1. Introduction.

2. Linear Block Codes.
   2.1 Binary Linear Block Codes.
   2.2 Generator Matrix.
   2.3 Parity Check Matrix and Syndrome Testing.

3. Convolutional Codes.
   3.1 Mathematical Representation.
   3.2 Tree diagram
   3.3 Trellis Diagram.
   3.4 Viterbi Algorithm.

4. Interleaver
5. Concatenated Codes
Linear block codes

Linear block codes are conceptually simple codes that are basically an extension of single-bit parity check codes for error detection. A single-bit parity check code is one of the most common forms of detecting transmission errors.

This code uses one extra bit in a block of $n$ data bits to indicate whether the number of 1s in a block is odd or even. Thus, if a single error occurs, either the parity bit is corrupted or the number of detected 1s in the information bit sequence will be different from the number used to compute the parity bit: in either case the parity bit will not correspond to the number of detected 1s in the information bit sequence, so the single error is detected.

Linear block codes extend this notion by using a larger number of parity bits to either detect more than one error or correct for one or more errors. Unfortunately linear block codes, along with convolutional codes, trade their error detection or correction capability for either bandwidth expansion or a lower data rate, as will be discussed in more detail below. We will restrict our attention to binary codes, where both the original information and the corresponding code consist of bits taking a value of either 0 or 1.

Binary Linear Block Codes

A binary block code generates a block of $n$ coded bits from $k$ information bits. We call this an $(n,k)$ binary block code. The coded bits are also called codeword symbols. The $n$ codeword symbols can take on $2^n$ possible values corresponding to all possible combinations of the $n$ binary bits. We select $2^k$ codewords from these $2^n$ possibilities to form the code, such that each $k$ bit information block is uniquely mapped to one of these $2^k$ codewords.

The rate of the code is $R_c = k/n$ information bits per codeword symbol. If we assume that codeword symbols are transmitted across the channel at a rate of $R_s$ symbols/second, Then the information rate associated with an $(n, k)$ block code is $R_o = R_c R_s = kn R_s$ bits/second. Thus we see that block coding reduces the data rate compared to what we obtain with uncoded modulation by the code rate $R_c$. A block code is called a linear code when the mapping of the $k$ information bits to the $n$ codeword symbols is a linear mapping.
The generator matrix is a compact description of how codewords are generated from information bits in a linear block code. The design goal in linear block codes is to find generator matrices such that their corresponding codes are easy to encode and decode yet have powerful error correction/detection capabilities. Consider an \((n, k)\) code with \(k\) information bits denoted as

\[
\mathbf{U}_i = [u_{i1}, \ldots, u_{ik}]
\]

That are encoded into the codeword

\[
\mathbf{C}_i = [c_{i1}, \ldots, c_{in}].
\]

We represent the encoding operation as a set of \(n\) equations defined by

\[
c_{ij} = u_{i1}g_{1j} + u_{i2}g_{2j} + \ldots + u_{ik}g_{kj}, \quad j = 1, \ldots, n,
\]

Where \(g_{ij}\) is binary (0 or 1) and binary (standard) multiplication is used. We can write these \(n\) equations in matrix form as

\[
\mathbf{C}_i = \mathbf{U}_i \mathbf{G},
\]

Where the \(k \times n\) generator matrix \(\mathbf{G}\) for the code is defined as

\[
\mathbf{G} = \begin{bmatrix}
g_{11} & g_{12} & \cdots & g_{1n} 
g_{21} & g_{22} & \cdots & g_{2n} 
\vdots & \vdots & \ddots & \vdots 
g_{k1} & g_{k2} & \cdots & g_{kn}
\end{bmatrix}.
\]
If we denote the \( l \)th row of \( G \) as \( \mathbf{g}_l = [g_{l1}, \ldots, g_{ln}] \) then we can write any codeword \( \mathbf{c}_i \) as linear combinations of these row vectors as follows:

\[
\mathbf{c}_i = u_{i1}\mathbf{g}_1 + u_{i2}\mathbf{g}_2 + \ldots + u_{ik}\mathbf{g}_k.
\]

Since a linear \((n, k)\) block code is a subspace of dimension \( k \) in the larger \( n \)-dimensional space, the \( k \) row vectors \( \{\mathbf{g}_l\}_1^k \) of \( G \) must be linearly independent, so that they span the \( k \)-dimensional subspace associated with the \( 2^k \) codewords. Hence, \( G \) has rank \( k \). Since the set of basis vectors for this subspace is not unique, the generator matrix is also not unique.

A **systematic** linear block code is described by a generator matrix of the form

\[
G = [I_k | P] = \begin{bmatrix}
1 & 0 & \ldots & 0 & p_{11} & p_{12} & \ldots & p_{1(n-k)} \\
0 & 1 & \ldots & 0 & p_{21} & p_{22} & \ldots & p_{2(n-k)} \\
\vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 1 & p_{k1} & p_{k2} & \ldots & p_{k(n-k)}
\end{bmatrix},
\]

Where \( I_k \) is a \( k \times k \) identity matrix and \( P \) is a \( k \times (n-k) \) matrix that determines the redundant, or parity, bits to be used for error correction or detection. The codeword output from a systematic encoder is of the form

\[
\mathbf{c}_i = \mathbf{u}_i G = \mathbf{u}_i[I_k | P] = [u_{i1}, \ldots, u_{ik}, p_1, \ldots, p_{(n-k)}].
\]

Where the first \( k \) bits of the codeword are the original information bits and the last \((n-k)\) bits of the codeword are the parity bits obtained from the information bits as

\[
p_j = u_{i1}p_{1j} + \ldots + u_{ik}p_{kj}, \quad j = 1, \ldots, n-k.
\]

Note that any generator matrix for an \((n, k)\) linear block code can be reduced by row operations and column permutations to a generator matrix in systematic form.
Parity Check Matrix and Syndrome Testing

The parity check matrix is used to decode linear block codes with generator matrix $G$. The parity check matrix $H$ corresponding to a generator matrix $G = [I_k | P]$ is defined as

$$H = [P^T | I_{n-k}].$$

It is easily verified that $GH^T = 0_{k,n}$, where $0_{k,n}$ denotes an all zero $k \times (n-k)$ matrix. Recall that a given codeword $C_i$ in the code is obtained by multiplication of the information bit sequence $U_i$ by the generator matrix $G$: $C_i = U_iG.$ Thus,

$$C_iH^T = U_iGHT = 0_{n-k}$$

For any input sequence $U_i$, where $0_{n-k}$ denotes the all-zero row vector of length $n-k$. Thus, multiplication of any valid codeword with the parity check matrix results in an all zero vector. This property is used to determine whether the received vector is a valid codeword or has been corrupted, based on the notion of syndrome testing, which we now define. Let $R$ be the received codeword resulting from transmission of codeword $C$. In the absence of channel errors, $R = C$. However, if the transmission is corrupted, one or more of the codeword symbols in $R$ will differ from those in $C$. We therefore write the received codeword as

$$R = C + e,$$

Where $e = [e_1, e_2, \ldots, e_n]$ is the error vector indicating which codeword symbols were corrupted by the channel. We define the syndrome of $R$ as

$$S = RH^T.$$

If $R$ is a valid codeword, i.e. $R = C_i$ for some $i$, then $S = C_iH^T = 0_{n-k}$ by (8.14). Thus, the syndrome equals the all zero vector if the transmitted codeword is not corrupted, or is corrupted in a manner such that the received codeword is a valid codeword in the code that is different from the transmitted codeword. If the received codeword $R$ contains detectable errors, then $S \neq 0_{n-k}$. If the received codeword contains correctable errors, then the syndrome identifies the error pattern corrupting the transmitted codeword, and these errors can then be corrected. Note that the
syndrome is a function only of the error pattern \( e \) and not the transmitted codeword \( C \), since

\[
S = RH^T = (C + e)H^T = CH^T + eH^T = 0_{n-k} + eH^T.
\]

Since \( S = eH^T \) corresponds to \( n - k \) equations in \( n \) unknowns, there are \( 2^k \) possible error patterns that can produce a given syndrome \( S \). However, since the probability of bit error is typically small and independent for each bit, the most likely error pattern is the one with minimal weight, corresponding to the least number of errors introduced in the channel. Thus, if an error pattern \( \hat{e} \) is the most likely error associated with a given syndrome \( S \), the transmitted codeword is typically decoded as

\[
\hat{C} = R + \hat{e} = C + e + \hat{e}.
\]

When the most likely error pattern does occur, i.e. \( ^\wedge e = e \), then \( ^\wedge C = C \), i.e. the corrupted codeword is correctly decoded

Let \( C_w \) denote a codeword in a given \((n, k)\) code with minimum weight (excluding the all-zero codeword). Then \( C_wH^T = 0_{n-k} \) is just the sum of \( d_{\min} \) columns of \( H^T \), since \( d_{\min} \) equals the number of 1s (the weight) in the minimum weight codeword of the code. Since the rank of \( H^T \) is at most \( n - k \), this implies that the minimum distance of an \((n, k)\) block code is upperbounded by

\[
d_{\min} \leq n - k + 1.
\]

**Convolutional Codes**

A convolutional code generates coded symbols by passing the information bits through a linear finite-state shift register, as shown in the Figure below. The shift register consists of \( K \) stages with \( k \) bits per stage. There are \( n \) binary addition operators with inputs taken from all \( K \) stages: these operators produce a codeword of length \( n \) for each \( k \) bit input sequence. Specifically, the binary input data is shifted into each stage of the shift register \( k \) bits at a time, and each of these shifts produces a coded sequence of length \( n \). The rate of the code is \( R_c = k/n \). The number of shift register stages \( K \) is called the constraint length of the code. It is clear from Figure 8.5 that a length-\( n \) codeword depends on \( kK \) input bits, in contrast to a block code which only depends on \( k \) input bits. Convolutional codes are said to have memory since the current codeword depends on more input bits \((kK)\) than the number input to the encoder to generate it \((k)\).
When a length-\(n\) codeword is generated by a convolutional encoder, this codeword depends both on the \(k\) bits input to the first stage of the shift register as well as the state of the encoder, defined as the contents in the other \(K - 1\) stages of the shift register. In order to characterize a convolutional code, we must characterize how the codeword generation depends both on the \(k\) input bits and the encoder state, which has \(2^K - 1\) possible values. There are multiple ways to characterize convolutional codes, including a tree diagram, state diagram, and trellis diagram.

The tree diagram represents the encoder in the form of a tree where each branch represents a different encoder state and the corresponding encoder output. A state diagram is a graph showing the different states of the encoder and the possible state transitions and corresponding encoder outputs. A trellis diagram uses the fact that the tree representation repeats itself once the number of stages in the tree exceeds the constraint length of the code. The trellis diagram simplifies the tree representation by merging nodes in the tree corresponding to the same encoder state. In this report we will focus on the trellis representation of a convolutional code since it is the most common characterization. The details of the trellis diagram representation are best described by an example.
Consider the convolutional encoder shown in Figure below with $n = 3$, $k = 1$, and $K = 3$. In this encoder, one bit at a time is shifted into Stage 1 of the 3-stage shift register. At a given time $t$ we denote the bit in Stage $I$ of the shift register as $S_i$. The 3 stages of the shift register are used to generate a codeword of length 3, $C_1C_2C_3$, where from the figure we see that $C_1 = S_1 + S_2$, $C_2 = S_1 + S_2 + S_3$, and $C_3 = S_3$. A bit sequence $U$ shifted into the encoder generates a sequence of coded symbols, which we denote by $C$. Note that the coded symbols corresponding to $C_3$ are just the original information bits. As with block codes, when one of the coded symbols in a convolutional code corresponds to the original information bits, we say that the code is systematic. We define the encoder state as $S = S_2S_3$, i.e. the contents of the last two stages of the encoder, and there are $2^2 = 4$ possible values for this encoder state. To characterize the encoder, we must show for each input bit and each possible encoder state what the encoder output will be, and how the new input bit changes the encoder state for the next input bit.

The trellis diagram for this code is shown in Figure below. The solid lines in the Figure indicate the encoder state transition when a 0 bit is input to Stage 1 of the encoder, and the dashed lines indicate the state transition corresponding to a 1 bit input. For example, starting at state $S = 00$, if a 0 bit is input to Stage 1 then, when the shift register transitions, the new state will remain as $S = 00$ (since the 0 in Stage 1 transitions to Stage 2, and the 0 in Stage 2 transitions to Stage 3, resulting in the new state $S = S_2S_3 = 00$). On the other hand, if a 1 bit is input to Stage 1 then, when the shift register transitions, the new state will become $S = 10$ (since the 1 in Stage 1 transitions to Stage 2, and the 0 in Stage 2 transitions to Stage 3, resulting in the new
state $S = S_2 S_3 = 10$). The encoder output corresponding to a particular encoder state $S$ and input $S_1$ is written next to the transition lines in the Figure. This output is the encoder output that results from the encoder addition operations on the bits $S_1$, $S_2$ and $S_3$ in each stage of the encoder. For example, if $S = 00$ and $S_1 = 1$ then the encoder output $C_1 C_2 C_3$ has $C_1 = S_1 + S_2 = 1$, $C_2 = S_1 + S_2 + S_3 = 1$, and $C_3 = S_3 = 0$. This output 110 is drawn next to the dashed line transitioning from state $S = 00$ to state $S = 10$ in the Figure. Note that the encoder output for $S_1 = 0$ and $S = 00$ is always the all-zero codeword regardless of the addition operations that form the codeword $C_1 C_2 C_3$, since summing together any number of 0s always yields 0. The portion of the trellis between time $t_i$ and $t_i + 1$ is called the $i^{th}$ branch of the trellis. The Figure indicates that the initial state at time $t_0$ is the all-zero state. The trellis achieves steady state, defined as the point where all states can be entered from either of two preceding states, at time $t_3$. After this steady state is reached, the trellis repeats itself in each time interval. Note also that in steady state each state transitions to one of two possible new states. In general trellis structures starting from the all-zero state at time $t_0$ achieve steady-state at time $t_K$.

For general values of $k$ and $K$, the trellis diagram will have $2^{K-1}$ states, where each state has $2^k$ paths entering each node, and $2^k$ paths leaving each node. Thus, the number of paths through the trellis grows exponentially with $k$, $K$, and the length of the trellis path.
The Viterbi Algorithm

The Viterbi algorithm, discovered by Viterbi in 1967, reduces the complexity of maximum likelihood decoding by systematically removing paths from consideration that cannot achieve the highest path metric. The basic premise is to look at the partial path metrics associated with all paths entering a given node (Node N) in the trellis. Since the possible paths through the trellis leaving node N are the same for each entering path, the complete trellis path with the highest path metric that goes through Node N must coincide with the path that has the highest partial path metric up to node N. This is illustrated in the next Figure, where Path 1, Path 2, and Path 3 enter Node N (at trellis depth n) with partial path metrics $P_i = \sum_{i=0}^{N} B_{i}^l$, $l = 1, 2, 3$ up to this node. Assume $P_1$ is the largest of these partial path metrics. The complete path with the highest metric, shown in bold, has branch metrics $\{B_k\}$ after node N. The maximum likelihood path starting from Node N, i.e., the path starting from node N with the largest path metric has partial path metric $\sum_{k=n}^{\infty} B_k$. The complete path metric for Path $i$, $i = 1, 2, 3$, up to node N and the maximum likelihood path after node N is $P_i + \sum_{i=n}^{\infty} B_k$, $l = 1, 2, 3$, and thus the path with the maximum partial path metric $P_i$ up to node N (Path 1 in this example) must correspond to the path with the largest path metric that goes through node N.

The Viterbi algorithm takes advantage of this structure by discarding all paths entering a given node except the path with the largest partial path metric up to that node. The path that is not discarded is called the survivor path. Thus, for the example of Figure 8.8, Path 1 is the survivor at node N and Paths 2 and 3 are discarded from further consideration. Thus, at every stage in the trellis there are $2^{K-1}$ surviving paths, one for each possible encoder state. A branch for a given stage of the trellis cannot be decoded until all surviving paths at a subsequent trellis stage overlap with that branch, as shown in the next Figure. This figure shows the surviving paths at time $t_{k+3}$. We see in this figure that all of these surviving paths can be traced back to a common stem from time $t_k$ to $t_{k+1}$. At this point the decoder can output the codeword $C_i$ associated with this branch of the trellis. Note that there is not a fixed decoding delay associated with how far back in the trellis a common stem occurs for a given set of surviving paths, this delay depends on $k$, $K$ and the specific code properties. To avoid a random decoding delay, the Viterbi algorithm is typically modified such that at a given stage in the trellis, the most likely branch $n$ stages back is decided upon based on the partial path metrics up to that point. While this modification does not yield exact maximum likelihood decoding, for $n$ sufficiently large (typically $n \geq 5K$) it is a good approximation.
Viterbi Algorithm

The Viterbi algorithm must keep track of $2^{K(K-1)}$ surviving paths and their corresponding metrics. At each stage, $2k$ metrics must be computed for each node to determine the surviving path, corresponding to the $2k$ paths entering each node. Thus, the number of computations in decoding and the memory requirements for the algorithm increase exponentially with $k$ and $K$. This implies that for practical implementations convolutional codes are restricted to relatively small values of $k$ and $K$.

Concatenated Codes

A concatenated code uses two levels of coding: an inner code and an outer code, as shown in the next Figure. The inner code is typically designed to remove most of the errors introduced by the channel, and the outer code is typically a less powerful code that further reduces error probability when the received coded bits have a relatively low probability of error (since most errors are corrected by the inner code). Concatenated codes may have the inner and outer codes separated by an interleaver to break up block errors introduced by the channel. Concatenated codes typically achieve very low error probability with less complexity than a single code with the same error probability performance. The decoding of concatenated codes is typically done in two stages, as indicated in the figure: first the inner code is decoded, and then the outer code is decoded separately. This is a suboptimal technique, since in fact both codes are working in tandem to reduce error probability. However, the ML decoder for a concatenated code, which performs joint decoding, is highly complex. It was discovered in the mid 1990s that a near-optimal decoder for concatenated codes can be obtained based on iterative decoding, and that is the basic premise behind turbo codes.
Concatenated Coding