Hand in 3
Information Theory

Fredrick Nilsson
fr2037ni-s

1st March 2024

1 Report

1.1 Part 1

It takes 2 rounds to decode the message.

Iteration 1:

$\hat{v}$: 0 1 0 0 1 0 0
Syndrome: 1 1 0 1 1 1
$\sigma^{(i)}$: 1 2 1 1 3 1 1

Iteration 2:

$\hat{v}$: 0 1 0 0 0 0 0
Syndrome: 1 1 1 0 0 0
$\sigma^{(i)}$: 1 3 2 1 1 0 0

Final:

$\hat{v}$: 0 0 0 0 0 0 0

Resulting in the decoded message: 0 0 0 0 0 0

The code for this part is found in section 2.1.

The code is an implementation of the algorithm described in lecture 9. It takes a message and a parity check matrix as input, and then decodes the message using the algorithm.

1.2 Part 2

The decoded message is: Life is what happens when you are busy making other plans.

The source code for this can be found in section 2.1. The only thing added to the code from part 1 is the ability to scan a file for the message and parity check matrix and the translatation of the bits back to a string.

1.3 Part 3

For the third part I translated the provided matlab example into python, and then continued it's implementation by using a proper decoder and parity check matrix. I also implemented a progressbar and made it multi-threaded to speed up the process.

The graph in figure 1 shows the biterrors for different values of ϵ. The source code for this part can be found in section 2.2. What the graph shows is that the biterrors increase as ϵ increases. This is expected as the probability of a bit being flipped increases, and beyond $\epsilon > 0.06$ the biterrors from using the algorithm is higher than the biterrors from using not having any error detection at all. Although this is using the parity matrix given in part 2, and there might be other parity matrixes which would provide better results.

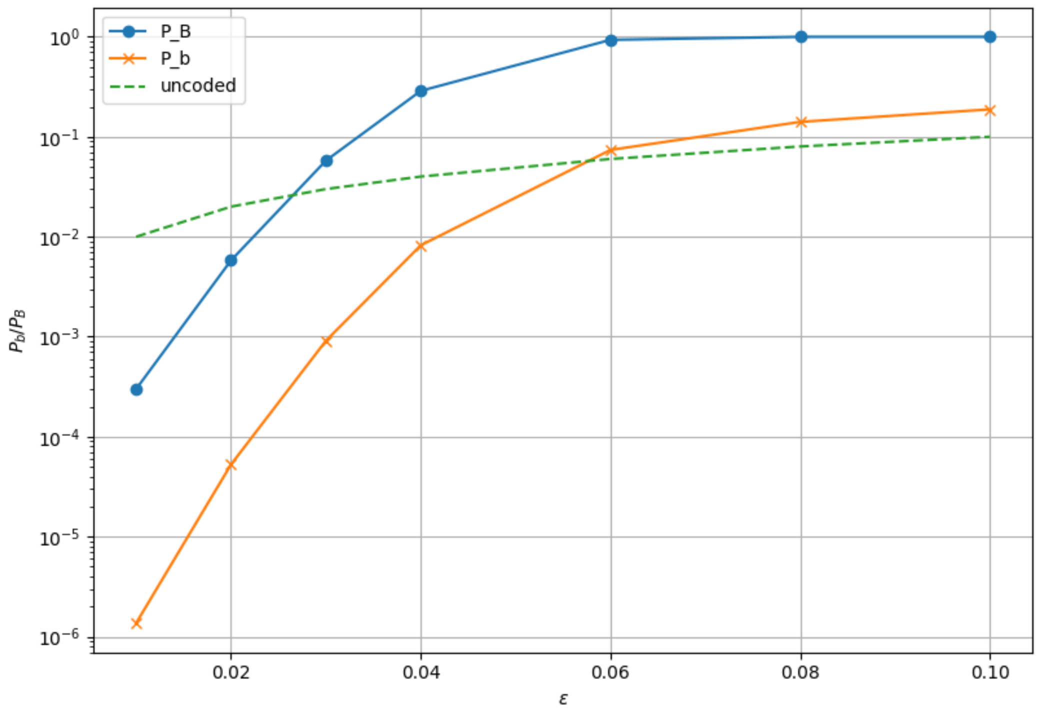

Figure 1: Graph of the biterrors for different values of ϵ .

2 Source code

2.1 Part 1 & 2

```python
import numpy as np
import matplotlib.pyplot as plt

def bit_flip_decode(r: np.ndarray, H: np.ndarray, imax = 100, should_print=False) ->
↪   np.ndarray:
    v_hat = np.copy(r)
    for i in range(imax):
        if should_print:
            print(f"v_hat at iteration {i}: {v_hat}")
        # Calculate syndrome
        syndrome = H.dot(v_hat) % 2
        if should_print:
            print(f"Syndrome at iteration {i}: {syndrome}")

        # If syndrome is 0, then the code is correct so we break the loop
        if all(x == 0 for x in syndrome):
            if should_print:
                print(f"No errors detected at iteration {i}. Decoding complete.")
            return v_hat

        # Compute number of invalid syndromes per codebit
        sigma = syndrome.dot(H)
        if should_print:
            print(f"Sigma at iteration {i}: {sigma}")
        # Flipping the bits with the highest number of invalid syndromes
        max = np.max(sigma)
        for i in range(len(sigma)):
            if sigma[i] == max:
                v_hat[i] = v_hat[i] ^ 1
    if should_print:
        print(f"Maximum iterations reached. Decoding may not be complete.")
    return v_hat


def array_from_file(filename: str) -> np.ndarray:
    data = []
    with open(filename, 'r') as file:
        for line in file:
            numbers = [int(num) for num in line.split()]
            data.append(numbers)

    ndarray = np.array(data)

    return ndarray

def bitarray_to_string(bitarray: np.ndarray) -> str:
    reshaped_array = np.reshape(bitarray, (len(bitarray) // 8, 8))

    bytes_array = np.packbits(reshaped_array, axis=-1)

    return ''.join([chr(byte) for byte in bytes_array.flatten()])


if __name__ == '__main__':

    # Part 1
    parity_matrix = np.array([[1, 1, 0, 0, 0, 0, 0],
                              [0, 1, 1, 0, 0, 0, 0],
                              [0, 1, 1, 1, 1, 0, 0],
                              [0, 0, 0, 1, 1, 0, 0],
                              [0, 0, 0, 0, 1, 1, 0],
                              [0, 0, 0, 0, 1, 0, 1]])
    bitvec = np.array([0, 1, 0, 0, 1, 0, 0])
    print("Part 1:")
    print(bit_flip_decode(bitvec, parity_matrix, should_print=True))

    # Part 2
    parity_matrix = array_from_file("H_1024_3_6.txt")
    bitvec = array_from_file("ChannelOutputBinary.txt")
    bitarray = bit_flip_decode(bitvec[0], parity_matrix)
    bitarray = bitarray[:512]
    with open("DecodedMessage.txt", 'w') as file:
        file.write(bitarray_to_string(bitarray))
    print("Part 2:")
    print(bitarray_to_string(bitarray))
```

2.2 Part 3

```python
import numpy as np
import matplotlib.pyplot as plt
from handin12 import bit_flip_decode, array_from_file, bitarray_to_string
from progress.bar import Bar
from concurrent.futures import ThreadPoolExecutor, as_completed
import threading

# Parameters
n = 1024  # Length of the codeword
k = 512   # Length of the message
R = k / n  # Code rate
numBlocks = 10000  # Number of blocks to transmit

# Channel error probabilities
pvec = np.array([0.01, 0.02, 0.03, 0.04, 0.06, 0.08, 0.1])
print (len(pvec))
# Initialize error counters
biterrors = np.zeros(len(pvec))
numbits = np.zeros(len(pvec))
blockerrors = np.zeros(len(pvec))

imax = 25
H = array_from_file("H_1024_3_6.txt")

lock = threading.Lock()

with Bar('Processing', max=len(pvec) * numBlocks) as bar:
    def simulate_epsilon(indp, p):
        for _ in range(numBlocks):
            # Simulate channel errors
            errors = np.random.rand(n) < p
            r = errors.astype(int)  # Received vector, all-zero codeword + errors
            print(r)
            vhat = bit_flip_decode(r, H, imax=imax, should_print=False)

            # Count errors
            errorpositions = (vhat != 0)  # Since v is all-zero
            biterrors[indp] += np.sum(errorpositions)
            blockerrors[indp] += np.any(errorpositions)
            numbits[indp] += n
            with lock:
                bar.next()

    with ThreadPoolExecutor(max_workers=np.max([len(pvec), 8])) as executor:
        for indp, p in enumerate(pvec):
            executor.submit(simulate_epsilon, indp, p)

# Calculate error probabilities
Pb = biterrors / numbits
Pblock = blockerrors / numBlocks

# Plot results
plt.figure()
plt.semilogy(pvec, Pblock, '-o', label='P_B')
plt.semilogy(pvec, Pb, '-x', label='P_b')
plt.semilogy(pvec, pvec, '--', label='uncoded')
plt.xlabel('$\epsilon$')
plt.ylabel('$P_b / P_B$')
plt.grid(True)
plt.legend()
plt.show()
```