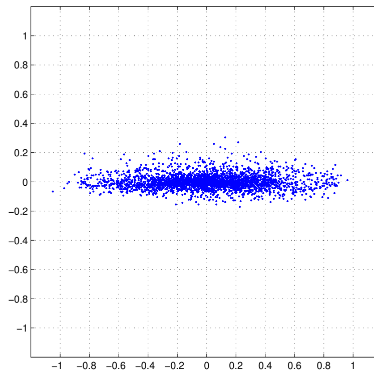
10 Transform coding

Consider pairs of consecutive samples from a speech signal.



Consecutive sampels are strongly correlated. If we quantize the samples scalarly, the quantizer for both sampels must be able to handle large variations in the signal values. If we instead describe the pairs in a new basis (another coordinate system) we remove the dependance between the samples and make it easier to do scalar quantization.

New basis vectors: $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$, $\frac{1}{\sqrt{2}}\begin{pmatrix} -1 \\ 1 \end{pmatrix}$.



Now we can use different quantizers for the different coordinates, and only one of the quantizers needs to handle large signal values. This will mean that we can get a more efficient coding (lower rate at the same distortion, or lower distortion at the same rate).

10.1 Main idea

1. Split the signal into blocks of size N (or $N \times N$ if the signal is twodimensional). Transform the blocks using a suitable, reversible transform to a new sequence.
2. Quantize the transform components.
3. Use some kind of source coding on the quantized transform components (fixed length coding, Huffman, arithmetic coding et c.)

10.2 Linear transforms

Block of N samples from the signal $\{x_n\}_{n=0}^{N-1}$ are transformed to a block $\{\theta_n\}_{n=0}^{N-1}$

$$\theta_n = \sum_{i=0}^{N-1} a_{n,i} \cdot x_i$$

All the components of x have the same statistics (variance et c.) but the components of θ will have different statistics, depending on position n.

The inverse transform, that recreates $\{x_n\}$ from $\{\theta_n\}$ is given by

$$x_n = \sum_{i=0}^{N-1} b_{n,i} \cdot \theta_i$$

The transform and the inverse transform can be written in matrix form as

$$\bar{\theta} = \mathbf{A} \cdot \bar{\mathbf{x}} \quad ; \quad \bar{\mathbf{x}} = \mathbf{B} \cdot \bar{\theta}$$

where

$$\bar{\mathbf{x}} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \end{pmatrix} \quad ; \quad \bar{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{N-1} \end{pmatrix}$$

and the matrix element at position (i,j) is given by

$$[\mathbf{A}]_{i,j} = a_{i,j} \quad ; \quad [\mathbf{B}]_{i,j} = b_{i,j}$$

The matrices $\mathbf{A}$ and $\mathbf{B}$ are the inverses of each other, ie $\mathbf{B} = \mathbf{A}^{-1}$.

10.3 Orthonormal transforms

We are usually only interested in *orthonormal* transforms, ie transforms where $\mathbf{B} = \mathbf{A}^{-1} = \mathbf{A}^T$.
Orthonormal transforms are energy preserving, ie the sum of the squares of the transformed signal is equal to the sum of the squares of the original signal

$$\begin{aligned} \sum_{i=0}^{N-1} \theta_i^2 &= \bar{\theta}^T \bar{\theta} \\ &= (\mathbf{A}\bar{\mathbf{x}})^T \mathbf{A}\bar{\mathbf{x}} \\ &= \bar{\mathbf{x}}^T \mathbf{A}^T \mathbf{A}\bar{\mathbf{x}} \\ &= \bar{\mathbf{x}}^T \bar{\mathbf{x}} = \sum_{i=0}^{N-1} x_i^2 \end{aligned}$$

This is often referred to as *Parseval's identity*.

10.4 The transform as a basis change

The transform can be seen as describing the signal in another basis, ie as a linear combination of new basis vectors

$$\begin{aligned}
\bar{\mathbf{x}} &= \mathbf{A}^T\bar{\theta} \\
&= \begin{pmatrix} a_{00} & \cdots & a_{N-1,0} \\ \vdots & \ddots & \vdots \\ a_{0,N-1} & \cdots & a_{N-1,N-1} \end{pmatrix} \begin{pmatrix} \theta_0 \\ \vdots \\ \theta_{N-1} \end{pmatrix} \\
&= \theta_0 \begin{pmatrix} a_{00} \\ \vdots \\ a_{0,N-1} \end{pmatrix} + \ldots + \theta_{N-1} \begin{pmatrix} a_{N-1,0} \\ \vdots \\ a_{N-1,N-1} \end{pmatrix}
\end{aligned}$$

The rows of the transform matrix (or the columns in the inverse transform matrix) are the basis vectors of the new basis.

10.5 Transform properties

Some desirable properties of the transform:

- The transform should concentrate the signal energy to as few components as possible.
- The transform should decorrelate the transform components, ie if possible we want $E\{\theta_i \cdot \theta_j\} = 0,\ i \neq j$. This means that we remove all dependance (memory) between the transform components.
- The transform should be robust with respect to changes in source statistics.
- The transform should be simple and fast to calculate.

All of these properties can not be found in a single type of transform.

10.6 The Karhunen-Loève-transform (KLT)

The KLT is a transform that will completely decorrelate the transform components and also give maximal energy concentration.
Assuming we have an input signal that is modelled as a stationary random process X_n with mean zero and auto correlation function $R_{XX}(k) = E\{X_n X_{n+k}\}$.
Given a block size of N, we have signal vectors

$$\bar{\mathbf{x}} = \begin{pmatrix} X_n \\ X_{n+1} \\ \vdots \\ X_{n+N-1} \end{pmatrix}$$

The *correlation matrix* $\mathbf{R}_X$ is the matrix

$$\mathbf{R}_X = E\{\bar{\mathbf{x}}\bar{\mathbf{x}}^T\}$$

The correlation matrix can be expressed using the auto correlation function

$$\mathbf{R}_X = \begin{pmatrix} R_{XX}(0) & R_{XX}(1) & \cdots & R_{XX}(N-1) \\ R_{XX}(1) & R_{XX}(0) & \cdots & R_{XX}(N-2) \\ \vdots & \vdots & \ddots & \cdots \\ R_{XX}(N-1) & R_{XX}(N-2) & \cdots & R_{XX}(0) \end{pmatrix}$$

The correlation matrix $\mathbf{R}_\theta$ of the transformed signal, given a transform $\mathbf{A}$, is given by

$$\mathbf{R}_\theta = E\{\bar{\theta}\bar{\theta}^T\} = E\{\mathbf{A}\bar{\mathbf{x}}(\mathbf{A}\bar{\mathbf{x}})^T\} = \mathbf{A}\mathbf{R}_X\mathbf{A}^T$$

If we want the transform to decorrelate the signal, ie diagonalize $\mathbf{R}_\theta$ (all values zero except for the main diagonal), we should choose the basis vectors (rows of $\mathbf{A}$) as the normalized eigenvectors of $\mathbf{R}_X$.
For a KLT, the variances of the transform components will be equal to the eigenvalues of the signal correlation matrix.
In addition to decorrelating the source, the KLT will also be the transform that gives the maximum energy concentration to a few transform components. This is the same as saying that the KLT is the transform that minimizes the geometric mean of the transform component variances

$$(\prod_{i=0}^{N-1} \sigma_i^2)^{1/N}$$

A disadvantage of the KLT is that it is signal dependent, so it has to be transmitted as side information. There is usually also no fast way to perform the transform.

10.7 The discrete cosine transform (DCT)

The transform matrix $\mathbf{C}$ is given by

$$[\mathbf{C}]_{ij} = \begin{cases} \sqrt{\frac{1}{N}} & ;\ i = 0 \\ \\ \sqrt{\frac{2}{N}} \cos \frac{(2j+1)i\pi}{2N} & ;\ i = 1, \ldots, N-1 \end{cases}$$

The DCT is a close relative of the discrete fourier transform (DFT). There are fast ways of doing a DCT, in the same way that there are fast fourier transforms (FFT).
The DCT will usually be very close to a KLT for sources where there is a high correlation between consecutive samples, which includes most natural audio and image sources.

To be more precise, there at least 8 slightly different versions of the DCT. The above transform (a type II DCT) is the most commonly used version, for instance in the JPEG and MPEG standards, so it is usually referred to just as "the DCT".

10.8 The discrete Walsh-Hadamard transform

A *Hadamard matrix* H_N of size $N = 2^k$ is given by

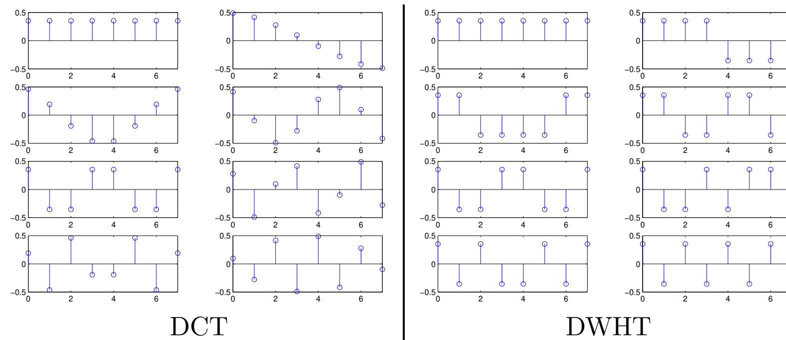
$$H_N = \begin{pmatrix} H_{N/2} & H_{N/2} \\ H_{N/2} & -H_{N/2} \end{pmatrix}$$

where $H_1 = 1$.

The transform matrix in DWHT is a Hadamard matrix, normalized with a factor $1/\sqrt{N}$. Usually the rows of the matrix are sorted in frequency order.
Since the transform matrix, apart from the normalizing factor, only contains ± 1, the transform is easy to calculate.
However, the DWHT does not give very good energy concentration, and since the basis vectors are very "square", any quantization errors will be very visible or audible.

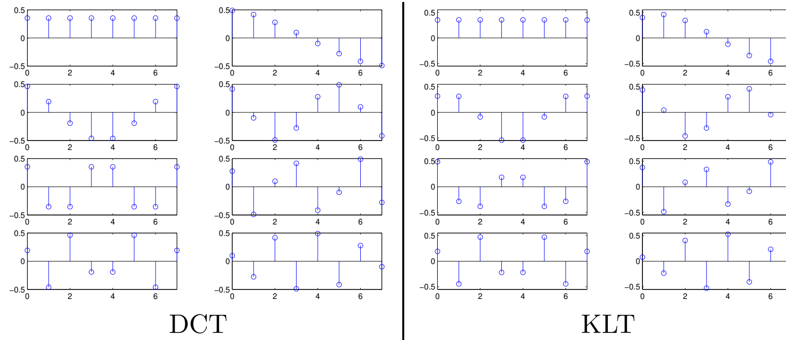
10.9 Comparison between DCT and DWHT

Basis vectors for 8-point DCT and DWHT



10.10 Comparison between DCT and KLT

Basis vectors for 8-point DCT and a KLT adapted to a music signal.



10.11 Twodimensional signals

For a twodimensional signal (eg an image) we usually take blocks of size $N \times N$ to transform.
In general we can view this block as a vector of N^2 samples and use a transform matrix of size $N^2 \times N^2$.
Usually a *separable* transform is used. We then consider the block as a matrix $\mathbf{X}$ instead of a vector. A onedimensional transform is applied first to the rows of $\mathbf{X}$ and then to the columns (or the other way, the order will not matter). The resultat is a matrix $\boldsymbol{\Theta}$ of transform components

$$\boldsymbol{\Theta} = \mathbf{A}\mathbf{X}\mathbf{A}^T$$

The inverse transform is given by

$$\mathbf{X} = \mathbf{A}^T\boldsymbol{\Theta}\mathbf{A}$$

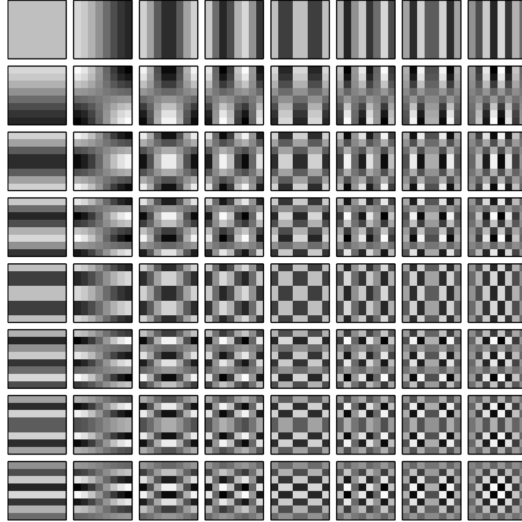
We can view the block $\mathbf{X}$ as a linear combination of new basis matrices α_{ij} given by

$$\alpha_{ij} = \bar{\mathbf{a}}_i^T \bar{\mathbf{a}}_j$$

where $\bar{\mathbf{a}}_i$ and $\bar{\mathbf{a}}_j$ are the i:th and j:th rows of $\mathbf{A}$.

$$\mathbf{X} = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} [\boldsymbol{\Theta}]_{ij} \cdot \alpha_{ij}$$

A separable transform can always be written as a general transform applied to a vector of N^2 elements, but the reverse is not true.
For a more general description, we can use rectangular blocks of size $N \times M$ instead of square blocks. This is used in some video coding standards.
Basis matrices for an 8×8 DCT:



10.12 Block size

A large block size N will give better concentration of the energy, but the transform will be more complicated to calculate. It will also be harder to adapt the coder if the source has different statistics in different parts (eg foreground and background in an image or different parts of a music signal). Large transforms can also give rise to more noticable quantization errors.
Typical block size for image coding is 8×8 pixels (JPEG, MPEG, DV)
Typical block sizes for audio coding are 256-2048 samples (Dolby Digital, MPEG AAC, Ogg Vorbis)

10.13 Distortion

For orthonormal transforms the distortion in the transform domain will be the same as the distortion in the signal domain. This is because an orthonormal transform preserves the length of all vectors.
Assume that we quantize and reconstruct the transform vector to $\hat{\theta}$ and inverse transform to the reconstructed vector $\hat{\mathbf{x}}$. The distortion is then

$$\begin{aligned}
D &= \frac{1}{N}||\bar{\mathbf{x}} - \hat{\mathbf{x}}||^2 = \frac{1}{N}(\bar{\mathbf{x}} - \hat{\mathbf{x}})^T(\bar{\mathbf{x}} - \hat{\mathbf{x}}) \\
&= \frac{1}{N}(\mathbf{A}^T\bar{\theta} - \mathbf{A}^T\hat{\theta})^T(\mathbf{A}^T\bar{\theta} - \mathbf{A}^T\hat{\theta}) \\
&= \frac{1}{N}(\bar{\theta} - \hat{\theta})^T\mathbf{A}\mathbf{A}^T(\bar{\theta} - \hat{\theta}) \\
&= \frac{1}{N}(\bar{\theta} - \hat{\theta})^T(\bar{\theta} - \hat{\theta}) = \frac{1}{N}||\bar{\theta} - \hat{\theta}||^2
\end{aligned}$$

The same reasoning also applies for random signals, with expectation.

10.14 Zonal coding

In zonal coding (or zonal sampling) we split the transformed vector (or block) into a number of parts (zones). All coefficients in the same zone are coded using the same quantizer and the same source coder.
If we have K zones and zone j has N_j coefficients, we of course have $N_1 + N_2 + \ldots + N_K = N$.
Given that zone j has the rate R_j bits/sample, the average rate R for the whole coder is

$$R = \frac{\sum_{j=1}^{K} N_j \cdot R_j}{N}$$

The zone division, quantization and source coding can be fixed for all blocks, or they can be changed when needed. This gives us a better possibility to adapt the coder to a varying signal, but it also means that we get more side information to transmit.
From now on, assume that we let each transform coefficient be its own zone (ie all $N_j = 1$) and that we keep the coders fixed and don't switch coders between blocks.
Transform component k is quantized and coded to R_k bits, with a resulting distortion D_k. Assuming fine quantization, the distortion can be approximated by

$$D_k \approx c_k \cdot \sigma_k^2 \cdot 2^{-2R_k}$$

We want to find the bit allocation that minimizes the average distortion

$$D = \frac{1}{N} \sum_{k=0}^{N-1} D_k \approx \frac{1}{N} \sum_{k=0}^{N-1} c_k \cdot \sigma_k^2 \cdot 2^{-2R_k}$$

under the condition that the average rate is fixed

$$R = \frac{1}{N} \sum_{i=0}^{N-1} R_k$$

For simplicity we assume that all transform components have the same type of distribution and that we use the same type of quantization and source coding. Then all c_k are equal. Lagrange optimization gives (see Sayood for details)

$$R_k = R + \frac{1}{2} \log_2 \frac{\sigma_k^2}{(\prod_{i=0}^{N-1} \sigma_i^2)^{1/N}}$$

Note that this can give some components a negative rate. In that case we set the rate for those components to 0, and redo the bit allocation for the other components, such that the average rate is still R.
For some types of quantization and coding (Lloyd-Max quantization, quantization followed by fixed length coding) we might have the condition that rates should be integers.

For optimal bit allocation the distortion for each component (given that our fine quantization assumption still holds) is

$$\begin{aligned} D_k &\approx c \cdot \sigma_k^2 \cdot 2^{-2R_k} = \\ &= c \cdot \sigma_k^2 \cdot 2^{-2R - \log_2 \frac{\sigma_k^2}{(\prod_{i=0}^{N-1} \sigma_i^2)^{1/N}}} = \\ &= c \cdot \sigma_k^2 \cdot \frac{(\prod_{i=0}^{N-1} \sigma_i^2)^{1/N}}{\sigma_k^2} \cdot 2^{-2R} = \\ &= c \cdot (\prod_{i=0}^{N-1} \sigma_i^2)^{1/N} \cdot 2^{-2R} \end{aligned}$$

We will thus get the same distortion for each transform component. The average distortion will of course also take this value.

10.15 Transform coding gain

One way of measuring how good a certain transform is, is the *transform coding gain*. The transform gives the average distortion and signal to noise ratio

$$D_t = \frac{1}{N} \sum_{i=0}^{N-1} D_i, \quad \mathrm{SNR}_t = 10 \cdot \log_{10} \frac{\sigma_x^2}{D_t}$$

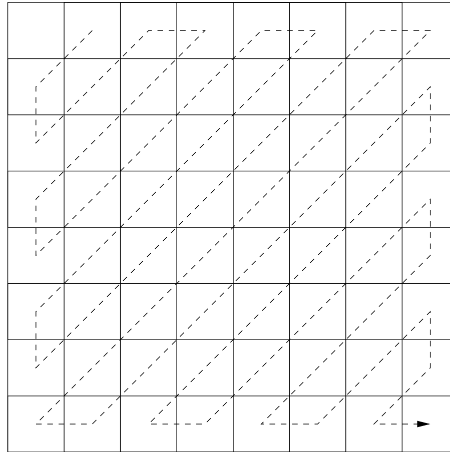
where σ_x^2 is the variance of the original signal.
Coding without transform to the same rate gives distortion D_o and signal to noise ratio SNR_o. The transform coding gain is the difference

$$\mathrm{SNR}_t - \mathrm{SNR}_o = 10 \cdot \log_{10} \frac{D_o}{D_t} \approx 10 \cdot \log_{10} \frac{\sigma_x^2}{(\prod_{i=0}^{N-1} \sigma_i^2)^{1/N}}$$

The final approximation holds when we have fine quantization and optimal bit allocation.

10.16 Threshold coding

For each transform block we tell which transform components that have a magnitude over a threshold value. Only these components are quantized and coded, the rest are set to zero. Which components that are above the threshold needs to be transmitted as side information for every block.
Often runlength coding of the zeros are used for this side information.
For twodimensional transforms a zigzag scanning of the components are usually performed, to get a onedimensional signal, before the runlength coding.
In practice, usually no separate thresholding is done. Instead, the components that are quantized to zero are the ones that are considered to be below the threshold.
Zigzag scanning for 8×8 transform. The DC level in the upper left corner is usually treated separately.



10.17 JPEG

JPEG is an ISO standard (1990) for still image coding. This is still the most common way of doing lossy image coding.
Uses DCT of size 8×8 pixels.
1-4 colour components.
Either 8 or 12 bits per colour components. The common file formats JFIF and EXIF only allow 8 bits per component.
No explicit thresholding, uniform quantization. The step size can be choosen freely for each of the 64 transform components. Typically the high frequency components are quantized harder than the low frequency components.
The source coding is either runlength coding of zeros followed by Huffman coding, or arithmetic coding. Since the arithmetic coder in the standard was protected by several patents, only Huffman coding is used in practice.
Image quality is controlled by the choice of the step sizes of the 64 quantizers. Since we can choose them freely and independently of each other, it might be hard to find the best choice of step sizes for a given average rate or a given average distortion.
In order to simplify, most JPEG coders (eg digital cameras) only let the user choose one quality parameter. Each quality parameter will correspond to a pre-chosen matrix of step sizes. A quantization matrix might look like this

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

The difference d from the DC level in the previous block is coded. The Huffman coding is not done directly on the difference values. Instead a category is formed according to

$$k = \lceil \log(|d| + 1) \rceil$$

Stastics are gathered for all categories and a Huffman code is constructed.
The codeword for a difference d consists of the Huffman codeword for k followed by k extra bits to exactly specify d.

k	d	extra bits
0	0	–
1	$-1, 1$	$0, 1$
2	$-3, -2, 2, 3$	$00, 01, 10, 11$
3	$-7, \ldots, -4, 4, \ldots, 7$	$000, \ldots, 011, 100, \ldots, 111$
$\vdots$	$\vdots$	$\vdots$

The components are ordered in zigzag order. All runs of zeros are replaced by the length of the run (min 0, max 15). Just as for the DC component, we form the category for each non-zero component l as

$$k = \lceil \log(|l| + 1) \rceil$$

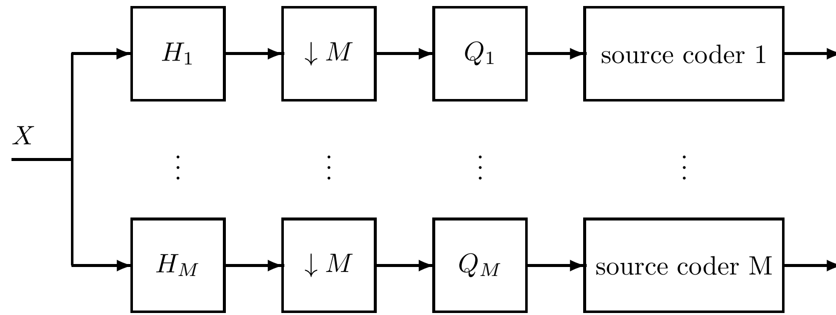
A new symbol alphabet is constructed, consisting of pairs (runlength, category). We gather statistics for the pairs and build a Huffman code for the new alphabet. Just as for the DC level, the codeword for each pair is followed by k bits that exactly tells us what value the non-zero component has.
In the Huffman code we also have two special symbols, (End Of Block) which is used when all the remaining components in a block are zero and ZRL (Zero Run Length) which is used when we have to code a run of zeros that is longer than 15. ZRL means 16 zeros. For example, a run of 19 zeros followed by category 5 is described as (ZRL)(3,5).

11 Subband coding

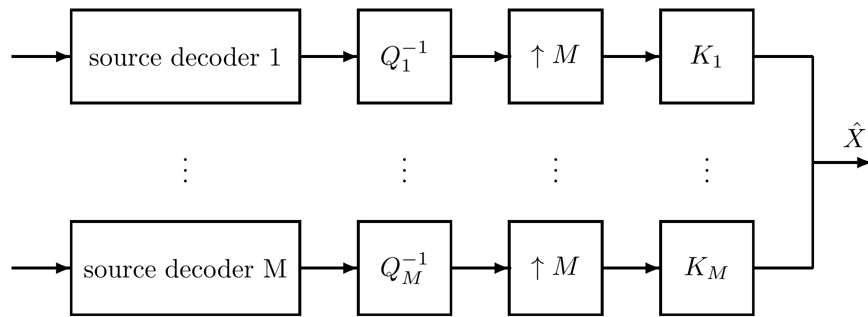
A *subband coder* works somewhat similarily to a transform coder. Instead of splitting the signal into different frequancies using a transform, we split the signal into different frequency bands using a number of bandpass filters. The different frequency signals can (in theory) be downsampled without destroying any information, since they have a smaller bandwith than the original signal. Quantize and source code the different frequency signals.

11.1 Subband coder (M bands)

A subband coder can be described using the following block schedule



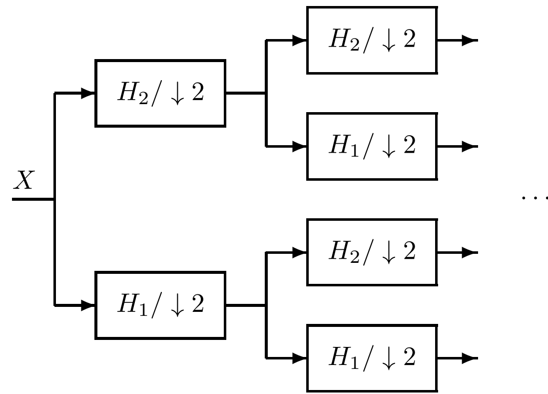
The bandpass filters H_i ; $i = 1 \ldots M$ are called *analysis filters.*
$\downarrow M$ denotes downsampling with a factor M, ie we only keep every M:th sample in each subband.
The source coders and quantizers can of course also depend on each other.
The corresponding subband decoder looks like



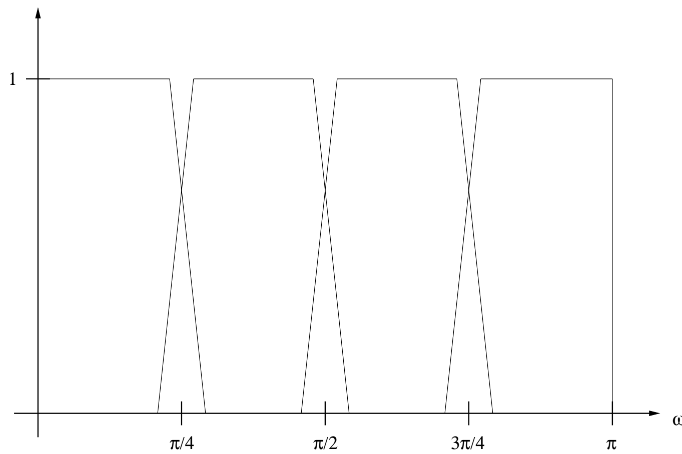
The bandpass filters K_i ; $i = 1 \ldots M$ are called *synthesis filters.*
$\uparrow M$ denotes upsampling with a factor M, ie $M - 1$ zeros are inserted after each sample.

11.2 Recursive filtering

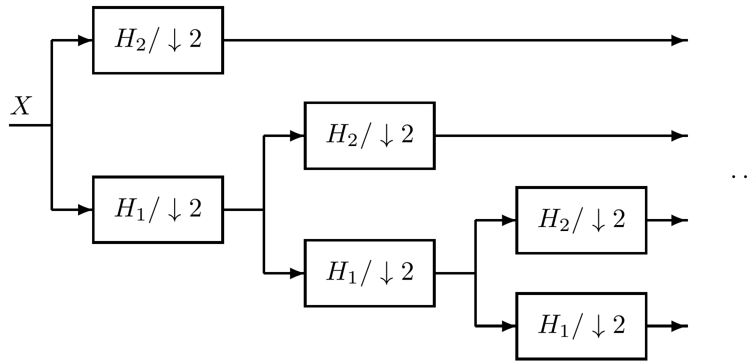
Either we have M actual filters, or we use only two filters (one highpass filter and one lowpass filter) and then apply the filters recursively to divide the signal into narrow bands.
If we do the filtering in two steps and do filtering on both the high- and lowpass branches, we get the following filter structure (the filtering and downsampling have been combined in the figure.)



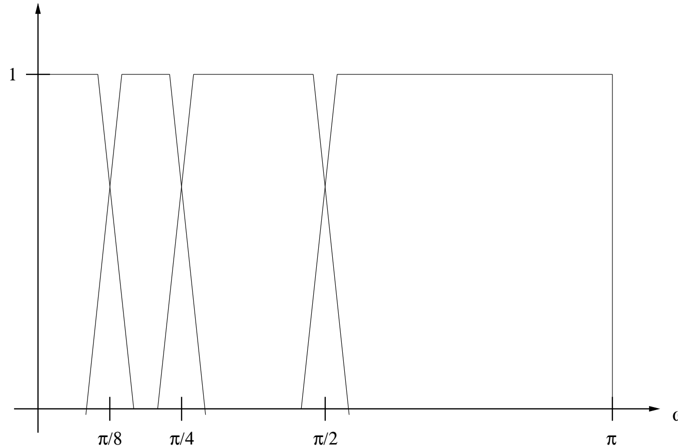
This gives us a flat filter bank (ie the bandwidth of all bands are the same). In the frequency domain it looks like



Alternatively, if we only do repeated splits on the lowpass branch, we get what is called a *dyadic* filterbank.



The division of the frequency axis using a dyadic filter bank with 4 bands looks like



11.3 Filter properties

If we consider the system coder-decoder for a 2-band subband coder without quantization, we can show (see Sayood) that the reconstructed signal, expressed in the z-transform, looks like

$$\begin{aligned}\hat{X}(z) &= \frac{1}{2}[H_1(z)K_1(z) + H_2(z)K_2(z)]X(z) + \\ &+ \frac{1}{2}[H_1(-z)K_1(z) + H_2(-z)K_2(z)]X(-z)\end{aligned}$$

We're usually interested in filters that give perfect reconstruction, ie filters where the reconstructed signal is equal to the original signal, apart from a constant gain and/or a time delay

$$\hat{X}(z) = c \cdot z^{-n_0} \cdot X(z)$$

Another common demand is that we only want to use filters with a finite impulse response (FIR).
There are several ways of finding suitable filters for subband coding, eg QMF, power symmetric filters, wavelets. A few examples:
Haar filter

$$\begin{array}{ll} H_1(z) = \frac{1}{\sqrt{2}}[1 + z^{-1}] & H_2(z) = \frac{1}{\sqrt{2}}[1 - z^{-1}] \\ K_1(z) = H_2(-z) & K_2(z) = -H_1(-z) \end{array}$$

LeGall filter

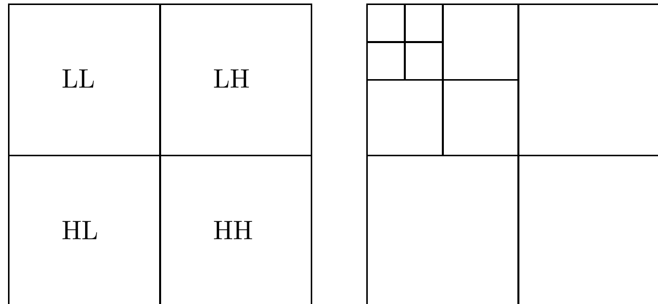
$$\begin{aligned}H_1(z) &= \frac{1}{4\sqrt{2}}[-z^2 + 2z + 6 + 2z^{-1} - z^{-2}] \\ H_2(z) &= \frac{1}{2\sqrt{2}}[-1 + 2z^{-1} - z^{-2}]\end{aligned}$$

$$\begin{aligned} K_1(z) &= H_2(-z) \\ K_2(z) &= -H_1(-z) \end{aligned}$$

Finding suitable filters for a subband coder is a broad field, and goes a bit outside of the scope of this course. If you are interested in more details, see Sayood's book.

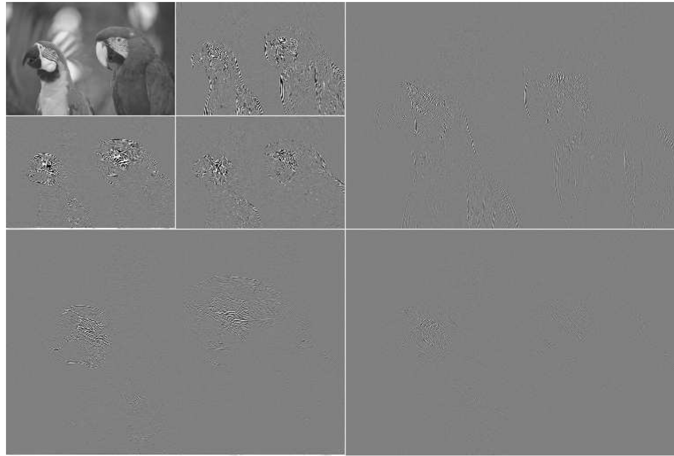
11.4 Twodimensional signals

Wehn coding twodimensional signals (ie images), usually only two filters (lowpass and highpass) are used. The image is filtered horizontally and then vertically with the filter pair so that we get four different frequency bands. Traditionally we only keep splitting the lowpass-lowpass filtered part. Typically this is done for a few steps, depending on the size of the image.



As an example, given this image

this is what we get after two steps of splits



The high frequency bands have been amplified to show the results more clearly.

11.5 Quantization and source coding

In principle we can use the same kinds of methods that are used in transform coding when we do quantization and source coding.
The most important part is to find an efficient way to do source coding.
In the high frequency bands most of the components will be quantized to 0, and there is a strong correlation between adjacent components in the same subband. There is also a correlation between components in different subbands at the same position in the image (eg an edge in the image will give large values in several subbands). This can be utilized by the source coder.
If we have a flat filter bank we can do bit allocation in exactly the same way as in in transform coding (zonal coding).
If we dont have uniform frequency bands, eg from using a dyadic filter bank, we have to take into account the different sample rates of the different bands. This is because we have performed different number of subsamplings for the different bands.

11.6 JPEG 2000

JPEG 2000 is an ISO standard for coding of still images.
The image is first split into a number of rectangular parts (*tiles*). Normally we will only have one tile covering the whole image.
Each tile is transformed using a dyadic subband transform (wavelet transform), using 0-32 splits.
The transformed image is divided into small rectangular blocks of $2^k \times 2^l$ ($2 \leq k, l \leq 10;\ k + l \leq 12$) coefficients for quantization and source coding.
The quantization is uniform.

The source coder is a binary arithmetic coder. The coefficents are coded one bitplane at a time. The coefficients are coded conditioned on surrounding coefficents in the same block. The similarity between different subbands is not used.

JPEG 2000 gives a progressive bitstream, ie its possible to decode just the beginning of the stream and still get a whole image, but with lower quality.

It's possible to specify a *region of interest*, ie a part of the image can be coded using higher quality than the rest of the image.

1-16384 colour components in the image. Can thus be used for multispectral and hyperspectral images.

The input image can have up to 38 bits per colour component.

There is also a lossless coding mode, giving slightly worse results than for instance JPEG-LS.