# CHANNEL CODING CODE DESIGN

EE539: Principles of Information Theory and Coding


Abstract

A report on the fulfillment of Exercise 5, focusing on Channel coding code design which takes as input a binary channel and designs a high-rate code.



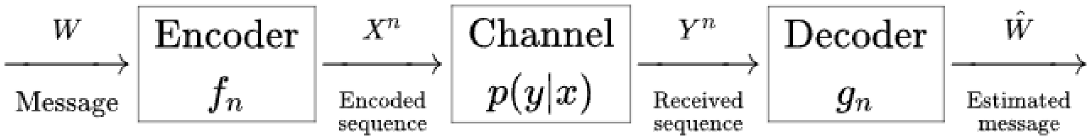
Jayant Som

MS Student in Quantum Science and Nanotechnology

# Error Correction (Channel coding)

Error detection and correction are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channels are subject to channel noise, and thus errors may be introduced during transmission from the source to a receiver. Error detection techniques allow detecting such errors, while error correction enables detection of errors and reconstruction of the original, error-free data.

**The basic mathematical model for a communication system:**




A message W is transmitted through a noisy channel by using encoding and decoding functions. An encoder maps W into a pre-defined sequence of channel symbols of length n. In its most basic model, the channel distorts each of these symbols independently of the others. The output of the channel is fed into a decoder which maps the sequence into an estimate of the message. In this setting, the probability of error is defined as:

$$P_e = \Pr\left\{\hat{W} \neq W\right\}$$

**Shannon's noisy-channel coding theorem**

This theorem was stated by Claude Shannon in 1948. It establishes that for any given degree of noise contamination of a communication channel, it is possible to communicate discrete data nearly error-free up to a computable maximum rate through the channel. Shannon's theorem has wide-ranging applications in both communications and data storage.

The Shannon theorem states that given a noisy channel with channel capacity C and information transmitted at a rate R, then if R<C, there exist codes that allow the probability of error at the receiver to be made arbitrarily small. This means that, theoretically, it is possible to transmit information nearly without error at any rate below a limiting rate, C.

The converse: If R>C, an arbitrarily small probability of error is not achievable. All codes will have a probability of error greater than a certain positive minimal level, and this level increases as the rate increases. So, information cannot be guaranteed to be transmitted reliably across a channel at rates beyond the channel capacity.

For every discrete memoryless channel, the channel capacity, defined in terms of the mutual information I(X;Y) as C = max I(X;Y). Formally, for any ϵ>0 and R<C, for large enough N, there exists a code of length N and rate ≥ R and a decoding algorithm, such that the maximal probability of block error is ≤ ϵ.

# Channel coding Code Design

I have designed the channel coding and segregated my program into 4 parts:

0. **Orchestrator:** This program reads the transition probabilities: p(0|0), p(1|0), p(0|1), p(1|1) and coordinates the entire workflow, running all components systematically, while managing data flow between them.
1. **Channel Capacity Calculator:** This program numerically calculates the maximum mutual information or channel capacity C, corrected to 4 places of decimal. For the channel given as input, it prints the value of C and the optimizing $q_*$.
2. **Codebook generator:** This creates M random codewords of length n using the optimal distribution q*, saving them to a file.
3. **Simulator:** This program randomly selects messages, simulates their transmission through the channel to generate outputs, performs maximum likelihood decoding, and calculates the success rate across 1000 trials while displaying sample transmissions.

# 1. Channel Capacity Calculator

The sequence of operations executed within the code and my explanations are as follows:

a. Input:

- Accepted channel transition probabilities as comma-separated values.
- Validated that each conditional probability distribution sums to 1.

b. Structuring the Data:

- Organized the probabilities into a nested dictionary pYgX for efficient access during calculations.

c. Calculating Mutual Information:

- Computed the output distribution Pr(Y) by marginalizing over inputs.
- Calculated the output entropy H(Y).
- Determined the conditional entropy H(Y|X) by averaging over input symbols.

d. Optimizing Capacity:

- Used numerical optimization (minimize_scalar) to find the input distribution q* that maximizes mutual information.
- Ensured the optimization stays within valid probability bounds (0 to 1).

I have used minimize_scalar to numerically find the input distribution $q*$ that maximizes the channel capacity $C$. Since the mutual information $I(X;Y)$ depends on a single variable $q=P(X=0)$, this method efficiently searches for the optimal $q$ within the range [0, 1]. It is the simplest and most efficient approach for finding the capacity-achieving input distribution $q*$ for a binary channel.

e. Output:

- Stored the results (pYgX, C, q*) in the IPython namespace for downstream use.
- Printed the channel capacity C (to 4 decimal places) and the optimal input distribution q*.

In [ ]:

```
import numpy as np
from scipy.optimize import minimize_scalar

# I am taking channel probabilities as input
input_str = input("Enter p(0|0), p(1|0), p(0|1), p(1|1) (comma-separated): "
p0_0, p1_0, p0_1, p1_1 = map(float, input_str.split(','))

# I am validating probability distributions
# Checking if the first pair of probabilities sums to approximately 1
assert np.isclose(p0_0 + p1_0, 1), "First probability pair does not sum to 1
# Checking if the second pair of probabilities sums to approximately 1
assert np.isclose(p0_1 + p1_1, 1), "Second probability pair does not sum to 

# I am structuring channel probabilities for easy access
pYgX = {
    0: {0: p0_0, 1: p1_0},
    1: {0: p0_1, 1: p1_1}
}

def mutual_info(q):
    # I am calculating output distribution Pr(Y)
    pY0 = q * p0_0 + (1-q) * p0_1
    pY1 = q * p1_0 + (1-q) * p1_1

    # I am computing output entropy H(Y)
    HY = 0.0
    if pY0 > 0:
        HY -= pY0 * np.log2(pY0)
    if pY1 > 0:
        HY -= pY1 * np.log2(pY1)

    # I am computing conditional entropy H(Y|X)
    HYgX = 0.0
    if p0_0 > 0:
        HYgX += q * (-p0_0 * np.log2(p0_0))
    if p1_0 > 0:
        HYgX += q * (-p1_0 * np.log2(p1_0))
    if p0_1 > 0:
        HYgX += (1-q) * (-p0_1 * np.log2(p0_1))
    if p1_1 > 0:
        HYgX += (1-q) * (-p1_1 * np.log2(p1_1))

    return HY - HYgX

# I am finding capacity by optimizing mutual information
result = minimize_scalar(lambda q: -mutual_info(q), bounds=(0,1), method='bo
C = mutual_info(result.x)
q_opt = result.x

# I am storing results for later use
get_ipython().push({'pYgX': pYgX, 'C': C, 'q_opt': q_opt})
print(f"\n• Computed Capacity: C = {C:.4f} bits\n• Optimal q* = {q_opt:.4f}"
```

## 2. Codebook generator

The sequence of operations executed within the code and my explanations are as follows:

a. Initializing parameters:

- Set the block-length n to 24 symbols.
- Calculated the code rate R as 75% of the channel capacity C.
- Calculated the number of codewords M using M = 2^(n*R)

b. Generating Codebook:

- Created each codeword by randomly generating n bits using the optimal input distribution q*.
- Stored all codewords in a list structure.

c. Writing codebook:

- Wrote the complete codebook to Codebook.txt, with one codeword per line.

d. Completion and Storage:

- Displayed a success message confirming codebook generation.
- Then finally, I pushed key variables (M, n, R) to the IPython namespace for downstream use.

In [ ]:

```
import numpy as np

# I am setting up code parameters using channel capacity results
n = 24  # Blocklength
R = 0.75 * C  # Rate (75% of capacity)
M = int(2 ** (n * R))  # Number of codewords

# I am displaying key parameters for verification
print(f"Parameters:")
print(f"- Blocklength (n): {n}")
print(f"- Rate (R): {R:.4f} (75% of capacity C = {C:.4f})")
print(f"- Number of codewords (M): {M}")
print(f"- Input distribution q*: [0 = {q_opt:.4f}, 1 = {1-q_opt:.4f}]")

# I am generating random codewords using the optimal distribution
codebook = []
for i in range(M):
    codeword = np.random.choice(['0','1'], size = n, p = [q_opt, 1-q_opt])
    codebook.append(''.join(codeword))

# I am saving the codebook to a text file
with open("Codebook.txt", "w") as f:
    f.write("\n".join(codebook))

# I am confirming successful generation and storing variables
print(f"\033[1;32m\nSuccessfully saved {M} codewords to Codebook.txt !\033[0
get_ipython().push({'M': M, 'n': n, 'R': R})
```

# 3. <u>Simulator</u>

The sequence of operations executed within the code and my explanations are as follows:

a. Initialization:

- Loaded the pre-generated codebook from Codebook.txt file and extracted its parameters (number of codewords M and blocklength n).

b. Defining simulation and decoding functions:

- Created the channel simulation function that corrupts transmitted bits according to given probabilities.
- Created the maximum likelihood decoder function that compares received sequences against all possible codewords.

c. Simulation Preparation:

- Set the total number of trials to 1000 and selecting 10 random trials to display.
- Initialized counters for success tracking and timing the execution.

d. Execution:

- For each trial, I randomly selected a message and its corresponding codeword.
- Then I simulated the channel transmission to generate a received sequence.
- Then I applied ML decoding to recover the most likely transmitted message.
- Finally checked whether the decoding was successful and displaying sample results.

e. Results:

- At last, I calculated the total runtime and success rate percentage.

In [ ]:

```
import numpy as np
import time
import random

# I am loading the codebook
with open("Codebook.txt") as f:
    codebook = [line.strip() for line in f]
M, n = len(codebook), len(codebook[0])

# I am using channel probabilities from capacity calculations
# pYgX = {0: {0: p0_0, 1: p1_0}, 1: {0: p0_1, 1: p1_1}}

# I am simulating the channel transmission for each bit
def simulate_channel(x_str):
    return ''.join(
        str(np.random.choice([0, 1], p = [pYgX[int(xi)][0], pYgX[int(xi)][1]
        for xi in x_str
    )

# I am implementing maximum likelihood decoding
def ml_decode(y_n):
    return max(range(M),
               key=lambda c: sum(np.log(pYgX[int(xi)][int(yi)])
                            for xi, yi in zip(codebook[c], y_n)
                            if pYgX[int(xi)][int(yi)] > 0))

# I am initializing simulation parameters
total_trials = 1000
sample_trials = 10
success_count = 0
start_time = time.time()

# I am setting up the results display header
print("="*60)
print(f"RUNNING {total_trials} TRANSMISSION TRIALS\n")
print(f"Displaying {sample_trials} random samples below:\n")

print(f"{'Trial':<6} | {'w':<4} | {'x^n(w)':<{n}} | {'y^n':<{n}} | {'w^':<4}
print("-" * (6 + 4 + n + n + 4 + 8 + 5*3))

# I am selecting random trials to display
sample_indices = set(random.sample(range(total_trials), sample_trials))

# I am running the main simulation loop
for trial in range(total_trials):

    # I am randomly selecting a message to transmit
    w = np.random.randint(0, M)
    x_w = codebook[w]

    # I am simulating channel transmission
    y_n = simulate_channel(x_w)

    # I am performing ML decoding
```

```
    w_ = ml_decode(y_n)

    # I am tracking successful decodings
    success = (w_ == w)
    success_count += success

     # I am displaying sample trial results
    if trial in sample_indices:
        result = "SUCCESS" if success else "ERROR"
        print(f"{trial+1:<6} | {w:<4} | {x_w:<{n}} | {y_n:<{n}} | {w_:<4} |

# I am calculating and displaying final results
runtime = time.time() - start_time
success_rate = 100 * success_count / total_trials

print("\n" + "="*60)
print(f"\033[1;32mSimulation completed in {runtime:.2f} seconds!\033[0m")
print("\033[1;31m\n==== FINAL RESULTS ====\n\033[0m")
print(f"- Total trials: {total_trials}")
print(f"\033[1;32m- Success rate: {success_rate:.2f}%\033[0m")
print("="*60)
```

# Orchestration and Results

## 1. Simulating a Binary Symmetric Channel (BSC) with 10% Error

**p(0|0), p(1|0), p(0|1), p(1|1) : 0.9, 0.1, 0.1, 0.9**

- Flips bits with 10% probability
- Capacity ≈ 0.531 bits

## 2. Simulating a Binary Symmetric Channel (BSC) with crossover probability 0.2

**p(0|0), p(1|0), p(0|1), p(1|1) : 0.8, 0.2, 0.2, 0.8**

- Flips bits with 20% probability

- Capacity ≈ 0.278 bits

In [1]:

```
# I am configuring the notebook files
%config InteractiveShell.ast_node_interactivity = 'all'
%reload_ext autoreload
%autoreload 2

# Running the components in the following sequence :
print("\033[1;31m\n\n==== 1. CHANNEL CAPACITY CALCULATION ====\n\033[0m")
%run -i 1_Channel_Capacity_Calculator.ipynb

print("\033[1;34m\n==== 2. CODEBOOK GENERATION ====\n\033[0m")
%run -i 2_Codebook_Generator.ipynb

print("\033[1;35m\n==== 3. TRANSMISSION SIMULATION ====\n\033[0m")
%run -i 3_Simulator.ipynb
```

```
==== 1. CHANNEL CAPACITY CALCULATION ====
```

```
• Computed Capacity: C = 0.5310 bits
• Optimal q* = 0.5000

==== 2. CODEBOOK GENERATION ====

Parameters:
- Blocklength (n): 24
- Rate (R): 0.3983 (75% of capacity C = 0.5310)
- Number of codewords (M): 753
- Input distribution q*: [0 = 0.5000, 1 = 0.5000]

Successfully saved 753 codewords to Codebook.txt !

==== 3. TRANSMISSION SIMULATION ====

============================================================
RUNNING 1000 TRANSMISSION TRIALS

Displaying 10 random samples below:

Trial  | w    | x^n(w)                   | y^n                      | w^   |
Result
----------------------------------------------------------------------------
---------
21     | 25   | 110000110101101111111101 | 110000110101101111110101 | 25   |
SUCCESS
205    | 4    | 100011010010110111000101 | 000011010010110111100101 | 4    |
SUCCESS
306    | 468  | 111101111111011110111101 | 111101111111010110011101 | 468  |
SUCCESS
331    | 629  | 001100110001010111010100 | 001100110001010111010000 | 629  |
SUCCESS
391    | 388  | 001000000110111000101011O | 001000000110111000101111O | 388  |
SUCCESS
469    | 549  | 101010110111010111100011 | 101010110111111111100010 | 549  |
SUCCESS
699    | 684  | 101110111111010001000001 | 101110111111110111001001 | 79   |
ERROR
759    | 194  | 100100000100000000101101 | 101100000100000000101101 | 194  |
SUCCESS
777    | 160  | 100101001110011001111000 | 100000001110011001111000 | 160  |
SUCCESS
786    | 591  | 010110101000111011100110 | 010110111000111001100110 | 591  |
SUCCESS

============================================================
Simulation completed in 26.68 seconds!

==== FINAL RESULTS ====

- Total trials: 1000
- Success rate: 89.00%
============================================================
```

In [1]:
```
# I am configuring the notebook files
%config InteractiveShell.ast_node_interactivity = 'all'
%reload_ext autoreload
%autoreload 2

# Running the components in the following sequence :
print("\033[1;31m\n\n==== 1. CHANNEL CAPACITY CALCULATION ====\n\033[0m")
%run -i 1_Channel_Capacity_Calculator.ipynb

print("\033[1;34m\n==== 2. CODEBOOK GENERATION ====\n\033[0m")
%run -i 2_Codebook_Generator.ipynb

print("\033[1;35m\n==== 3. TRANSMISSION SIMULATION ====\n\033[0m")
%run -i 3_Simulator.ipynb
```

```
==== 1. CHANNEL CAPACITY CALCULATION ====
```

```
• Computed Capacity: C = 0.2781 bits
• Optimal q* = 0.5000

==== 2. CODEBOOK GENERATION ====

Parameters:
- Blocklength (n): 24
- Rate (R): 0.2086 (75% of capacity C = 0.2781)
- Number of codewords (M): 32
- Input distribution q*: [0 = 0.5000, 1 = 0.5000]

Successfully saved 32 codewords to Codebook.txt !

==== 3. TRANSMISSION SIMULATION ====

============================================================
RUNNING 1000 TRANSMISSION TRIALS

Displaying 10 random samples below:

Trial  | w    | x^n(w)                   | y^n                      | w^   |
Result
----------------------------------------------------------------------------
---------
118    | 16   | 101010111101111011110010 | 101110111000110011010010 | 16   |
SUCCESS
412    | 26   | 000000000110101111001011 | 010000000101101111001000 | 26   |
SUCCESS
427    | 20   | 000001111011101101010100 | 011011111010100000010100 | 20   |
SUCCESS
492    | 28   | 010111101001100010001010 | 010111000001100110001110 | 28   |
SUCCESS
542    | 14   | 001100100110111001111011 | 011100110110111111110101 | 2    |
ERROR
560    | 12   | 110000000001001010110010 | 000100000011001011111111 | 12   |
SUCCESS
691    | 30   | 111001101001101110010000 | 111001001101101010010000 | 30   |
SUCCESS
696    | 24   | 011010010110010010111001 | 011000010110000110111001 | 24   |
SUCCESS
725    | 29   | 001000111001001110101000 | 001111101011001100001000 | 17   |
ERROR
869    | 5    | 101001101100010001101000 | 101000101000010111101011 | 5    |
SUCCESS

============================================================
Simulation completed in 1.59 seconds!

==== FINAL RESULTS ====

- Total trials: 1000
- Success rate: 81.00%
============================================================
```

Codebook.txt - Notepad

File Edit Format View Help

```
010101110111101010101111
000111011110000011101011
000100101010110000011001
111001001000111101101100
101000111001010000111001
010011110001110111110000
111100100110101010110000
110000100101111011110101
101011001110000001101001
100111000011110011010011
010010000001100011001011
110110010100100000010001
110010011000011100111100
110110101011011100111101
011111000100101111011001
001010011110111110010000
101100111010100100011101
001000000111001000000011
101001101000111110011100
110100101110101101010000
010010001010111011001111
100110011000100000101000
001000100100000010111101
111000100101000011000010
001111110001100110111011
110000010010110100001111
100100110100110011000010
111110000001110000101001
011011100010110101101101
110100001110011110100111
```