MEng Project Weekly Notes

Rochelle Barsz

Fall 2023

1 Week 1: September 23 - 29

1.1 Reading Paper

Polar codes are a mathematical calculation such that with the combination of a specially-devised encoder and decoder, bits can be either communicated with low error probability or with high probability. It is possible to know which channels are considered good and which are poor channels so that message bits can be transmitted over the poor channels and random bits, called frozen bits, can be sent on the poor channels. This guarantees that message bits can be recovered after being sent on the channel. Bits are first encoded, then sent through a channel which I will simulate with a Binary Symmetric Channel (BSC), and then decoded upon arriving on the other end of the channel. My MEng project will be to simulate this channel by implementing the endoder and decoder using ideas within polar codes to obtain a result that is implementable in practice.

1.2 Implementing Encoder

1.2.1 Explanation

The encoder takes an input u^N and produces combination of the u bits, denoted x^N. The x bits are computed by multiplying the u bits by the matrix G_N, defined mathematically below. B_N is a bit reversal matrix, taking a string 0010001 and producing 1000100. $F^{\otimes n}$ is the n-fold produce of the F matrix shown below. Combined, G_N is a special matrix that produces a linear combination of the u bits.

1.2.2 Important Equations

$$G_N = B_N F^{\otimes n}$$

where $F = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ and $N = 2^n, n \geq 0$ and

$$A \otimes B = \begin{bmatrix} A_{11}B & \cdots & A_{1n}B \\ \vdots & \ddots & vdots \\ A_{m1}B \cdots A_{mn}B \end{bmatrix}$$

$A^{\otimes n} = A \otimes A^{\otimes(n-1)}$ for all $n \geq 1$ and $A^{\otimes 0} = [1]$

1.2.3 Encoder Code

```
def encoding(u):
    N = len(u)

    # Check if N is a power of 2, alters the value of pow
    _,pow = isPowerOfTwo(N)
    # assert powOfTwo
    n = pow # 2^n = N

    u_N = np.zeros(N)
    for i in range(len(u)):
        u_N[i] = int(u[i])
    u_N = u_N.astype(int)

    # G_N = B_N F^(Ox)n
    # B_N is permutation matrix = bit-reversal
    # F = [[1 0], [1 1]]
    # N = 2^n

    # Let B_N = a permutation matrix -> one 1 per column and row
    B_N = permutation_mat(N)
    F = np.array([[1,0], [1,1]])

    F_N = calc_F(n,F)

    G_N = np.matmul(B_N, F_N)
    x_N = np.matmul(u_N,G_N)
```

1

```python
    x_N = x_N.astype(int)
    x_N = x_N % 2
  # convert back to binary bitstring
    x = ''.join(map(str, x_N))  # Convert matrix to string

    # print("u's: ", u)
    # print("x's: ", x)

    return G_N, x

def isPowerOfTwo(n):
    pow = 0
    if n==0:
        return False
    while n != 1:
        if n%2 != 0:
            return False
        pow += 1
        n = n // 2
    return True, pow

def calc_F(n,F): # count backwards from n
    if n == 0:
        return np.ones(1)
    F_N = np.kron(F, calc_F(n-1,F))      # kronecker product
    return F_N

def permutation_mat(L):
    mat = np.zeros((L,L))
    desired_length = len(bin(L-1)[2:])
    for N_indx in range(L):
        new_index = int('{0:0{1}b}'.format(N_indx, desired_length)[::-1], 2)
        mat[new_index][N_indx] = 1
    return mat
```

2

2 Week 2: September 30 - October 13

2.1 Implementing Decoder

2.1.1 Important equations:

$$L_N^{(i)}(y_1^n, \hat{u}_1^{i-1}) = \frac{W_N^{(i)}(y_1^N, \hat{u}_1^{i-1}|0)}{W_N^{(i)}(y_1^N, \hat{u}_1^{i-1}|1)}$$

$$\hat{u}_i = \begin{cases} 0 & \text{if } L_N^{(i)}(y_1^N, \hat{u}_1^{i-1}) \geq 1 \\ 1 & \text{otherwise} \end{cases}$$

Compute the above recursively:

$$L_N^{(2i-1)}(y_1^N, \hat{u}_1^{2i-1}) = \frac{L_{N/2}^{(i)}\left(y_1^{N/2}, \hat{u}_{1,o}^{2i-2} \oplus \hat{u}_{1,e}^{2i-2}\right) L_{N/2}^{(i)}\left(yN/2 + 1^N, \hat{u}_{1,e}^{2i-2}\right) + 1}{L_{N/2}^{(i)}\left(y_1^{N/2}, \hat{u}_{1,o}^{2i-2} \oplus \hat{u}_{1,e}^{2i-2}\right) + L_{N/2}^{(i)}\left(y_1^{N/2}, \hat{u}_{1,e}^{2i-2}\right)}$$

$$L_N^{(2i)}(y_1^N, \hat{u}_1^{2i-1}) = \left[L_{N/2}^{(i)}\left(y_1^{N/2}, \hat{u}_{1,o}^{2i-2} \otimes \hat{u}_{1,e}^{2i-2}\right)\right]^{1-e\hat{u}_{2i-1}} \cdot L_{N/2}^{(i)}\left(y_{N/2+1}^N, \hat{u}_{1,e}^{2i-2}\right)$$

Down to the recursive case:

$$L_1^{(1)}(y_i) = \frac{W(y_i|0)}{W(y_i|1)}$$

2.1.2 Explanation

The decoder converts the y bit string to estimations $\hat{u}^N$. Each $\hat{u}_i$ is computed as two probabilities: the probability of being a 0 or a 1, and the higher probability is the result that is chosen. Calculating these probabilities is quite involved and has high computation time, so an alternative is the recursive definition defined mathematically above. This algorithm is $O(N^2)$.

3

3   Week 3: October 14 - October 20

3.1   Continued Implementation of Decoder

We chose to send all data indices through the decoder, not taking into consideration whether a certain bit is frozen as we are not yet sure how to obtain information on whether a bit is flipped.

3.1.1   Decoder Code

```
def decoding(y, epsilon):
    N = len(y)

    # Convert bit string to numpy array of ints
    y_N = np.zeros(N)
    for i in range(N):
        y_N[i] = int(y[i])
    y_N = y_N.astype(int)

    u_hat_N = np.zeros(N)

    for i in range(N):
        L_i = likelihood(y_N, u_hat_N[:i], epsilon)
        if L_i >= 1:
            u_hat_N[i] = 0
        else:
            u_hat_N[i] = 1

    u_hat_N = u_hat_N.astype(int)
    u_hat_N = u_hat_N % 2       # convert back to binary bitstring
    u_hat = ''.join(map(str, u_hat_N))   # Convert matrix to string

    # print("uhs: ", u_hat)
    return u_hat_N

def likelihood(y_N, u_hat, epsilon):
    N = len(y_N)
    # print(y_N)
    if N == 1:
        if y_N == 0: # y_N is a single value here
            return (1-epsilon)/epsilon
        else: # y_1 = 1
            return  epsilon/(1-epsilon)
    else:
        # y's needed as part of definition of new recursive likelihoods
        firsthalf_y = y_N[:N//2]
        lasthalf_y = y_N[N//2:]

        u_hato = u_hat[::2] # get only odd rows 1,3,...
        u_hate = u_hat[1::2] # get only even rows 2,4,...

        # Kronecker sum - add componentwise
        new_uhat = np.zeros(len(u_hate))
        for p in range(len(new_uhat)):
            if len(u_hat)%2 == 1 and p == (len(new_uhat) - 1):
            # when the length of uhat is odd
                new_uhat[p] = u_hato[p]
            else:
                new_uhat[p] = (u_hato[p] + u_hate[p]) % 2
                # take the mod 2 to keep 0s and 1s
```

4

```
        like1 = likelihood(firsthalf_y, new_uhat, epsilon)
        like2 = likelihood(lasthalf_y, u_hate, epsilon)

        if len(u_hat) % 2 == 0: # Equation 75
            return (like1*like2 + 1) / (like1 + like2)
        else: # i is even, Equation 76
            power = 1 - 2*u_hat[len(u_hat)-1] # either 1 or -1
            return (like1)**power * like2
```

Must now read other paper to determine if bits are frozen or not

3.2   Simulating BSC and Integration with Encoder and Decoder

3.2.1   BSC Explanation

We simulating passing an length N bit string through N independent BSC's $x_i \rightarrow y_i$ by flipping x_i with probability ϵ.

3.2.2   BSC Code

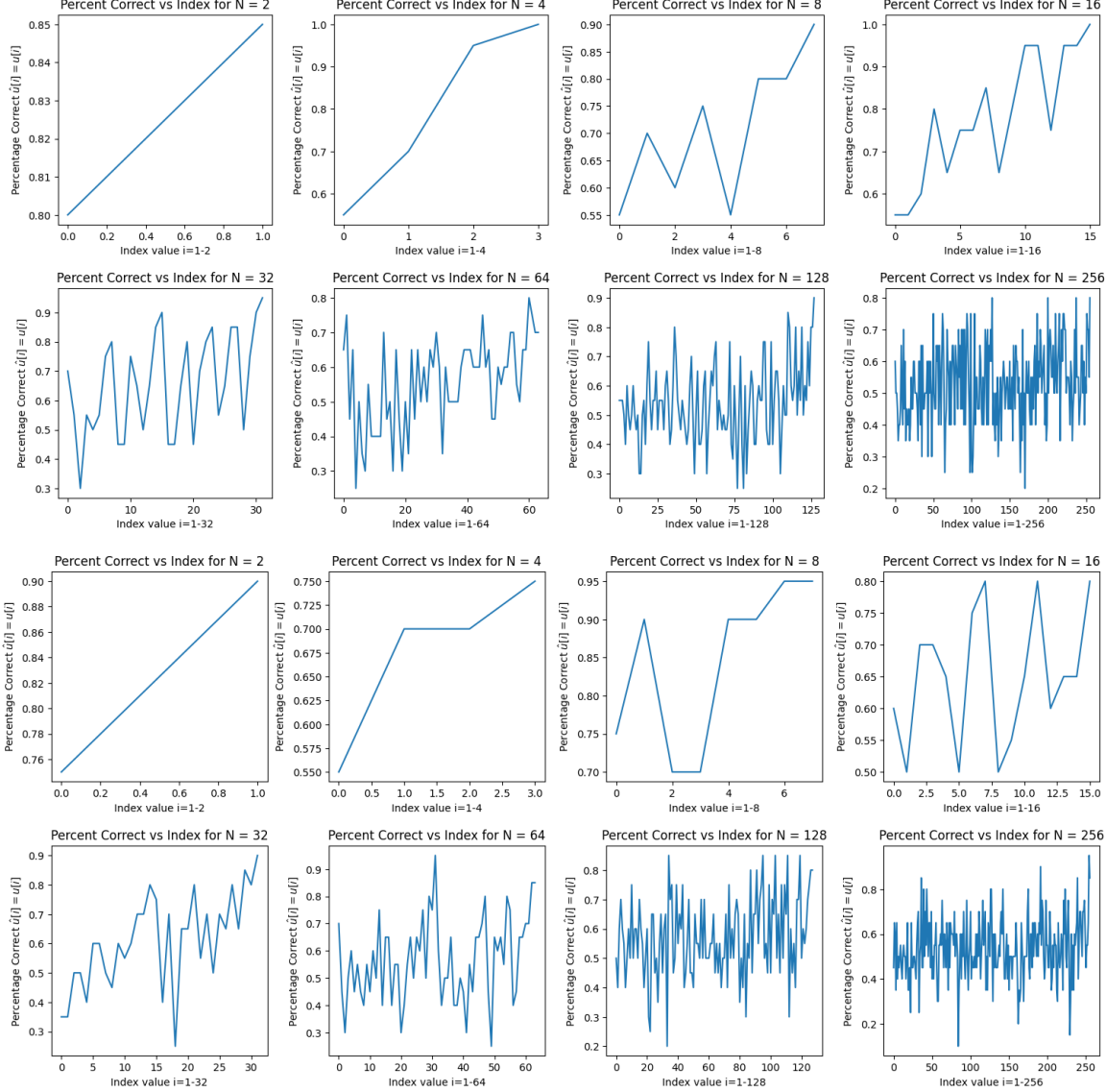
```
def bse(x, epsilon):
    N = len(x)

    # Convert bit string to numpy array of ints
    x_N = np.zeros(N)
    for i in range(len(x)):
        x_N[i] = int(x[i])
    x_N = x_N.astype(int)

    y_N = np.zeros(len(x_N))
    for i in range(len(x_N)):
        y_N[i] = flip(x_N[i], epsilon)

    y_N = y_N.astype(int)
    y_N = y_N % 2        # convert back to binary bitstring
    y = ''.join(map(str, y_N))  # Convert matrix to string

    print("y's: ", y)

    return y

def flip(elem, epsilon):
    # random.random creates a uniformly distributed random floating point number in [0,1)
    return elem if random.random() >= epsilon else int(not elem)
```

5

4 Week 4: October 21 - October 27

4.1 Analysis on the index of frozen bit

I randomly generate a u bit string of length $N = 2^n$, for $n \in [1,8]$ as bit strings longer than 2^8 take too long to run. Through trials of u bit strings of a certain length N, I set the values of a matrix to count the number of times $u[i] = \hat{u}[i]$ for a given i. The goal is to test if the frozen bits are always the same indices throughout the trials. I then divide each value of the matrix by N so that each entry is the fraction of the time for which $u[i] = \hat{u}[i]$, and I plotted these values. This is the resulting plots over 20 trials for each value of N.

4.1.1 $\epsilon = 0.1$

I also recorded the average total number of bits that are correct in a single run of a trial for a given N:

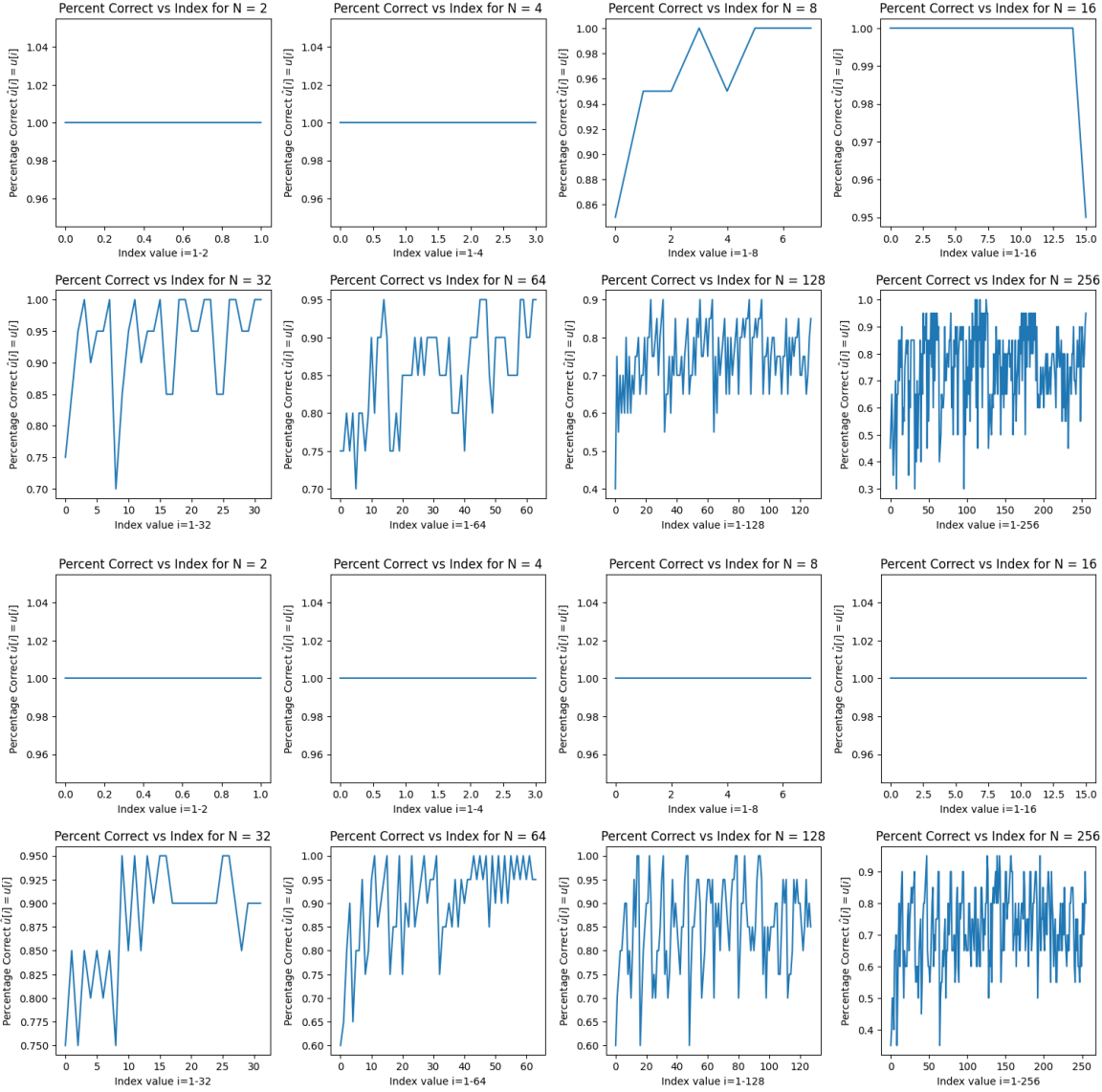
N Value	Average Correct	Average Fraction Correct
$N = 2$	1.65	0.83
$N = 2^2$	3.2	0.8
$N = 2^3$	5.7	0.71
$N = 2^4$	12.5	0.78
$N = 2^5$	21.5	0.67
$N = 2^6$	35.4	0.55
$N = 2^7$	69.3	0.54
$N = 2^8$	135.4	0.53

6

The fraction of good channels is the capacity, which should be $1 - H_b(\epsilon)$.

$$1 - H_b(\epsilon) = 1 + \epsilon \log \epsilon + (1 - \epsilon) \log(1 - \epsilon) = 1 + 0.1 \log 0.1 + 0.9 \log 0.9 = 1 - 0.1 * 3.322 - 0.9 * 0.152 = 1 - 0.469 = 0.53$$

This is approximately the fraction correct for all N, and should be the fraction correct as N increases.

4.1.2 $\epsilon = 0.01$

It doesn't seem that any one particular index is a known frozen bit throughout a bit string of length N, but here does seem to be a trend that only a couple of bits have low fraction of $u[i] = \hat{u}[i]$, a few more indices of high fraction of equality, and most of the other indices hovering over $\frac{1}{2}$. The possible fractional values decrease as N increases.

I also recorded the average total number of bits that are correct in a single run of a trial for a given N:

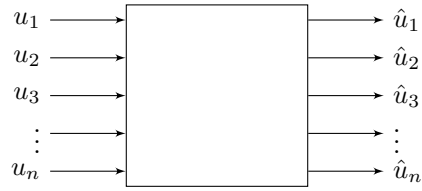
N Value	Average Correct	Average Fraction Correct
$N = 2$	2.0	1.0
$N = 2^2$	4.0	1.0
$N = 2^3$	7.7	0.96
$N = 2^4$	15.95	1.0
$N = 2^5$	29.85	0.93
$N = 2^6$	54.7	0.85
$N = 2^7$	95.85	0.75
$N = 2^8$	189.85	0.74

Capacity should be $1 - H_b(\epsilon)$.

$$1 - H_b(\epsilon) = 1 + \epsilon \log \epsilon + (1 - \epsilon) \log(1 - \epsilon) = 1 + 0.01 \log 0.01 + 0.99 \log 0.99 = 1 - 0.01 * 6.644 - 0.99 * 0.0145 = 0.92$$

7

Goes below capacity...

4.2 u to $\hat{u}$ is a single block

If we apply G_N^{-1} to the left of u_i's and G_N to the right of the $\hat{u}_i$'s, this should result in a block with x inputs and y outputs.

$$x = G_N^{-1}u \qquad y = \hat{u}G_N$$

Append GN-1 and GN to the back and front of the full block
Approximating $\epsilon = 0.2$:

N Value	Epsilon Approximation
$N = 2$	0.0
$N = 2^2$	0.0
$N = 2^3$	0.0
$N = 2^4$	0.19
$N = 2^5$	0.19
$N = 2^6$	0.23
$N = 2^7$	0.31
$N = 2^8$	0.25

Approximating $\epsilon = 0.02$:

N Value	Epsilon Approximation
$N = 2$	0.0
$N = 2^2$	0.0
$N = 2^3$	0.0
$N = 2^4$	0.0
$N = 2^5$	0.0
$N = 2^6$	0.016
$N = 2^7$	0.0078
$N = 2^8$	0.0195

8

5 Week 5: October 28 - November 10

Above doesn't work just yet, need to figure out which bits are frozen before decoding, since this is necessary information for the decoder.

Next working on figuring out which channels are good and which are bad. This is complicated to do with the original channel since it has a large and complex output. Instead, we can degrade the channel and see which indices result in the highest capacities, meaning they are good channels. A good degraded channel means the original channel is also good.

This requires implementation of Algorithms A and C in paper 'How to Construct Polar Codes' by Tal and Vardy. We don't need information on the upgraded channel right now - unless we want to check if a degraded channel is bad and the actual channel is good. This could eventually be used for optimization?

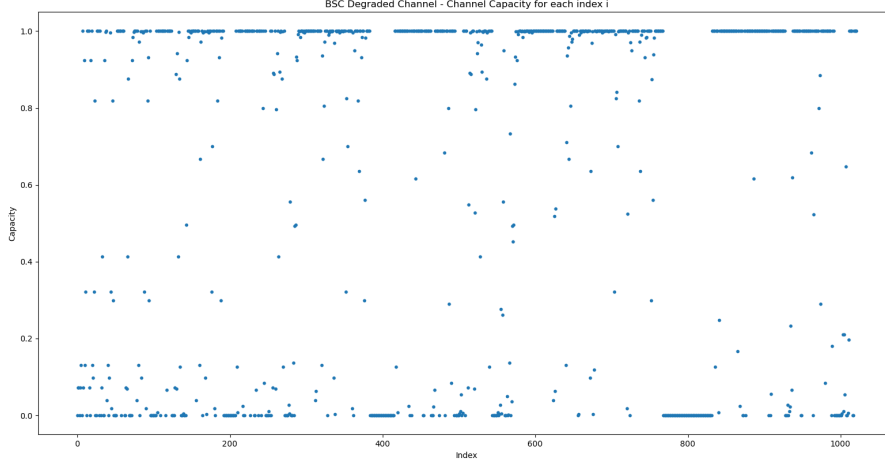
Square and circle functions increase the output size of the function to Y^2 and $X \times Y^2$, respectively, and the degrading merge procedure decreases the output size. This process occurs for each channel index i as it is broken down to its binary equivalent.

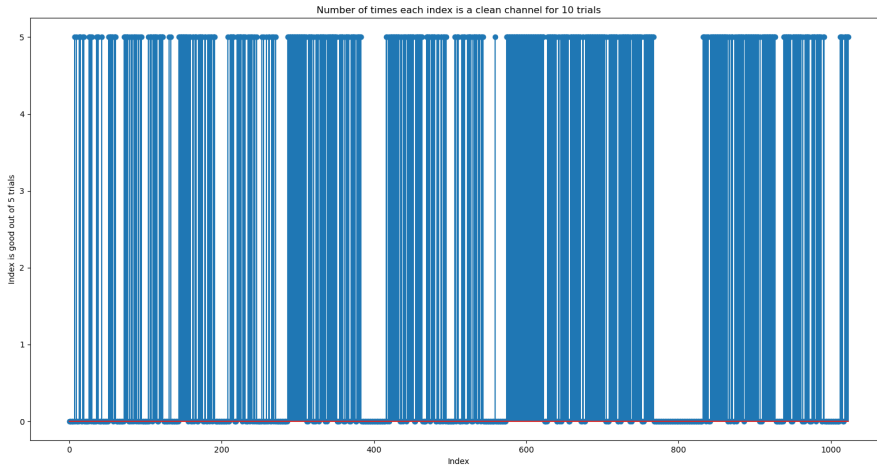
9

6 Week 6: November 11 - November 17

6.1 Clean vs Poor Channels Analysis

Getting the code above to work is difficult. Maintaining heaps and making sure each row of a channel always sums to 1 since this is a probability transition matrix. The square and circle functions preserve this, degrading does not, so just make sure to rescale after running the degrading merge procedure on the channel.

For N=1024, get this plot of capacities:

Using a cutoff of capacity of 0.5, 658 channels are deemed clean and 366 are seen as poor. Cutoff of 0.75, 631 indices are clean and 393 indices are poor. Cutoff of 0.9, 601 indices are clean and 423 are poor. Even at high cutoff for calling a channel clean, more than half are good. This is promising for performance of sending lots of data through. These values are the same after many runs.

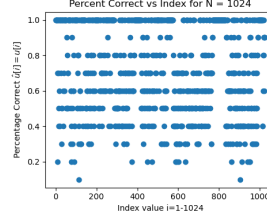
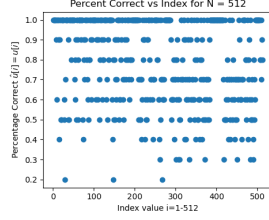
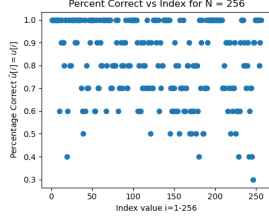
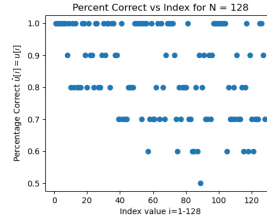
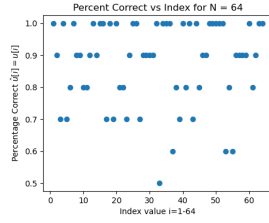
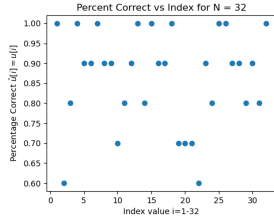
Now to find the indices that are good vs. poor. I will be using the 0.9 cutoff. Each index is always a good channel or always a bad channel by this criteria, so over 5 trials, the plot looks as follows where each index either has 5 good trials or 0.

6.2 Using channel information to implement decoder

Now I can use the information on which channels are good and which are poor to properly decode to the correct $\hat{u}$. I return the list of indices with information on whether each is a good or bad channel. If a channel i is poor, meaning that noise was sent on the channel originally, this must be a frozen bit so the $\hat{u}_i$ prediction is set to original value u_i. Otherwise, this is a message bit and we want to decode it using the likelihood calculations with all y_1^N bits and previous $\hat{u}_1^{i-1}$ and epsilon.

With this new involvement of clean and poor channels, my outputs of $\hat{u}$ are sometimes exactly the same as u, or differ by a few indices of message bits. These seem to be in batches of consecutive indices. Using randomly generated u's spanning from size 32 to 1024, here is the plot of fraction correct for each index in 10 trials:

10

Percent Correct vs Index for N = 32   Percent Correct vs Index for N = 64   Percent Correct vs Index for N = 128

Percent Correct vs Index for N = 256   Percent Correct vs Index for N = 512   Percent Correct vs Index for N = 1024

```python
import numpy as np
from dataclasses import dataclass
import math
import sys
import matplotlib.pyplot as plt

# input: An underlying BMS channel W, a bound mu = 2*nu on the output
#         alphabet size, a code length N = 2^n and index i with binary representation
#         i = <b1,b2,...,bm>_2
# output: BMS channel that is degraded wrt bit channel W_i

# New Data dataclass
@dataclass
class Data_Element:
    a = 0.0
    b = 0.0
    a_prime = 0.0
    b_prime = 0.0
    deltaI = 0.0
    left = None
    right = None
    h = 0 # index of the data element in the heap array

@dataclass
class MinHeap:
    def __init__(self):
        self.heap = []
    def insert(self,item):
        self.heap.append(item)
        index = len(self.heap) - 1
        self._heapify_up(index)
        self._left_right_update(index)
    def _left_right_update(self,index):
        item = self.heap[index]
        # Update left and right for newly inserted element
        if index > 0:
            parent_index = (index - 1) // 2
            if index % 2 == 0: # Right child
                self.heap[parent_index].right = item
            else: # Left child
                self.heap[parent_index].left = item
    def extract_min(self):
```

11

```python
        if len(self.heap) == 0:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()
        min_value = self.heap[0]
        self.heap[0] = self.heap.pop()

        self._heapify_down(0)

        ## Update left and right of elements in heap
        # min_elem.left = min_value
        # min_elem.right = None

        return min_value
    def _heapify_up(self, index):
        orig_index = index
        while index > 0:
            parent_index = (index - 1) // 2
            if self.heap[index].deltaI < self.heap[parent_index].deltaI:
                self.heap[index], self.heap[parent_index]
                              = self.heap[parent_index], self.heap[index]
                index = parent_index
            else:
                break
        # update left and right
        self._left_right_update(orig_index)

    def _heapify_down(self, index):
        orig_index = index
        while True:
            left_child_index = 2 * index + 1
            right_child_index = 2 * index + 2
            smallest = index

            if (left_child_index < len(self.heap) and
                    self.heap[left_child_index].deltaI < self.heap[smallest].deltaI):
                smallest = left_child_index

            if (right_child_index < len(self.heap) and
                    self.heap[right_child_index].deltaI < self.heap[smallest].deltaI):
                smallest = right_child_index

            if smallest == index:
                break

            self.heap[index], self.heap[smallest] = self.heap[smallest], self.heap[index]
            index = smallest

        # update left and right
        self._left_right_update(orig_index)

    def get_min(self):
        if len(self.heap) == 0:
            return None
        return self.heap[0]
    def find_index(self, d):
        for i, element in enumerate(self.heap):
            if element.deltaI == d.deltaI and element.a == d.a and element.b == d.b:
                return i
        # When the element is not found
```
12

```python
                return -1
        def update_order(self,index):
            # decide heapify_up or heapify_down based on comparing to parent
            Pindex = (index-1)//2
            if index >= 0 and self.heap[index].deltaI < self.heap[Pindex].deltaI:
                self._heapify_up(index)
            else:
                self._heapify_down(index)
        def update_vals(self,index,new_deltaI, new_a_prime, new_b_prime):
            self.heap[index].deltaI = new_deltaI
            self.heap[index].a_prime = new_a_prime
            self.heap[index].b_prime = new_b_prime
        def __str__(self):
            elements = [f"deltaI={item.deltaI}, left={item.left}, right={item.right},
                    a={item.a}, b={item.b}\n" for item in self.heap]
            return f"MinHeap([{', '.join(elements)}])"
        def get_size(self):
            size = 0
            for _ in self.heap:
                size += 1
            return size

# Algorithm A
def degrading_procedure(W, mu, b):
    m = len(b)
    Q = degrading_merge(W,mu)
    for j in range(0,m):
        if b[j] == 0:
            script_W = square(Q)
            if not ohno(script_W): print("issue with square")
        else: # b[j] == 1
            script_W = circle(Q)
            if not ohno(script_W): print("issue with circle")
        Q = degrading_merge(script_W,mu)
        if not ohno(Q):
            print("issue in degrading")
            sys.exit()
    return Q

def ohno(W):
    # calculate sum of each row
    row_sums = W.sum(axis=1)

    pos_arr = np.all(W >= 0)
    less_1 = np.all(W <= 1)
    if not pos_arr or not less_1:
        print("arr is not in right form...")

    # Check if each row sum is equal to 1
    return all(np.isclose(row_sum, 1.0) for row_sum in row_sums)

# Arikan channel transformations
def square(W):
    # define new matrix
    new_W = np.zeros((W.shape[0],W.shape[1]**2))
    for u1 in range(W.shape[0]): # i = 0,1
        for y1 in range(W.shape[1]): # i = 00,01,10,11 or higher dimension
            for y2 in range(W.shape[1]):
                val = 0
                for u2 in [0,1]: # since W has binary input
```

13

```
                           # W( y1 | u1^u2 )  W( y2 | u2 )
                           val  += W[ u1^u2 ,  y1 ]  ∗ W[ u2 , y2 ]

                  stry1  =  str ( bin ( y1 ) [ 2 : ] . z f i l l ( int ( math . log2 (W. shape [ 1 ] ) ) ) )
                  stry2  =  str ( bin ( y2 ) [ 2 : ] . z f i l l ( int ( math . log2 (W. shape [ 1 ] ) ) ) )
                  ind  =  bin ( int ( stry1  +  stry2 ,  2 ) ) [ 2 : ]
                  ind  =  int ( ind , 2 )

                  new_W [ u1 ,  ind ]  =  val  /  2

        return  new_W

def  circle (W) :
    new_W  =  np . zeros ( (W. shape [ 0 ] ,   (W. shape [ 1 ] ∗∗2 ) ∗W. shape [ 0 ] ) )
    for  u1  in  range (W. shape [ 0 ] ) :
        for  y1  in  range (W. shape [ 1 ] ) :
            for  y2  in  range (W. shape [ 1 ] ) :
                for  u2  in  range (W. shape [ 0 ] ) :
                    # compute  W( y1 | u1^u2 )  W( y2 | u2 )
                    val  = W[ u1^u2 ,  y1 ]  ∗ W[ u2 , y2 ]

                    # Convert  each  u1 , y1 , y2  to  str  &  concat  to  form  output  index
                    stry1  =  str ( bin ( y1 ) [ 2 : ] . z f i l l ( int ( math . log2 (W. shape [ 1 ] ) ) ) )
                    stry2  =  str ( bin ( y2 ) [ 2 : ] . z f i l l ( int ( math . log2 (W. shape [ 1 ] ) ) ) )
                    stru1  =  str ( bin ( u1 ) [ 2 : ] . z f i l l ( int ( math . log2 (W. shape [ 0 ] ) ) ) )
                    ind  =  bin ( int ( stry1  +  stry2  +  stru1 ,   2 ) ) [ 2 : ]

                    ind  =  int ( ind , 2 )

                    new_W [ u2 ,  ind ]  =  val  /  2
        return  new_W

# Algorithm  C
def  degrading_merge (W, mu ) :
    # W:  X −> Y
    Y  = W. shape [ 1 ]
    L  = Y  //  2

    v  =  mu  //  2

    # Degraded  W is  W  i t s e l f
    if  L <=  v :
        return  W

    for  i  in  range ( 1 , L ) :
        d  =  Data_Element ( )
        d . a  = W[ 0 , i −1]
        d . b  = W[ 1 , i −1]
        d . a_prime  = W[ 0 , i ]
        d . b_prime  = W[ 1 , i ]
        d . deltaI  =  calcDeltaI ( d . a , d . b , d . a_prime , d . b_prime )
        insertRightmost ( d )

    l  = L

    # Here ,  the  heap  and  l i s t  have  L−1  elements  each
    # Number  of  elements  in  heap / l i s t  w i l l  now  be  decreased  to  mu//2 − 1

    while  l  >  v :
```

14

```
        d = getMin()
        a_plus = d.a + d.a_prime
        b_plus = d.b + d.b_prime
        dLeft = d.left
        dRight = d.right
        removeMin()
        l -= 1
        if dLeft is not None:
            # Find where dLeft is in the heap
            index = heap.find_index(dLeft)
            new_a_prime = a_plus
            new_b_prime = b_plus
            new_deltaI = calcDeltaI(dLeft.a, dLeft.b, a_plus, b_plus)
            valueUpdated(index, new_deltaI, new_a_prime, new_b_prime)
        if dRight is not None:
            # Find where dRight is in the heap
            index = heap.find_index(dRight)
            new_a_prime = a_plus
            new_b_prime = b_plus
            new_deltaI = calcDeltaI(dRight.a, dRight.b, a_plus, b_plus)
            valueUpdated(index, new_deltaI, new_a_prime, new_b_prime)

    # Initialize Q
    Q = np.zeros((W.shape[0], heap.get_size()))
    min_elem = removeMin()
    i=0
    while min_elem is not None:
        Q[0,i] = min_elem.a
        Q[1,i] = min_elem.b
        i += 1
        min_elem = removeMin()

    # rescale Q
    Q = rescale(Q)

    return Q

def rescale(W):
    # Sum of each row
    row_sums = W.sum(axis=1)

    # Rescale each element of each row
    rescaled = W / row_sums[:, np.newaxis]

    return rescaled

# Initialize deltaI field for a new data element
def calcDeltaI(a, b, a_prime, b_prime):
    a_plus = a + a_prime
    b_plus = b + b_prime
    return C(a,b) + C(a_prime, b_prime) - C(a_plus, b_plus)

# Helper function for calcDeltaI()
def C(a,b):
    if a + b == 0:
        term1 = 0
    else:
        term1 = -(a+b)*math.log2((a+b)/2)
    if a == 0:
        term2 = 0
```
15

```
        else:
            term2 = a*math.log2(a)
        if b == 0:
            term3 = 0
        else:
            term3 = b*math.log2(b)
        return term1 + term2 + term3

# Inserts element as rightmost element of list and updates heap accordingly
def insertRightmost(d):
    list.append(d)
    heap.insert(d)
    d.h = len(list) - 1

# Returns the data element with smallest delta I
def getMin():
    return heap.get_min()

# Removes the element returned by getMin from both the list and the heap
def removeMin():
    min_elem = heap.get_min()
    if min_elem is not None:
        min_elem = heap.extract_min()
        list[min_elem.h] = 0
        # remove element by setting equal to 0 -- do not alter size of list
        return min_elem
    else:
        return None

# Updates the heap due to a change in deltaI resulting from a merge, no change to list
def valueUpdated(index, new_deltaI, new_a_prime, new_b_prime):
    heap.update_vals(index, new_deltaI, new_a_prime, new_b_prime)
    heap.update_order(index)

# ——————————————————————————————————————————————— #

def execute(N, e, mu):

    m = int(math.log2(N)) # such that N = 2^m

    # BSC flipping probability
    epsilon = e

    # initialize BSC channel with crossover probability epsilon
    W_init = np.array([[1-epsilon, epsilon],[epsilon, 1-epsilon]])

    # initialize heap and list to be altered globally
    global heap
    global list
    heap = MinHeap()      # sorted according to deltaI field
    list = []             # ordered according to corresponding LR value

    capacity = np.zeros(N)

    for i in range(N):
        # Binary representation in list b
        bin_i = bin(i)[2:]
        strbi = str(bin_i)
        b = np.zeros(m)
        for ind in range(len(strbi)-1,-1,-1):
```

```
            if strbi[ind] is not None:
                b[(m-1)-ind] = strbi[ind]
        b = np.flip(b)

        channel = degrading_procedure(W_init,mu,b)

        # print((i,channel))

        # Compute capacity
        # H(Y) - 0.5*H(Y|X=1) - 0.5*H(Y|X=0)

        HY = 0
        for y in range(channel.shape[1]):
            PY = .5*channel[0,y] + .5*channel[1,y]
            if PY != 0: # using 0log0 = 0
                HY += PY * math.log2(1/PY)

        HYX0 = 0
        for y in range(channel.shape[1]):
            PYX0 = channel[0,y]
            if PYX0 != 0:
                HYX0 += PYX0 * math.log2(1/PYX0)

        HYX1 = 0
        for y in range(channel.shape[1]):
            PYX1 = channel[1,y]
            if PYX1 != 0:
                HYX1 += PYX1 * math.log2(1/PYX1)

        capacity[i] = (HY - 0.5*HYX0 - 0.5*HYX1)

    return capacity

def plotting():
    N = 1024
    capacity = execute(N, 0.01, 10)

    num_poor = 0
    num_good = 0
    for elem in capacity:
        if elem < 0.9:
            num_poor += 1
        else:
            num_good += 1
    print("(good,poor) =" + str(num_good) +","+str(num_poor))

    # Plotting now
    i_vector = np.arange(N)
    plt.scatter(i_vector,capacity,s=10)
    plt.title('BSC Degraded Channel - Channel Capacity for each index i')
    plt.xlabel('Index')
    plt.ylabel('Capacity')
    plt.show()

    numtimesgood = np.zeros(N)
    for n in range(5): # will run code 5 times
        capacity = execute(N, 0.01, 10)
        for i in range(N):
            if capacity[i] > 0.9:
                numtimesgood[i] += 1
```

17

```python
    plt.stem(i_vector,numtimesgood)
    plt.xlabel('Index')
    plt.ylabel('Index is good out of 5 trials')
    plt.title('Number of times each index is a clean channel for 10 trials')
    plt.show()

def get_good_ind(N, e, mu):
    m = int(math.log2(N)) # such that N = 2^m

    # BSC flipping probability
    epsilon = e

    # initialize BSC channel with crossover probability epsilon
    W_init = np.array([[1-epsilon, epsilon],[epsilon, 1-epsilon]])

    # initialize heap and list to be altered globally
    global heap
    global list
    heap = MinHeap()      # sorted according to deltaI field
    list = []             # ordered according to corresponding LR value

    capacity = np.zeros(N)

    for i in range(N):
        # Binary representation in list b
        bin_i = bin(i)[2:]
        strbi = str(bin_i)
        b = np.zeros(m)
        for ind in range(len(strbi)-1,-1,-1):
            if strbi[ind] is not None:
                b[(m-1)-ind] = strbi[ind]
        b = np.flip(b)

        channel = degrading_procedure(W_init,mu,b)

        # Compute capacity
        # H(Y) - 0.5*H(Y|X=1) - 0.5*H(Y|X=0)

        HY = 0
        for y in range(channel.shape[1]):
            PY = .5*channel[0,y] + .5*channel[1,y]
            if PY != 0: # using 0log0 = 0
                HY += PY * math.log2(1/PY)

        HYX0 = 0
        for y in range(channel.shape[1]):
            PYX0 = channel[0,y]
            if PYX0 != 0:
                HYX0 += PYX0 * math.log2(1/PYX0)

        HYX1 = 0
        for y in range(channel.shape[1]):
            PYX1 = channel[1,y]
            if PYX1 != 0:
                HYX1 += PYX1 * math.log2(1/PYX1)

        capacity[i] = (HY - 0.5*HYX0 - 0.5*HYX1)
```

18

```python
    good_ind = np.zeros(N)

    for i in range(N):
        if capacity[i] > 0.9:
            good_ind[i] = 1

    return good_ind

print(execute(256, 0.01, 10))
```

19

7   Week 7: November 18-December 4

Should not need to rescale Q, so I continued debugging my code... Below is my updated code.

```python
import numpy as np
from dataclasses import dataclass
import math
import matplotlib.pyplot as plt

# input: An underlying BMS channel W, a bound mu = 2*nu on the output
#         alphabet size, a code length N = 2^n and index i with binary representation
#         i = <b1,b2,...,bm>_2
# output: BMS channel that is degraded wrt bit channel W_i

# New Data dataclass
@dataclass
class Data_Element:
    a = -1.0
    b = -1.0
    a_prime = -1.0
    b_prime = -1.0
    deltaI = -1.0
    left = None
    right = None
    h = -1 # index of the data element in the heap array

@dataclass
class MinHeap:
    def __init__(self):
        self.heap = []
    def insert(self,item):
        self.heap.append(item)
        index = len(self.heap) - 1

        # Update left and right to be left and right probabilities
        if index != 0:
            item.left = self.heap[index - 1]
            self.heap[index -1].right = item
        # do not sort yet, will do this after creating full heap
    def extract_min(self, fix):
        min_elem = self.heap[0]      # minimum element always has index 0 in the heap
        heapsize = len(self.heap)

        # Fix left and right fields if this is not the last element of the heap
        # Don't do this when filling out Q at the end
        if heapsize > 1 and fix:
            if min_elem.left is None: # left-most element in list
                rightelem = min_elem.right
                ind = rightelem.h
                self.heap[ind].left = None
                # previous .right now has .left of None
            elif min_elem.right is None: # right-most element in list
                leftelem = min_elem.left
                ind = leftelem.h
                self.heap[ind].right = None
                # previous .left now has .right of None
            else: # somewhere else in the list
                leftelem = min_elem.left
                Lind = leftelem.h
                self.heap[Lind].right = min_elem.right
                # .left now has .right of min.right
```

20

```python
                rightelem = min_elem.right
                Rind = rightelem.h
                self.heap[Rind].left = min_elem.left
                # .right now has .left of min.left

        self.heap.pop(0) # remove min element
        if heapsize > 1 and fix:
            self.min_sort() # sort heap after removing min
        return min_elem
    def min_sort(self):
        size = len(self.heap)
        if size == 1: # no sorting needed
            self.heap[0].h = 0

        for i in range(size//2 - 1, -1, -1):
            self.heapify(size, i)
    def heapify(self, size, i):
        minI = i
        L = 2*i + 1
        R = 2*i + 2

        if minI < size: self.heap[minI].h = minI
        if L < size: self.heap[L].h = L
        if R < size: self.heap[R].h = R

        if L < size and self.heap[L].deltaI < self.heap[minI].deltaI:
            minI = L
        if R < size and self.heap[R].deltaI < self.heap[minI].deltaI:
            minI = R
        if minI != i:
            # perform the swap
            self.heap[i], self.heap[minI] = self.heap[minI], self.heap[i]

            # Update .h values after the swap
            self.heap[i].h = i
            self.heap[minI].h = minI

            self.heapify(size, minI)
    def get_min(self):
        if len(self.heap) == 0:
            return None
        return self.heap[0]
    def update_vals(self, index, new_deltaI, new_a, new_b, new_a_prime, new_b_prime):
        self.heap[index].deltaI = new_deltaI
        if new_a is not None:
            self.heap[index].a = new_a
        if new_b is not None:
            self.heap[index].b = new_b
        if new_a_prime is not None:
            self.heap[index].a_prime = new_a_prime
        if new_b_prime is not None:
            self.heap[index].b_prime = new_b_prime
    def __str__(self):
        elements = [f"h={item.h}, deltaI={item.deltaI}, left={item.left},
                right={item.right}, a={item.a}, b={item.b}, a'={item.a_prime},
                b'={item.b_prime}\n" for item in self.heap]
        return f"MinHeap([{', '.join(elements)}])"
    def get_size(self):
        return len(self.heap)
```

21

```python
# Algorithm A
def degrading_procedure(W, mu, b):
    m = len(b)
    Q = degrading_merge(W,mu)
    for j in range(0,m):
        if b[j] == 0:
            script_W = square(Q)
        else: # b[j] == 1
            script_W = circle(Q)
        Q = degrading_merge(script_W,mu)

    return Q

# Arikan channel transformations
def square(W):
    # define new matrix
    new_W = np.zeros((W.shape[0],W.shape[1]**2))
    for u1 in range(W.shape[0]): # i = 0,1
        for y1 in range(W.shape[1]): # i = 00,01,10,11 or higher dimension
            for y2 in range(W.shape[1]):
                val = 0
                for u2 in range(W.shape[0]):
                    # W(y1|u1^u2) W(y2|u2)
                    val += W[u1^u2, y1] * W[u2,y2]
                stry1 = str(bin(y1)[2:].zfill(int(math.log2(W.shape[1]))))
                stry2 = str(bin(y2)[2:].zfill(int(math.log2(W.shape[1]))))

                ind = bin(int(stry1 + stry2, 2))[2:]
                ind = int(ind,2)

                new_W[u1, ind] = val / 2
    return new_W

def circle(W):
    new_W = np.zeros((W.shape[0], (W.shape[1]**2)*W.shape[0]))
    for u1 in range(W.shape[0]):
        for y1 in range(W.shape[1]):
            for y2 in range(W.shape[1]):
                for u2 in range(W.shape[0]):
                    # compute W(y1|u1^u2) W(y2|u2)
                    val = W[u1^u2, y1] * W[u2,y2]

                    # Convert each u1,y1,y2 to str & concat to form output index
                    stry1 = str(bin(y1)[2:].zfill(int(math.log2(W.shape[1]))))
                    stry2 = str(bin(y2)[2:].zfill(int(math.log2(W.shape[1]))))
                    stru1 = str(bin(u1)[2:].zfill(int(math.log2(W.shape[0]))))
                    ind = bin(int(stry1 + stry2 + stru1, 2))[2:]
                    ind = int(ind,2)

                    new_W[u2, ind] = val / 2
    return new_W

# Algorithm C
def degrading_merge(W,mu):
    # W: X -> Y
    Y = W.shape[1]
    L = Y // 2

    v = mu // 2
```

```python
# Degraded W is W itself if output size already <= mu
if L <= v:
    return W

# 1 <= LR(y1) <= LR(y2) <= ... <= LR(y_L)
LR = W[0,:W.shape[1]//2] / W[1,:W.shape[1]//2]
# compute LR of first half of W, second half will be reciprocal

# Want to pick one representative from corresponding columns in first
# & second halves but want all LR >= 1
ge1 = LR >= 1 # all indices are TRUE where LR >= 1
W[:, :W.shape[1]//2][:,~ge1] = W[::-1, :W.shape[1]//2][:, ~ge1]
# reverse columns where LR < 1
new_W = W[:,:W.shape[1]//2]

LR1 = new_W[0,:] / new_W[1,:]
LRind = np.argsort(LR1)

new_W = new_W[:,LRind]

for i in range(1,L):
    d = Data_Element()
    d.a = new_W[0,i-1]
    d.b = new_W[1,i-1]
    d.a_prime = new_W[0,i]
    d.b_prime = new_W[1,i]
    d.deltaI = calcDeltaI(d.a,d.b,d.a_prime,d.b_prime)
    insertRightmost(d)

# Now that heap is built, need to arrange heap according to deltaI value
heap.min_sort()

l = L

# Here, the heap has L-1 elements each
# Number of elements in heap will now be decreased to mu-2
while l > v:
    d = getMin()
    a_plus = d.a + d.a_prime
    b_plus = d.b + d.b_prime
    dLeft = d.left
    dRight = d.right
    removeMin()

    l -= 1

    if dLeft is not None:
        lind = dLeft.h # index of dLeft in heap
        new_a_prime = a_plus
        new_b_prime = b_plus
        new_deltaI = calcDeltaI(dLeft.a, dLeft.b, a_plus, b_plus)
        valueUpdated(lind, new_deltaI, None, None, new_a_prime, new_b_prime)
    if dRight is not None:
        rind = dRight.h # index of dRight in heap
        new_a = a_plus
        new_b = b_plus
        new_deltaI = calcDeltaI(a_plus, b_plus, dRight.a_prime, dRight.b_prime)
        valueUpdated(rind, new_deltaI, new_a, new_b, None, None)

# Initialize Q, Y output space of mu
```

23

```python
        heapsize = heap.get_size()
        Q = np.zeros((W.shape[0],mu))

        for i in range(heapsize):
            min_elem = removeMin(fix = 0)
            Q[0,i] = min_elem.a
            Q[1,i] = min_elem.b
            Q[0,(Q.shape[1]-1)-i] = min_elem.b
            Q[1,(Q.shape[1]-1)-i] = min_elem.a
            if min_elem.right is None:
                sp_ind = mu//2 - 1
                Q[0,sp_ind] = min_elem.a_prime
                Q[1,sp_ind] = min_elem.b_prime
                Q[0,sp_ind+1] = min_elem.b_prime
                Q[1,sp_ind+1] = min_elem.a_prime

        return Q

# Initialize deltaI field for a new data element
def calcDeltaI(a,b,a_prime,b_prime):
    a_plus = a + a_prime
    b_plus = b + b_prime
    return C(a,b) + C(a_prime,b_prime) - C(a_plus,b_plus)

# Helper function for calcDeltaI()
def C(a,b):
    if a + b == 0:
        term1 = 0
    else:
        term1 = -(a+b)*math.log2((a+b)/2)
    if a == 0:
        term2 = 0
    else:
        term2 = a*math.log2(a)
    if b == 0:
        term3 = 0
    else:
        term3 = b*math.log2(b)
    return term1 + term2 + term3

# Inserts element as rightmost element of list and updates heap accordingly
def insertRightmost(d):
    heap.insert(d)

# Returns the data element with smallest delta I
def getMin():
    return heap.get_min()

# Removes the element returned by getMin from heap
def removeMin(fix = 1):
    min_elem = heap.get_min()
    if min_elem is not None:
        min_elem = heap.extract_min(fix)
        return min_elem
    else:
        return None

# Updates the heap due to a change in deltaI resulting from a merge, no change to list
def valueUpdated(index, new_deltaI, new_a, new_b, new_a_prime, new_b_prime):
    heap.update_vals(index, new_deltaI, new_a, new_b, new_a_prime, new_b_prime)
```

24

```python
        heap.min_sort()

# ————————————————————————————————————————————————— #

def execute(N, e, mu):
    m = int(math.log2(N))  # N = 2^m
    epsilon = e    # BSC flipping probability

    # initialize BSC channel with crossover probability epsilon
    W_init = np.array([[1-epsilon, epsilon],[epsilon, 1-epsilon]])

    # initialize heap to be altered globally
    global heap
    heap = MinHeap()     # sorted according to deltaI field

    capacity = np.zeros(N)

    for i in range(N):
        # Binary representation in list b
        strbi = str(bin(i)[2:]).zfill(m)
        b = list(map(int, list(strbi)))

        channel = degrading_procedure(W_init,mu,b)

        # Compute capacity        H(Y) - 0.5*H(Y|X=1) - 0.5*H(Y|X=0)
        HY = 0
        for y in range(channel.shape[1]):
            PY = .5*channel[0,y] + .5*channel[1,y]
            if PY != 0: # using 0log0 = 0
                HY += PY * math.log2(1/PY)

        HYX0 = 0
        for y in range(channel.shape[1]):
            PYX0 = channel[0,y]
            if PYX0 != 0:
                HYX0 += PYX0 * math.log2(1/PYX0)

        HYX1 = 0
        for y in range(channel.shape[1]):
            PYX1 = channel[1,y]
            if PYX1 != 0:
                HYX1 += PYX1 * math.log2(1/PYX1)

        capacity[i] = (HY - 0.5*HYX0 - 0.5*HYX1)

    # print(capacity)
    return capacity

def plotting():
    N = 1024
    capacity = execute(N, 0.01, 16)

    num_poor = 0
    num_good = 0
    for elem in capacity:
        if elem < 0.9:
            num_poor += 1
        else:
            num_good += 1
    print("(good,poor) =" + str(num_good) +","+str(num_poor))
```

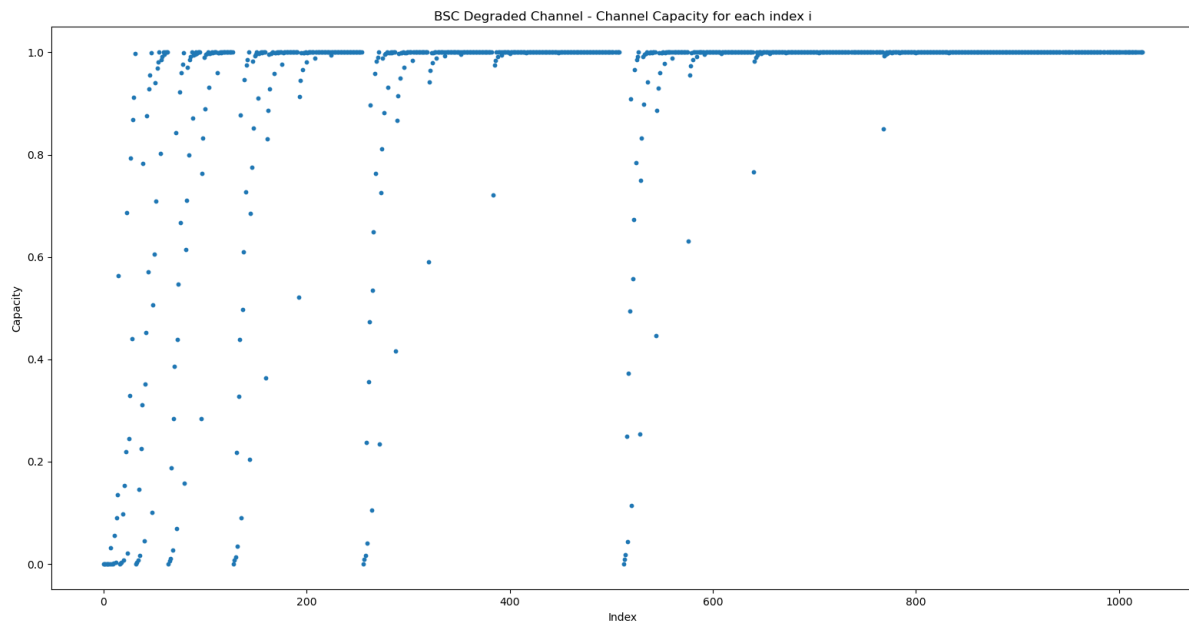
25

```python
    # Plotting now
    i_vector = np.arange(N)
    plt.scatter(i_vector, capacity, s=10)
    plt.title('BSC Degraded Channel - Channel Capacity for each index i')
    plt.xlabel('Index')
    plt.ylabel('Capacity')
    plt.show()

def get_good_ind(N, e, mu):
    m = int(math.log2(N)) # such that N = 2^m

    # BSC flipping probability
    epsilon = e

    # initialize BSC channel with crossover probability epsilon
    W_init = np.array([[1-epsilon, epsilon],[epsilon, 1-epsilon]])

    # initialize heap to be altered globally
    global heap
    heap = MinHeap()     # sorted according to deltaI field

    capacity = np.zeros(N)

    for i in range(N):
        # Binary representation in list b
        bin_i = bin(i)[2:]
        strbi = str(bin_i)
        b = np.zeros(m)
        for ind in range(len(strbi)-1,-1,-1):
            if strbi[ind] is not None:
                b[(m-1)-ind] = strbi[ind]
        b = np.flip(b)

        channel = degrading_procedure(W_init,mu,b)

        # Compute capacity
        # H(Y) - 0.5*H(Y|X=1) - 0.5*H(Y|X=0)

        HY = 0
        for y in range(channel.shape[1]):
            PY = .5*channel[0,y] + .5*channel[1,y]
            if PY != 0: # using 0log0 = 0
                HY += PY * math.log2(1/PY)

        HYX0 = 0
        for y in range(channel.shape[1]):
            PYX0 = channel[0,y]
            if PYX0 != 0:
                HYX0 += PYX0 * math.log2(1/PYX0)

        HYX1 = 0
        for y in range(channel.shape[1]):
            PYX1 = channel[1,y]
            if PYX1 != 0:
                HYX1 += PYX1 * math.log2(1/PYX1)

        capacity[i] = (HY - 0.5*HYX0 - 0.5*HYX1)

    good_ind = np.zeros(N)
```

26

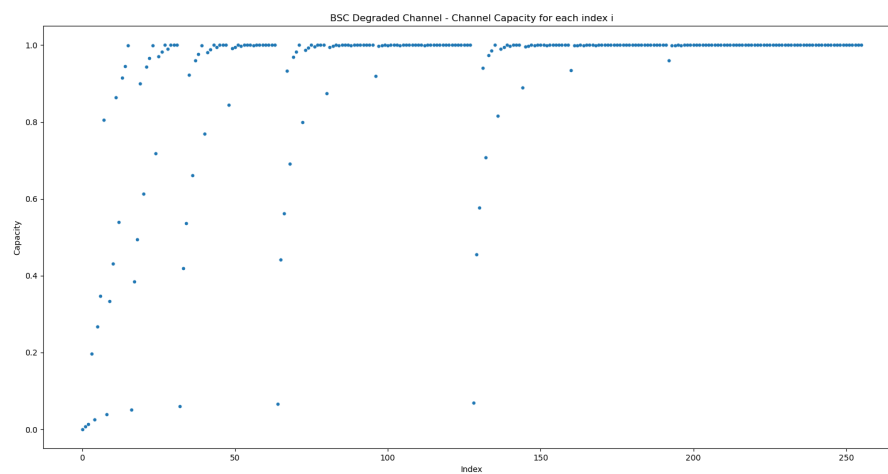
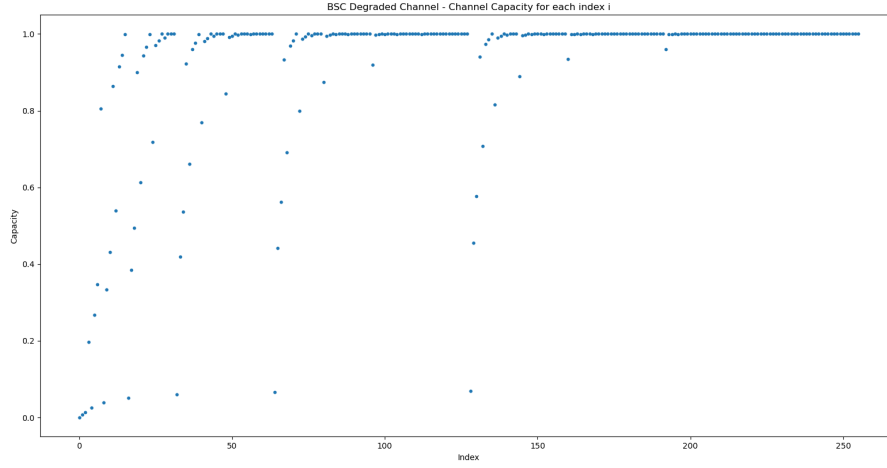
```
    for i in range(N):
        if capacity[i] > 0.9:
            good_ind[i] = 1

    return good_ind

plotting()
```

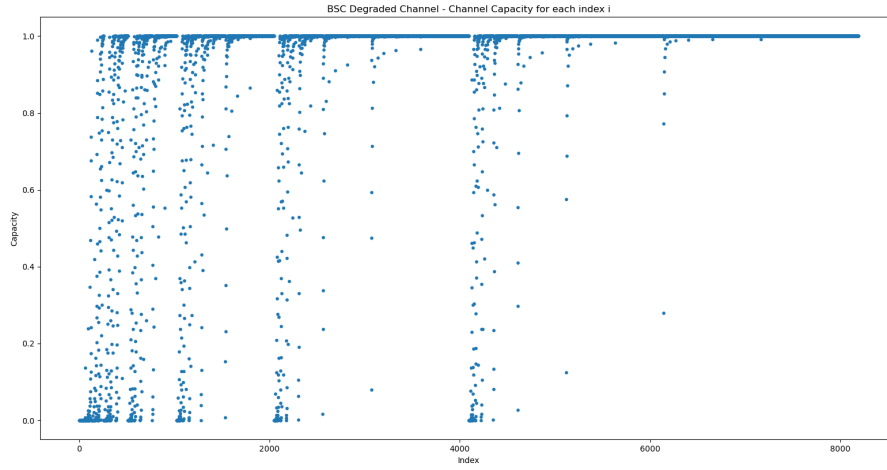
7.1 Capacity and Guessing Probability – Plots

This results in the capacities plot below for $N = 1024$, $\mu = 16$, $\epsilon = 0.01$.

$\mu = 32$ takes much more time than $\mu = 16$, so I plotted $N = 256$ where $\mu = 16, 32$:
$\mu = 16$:

$\mu = 32$:

27

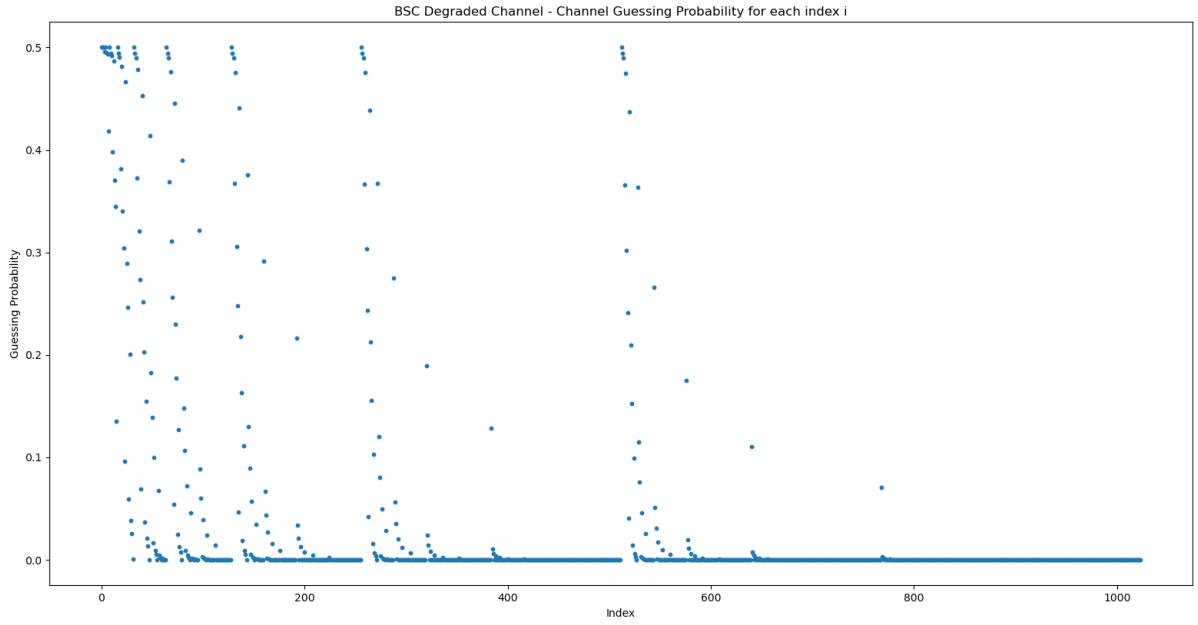
BSC Degraded Channel - Channel Capacity for each index i

These plots look the same... The only difference is the time it takes the algorithm to complete....
To further prove this, I tried $\mu = 4$ with $N = 8192$.

BSC Degraded Channel - Channel Capacity for each index i

The fraction of good to bad channels is $881{:}143 = 86\%$. This value should approach the capacity of the channel, $1 - H_b(\epsilon) = 0.92$ for $\epsilon = 0.01$. As the number of bits increases over 1024, I expect this to happen. The distinction of good and bad classes is based on capabity. I will below use an alternative distinction which is the the guessing probability:

$$\sum_{i=1}^{m} \min p_1, q_i \cdot \frac{1}{2} \leq \sum_{i=1}^{m} \sqrt{p_i q_i} \cdot \frac{1}{2}$$

which computes the probability of getting a bit incorrect. The second equation is Eq. 7 of Arikan's paper which is Z, defining the rate of polarization and allowing me to sort the indices into the low noise and high noise.
For a small $N = 32$ and using a threshold of 0.025 (values less than 0.025 are good channels), I see that the same channels are denoted good in capacity and in guessing probability. Below is the plot of guessing probability for $N = 1024, \mu = 16, \epsilon = 0.01$.

28

BSC Degraded Channel - Channel Guessing Probability for each index i

For this configuration, the fraction of good to bad channels using the guessing probability is $886{:}138 = \approx 87\%$. There does not seem to be a big difference between capacity and guessing probability distinctions. I will be using guessing probability from this point on with threshold 0.025.

7.2 Good vs Bad channels in decoder

It doesn't seem to be true that good channels are actually good, the decoder does provide a $\hat{u}$ that almost perfectly matches the input u...

7.3 Simulator

*** INSERT PIC OF CHANNEL ***

29

8 Week 8: December 5 - December 8

8.1 Permuting good indices

When I multiplied the indices by B_N, the indices array of 1s (good channels) and 0s (bad channels) are largely unchanged. For small N, this array is entirely unchanged, but for $N >= 128$, only a few indices are changed. So this does not seem to be the issue.

8.2 Brute-Force Channel Representation

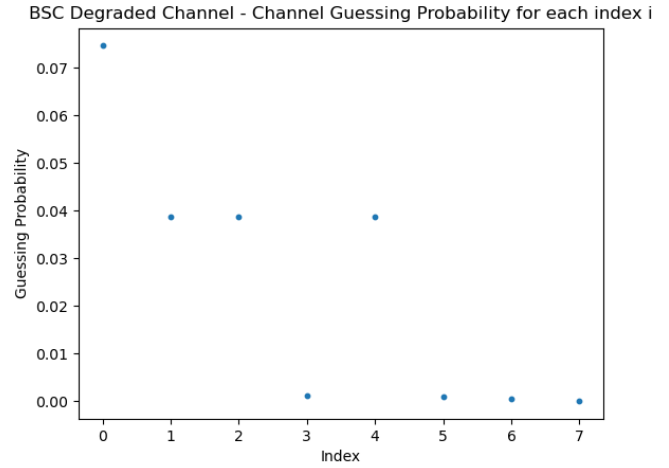
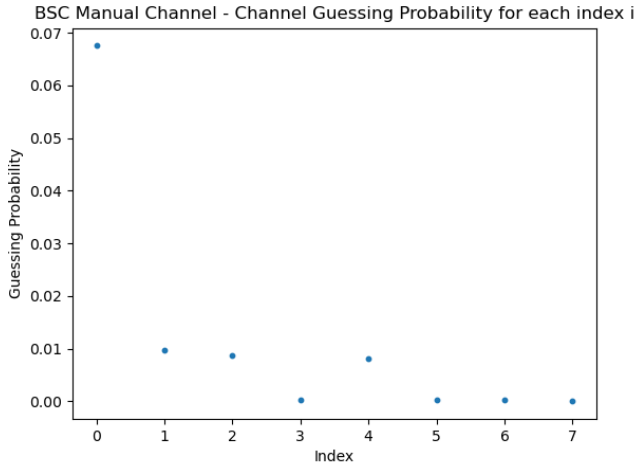
I wanted to construct the channel for $N = 8$. For each index i, the channel be of dimensions $2 \times 2^{index+N}$. Each probability will be as follows:

$$W_N^{(i)} = (y_1^N, \hat{u}_1^{i-1}|\hat{u}_i) = \sum_{\hat{u}_{i+1}^N \in \mathcal{X}^{N-i}} \frac{1}{2^N - 1} W_n(y_1^N|\hat{u}_1^N)$$

where

$$W_n(y_1^N|\hat{u}_1^N) = W^N(y_1^N|u_1^N G_N) = W^N(y_1^N|\hat{x}_1^N)$$

and each $W(Y_i|\hat{x}_i) = 1 - \epsilon$ if $y_i = \hat{x}_i$ and ϵ otherwise.

For $N = 8$, there is not much polarization, but I still calculate and plot the guessing probabilities for this matrix, which is the smallest value of N for which this can be computed. Using the above equation, I was unable to get the rows of the matrix to sum to 1, while I know this must be true. For now, I am scaling the rows of the matrix so that they sum to 1. Below are the plots for the manual channel calculation of guessing probability and the degraded channel calculation of guessing probability. Manual channel on the left and degraded channel on the right...

8.3 Checking decoder...

There are many likelihoods of 1.0. While some of the misclassified bits have a likelihood of 1.0, there are also some with very high or very low likelihood calculations, meaning that the decoder is fairly confident that a certain bit of $\hat{u}$ should be a 1 or 0. But this is not correct...

8.4 Identify subchannels that are the culprit

I want to identify subchannels that are the culprit to getting a $\hat{u}$ incorrect and run the degraded channel algorithm on just this subchannel to see if it really should be a good channel as described. Each estimate of $\hat{u}_i$ relies on all previous values $\hat{u}_1^{i-1}$, so a single error can cause many errors in the estimation of the full $\hat{u}_1^n$.

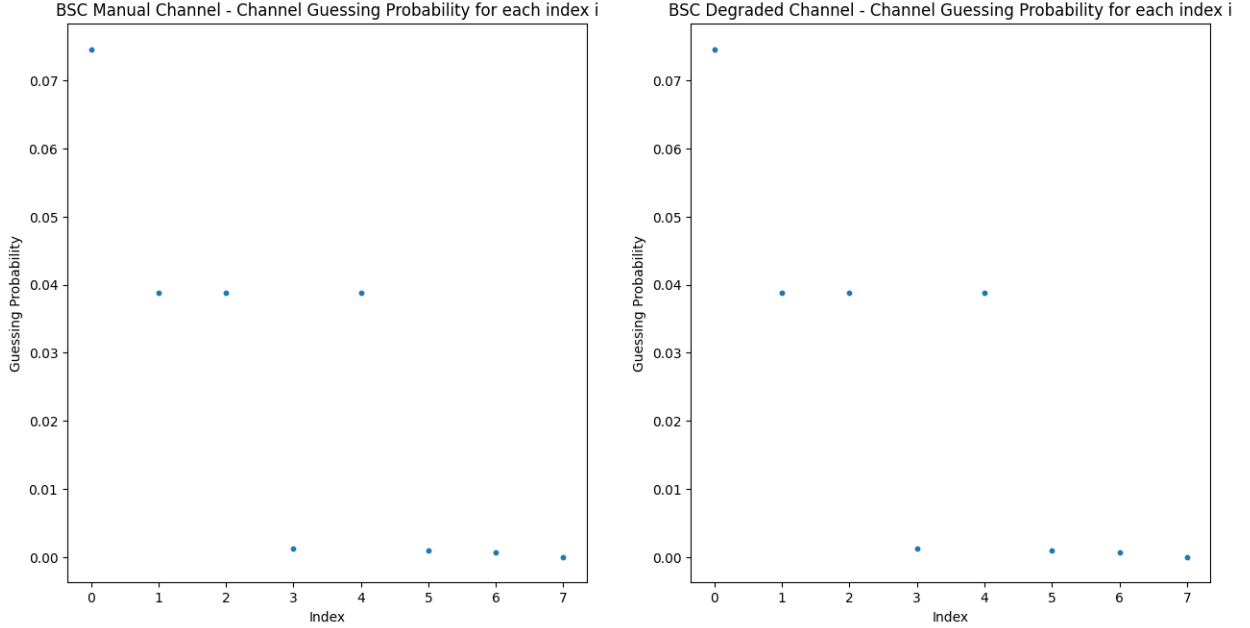
I have noticed that the subchannel culprit, i.e. the first index for which the estimate of $\hat{u}_i \neq u_i$, has a likelihood of 1.0 most of the time. Occassionally, the value will be smaller or larger, but only by an order of 3 in either direction (10^3 or 10^{-3}).

It's possible that there is an error in the decoder that predicts a likelihood of 1.0 when the calculation is another value...

30

9 Week 9: December 9 - December 13

9.1 Updated: Brute-Force Channel Representation

Upon fixing the brute force channel composition code without having to rescale the formed channel rows, I get the same plot as found by the degraded channel algorithm, shown in the image below.

From these calculations for small $N = 8$ (large N calculations take too much time), I am confident that the degrading algorithm is selecting the correct channels to classify as 'good' and performing guess probability calculations correct.

9.2 Decoder

The first fix I made to my decoder is the following line addition:

```
u_hato = (u_hat[::2]).astype(int) # get only odd rows 1,3,...
u_hate = (u_hat[1::2]).astype(int) # get only even rows 2,4,...

if len(u_hato) != len(u_hate):
        u_hato = u_hato[:len(u_hato)-1]

# Kronecker sum - add componentwise
new_uhat = np.bitwise_xor(u_hato, u_hate)
```

Before, I had an extra entry in the new-uhat array when the size of the original u-hat was odd, so the odd array had one more element than the even array. The equations in paper [1] actually state that we want to disregard of this element. This decreased the number of indices in $\hat{u}$ that were different than u, but this number was still inconsistently 0 and very variable.

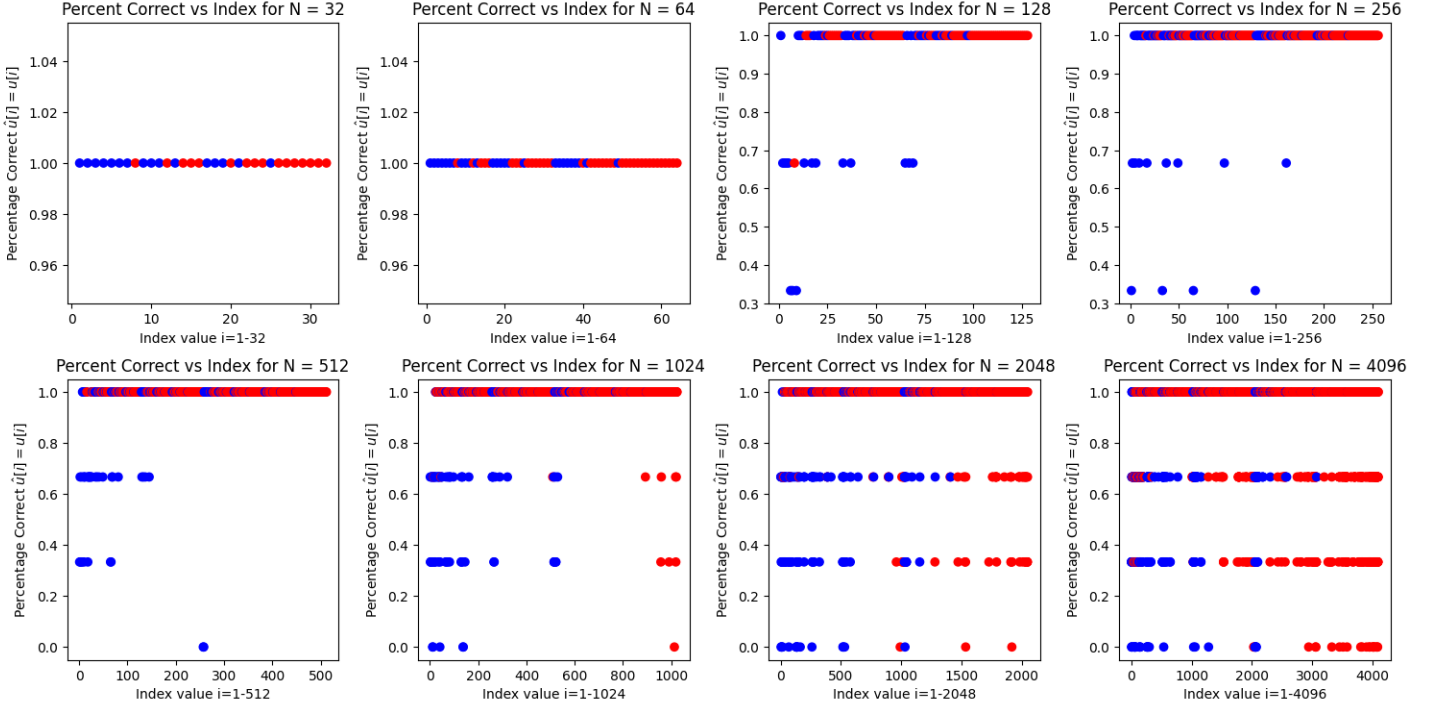
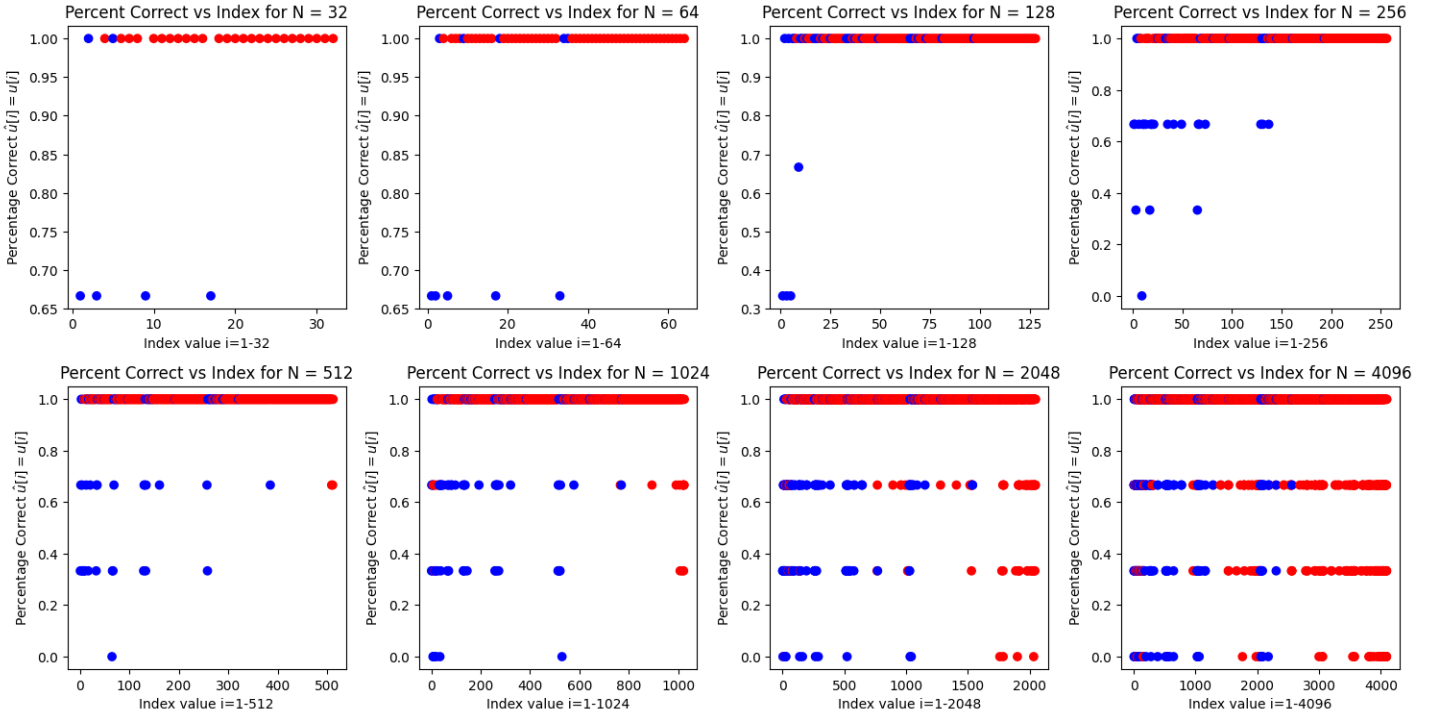
The next fix is the following: In the previous meeting, I tried decreasing the threshold that I choose for classification of a good channel. I originally had 0.001, which led to many incorrect bits in the comparison of $\hat{u}$ to u. I knew that this threshold would be too high because the subchannel 'culprit' that would be the first incorrect bit estimation would have a likelihood of 1.0, meaning this was more of a decent channel than a 'good' channel by our necessity. By lowering the threshold even by a factor of 10 to 0.0001, the number of bits incorrect from $\hat{u} \to u$ is very consistently 0, and sometimes a small number like 1 or 2. This is for an $N = 512$. This threshold results in 376 good channels, which is a fraction of 73% of channels that are being denoted 'good' vs 'poor'. I expect this number to approach the capacity of 92% (for $\epsilon = 0.01$) when N is large.

9.3 Quality of each channel - My own analysis

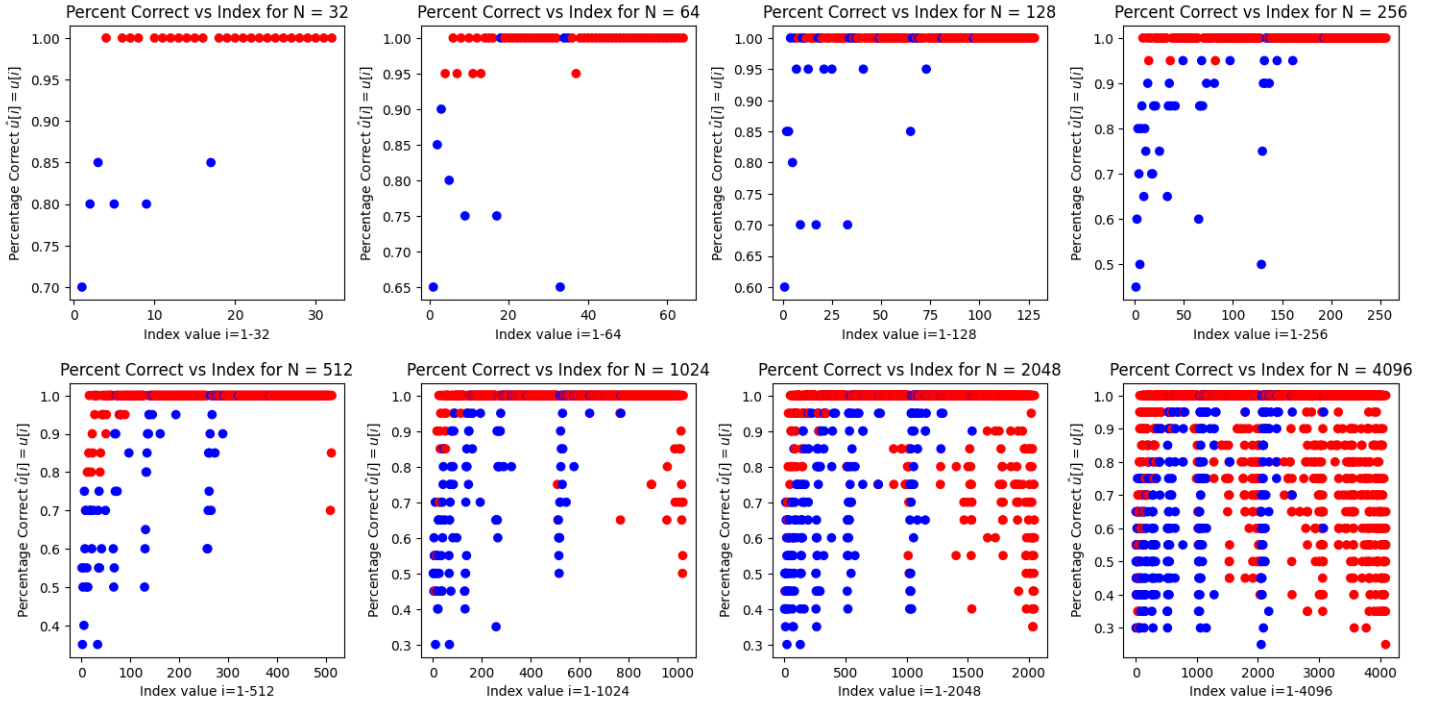
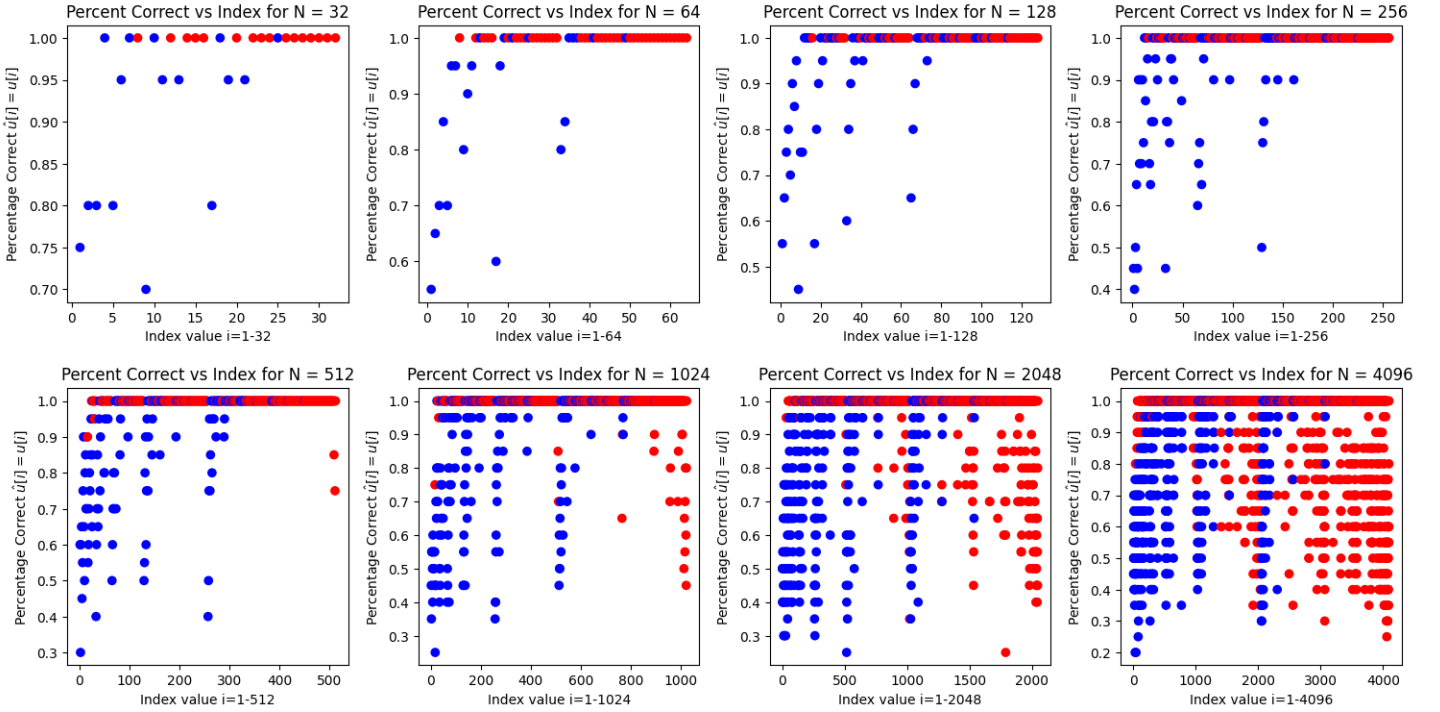
When I increase $N = 2048$, it seems harder to decrease the threshold to a good value - this threshold will need to be very small. It would be helpful to know what to set the threshold to based on the value of N.

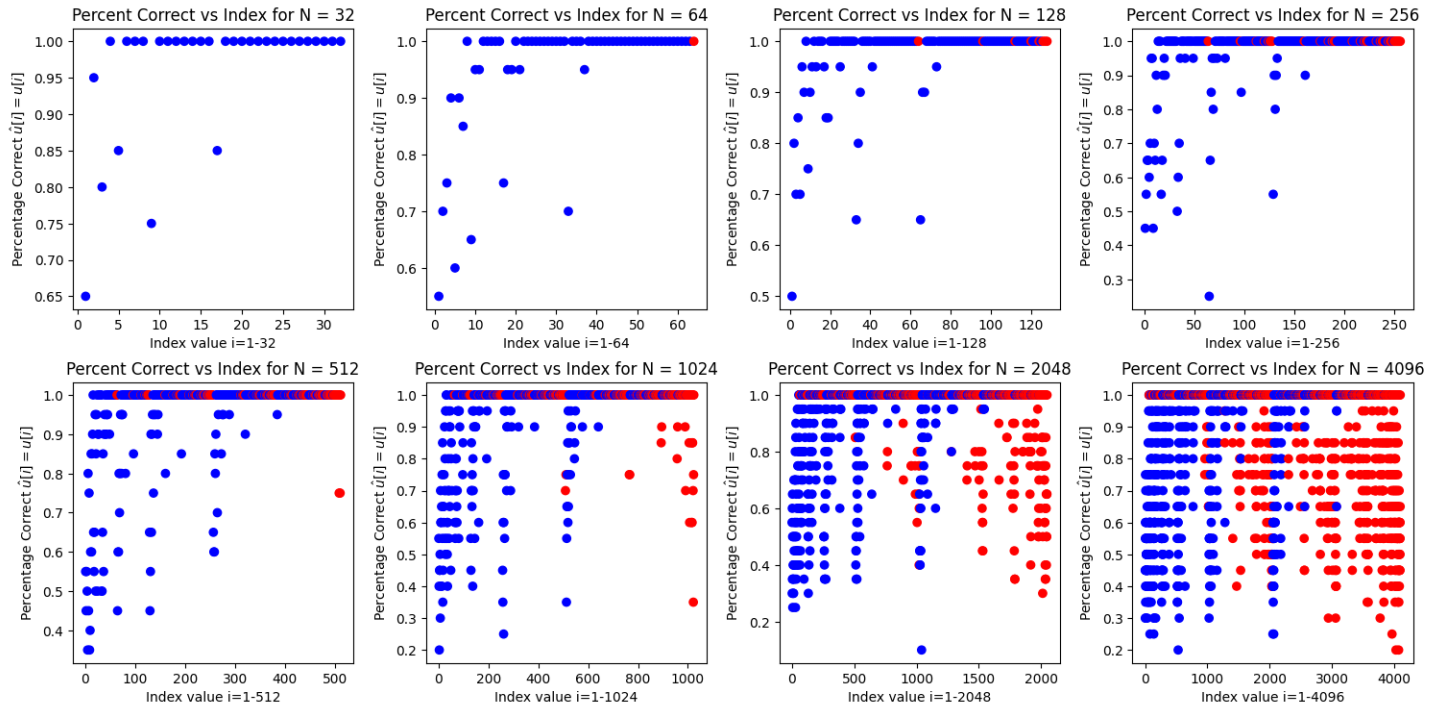
31

I would expect that the probability that a bit channel is incorrect to scale as $\frac{1}{N}$ but this does not seem to be happening. I will conduct an experiment where I look at each individual index and set all previous $\hat{u}_1^{i-1}$ to be u_1^{i-1} to prevent previous errors to isolate how often a certain index i is correct.

Using a threshold of 0.0001, a total of only 3 trials, and color-coding the plot so that red points denote a good channel using the threshold and blue points denote a poor channel.

Using a threshold of $\frac{1}{N}$, which is what I would expect should work, I get the following plot (also with only 3 trials):

These plots look relatively similar, but number of 'good' channels that have a low fraction of being correct is greater in higher N cases. This indicates that the threshold perhaps must be less than $\frac{1}{N}$.

Now I want to see the impact of increasing total number of trials to 20 while keeping the threshold as $\frac{1}{N}$:
Below is the resulting plot from this experiment, it is still true that the number of 'good' channels that do not perform well over the trials is greater for larger N.

32

I will now try a threshold of $\frac{1}{N^2}$.

I made the threshold very small, 1E-15, and I get a similar plot as earlier:

33

The increased amount of errors (misclassied good indices) for large N does not seem to depend on the chosen threshold...

34

10   Week 10: December 14 - January 2

At the previous meeting, we noticed that there were runtime warning associated with higher N calculations. In the recursion of the likelihood calculations, the likelihood values are taking on values as high as 10^{250} for $N = 256$, and even higher for higher N. I attempted to utilize increased overflow to allow larger numbers to be represented in Python, but as N increases, this value will increase so the highest value to be calculated will blow up.

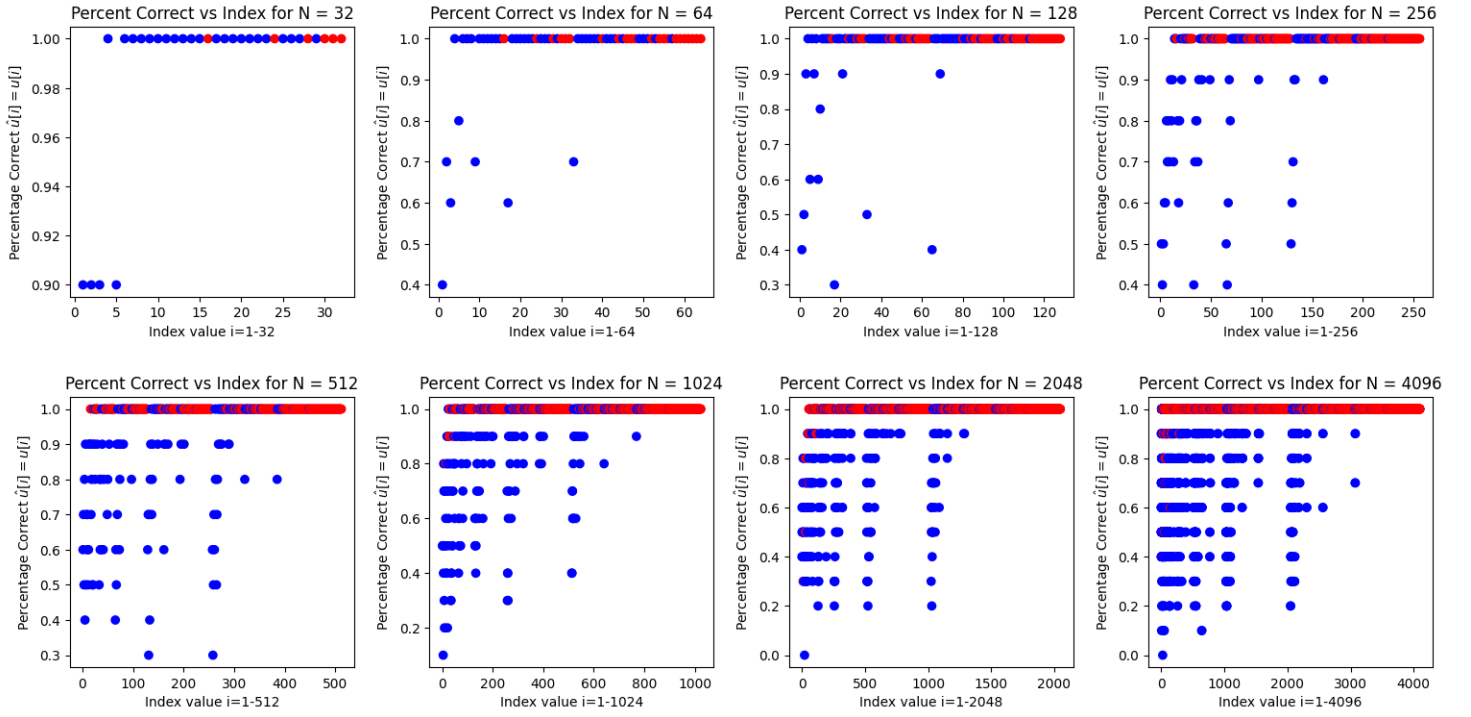
Since we don't really need to know the exact value of the likelihood value but rather it's value compared to 1, I decided to change the code to instead conduct calculations in terms of the order of magnitude of the number instead of the number itself. Here is the logic for this new code:

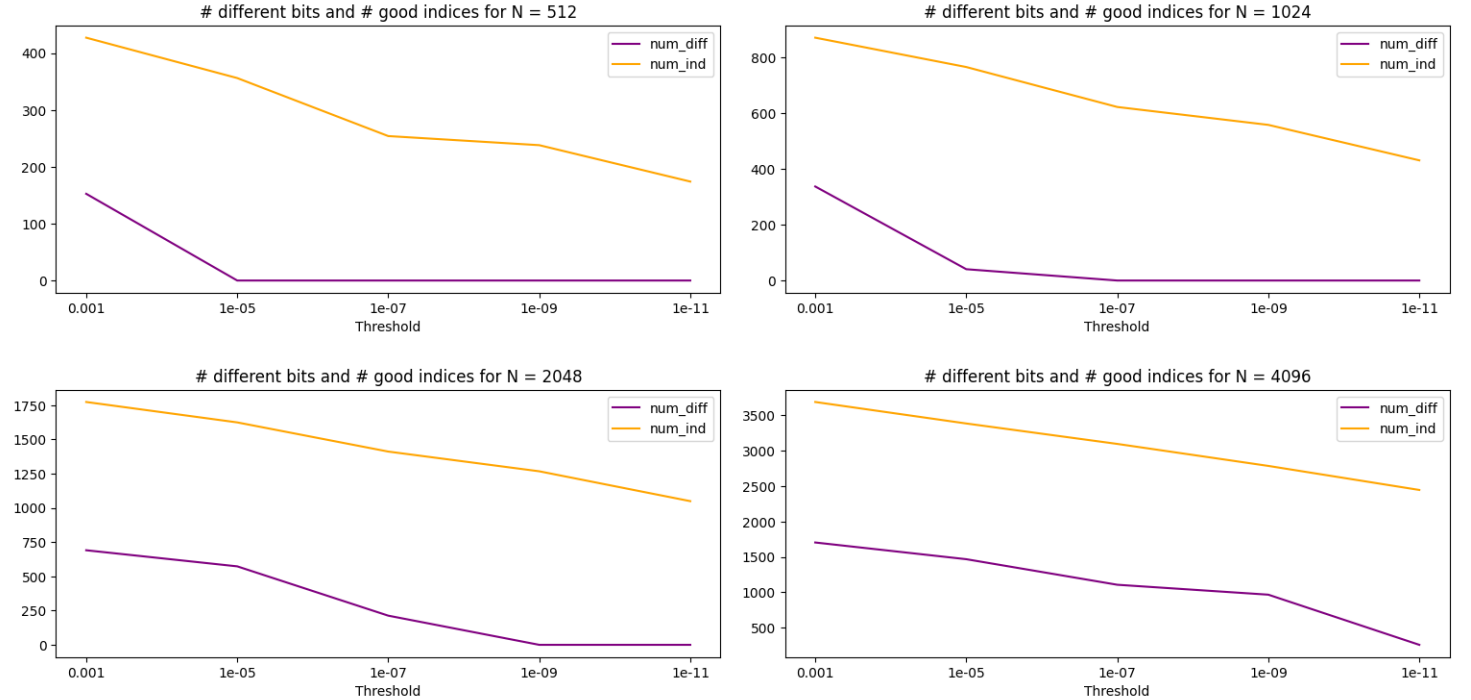
```python
def decoding(u, y, epsilon, mu, thresh):
    N = len(y)

    # Convert bit string to numpy array of ints
    u_N = np.array([int(bit) for bit in u], dtype=int)
    y_N = np.array([int(bit) for bit in y], dtype=int)

    u_hat_N = np.zeros(N)

    good_indices = frozenind.get_good_ind(N, epsilon, mu, thresh)

    for i in range(N):
        if good_indices[i]: # This is an information bit
            L_i = likelihood(y_N, u_hat_N[:i], epsilon)
            if L_i >= 0: # likelihood >= 1, order of mag of likelihood >= 0
                u_hat_N[i] = 0
            else:
                u_hat_N[i] = 1
        else: # This is a frozen bit
            u_hat_N[i] = u_N[i]

    # Convert matrix to string
    u_hat = ''.join(map(str, u_hat_N.astype(int)))

    return u_hat

def likelihood(y_N, u_hat, epsilon):
    N = len(y_N)
    if N == 1:
        if y_N == 0: # y_N is a single value here
            val = (1-epsilon) / epsilon
            return order_mag(val)
        else: # y_1 = 1
            val = epsilon / (1-epsilon)
            return order_mag(val)
    else:
        # y's needed as part of definition of new recursive likelihoods
        firsthalf_y = y_N[:N//2]
        lasthalf_y = y_N[N//2:]

        u_hato = (u_hat[::2]).astype(int) # get only odd rows 1,3,...
        u_hate = (u_hat[1::2]).astype(int) # get only even rows 2,4,...

        if len(u_hato) != len(u_hate):
            u_hato = u_hato[:len(u_hato)-1]

        # Kronecker sum - add componentwise
        new_uhat = np.bitwise_xor(u_hato, u_hate)

        like1 = likelihood(firsthalf_y, new_uhat, epsilon)
        like2 = likelihood(lasthalf_y, u_hate, epsilon)
```

35

```
            # like1 and like2 will represent the order of magnitude of the likelihood calculations

            if len(u_hat) % 2 == 0: # Equation 75
                return safe_compute_even(like1, like2)
            else: # i is even, Equation 76
                power = 1 - 2*u_hat[len(u_hat)-1] # either 1 or -1
                return safe_compute_odd(like1, like2, power)

def order_mag(value):
    order = round(math.log10(abs(value)))
    return order

def safe_compute_even(like1, like2):
    # (like1*like2 + 1) / (like1 + like2)
    order1 = like1
    order2 = like2

    # Product of 2 likelihood values
    numerator = order1 + order2

    # Add 1 to the sum above
    if order1 + order2 < 1:
        numerator = 1
    if order1 < order2:
        denominator = order2
    else:
        denominator = order1

    # Division of numerator and denominator
    return numerator - denominator

def safe_compute_odd(like1, like2, power):
    # (like1)**power * like2
    if power == -1:
        order1 = -like1
    elif power == 1:
        order1 = like1
    order2 = like2

    # Multiplication of two terms
    return order1 + order2
```

As a result, here is a replication of the above plots where we do not see that with larger N bad and good channels are correctly classified with my own calculations of percentage of time correct.:

36

I also see my code performing much better for larger N such that $u_i = \hat{u}_i$ with very high probability.

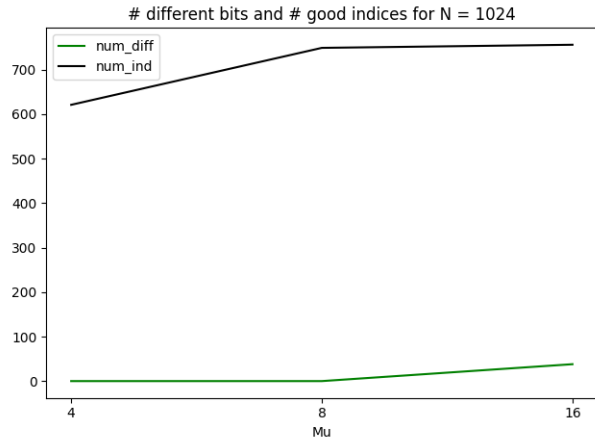
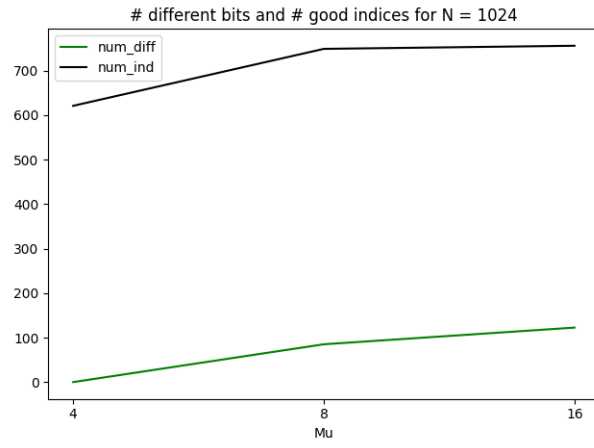
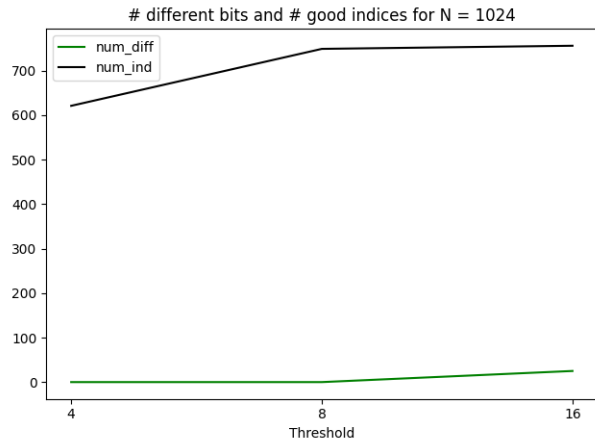
I conducted an experiment to see which threshold suits values of N from 512 to 4096. Below are the results, I plotted the number of bits that are on average different between u and $\hat{u}$ (in purple) as well as the number of indices that are considered good using that threshold (in orange). I kept $\mu = 4$ for the full test.

This is averaged data over 4 trials for the number of different bits. It appears that for the trials conducted, the working threshold increases with N, and it looks like a threshold smaller by a power of 2 is needed for increasing N. I am still not sure of the relationship between N and threshold..

I also conducted a test for the value of μ while keeping $N = 1024$ and threshold=10^{-7}. μ takes on values $4, 8$, and $2\log_2 N$ floored to the nearest power of 2. I ran this 3 times, taking the average of 4 trials each and below are the results:

37

different bits and # good indices for N = 1024

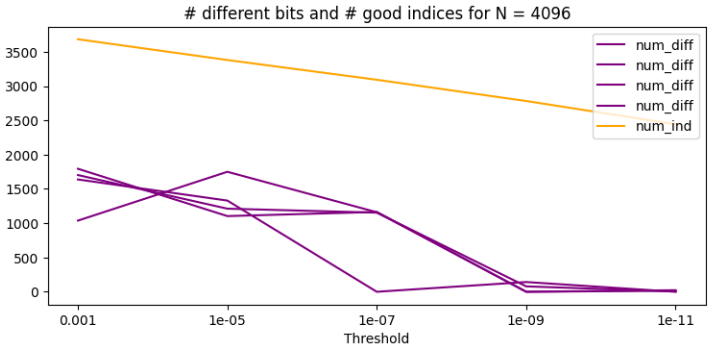
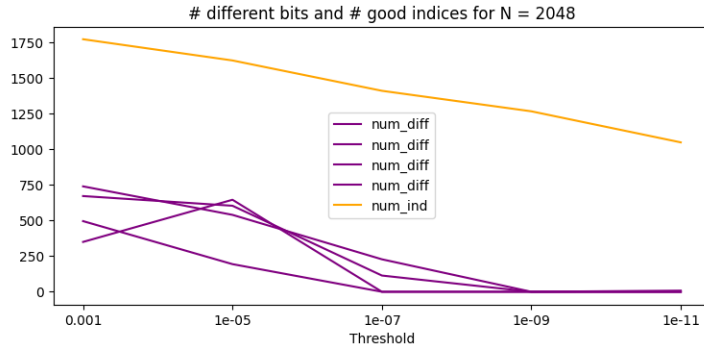
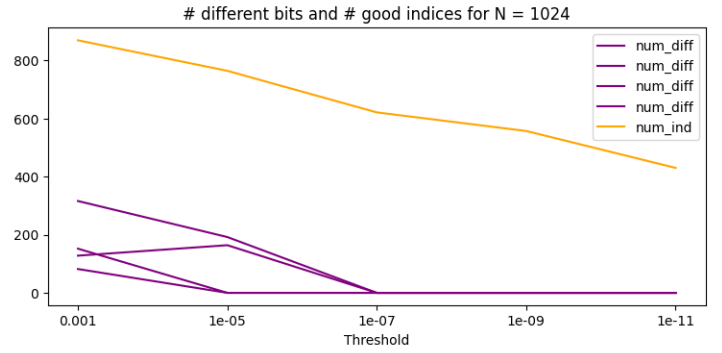
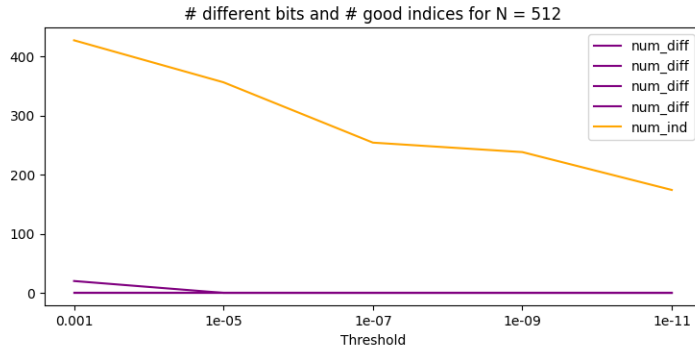
different bits and # good indices for N = 1024

different bits and # good indices for N = 1024

We see that as μ increases, the channel is less degraded and the number of indices that are denoted good channels also increases. This results in an increased number of channels that may be incorrectly classified using the threshold of 10^{-7}. If we want to use a higher threshold, it is required that we use a smaller threshold for the same results as with a smaller μ. Trials wilth a larger μ take longer than those with smaller μ.

38

11 Week 11: Janurary 3 - January 9

I found that I was rounding too much in my previous code in the log additions and order_mag() function. I removed the rounding step in order_mag(). I also altered parts of safe_compute_even() to treat a number smaller than 10^{-5} as negligible as compared to 1 and numbers as off by an order of 5 as negligible as compared to each other. I will test whether 5 is the correct value to use... This should increase precision in my calculations while not running into overflow or divide by 0 errors. Below is updated code and updated plots. I changed the plot so that instead of an average of 4 trials for each N, I plot each of the 4 trials to get a better idea of the output indices that are different between $\hat{u}$ and u.

```python
## Calculate order of magnitude of a number - only used in base case of recursion
def order_mag(value):
    order = math.log10(abs(value))
    return order

# Likelihood ratio for even indices used in the likelihood function
# Avoids overflow using a log representation
def safe_compute_even(like1, like2):
    # (like1*like2 + 1) / (like1 + like2)
    order1 = like1
    order2 = like2

    # Product of 2 likelihood values
    numerator = order1 + order2

    # Add 1 to the sum above
    if numerator < -5:
        numerator = 0 # equivalent to log(1) since other term is small
    elif abs(numerator) < 5:
        numerator = math.log10(1 + 10**numerator)
    # numerator > 10**5 => adding 1 is negigible

    if order1 < order2 - 5: # order1 much larger than order2
        denominator = order2
    elif order1 > order2 + 5: # order1 much smaller than order2
        denominator = order1
    else:
        # log(order1 + order2)
        # log(a + b) = log(a) + log(1 + b/a)
        denominator = order1 + math.log10(1+10**(order2-order1))

    # Division of numerator and denominator
    return numerator - denominator
```

Trying 6 as the order of magitude difference to determine if one number is negligible compared to another.

Trying 4.

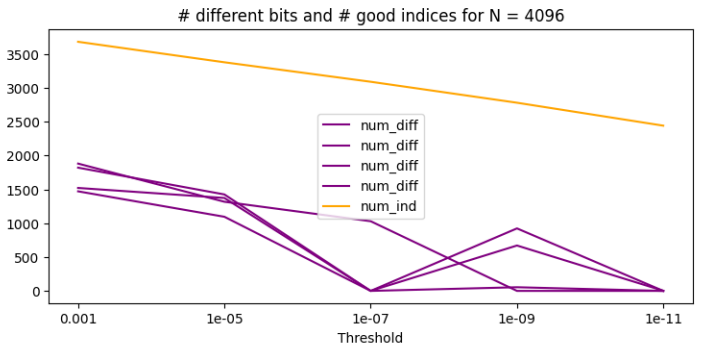
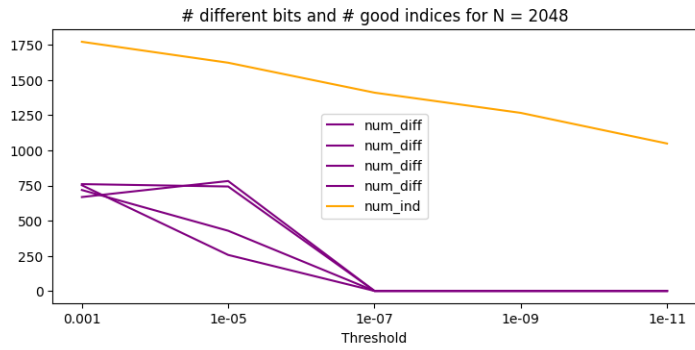
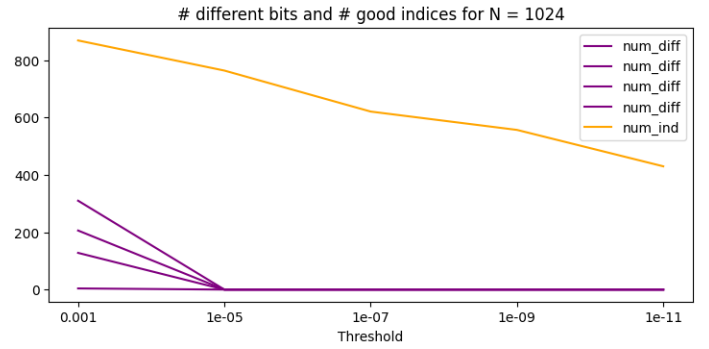
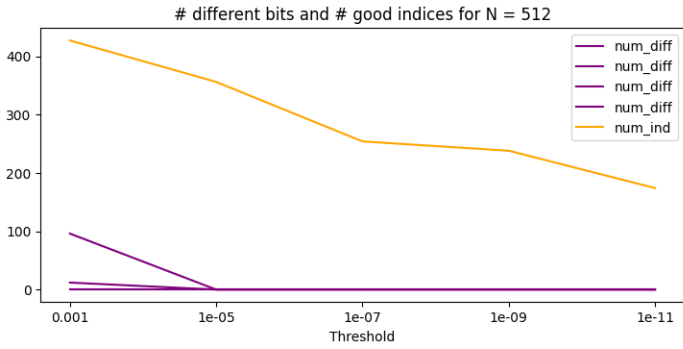
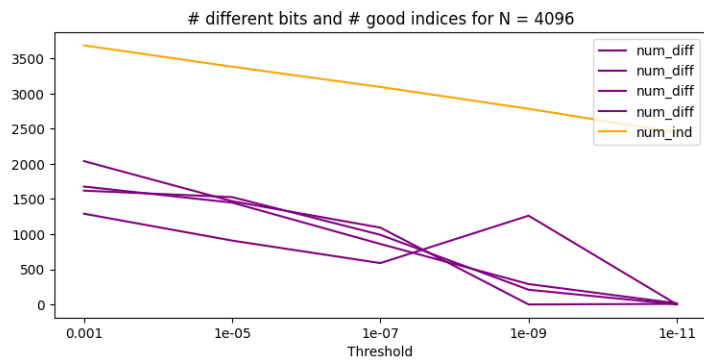
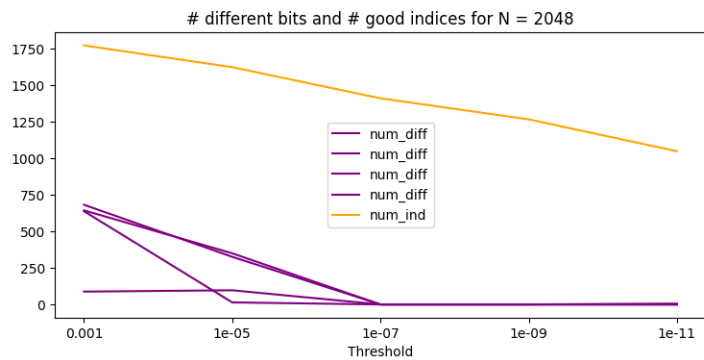
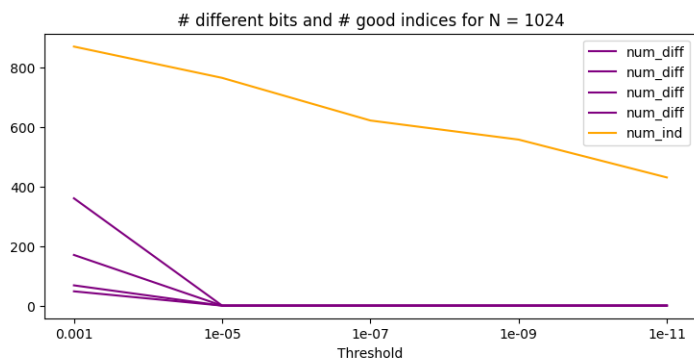
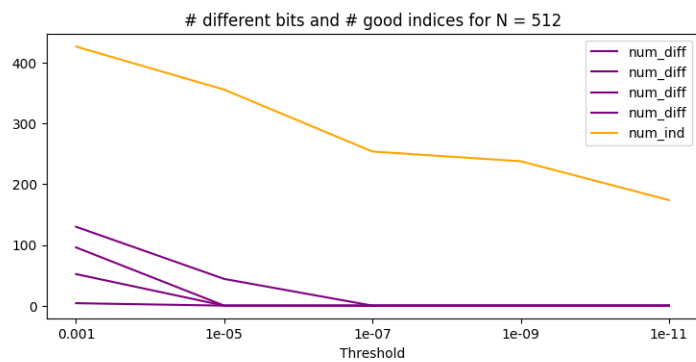
40

different bits and # good indices for N = 512

different bits and # good indices for N = 1024

different bits and # good indices for N = 2048

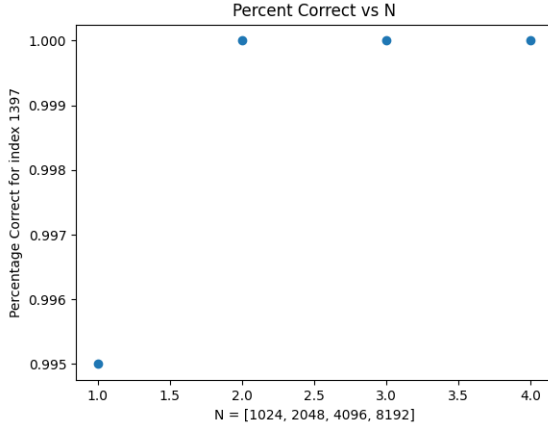
different bits and # good indices for N = 4096

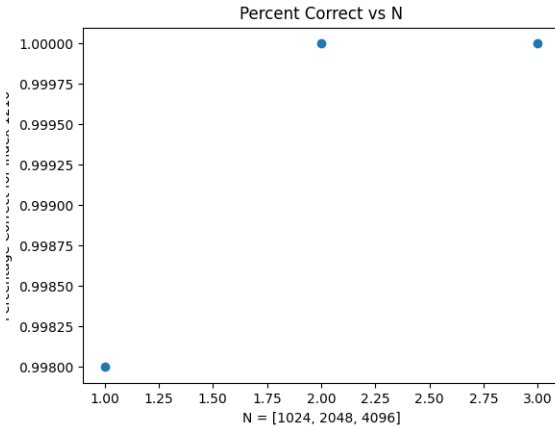
41

12   Week 12: January 10 - January 17

Making sense of the small nature of the error probability threshold value. As in the above experiments, the value of this threshold must be very small for large N in order to ensure suitable performance of the decoder, meaning $\hat{u} = u$ with high probability for each index i.

I altered one of the previous tests where I looked at each index running independently through the decoder. Here I set the threshold to 0.1, meaning that the probability of error is 0.1, and the number of trials to 200 so that we would expect the number of incorrect trials for a certain index under this threshold will be less than $200 \times 0.1 = 20$. For each value of N, I choose an index at random that has error probability under this threshold and run the many trials using the same index.

12.0.1   Trials = 200, Thresh = 0.1

Here, we would expect less than 20 trials to be incorrect for each N.

12.0.2   Trials = 1000, Thresh = 0.01

Here, we would expect less than 10 trials to be incorrect for each N.

Note: 1024 does not always have an error, these are just 2 random runs of the test...

Significance of the threshold for error probability:

$$\mathbb{P}(\text{At least one error in string}) \leq \sum_{i=1}^{n} \mathbb{P}(\text{Error at bit index } i) \leq N \times \text{thresh}$$

For $N = 4096$ and thresh = 0.0001, for example:

$$\mathbb{P}(\text{At least one error}) \leq 4096 \times 0.0001 = 0.4096$$

For $N = 4096$ and thresh = 10^{-7}:

$$\mathbb{P}(\text{At least one error}) \leq 4096 \cdot 10^{-7} = 0.00004096$$

42

With the above calculations, we must think of the threshold as being $\mathbb{R}$(At least one error), and we must select a threshold value such that the probability of at least one error is small 'enough'. With the two examples above, we would expect that:

1. With 10 trials, a maximum of around 4 trials will have at least one error.

2. With 1000 trials, less than 4 trials will have an error, meaning we should expect no errors in a few trials.

I have an experiment of the last item above, which is contrary to the results we are getting. Using the plots from last week, we see that a threshold of 10^{-7} has errors for all 4 of the trials. I also see that as N doubles, the threshold necessary decreases on the order of 10^2 instead of by 2, which is suggested by the union bound. When I de-couple the bit indices, ensuring that the previous bits are always correct when using the decoder, I see results that make more sense, where the probability of a good channel containing an error is small as expected.

There must be an issue with the decoder since the union bound should still provide an upper bound to the probability of error for the entire string.

Elements that could potentially invalidate the union bound used in this way:

- Error prob of a channel $\Rightarrow$ decoder output. Error in decoder calculations is not the same as error in the channels themselves. Using the recursive algorithm may introduce complication to the union bound calculation.

12.1   Using Degraded Channel as Decoder

To try to find the issue with threshold not matching intuition above, I will try to use the degrading algorithm as a decoder. In the degrading procedure, I return a large matrix of the degraded channels for each index. I then use this channel to locate the degraded output section given the string of y bits and preceding $\hat{u}$ bits and take the quotient of the probability given an input $u_i = 0$ and an input of 1. This is still a work in progress as I believe I will need to alter the degrading procedure to include a variable to explain where each normal output is within the degraded channel.

43

13 Week 13: January 18 - January 24

13.1 Threshold Test

Instead of the test done in the prior week, I want to compare the error probability and how good each channel is. There is no need for a global error probability, but instead I want to calculate the error probability for a certain subchannel, and then simulate the channel many times, equal to 1/error prob. I used a threshold of 10^{-3} for all trials when deciding whether a certain channel is good or poor.

I limited the number of trials to 500 since I wanted the code to run fast. The number of incorrect trials over the total number of trials should a good approximation for the error probability.

N=256 Trials		
Number of Trials	Error Prob	Approx
172	0.005801	0.005814
343	0.0029	0
208	0.0047995	0

N=512 Trials		
Number of Trials	Error Prob	Approx
288	0.00346	0
104	0.00955	0
263	0.003797	0
104	0.00955	0.00962

N=1024 Trials		
Number of Trials	Error Prob	Approx
144	0.006906	0.0069444
104	0.00954	0
278	0.003584	0.003597

N=2048 Trials		
Number of Trials	Error Prob	Approx
451	0.0022157	0.002217
112	0.00888	0

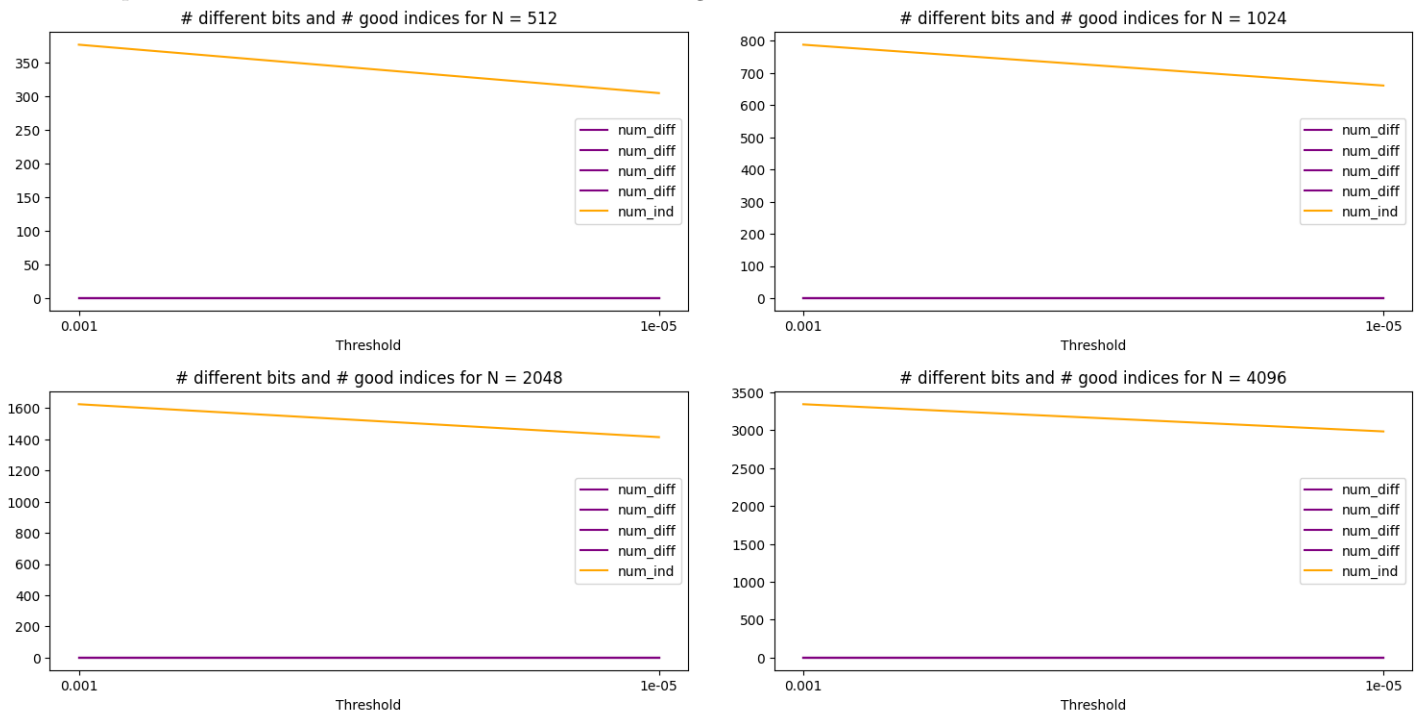
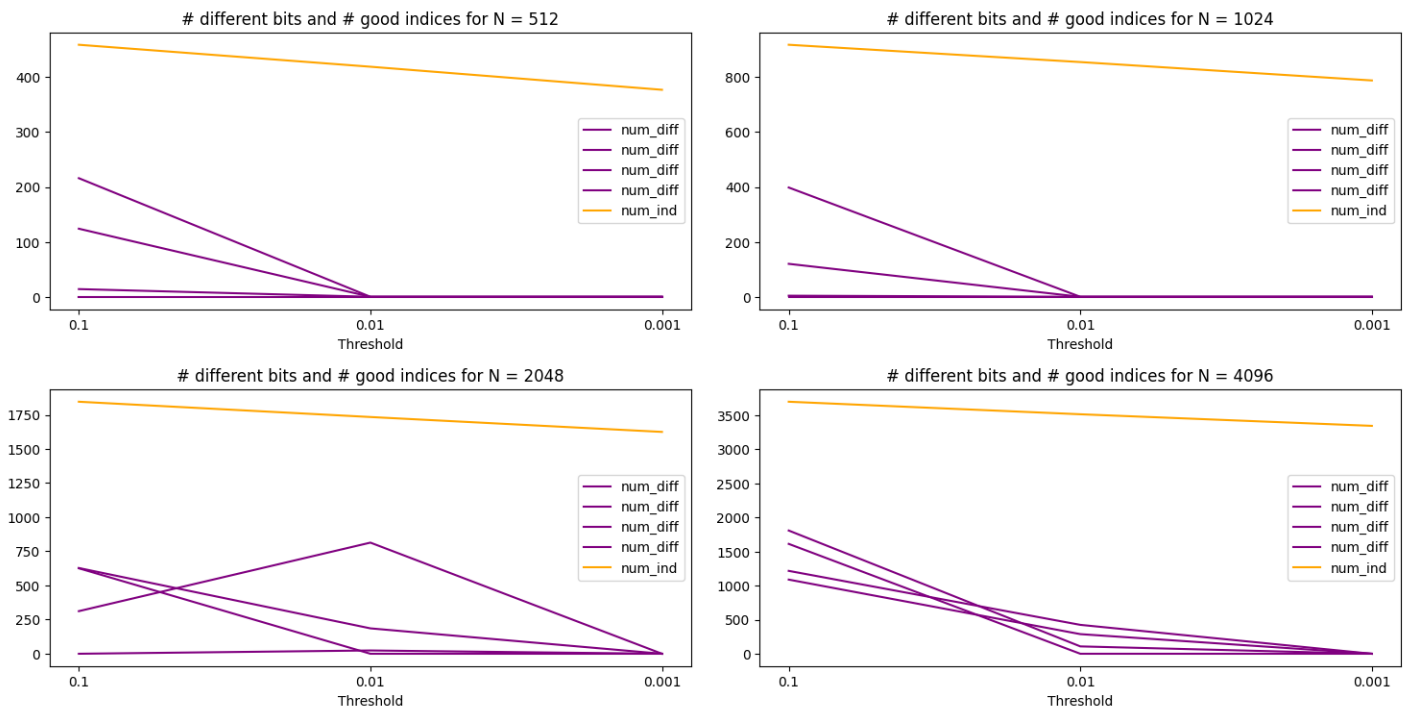
N=4096 Trials		
Number of Trials	Error Prob	Approx
407	0.00246	0
110	0.009058	0

When an index is independently decoded while all preceding $\hat{u}$'s are correct, the error probability calculated from the degraded is a good approximation for the experimental probability of error. Is there an issue in the decoder?

I might've fixed the decoder... This is coming from a changed method in the degraded execution code - how to get an array of the bits of an index. For example index 5 out of 128 bits would be 0000101. After simplifying this, the decoder has behavior closer to what we would expect using the union bound calculation.The previous code also had an error in calculation of this matrix, but the new code is correct. Here is the code I substituted in:

```
for i in range(N):
    # Method 1
    bin_i = bin(i)[2:]
    strbi = str(bin_i)
    b1 = np.zeros(m)
    for ind in range(len(strbi)-1,-1,-1):
        if strbi[ind] is not None:
            b1[(m-1)-ind] = strbi[ind]
    b1 = np.flip(b1)
    b1 = np.array(b1)
    b1 = b1.astype(int)
    b1 = b1.tolist()

    # Method 2
    strbi = str(bin(i)[2:]).zfill(m)
    b2 = list(map(int,list(strbi)))
```

44

Here is a repeated version of the decoder from a few weeks ago:

I tried the same experiment with smaller threshold values:

This looks more like we would expect, with the threshold values from one N to the next power of 2 being off by an order of 2 vs 100 as previously. Because of this, we no longer need to use the degraded procedure as a decoder as that would have been for debugging this issue.

Questions for this week's meeting:

- How to decide how many message bits there will be? Can this just be any number of bits less than the number of good channels?

- Error probability threshold prediction for large N. Suitable thresholds for a variety of N values?

  - $N = 512$: Error prob thresh = 0.01
  - $N = 1024$: Error prob thresh = 0.01

45

- $N = 2048$: Error prob thresh $= 0.001$
- $N = 4096$: Error prob thresh $= 0.001$
- Could use approximately $log_2(N)/4$? This works for the above pattern...

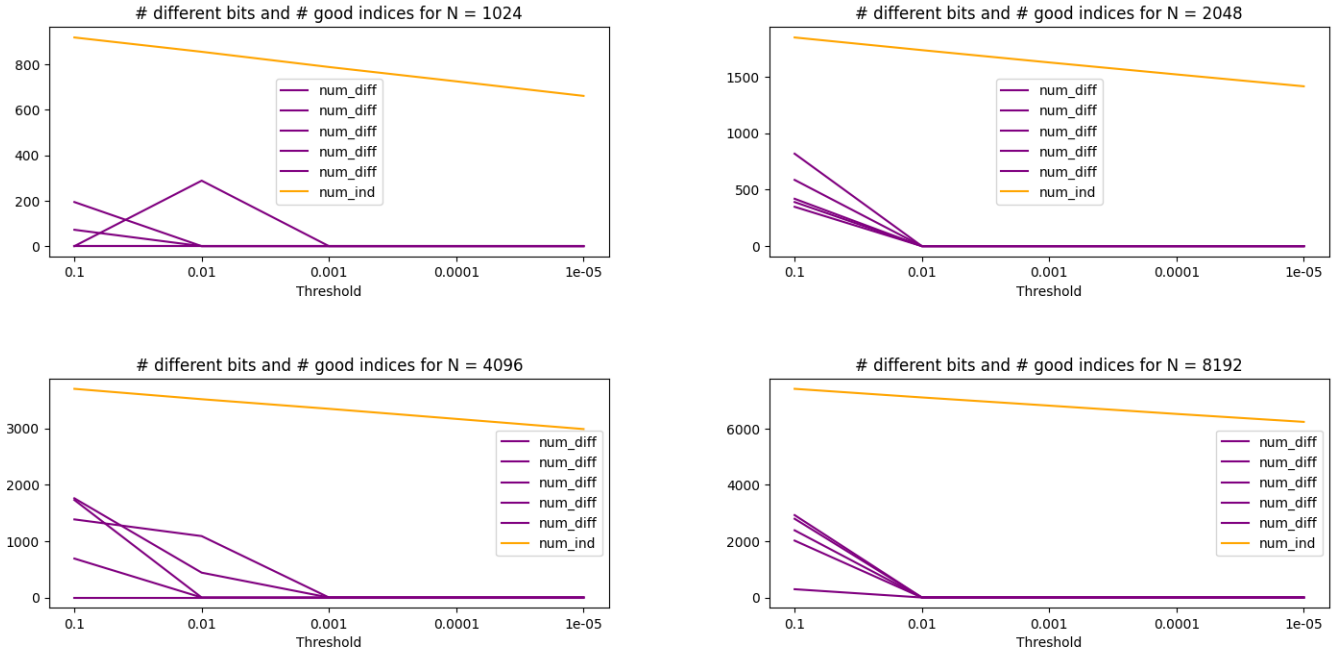
- Simulation: Chosen epsilon for the internal BSC is correctly approximated by the simulator. $y \mapsto \hat{x}$ closely resembles the chosen epsilon value. What are the next steps?

46

14   Week 14: January 25 - February 2

14.1   Speed up Decoder

Sharang implemented an $O(N \log N)$ decoder without altering the decoder implementation. The only change is the inclusion of *@cache_tools.memoize* before the declaration of the likelihood function. This refers to a python code that includes a memoized implementation of the likelihood calculation, using Python's features in the cache to speed up the recursive likelihood calculations. The correctness of the code is not compromised as no code was actually altered. According to Sharang's tests, the speedup is negligible for small N, up to $N = 512$ even, but for large N, it is considerable. For example, for $N = 8192$, the memoized version was nearly $4\times$ faster.

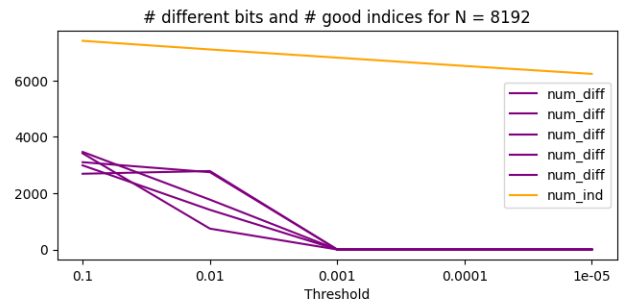
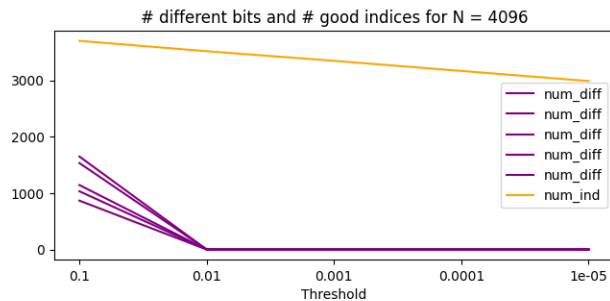
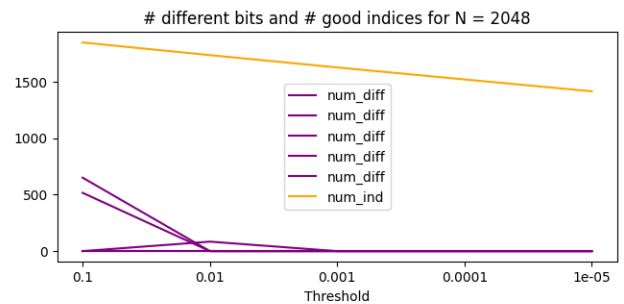
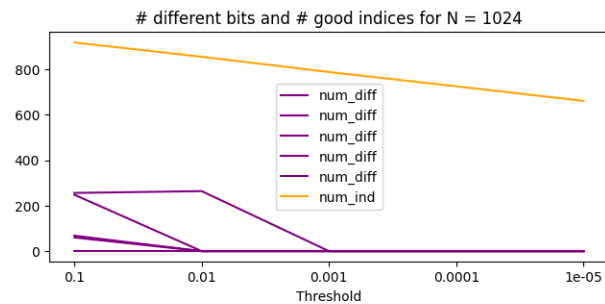
I conducted the threshold experiment as above with more threshold variations, 10 trials each, and for larger $N = 1024, 2048, 4096, 8192$. Here are the results:

14.2   Log Optimization

Sharang also implemented an optimization for the log calculation of $\log(1 + 10^{\mathrm{order1} + \mathrm{order2}})$ below:

```
def log_one_plus_x( log_x ):
    if log_x < −745: ans = 0
    elif log_x < −37: ans = np.exp(log_x)
    elif log_x > 37: ans = log_x + np.exp(−1*log_x)
    else: ans = np.log( 1 + np.exp( log_x ) )
    return ans

def safe_compute_even( log_like_1 , log_like_2 ):
    # (like1*like2 + 1) / (like1 + like2)
    #Compute Numerator
    log_like = log_like_1 + log_like_2
    numerator = log_one_plus_x( log_like )

    #Compute Denominator
    max_term = max( log_like_1 , log_like_2 )
    min_term = min( log_like_1 , log_like_2 )
    denominator = max_term + log_one_plus_x( min_term − max_term )

    return numerator − denominator
```

I redid the threshold experiment as above with this optimization as it should allow for increased correctness, and therefore increased performance.

47

different bits and # good indices for N = 1024
different bits and # good indices for N = 2048
different bits and # good indices for N = 4096
different bits and # good indices for N = 8192

There does not seem to be a large difference in the results, but I will use the log optimization code from now on.

14.3 $Y \mapsto X$ Simulator - Binomial Distribution

The simulator takes an input y and produced $\hat{x}$ and the below plots run the simulator for $N = 2048$ for 10 trials and 100 trials, respectively. The BSC transition probability $\epsilon = 0.4$, so the average number of incorrect bits will be 81.92. This is the center of the plots shown below, and the plots follow a binomial distribution.

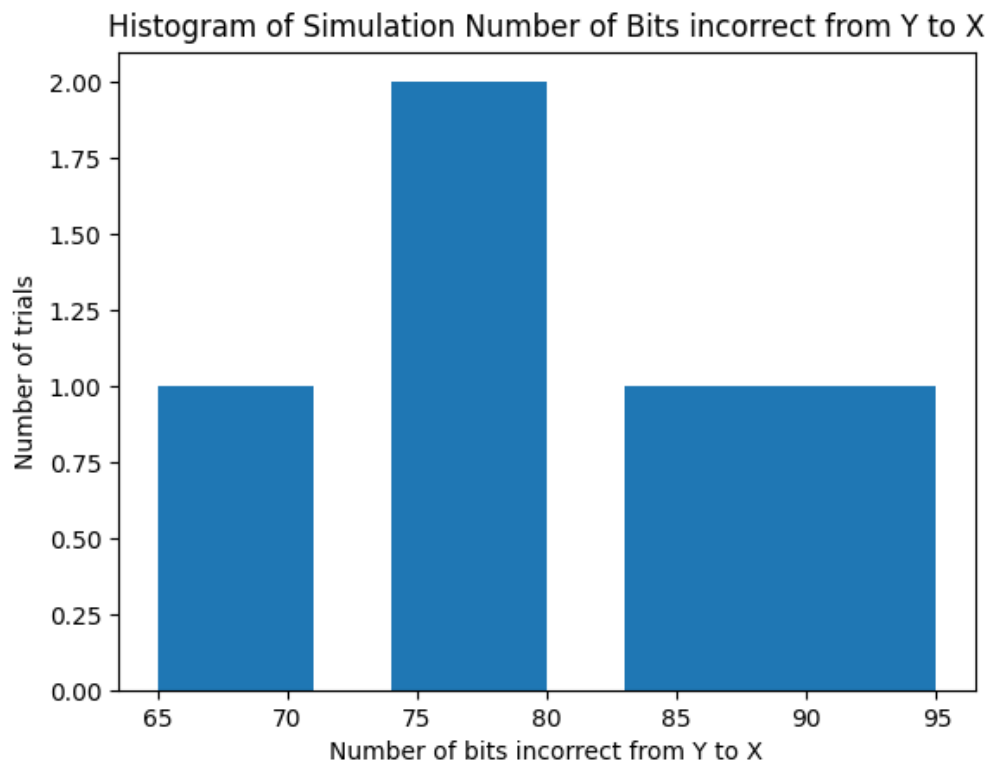
Trials = 10, # of bins = 3:

Histogram of Simulation Number of Bits incorrect from Y to X

48

Trials = 100, # of bins = 20:

Trials = 100, # of bins = 100:

49

14.4 Ranking of subchannel - Best to Worst

14.4.1 $Y \mapsto X$ Simulator

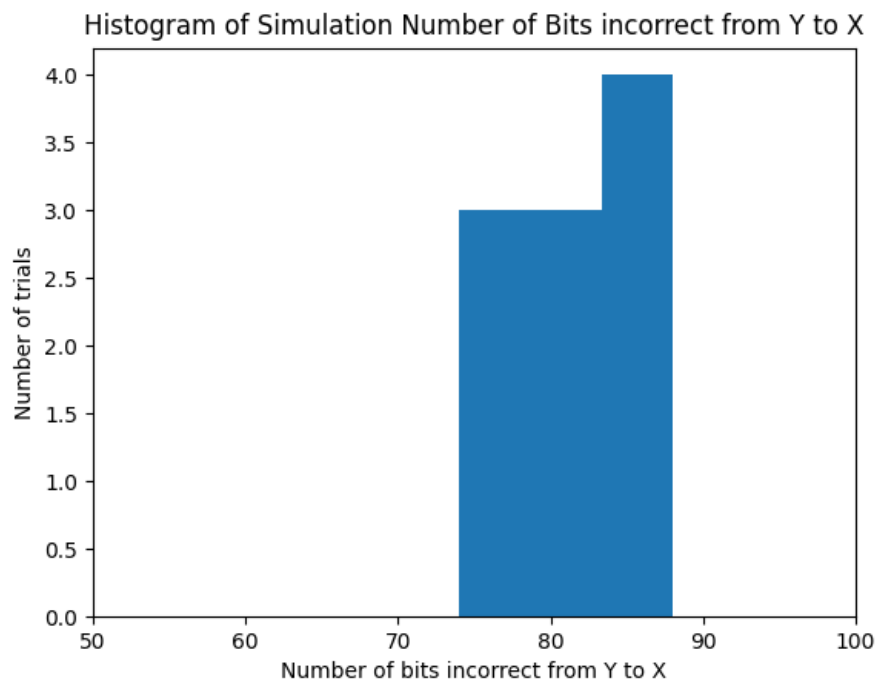
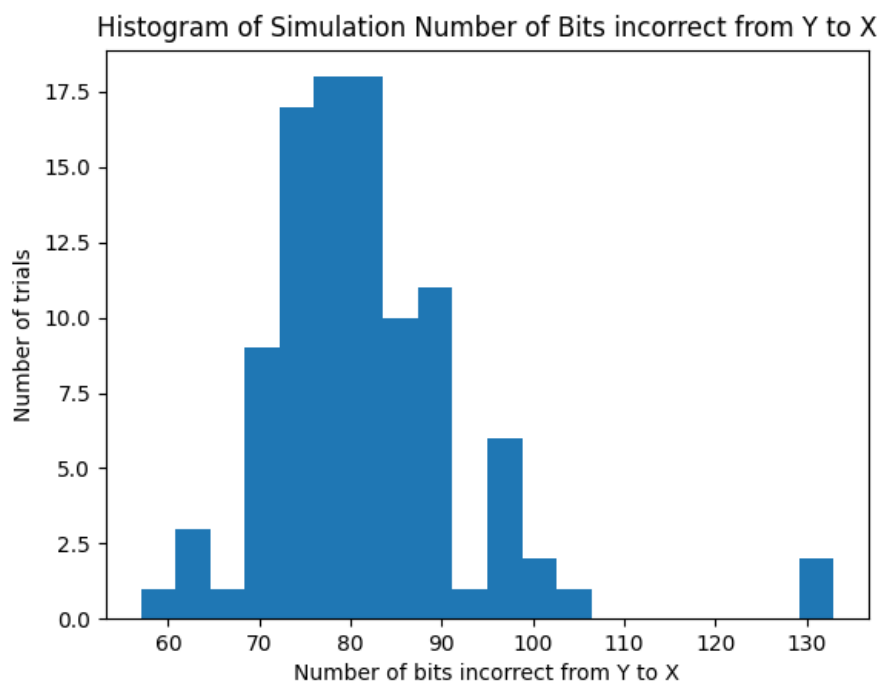
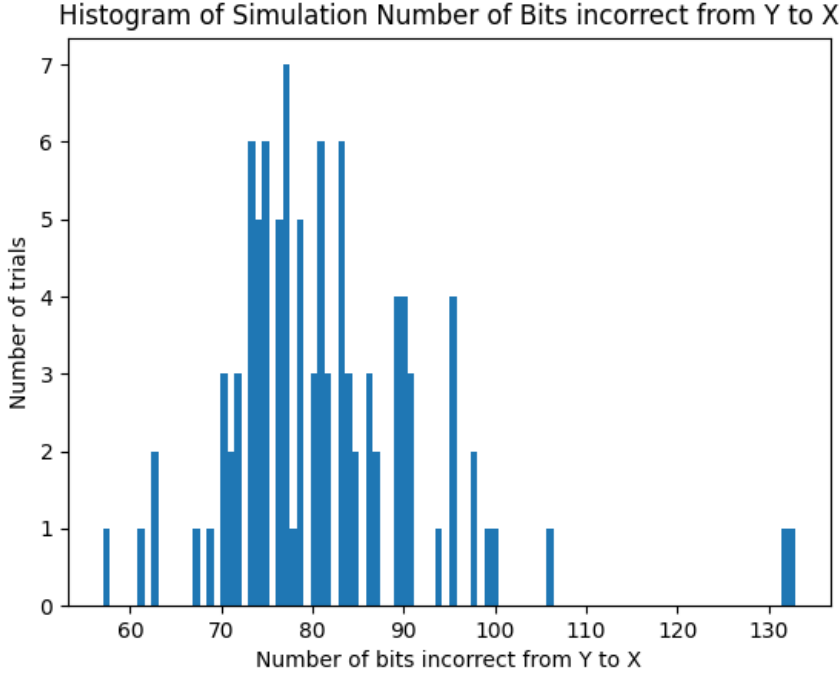
The above results are with calculation of the good channels by going from index $0 \to 2048$ and choosing the first p channels. Instead, I will now rank the subchannels according to their error probability calculated from the degraded channel, where the lower error probability is the best channel. This will assist in accuracy in calculations. The below plots will redo the above histogram experiments:

Trials = 10, # of bins = 3:

Trials = 100, # of bins = 20:

50

Trials = 100, # of bins = 100:

Histogram of Simulation Number of Bits incorrect from Y to X

14.4.2 Rank of subchannels changes with ϵ?

I am testing if the ranking of the subchannels changes with ϵ, the cross-over probability of the BSC. For $N = 2048$, I chose $\epsilon = 0.001, 0.01, 0.05, 0.1$. I see that none of the arrays are exactly the same. The value of ϵ changes the number of channels that are considered good, so as ϵ increases, the number of good channels decreases. However, the ordering is preserved up until there are no more good channels.

For example, for $N = 16$, these are the sorted good subchannels corresponding to $\epsilon = 0.001, 0.01, 0.05, 0.1$:

```
[15, 14, 13, 11, 7, 12, 10, 6, 9, 5, 3, 8, 1, 4, 2]
[15, 14, 13, 11, 7, 12, 10, 6, 9, 5, 3]
[15, 14, 13, 11, 7]
[15]
```

We see increased performance when using the best subchannels first, as opposed to varying which of the 'good' channels are used to send message bits. The histograms have a decreased spread, so the number of bits that are incorrect are more consistent across trials, and are more aligned with the expected percentage ϵ.

14.5 Try GPU for better performance?

I do not see increased performance with use of a GPU as is, I can try wrapping it in a special way to utilize speedup, but I don't think this is necessary at the moment...

14.6 More trials for Decoupled Error Prob Experiment

I want to run many more trials for the decoupled error probability experiment that I ran in the previous week. This will give more context for the error probability value that I am calculating from the degraded channel to the experimental value. I ran this on the server so I could run the code concurrently many times and have it run in the background. I used FileZilla to transfer the files over. The only issue on the server is that for the optimized likelihood calculations, a specific package is used for the cache_tools file, but I need an upgraded Python version to run it. I do not have sudo access, so I have no way of running the optimized code. These results are the same as I would expect the optimized code to be, except take longer...

I ran 10,000 trials for each, and I made sure that the error probability calculated was less than 0.0001 so that the experiment would yield helpful results. Here are the results from the experiments:

51

N=4096 Trials	
Error Prob	Approx
0.001012	0
0.000144	0
0.006902	0.0015
0.001500	0.0003

The results above show that the approximated error probability is always less than the expected error probability, so the expected error probability is an upper bound for the experimental results.

14.7 $\epsilon \to \#$ bits to flip

Instead of quantifying randomness as the BSC flipping probability ϵ, I instead want to flip a specific number of bits randomly. This is the new BSC code:

```
# BSC to flip num_flip # of bits x to form elements y
def bsc_numflip(x, num_flip):
    N = len(x)

    # Form arrays
    x_N = np.array([int(bit) for bit in x], dtype=int)
    y_N = np.zeros(len(x_N))

    # Randomly choose which num_flip # of bits will be flipped
    flip_inds = np.random.choice(N, num_flip, replace=False)

    # Flip the bits found above
    x_N[flip_inds] = x_N[flip_inds] ^ 1 # XOR with 1 will flip the bit

    y_N = x_N

    # Convert matrix to string
    y = ''.join(map(str, y_N.astype(int)))
    return y
```

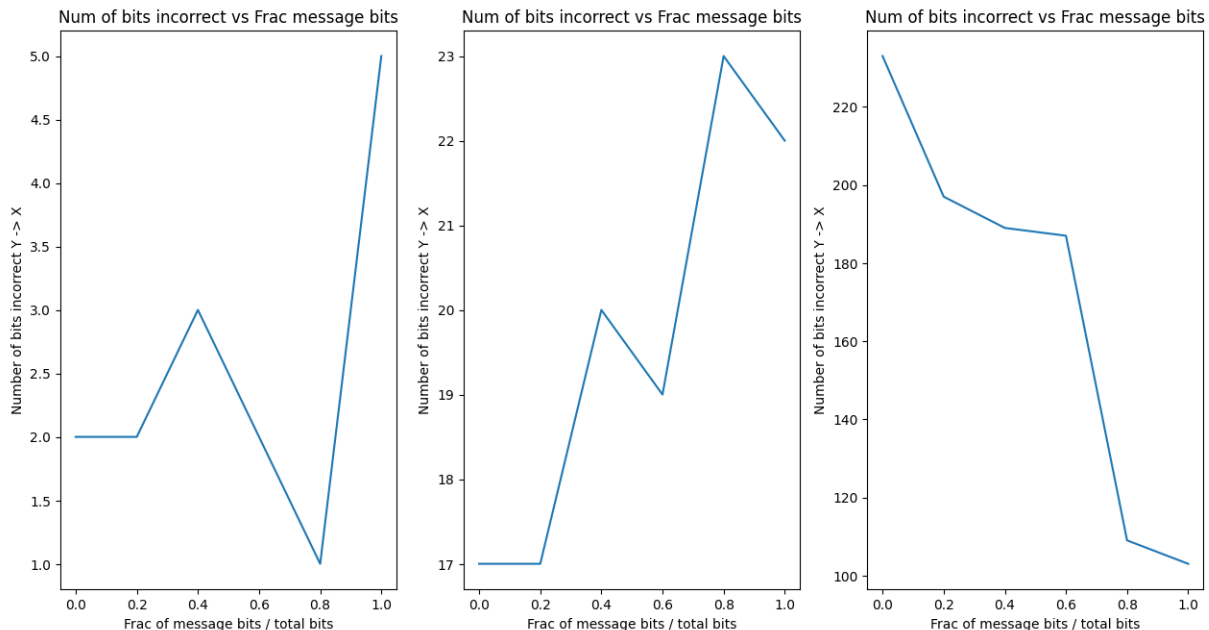
What value of epsilon ϵ should be used in the the degrading merge procedure AND likelihood calculations (initial channel involves ϵ)? Should it be the number of bits to be flipped / N? What are the next steps with this formulation?

Using this new definition of randomness, the estimated ϵ computed from number of bits incorrect $Y \to X$ in the simulator appears to be exactly ϵ, and if not is a much better approximation for ϵ.

52

15 Week 15: February 3 - February 9

15.1 Simulator dependence on ϵ

The simulator results of number of bits incorrect from $Y \rightarrow X$ should not depend on ϵ, but this number seems to be, on average, the value of $\epsilon \times N$. I ran the experiment below to prove this, where I sweep the ratio of message bits to N from 0 to 1, and I also use 3 values of ϵ and saw that the average number incorrect indeed follows $\epsilon \times N$, averaging 2 in the first plot, 20 in the second, and 200 in the third. Here are the data plots:

The plots above seem to suggest that the number of bits chosen as message bits do not make a difference in the number of bits that are correct on the two ends of the simulator. I should not be expecting an ϵ dependence, so I may be formulating the simulator incorrectly. There are several usages of ϵ within the simulator code:

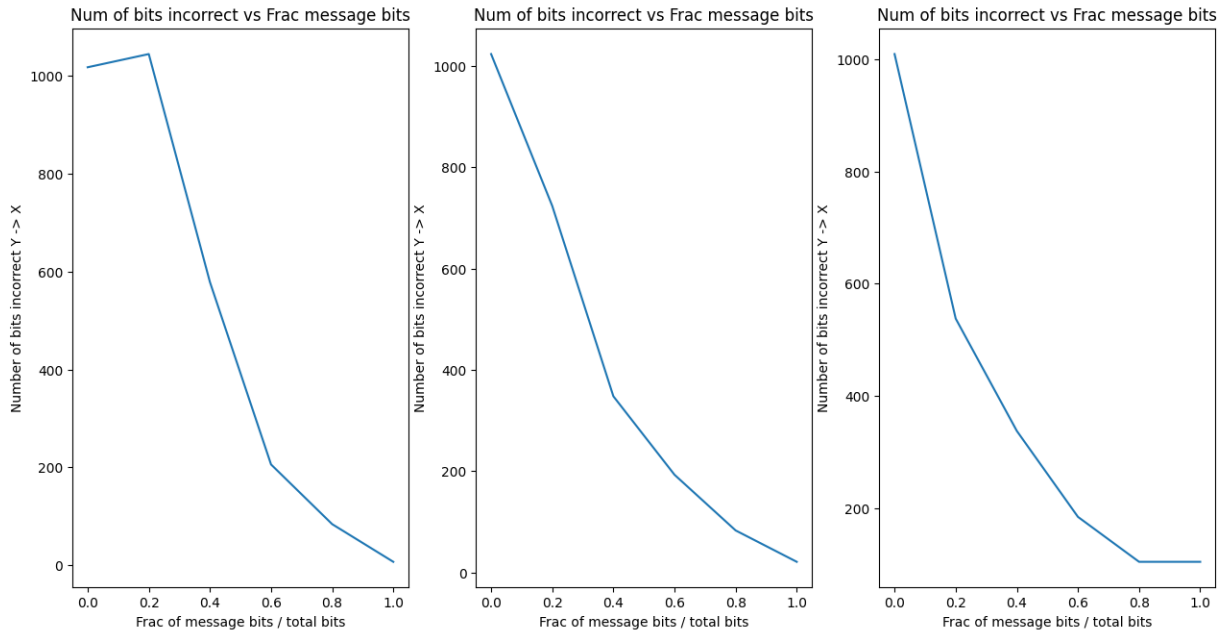
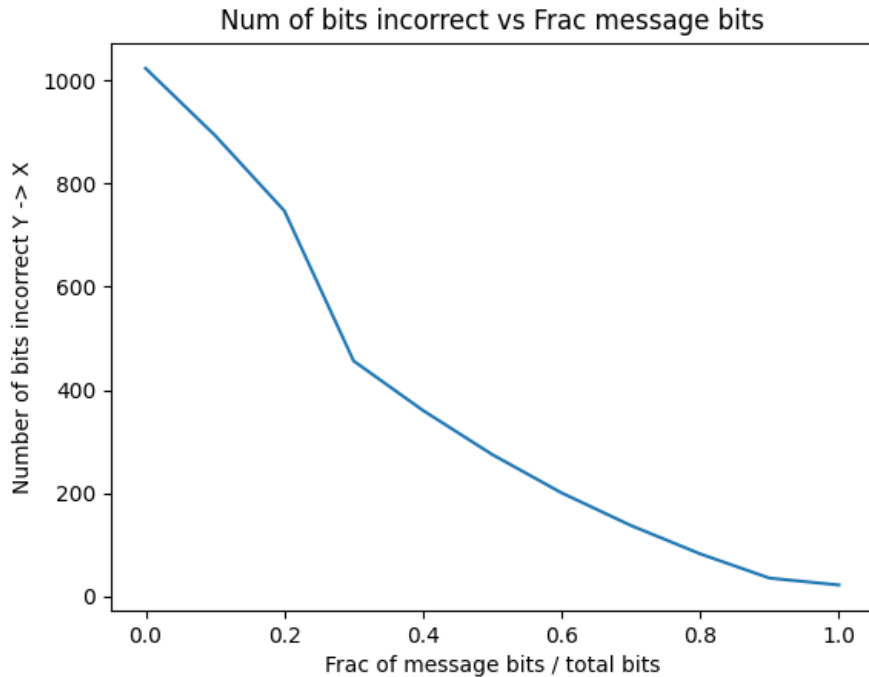
1. Generating an initial u sequence at the beginning of the simulator - this may be the issue, and also may not be needed

2. ML likelihood calculation - this shouldn't depend on the value of ϵ, only if epsilon is < 0.5, but not on the specific value.

3. Degrading procedure - as I showed last week, the good subchannel indices do not depend on the value of epsilon (other than the number of them), so this should not impact the code...

As explained in the first bullet, it is not necessary to generate a starting u, then encode to produce an x, and then send through a BSC to get y. This introduces another unnecessary/incorrect dependence on ϵ. Instead, I randomly generate a y bit string, and compute $\hat{u}$ as previously, except frozen indices which I am setting to 0 as these values will not matter.

I am no longer seeing an ϵ dependence in the plots above, here are the updated results where the fraction of incorrect bits is a function of k, which is the fraction of good channels to N (total channels). This is the median of 3 trials each:

53

These are the same results with 10 trials to average over and a wider range of k values, choosing just one value of epsilon:

15.2   Running simulator on large N

When I run the simulator for $N = 2^{24} = 16777216$, I get the following error:

```
File "/home/rsb359/MEngProject/frozenind.py", line 282, in C
    term1 = −(a+b)*math.log2((a+b)/2)
ValueError: math domain error
```

I am certain that the values of a and b are positive (greater than 0 and non-negative), so this may mean that the value of $a + b$ is very small and close to 0. I updated by code to take this into consideration:

54

```
# Helper function for calcDeltaI()
def C(a,b):
    if a + b < 1e-10: # checking for small value close to 0
        term1 = 0
    else:
        term1 = -(a+b)*math.log2((a+b)/2)
    if a < 1e-10:
        term2 = 0
    else:
        term2 = a*math.log2(a)
    if b < 1e-10:
        term3 = 0
    else:
        term3 = b*math.log2(b)
    return term1 + term2 + term3
```

This seemed to fix the issue...

55

16 Week 16: February 10 - February 16

16.1 Optimizing Degrading Procedure

I am working on optimizing the square and circle functions as there are many loops within the functions. The goal is to vectorize this code to take less time, making the algorithm run faster for large values of N.

Here are the updated algorithms, which no longer include for loops, improving performance:

```
# Arikan channel transformation 1
def square(W):
    # Original channel W is X-by-Y
    num_rows, num_cols = W.shape
    # New channel new_W is X-by-Y^2
    new_W = np.zeros((num_rows, num_cols**2))

    u = np.arange(num_rows)
    y = np.arange(num_cols)

    XORs = u[:, np.newaxis] ^ u[np.newaxis,:]

    firstterm = W[XORs[:,:,np.newaxis],y[np.newaxis,np.newaxis,:]]
    secondterm = W[u[:,np.newaxis],y[np.newaxis,:]]

    # Compute W(y1|u1^u2)*W(y2|u2), then sum over u2
    val = np.sum(firstterm[:,:,:,np.newaxis] * secondterm[:,np.newaxis,:], axis=1)
    val_reshaped = np.reshape(val,(num_rows, num_cols * num_cols))

    # Convert each y1,y2 to str and concatenate to form the output index
    stry = np.vectorize(np.binary_repr)(y, int(math.log2(num_cols)))
    ind = np.array([int(s1 + s2, 2) for s1 in stry for s2 in stry])

    # Assign values to new_W
    new_W[u[:, np.newaxis], ind] = val_reshaped[u, :] / 2

    return new_W

# Arikan channel transformation 2
def circle(W):
    # Original channel W is X-by-Y
    num_rows, num_cols = W.shape
    # New channel new_W is X-by-XY^2
    new_W = np.zeros((num_rows, (num_cols**2)*num_rows))

    u = np.arange(num_rows)
    y = np.arange(num_cols)

    XORs = u[:, np.newaxis] ^ u[np.newaxis,:]

    firstterm = W[XORs[:,:,np.newaxis],y[np.newaxis,np.newaxis,:]]
    secondterm = W[u[:,np.newaxis],y[np.newaxis,:]]

    # Compute W(y1|u1^u2)*W(y2|u2)
    val = firstterm[:,:,:,np.newaxis] * secondterm[:,np.newaxis,:]
    val_flip = np.transpose(val, (1, 0, 2, 3))
    val_reshaped = np.reshape(val_flip,(num_rows, num_rows * num_cols * num_cols), order='F')

    # Convert each u1,y1,y2 to str and concatenate to form the output index
    stry = np.vectorize(np.binary_repr)(y, int(math.log2(num_cols)))
    stru = np.vectorize(np.binary_repr)(u, int(math.log2(num_rows)))
    ind = np.array([int(s1 + s2 + u1, 2) for s1 in stry for s2 in stry for u1 in stru])
```

56

```
    # Assign values to new_W
    new_W[u[:, np.newaxis], ind] = val_reshaped[u, :] / 2

    return new_W
```

I also optimized the heap sorting algorithm. I only sort the entire heap when all items are first added to the heap. After this point, there are different scenarios that can occur. Either the first element of the heap is removed and we have to figure out where this first item belongs OR the ΔI value of an element changes so this element should either move up or down within the heap depending if the value increased or decreased.

After implementing these changes, I did not see much of a time difference between the optimized code and the original code, so the slowdown for large N must be within the decoder algorithm.

16.2   Setting Frozen Bits Randomly

Instead of setting frozen bits to 0 by default, as I was doing previously, I instead want to set these randomly to 0 or 1. The values of the frozen bits are constant throughout the simulator code.

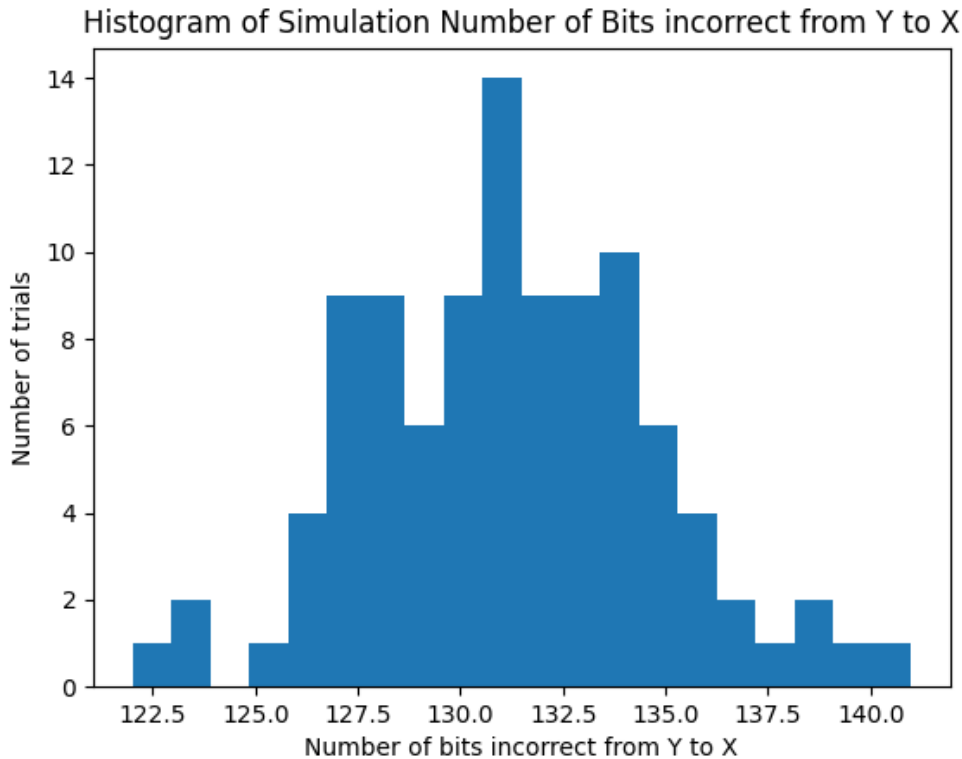
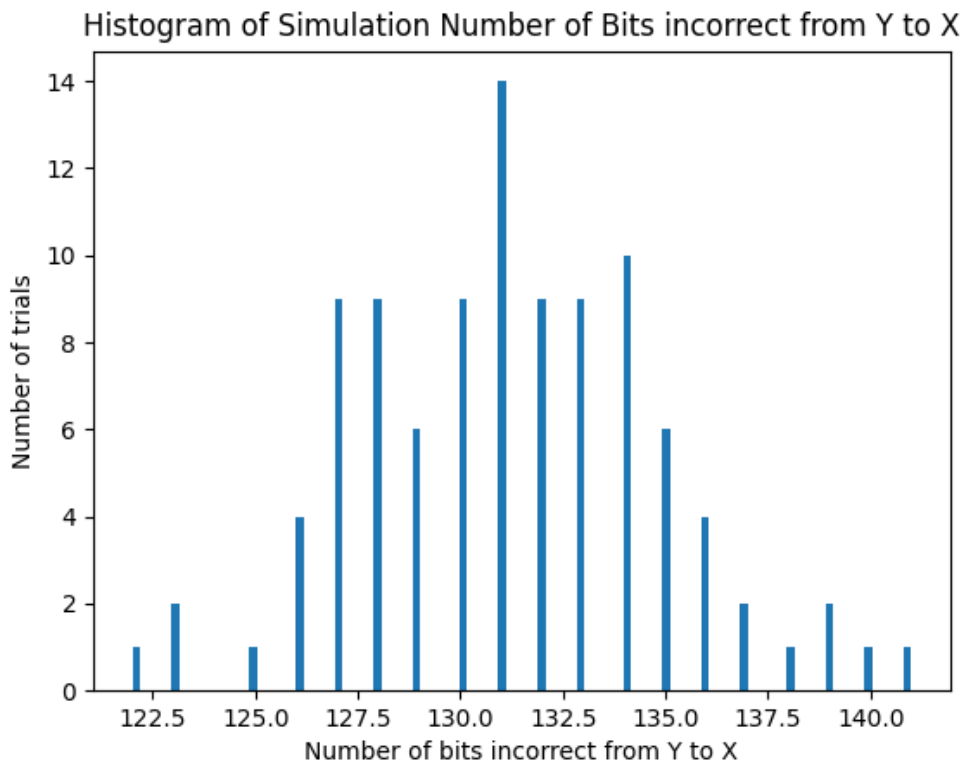
16.3   Redo Simulation Plots

With the changes to the simulator last week, I am re-running the histogram plots to show how many bits are incorrect in each run.

The simulator takes an input y and produced $\hat{x}$ and the below plots run the simulator for $N = 2048$ for 10 trials and 100 trials, respectively. The fraction of message bits I am using is 0.7 (good channels / total channels). The plots will follow a binomial distribution.

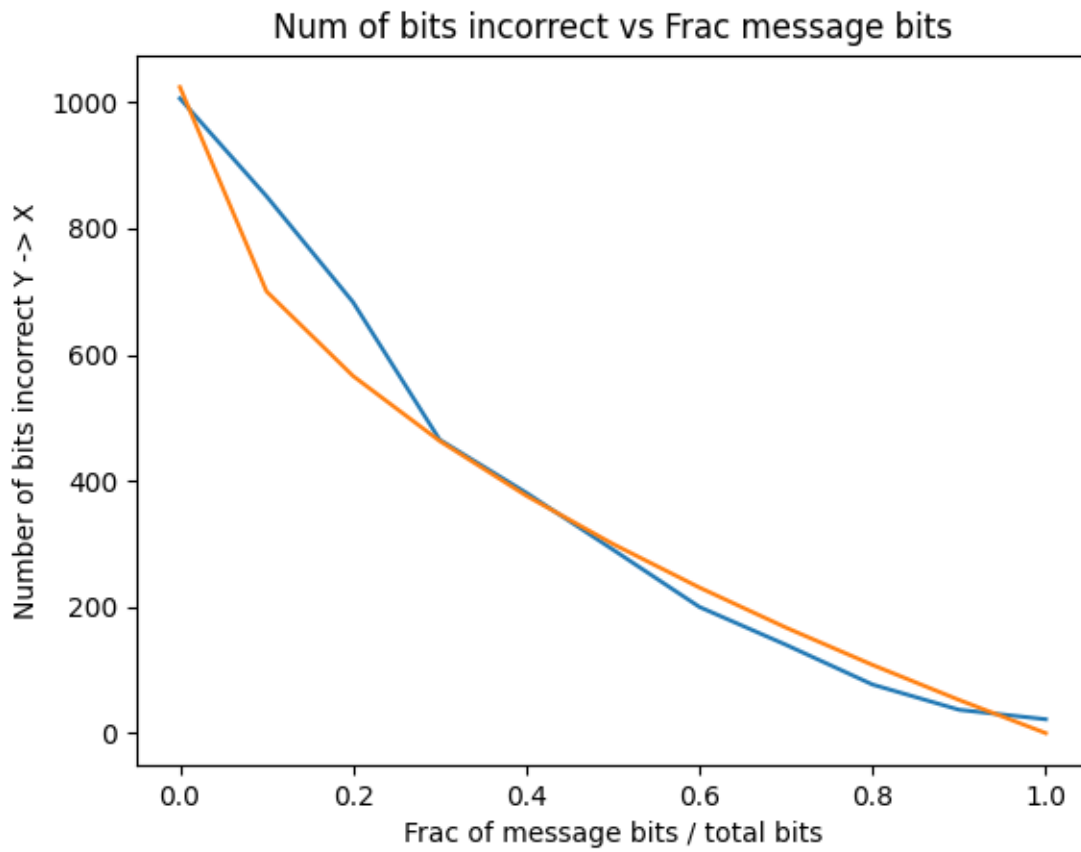
Trials = 100, # of bins = 20:

Trials = 100, # of bins = 100:

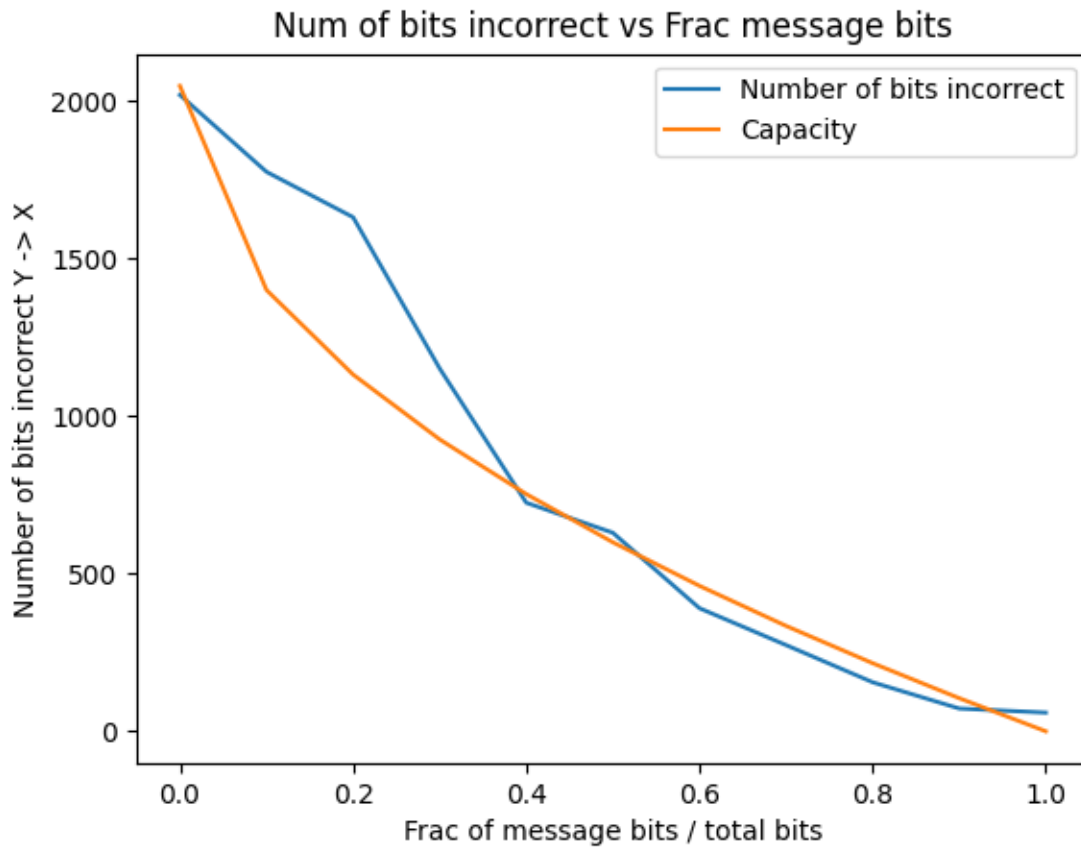
Histogram of Simulation Number of Bits incorrect from Y to X

16.4 Rate vs # bits incorrect

The plot of number bits incorrect vs fraction of bits that are message bits should follow the equation for capacity of the channel $C = H_b^{-1}(1 - R)$. I plotted these on the same graph, and they look very similar.

58

Num of bits incorrect vs Frac message bits

I redid this plot using $N = 4096$ and averaging 3 trials for each number of bits incorrect test.

Num of bits incorrect vs Frac message bits

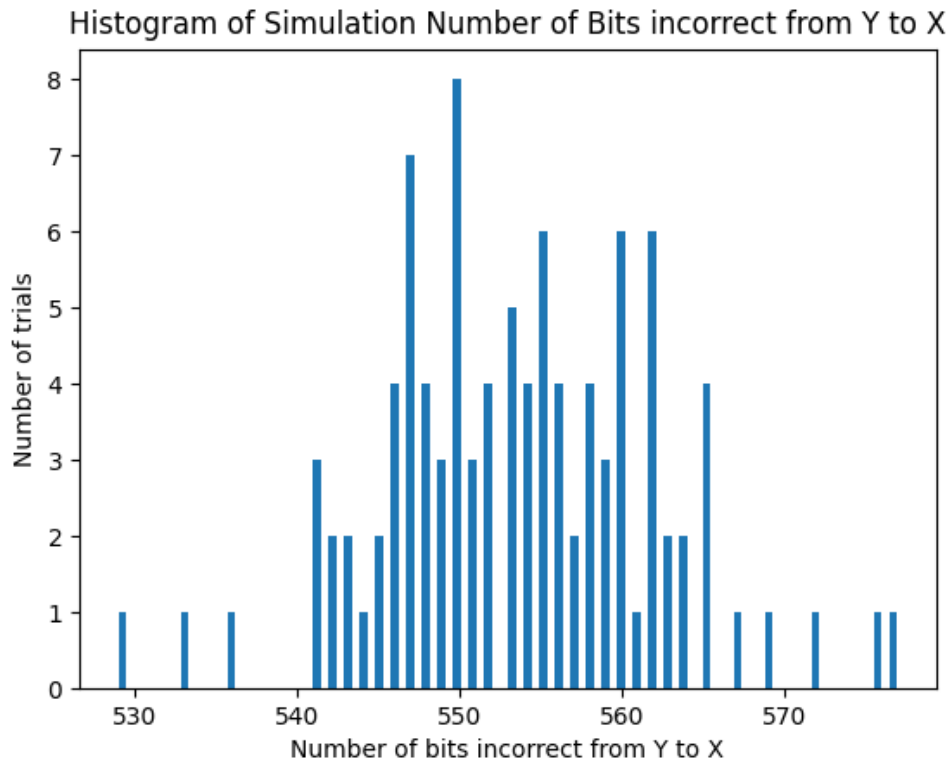
59

17 Week 17: February 17 - February 22

17.1 Degrading to file

I produced the indices for each N from 1024 to 262144 in order of how good of a subchannel each of the indices are. This means that I will not have to run the degrading procedure anymore for these values of N.

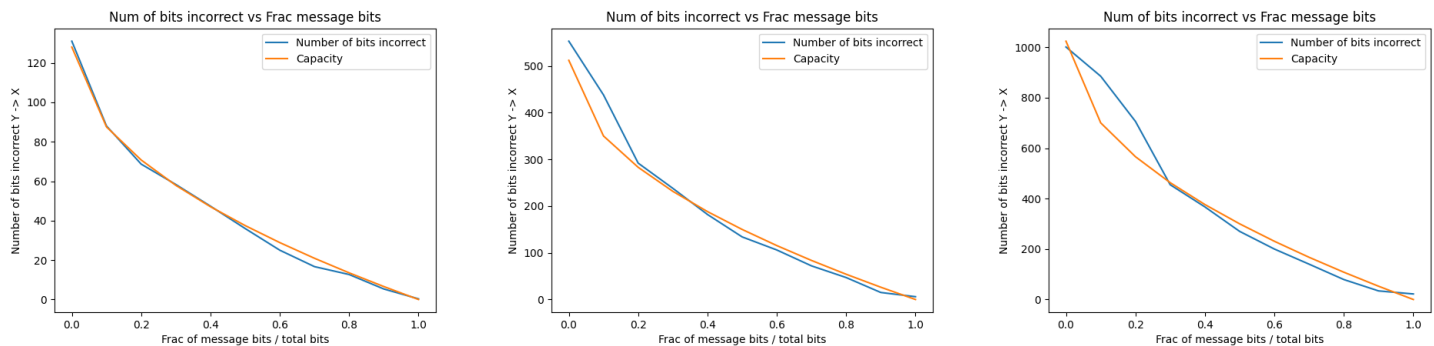
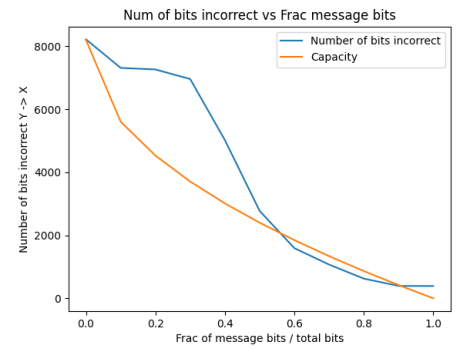
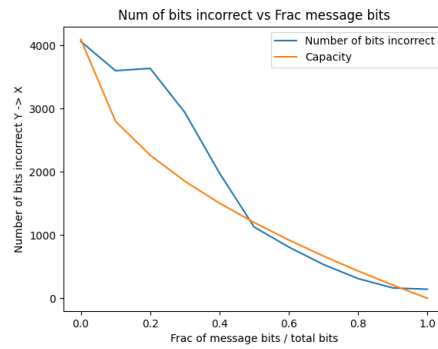
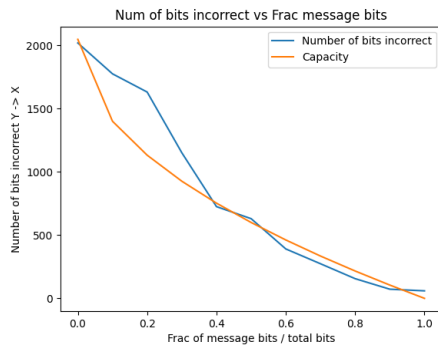
17.2 Simulation Plots for larger N

I will be redoing the plots from last week for $N = 4096$. The plots will follow a binomial distribution.

Trials = 100, # of bins = 100:

17.3 Rate / # of bits incorrect for larger N

I redid the plot from last week for $N = 16384$. As N increases, I expect that the gap between this rate vs # of bits incorrect and the capacity of the channel will decrease. However, I am seeing the opposite to be true:

$N = 256, 1024, 2048, 4096, 8192, 16384$:

60

I am not sure why this is occuring...

Simulator check: Are the agreed frozen bits the same as the frozen-indices of y? Both are randomly generated...
NOT THE SAME!!!

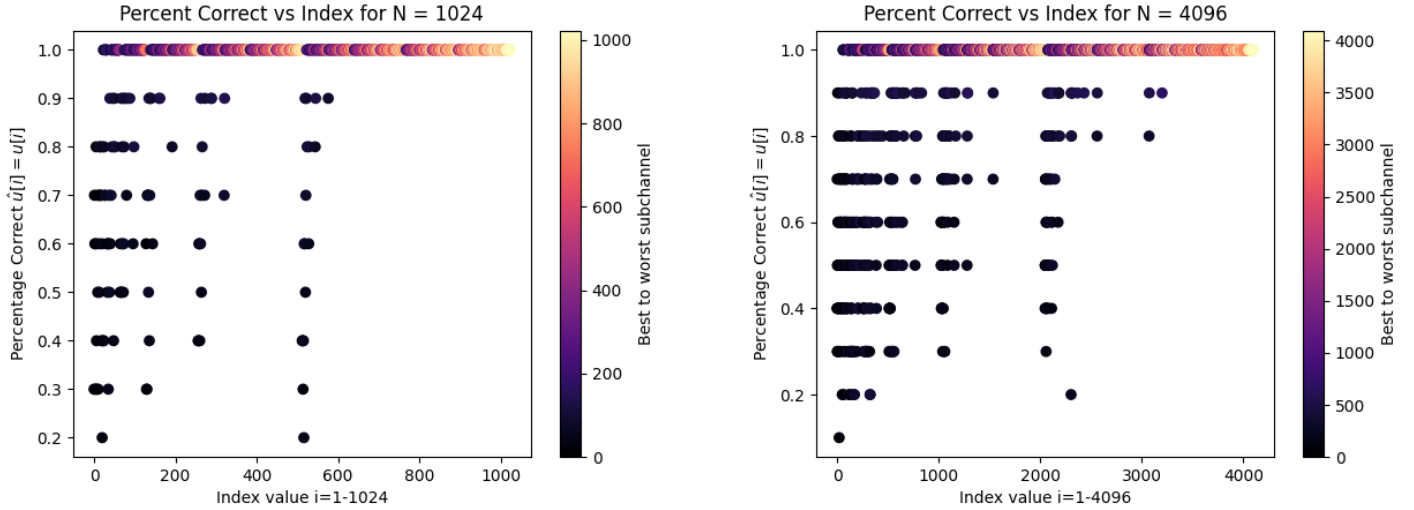
17.4 Vector Quantization

This will be used to simulate a BSC. Currently, my simulator is over a single cell. Next, given an x, fix a d such that d bits will be flipped at random. This will be simulated over a ball.
What is the difference in the code when simulating over a ball vs a cell? What must change within the simulator?
Will the $\hat{u}$ bits be flipped as they cross the channel?

61

18   Week 18: February 23 - March 1

18.1   Independent Subchannels Test

I re-ran the tests from previously to examine which subchannels are good if they are measured independent of the other subchannels. Part of the test is to see if my ranking of subchannels also makes sense given the independent experiments, so I color-coded the plot of fraction of bits correct for each subchannel when examined independently. The ligher the color, the lower the error probability is and the better the subchannel is. The darker colors represent the worst subchannels. I received this plot below, showing that indeed the best subchannels are performing the best when subchannels are tested independently, and the worst channel are not used as much. The results below are for $N = 1024, 4096$.

From the results above, the subchannels seem to be ranked properly and are not causing the results we see in the simulator.

18.2   Uniform Cell Test

I would expect that for a distinct choice of frozen bits used in the simulator, a unique $\hat{x}^N$ will be produced. I set up an experiment to test this by choosing a small $N = 4$ and running the simulator on all possible frozen bit inputs and comparing the outputs $\hat{x}^N$. My results show that the outputs $\hat{x}^N$ are unique across all possible possible arrangements of frozen bits, so this is not an issue.

18.3   Numerical approximations

I looked at the approximations used in the log likelihood calculations, and when I vary the order of magnitude threshold for which I determine that one number is negligible compared to the other, I do not see an impact in the simulation plot that does not match capacity. The number of incorrect bits does not change, leading me to believe that this is not the source of the issue.

18.4   Permute Indices

I tested if permuting the in-order indices as done in the encoder makes a difference in the simulator results. In previous tests with the decoder, I saw that for small N, whether a subchannel was good or not was preserve when the indices were sent through the bit reversal matrix B_N. However, for large N, this diverged slightly, but mostly held constant.
Since we were seeing simulator results diverge from the expected capacity measurement for large N, I thought this may be a cause of the issue.
I multiplied the in-order indices by matrix B_N, the bit reversal permutation matrix, and I see that the simulator now has results with 50% of the indices incorrect regardless of the value of k, the fraction of message bits. So this is making the error worse and is not the solution.

62

19 Week 19: 3/2-3/8

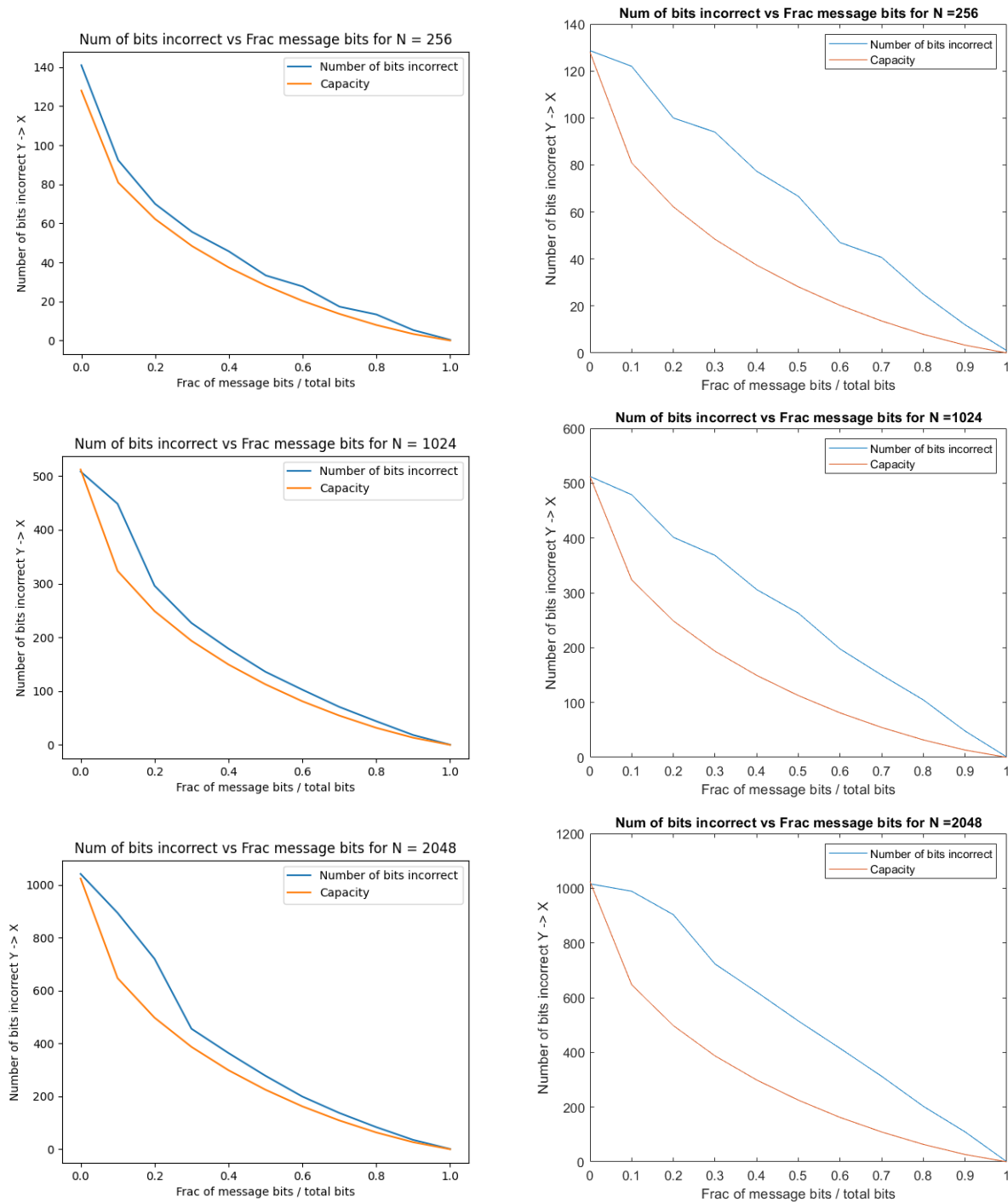
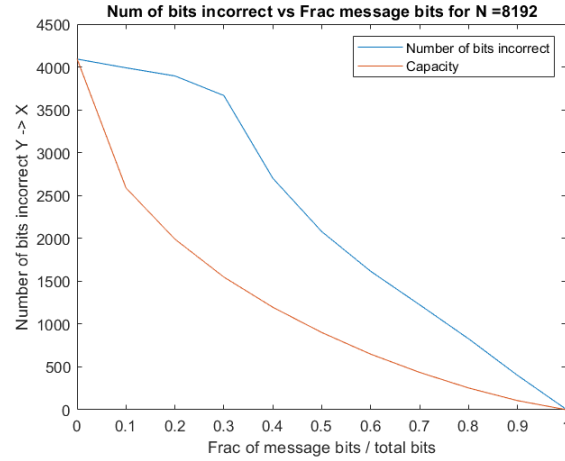
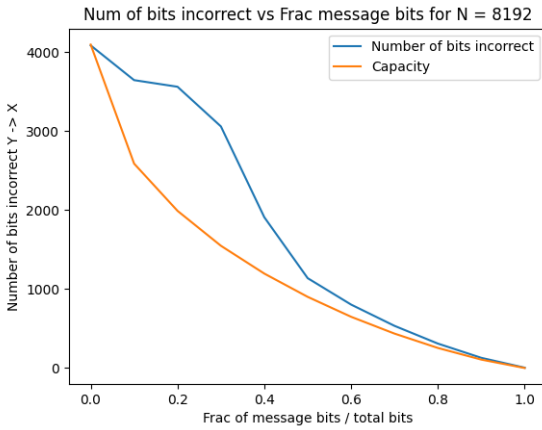
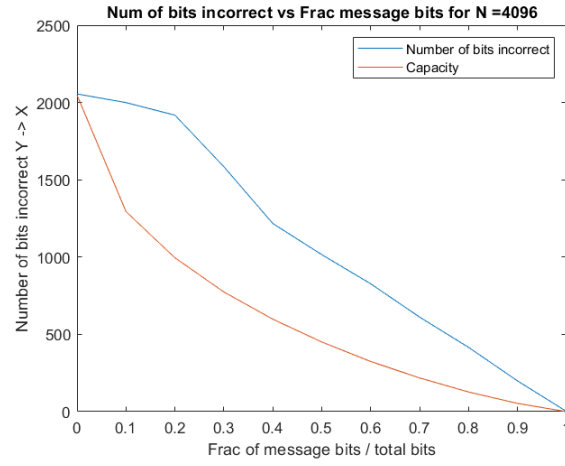
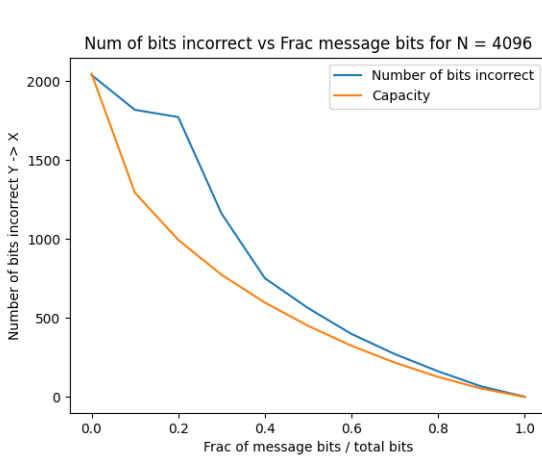
19.1 Reproducing plots with alternate encoder & decoder implementation

19.1.1 List size = 0, CRC = 0

External code is found here: https://github.com/tavildar/Polar?tab=readme-ov-file

Below are the resulting using the MATLAB implementation of the code above, the left are the results from my code, and the right are the results from Sourab's code using the same test code:

63

Num of bits incorrect vs Frac message bits for N = 4096

(These plots are the same for varying ϵ)

All plots are from an average of 3 trials for each value of k [fraction of message bits] and both use the same simulator code, only the communication encoder and decoder functions are swapped from my implementation and Saurabh's implementation. I am using the *PolarCode.polar_encode()* and *PolarCode.polar_decode()* functions for the plots above.

It looks like Saurabh's results are much further from the capacity curve than mine and the same flattening of the curve occurs for small values of k.

Is this an effect of an error in my simulator? Or is the capacity curve not a good metric to compare to? I will also continue to verify my simulator implementations and utilization of Saurabh's code are correct.

After looking further into the above, I realized that Saurabh's code is assuming frozen bits are always assumed to be 0. So, I decided to correct this is in my code. Prior, I was selecting random frozen bits and sending a zero y_N. This difference may be causing the higher error probabilities in the plot above.

When I swap this, always selecting frozen bits to be 0 and sending a random y_N, I run into an error in the decoder where a matrix in the *vnop* function is sometimes zero for random y_N, causing a divide by 0 error. This occurs very often, especially for larger values of N. Is it possible that my definition of y_N is different than the vector that is intended for input here?

Another difference between my and Saurabh's code is that my code returns decoded bits in order of index $i = 0, ..., N-1$ and Saurabh's code returns decoded bits in order of the message_inds = info_bits¿ For example, if $N = 16$, the decoded bits will be returned in order of indices $i = 16, 15, 14, 12, 8, 13, ...$ However, the PolarCode class function *PolarCode.polar_decode* is ordered indices $i = 0, ..., 16$. This should be fine, I just need to account for this difference in my simulator code.

19.1.2   List size = VARIABLE, CRC = 0

I am using the function *decode_scl_pl(obj,p1,p0,list_size)*, where $p1$ is the input y_N, $p0$ is the binary inverted version of $p1$, and *list_size* is the list size as input. The only code that is changing is the simulation encoder:

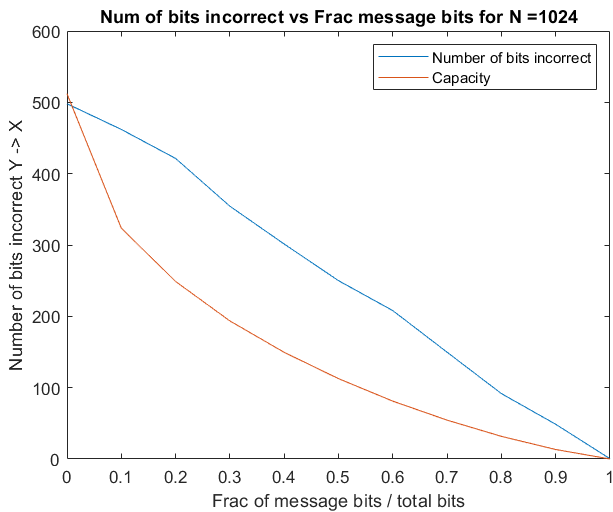
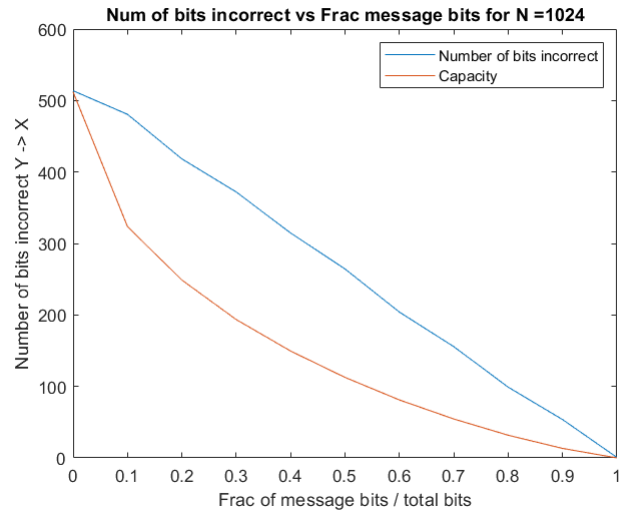
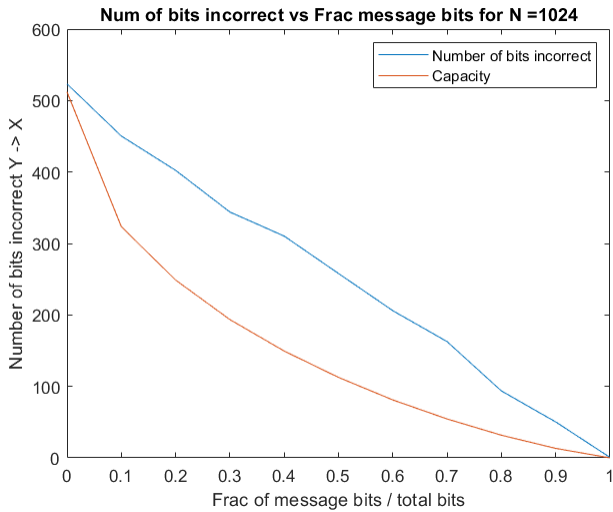
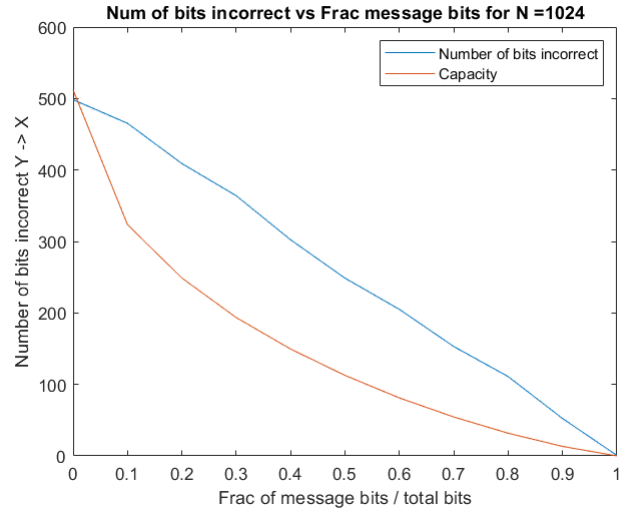
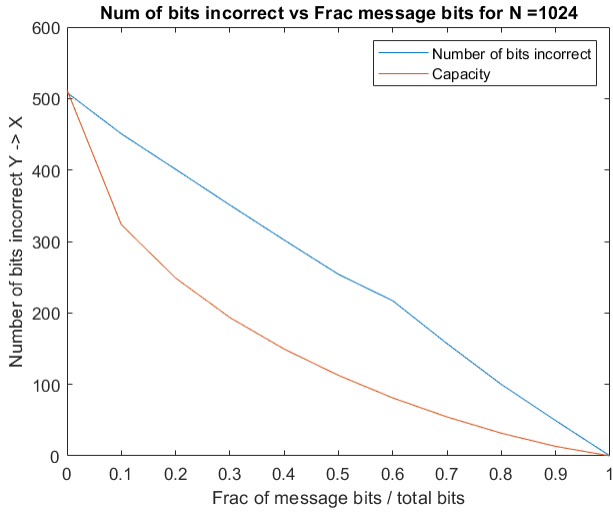
```
function message_uhats = sim_encoder(obj, y_N)
    p1 = y_N;
```

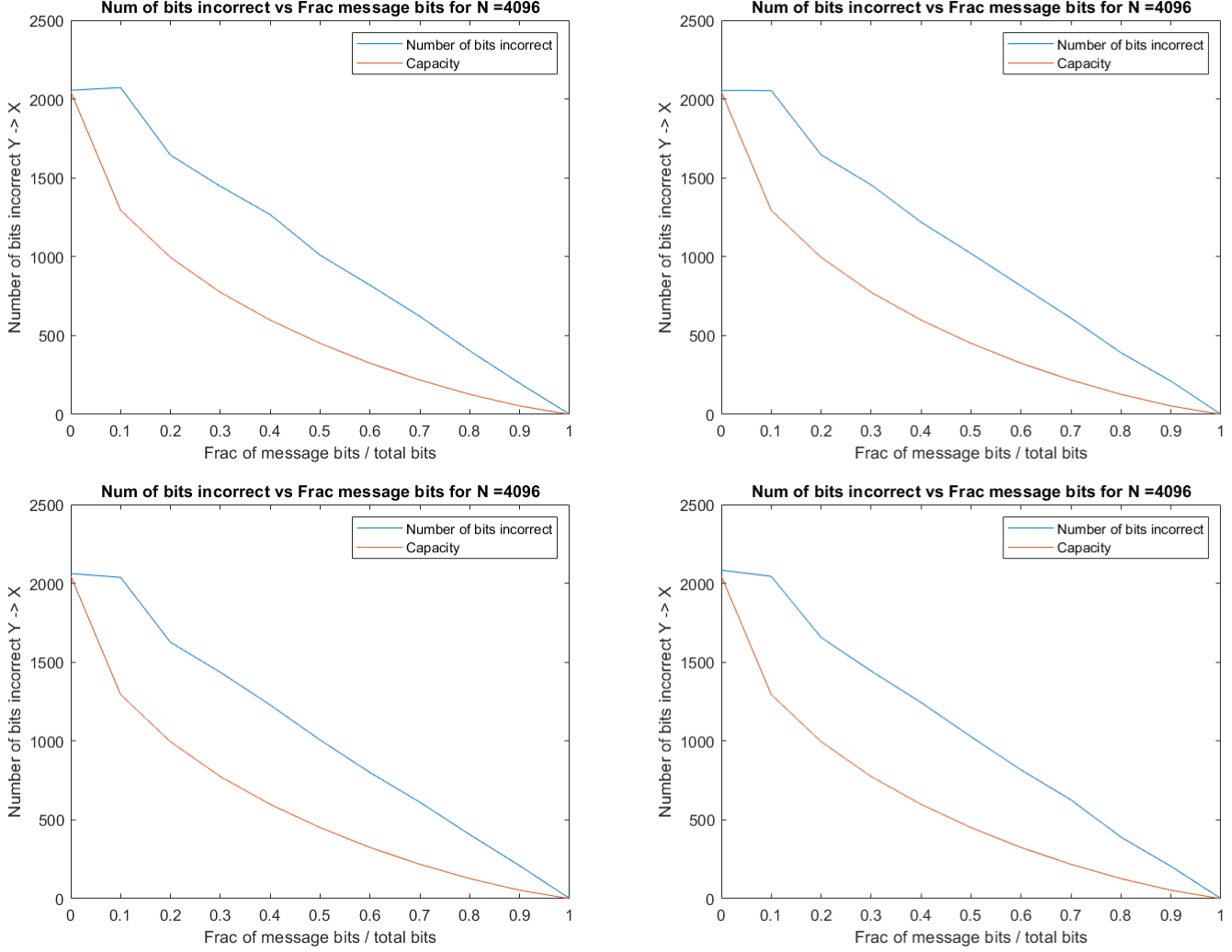
64

```
        p0 = ~p1;
        list_size = 10;

        message_uhats_outoforder = obj.decode_scl_p1(p1, p0, list_size);
        [~,order] = sort(obj.info_bits);
        message_uhats = message_uhats_outoforder(order);
end
```

This function takes more time that the other decoder algorithm.

The only inputs that work with this code are $y_N = \overrightarrow{0}$ and $y_N = \overrightarrow{1}$, and I chose to evaluate this code with $y_N = \overrightarrow{0}$. Here are the results for list_size $= 1, 2, 5, 10, 20$.

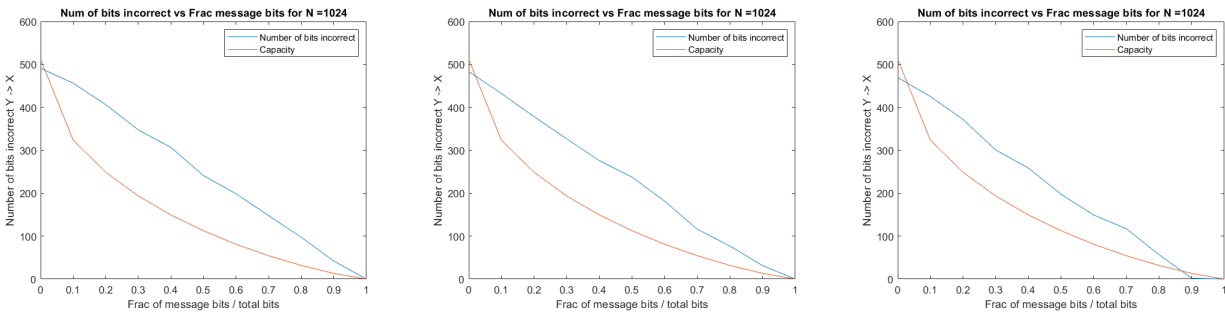
65

The list size does not seem to make a significant difference in the performance of the simulator for $N = 1024$.

I redid this with larger $N = 4096$ to further see the results:

I still do not see a difference in simulator performance.

19.1.3   List size = 1, CRC = VARIABLE

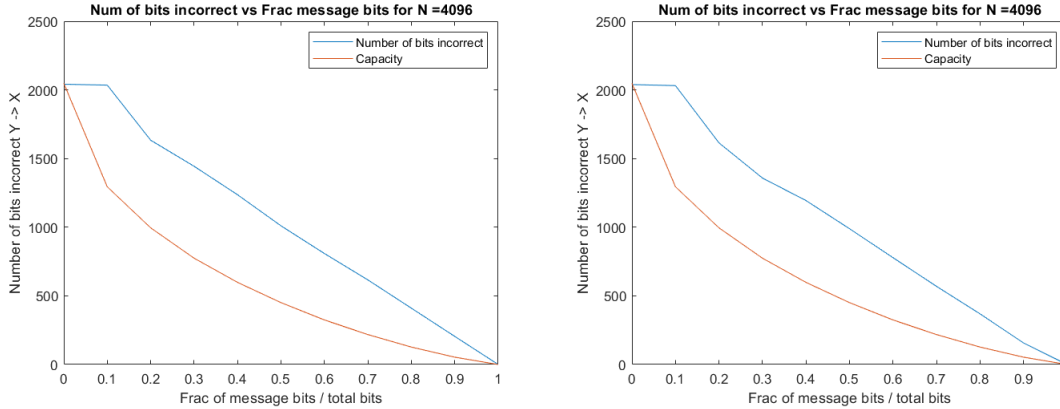
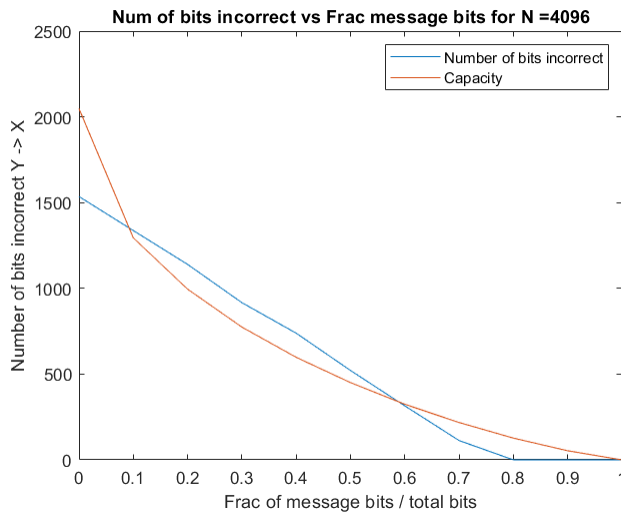
I altered my matlab simulator code to use the *obj* object that is defined in the PolarCode class. The following code is added to change the number of error correction bits to 10, for example.

```
CRC = 10;
if CRC + p > N
    CRC = N-p;
end
obj = PolarCode(N,p,epsilon, CRC);
```

I ran this code with $N = 1024$ and the following plots are CRC $= 10, 50, 100$:

I see that the addition of error correction bits is decreasing the fraction of bits incorrect from $y \to \hat{x}$, but still is not helping

66

in bringing the fraction incorrect curve down to the capacity curve.

I redid the plots with $N = 4096$ and $\text{CRC} = 10, 100$

I do not see a difference in these plots. When I make the number of error correction bits very large, 1000 for example, I see the following results:

The performance is better, but is the curve of number of bits incorrect is now below the capacity curve.

19.1.4 Compare ranking of Subchannels

The ranking of subchannels between my code and Saurab's code is different, and this difference increases with increasing N. However, many adjacent subchannels are simply swapped. This may be a result of using the degraded channnged vs the Bhattacharya parameters.

19.2 Prefix Condition

I assert the prefix condition by prepending the binary representation of the number of message bits to the actual message bits as the encoder of these message bits. By doing this, a string of many message bits and their lengths are sent to the simulator at one time, allowing for a continuous stream of data. The message bits will now be an input to the simulator as shown in this code here:

```
# Takes a list of input strings and create a single string with prefix condition
# length_bits1 + log2(N1) + message bits 1 + length_bits2 + log2(N2) + message bits 2 + ...
def append_prefix(message_arr,N):
    encode_str = ""
    for msg_bits in message_arr:
        p = len(msg_bits)
        length = str(bin(p)[2:].zfill(int(np.log2(N))))
        encode_str += length + msg_bits
    return encode_str

# Take an input string of many messages prepended by N bits denoting their length
```

67

```python
# Return a list of deciphered bits
def decode_bit_arr(full_message_str, N):
    message_arr = []
    ind = 0
    while ind < len(full_message_str):
        # Get length of message bits
        length_bits = full_message_str[ind:ind + int(np.log2(N))]
        length = int(length_bits, 2) # decipher bits to get # of message bits
        ind += int(np.log2(N))

        # Extract message bits
        curr_message = full_message_str[ind:ind+length]
        message_arr.append(curr_message)
        ind += length
    return np.array(message_arr)

def simulation_prefix(N, epsilon, message_bits):
    p = len(message_bits) # number of message bits

    file_name = "indices_" + str(N) + ".txt"
    with open(file_name) as f:
        ordered_inds = [int(line) for line in f.readlines()]

    message_inds = ordered_inds[:p] # This will hold all message indices

    print("# Message bits / # total bits: " + str(p) + "/" + str(N) + "=" + str(p/N))

    # Generate y bits given input message_bits
    y_N = np.zeros(N)
    j = 0
    for i in message_inds:
        y_N[i] = message_bits[j]
        j += 1

    # randomly assign frozen bits
    froz_N = np.random.randint(2, size=N-p)

    # Simulation encoder to obtain message bits
    u_hat_message_N = sim_encoder(message_inds, y_N, epsilon, N, froz_N)

    # Send message bits across channel...
    print("Sending u_hat message bits...")
    # nothing happening in the channel now, will at some point

    x_hat_N = sim_decoder(message_inds, u_hat_message_N, N, froz_N)

    # Calculate num incorrect from x_hat_N to y_N elementwise
    num_incorrect = sum(x != y for x, y in zip(x_hat_N, y_N))

    print("Frac incorrect = " + str(num_incorrect / N))
    print("Num incorrect = " + str(num_incorrect))

    return num_incorrect

# Send random strings of bits, each of size N
def send_bits(N, p_arr, num_messages, epsilon):
    assert num_messages == len(p_arr)

    message_arr = []
    for i in range(num_messages):
```

68

```
        message = np.random.randint(2, size=p_arr[i])
        message_arr.append(''.join(map(str, message.astype(int))))
    message_arr = np.array(message_arr)

    # Append prefix
    full_str = append_prefix(message_arr, N)

    # Get message bits individually
    all_message_bits = decode_bit_arr(full_str, N)

    for message_bits in all_message_bits:
        simulation_prefix(N, epsilon, message_bits)

# Example run
send_bits(N=1024, p_arr=[700,900, 500, 1000], num_messages=4, epsilon=0.01)
```

where the *send_bits* function is the main function to run if you want to send p_i message bits, total block length N, and num_messages messages with a given ϵ.

Now that I have code that upholds the prefix condition, there are additional metrics that we can compare to my current performance.

69

20 Week 20: March 9 - March 15

20.1 Fixed Saurabh's Code

Last week, I though the input $p1$ to the decoder can be the same as simulation encoder input y_N. However, this is actually a soft decoding input, meaning I need $p1$ to represent the input as given by the preceding BSC, which does not appear in the simulator but is in the communication scheme. Therefore, I have these probability measurements:

$$P_1[i] = Pr(X[i] = 1 | Y[i] = y[i]) = \begin{cases} 1 - \epsilon & \text{if } y[i] = 1 \\ \epsilon & \text{o.w.} \end{cases}$$

$$P_0[i] = 1 - P_1[i]$$

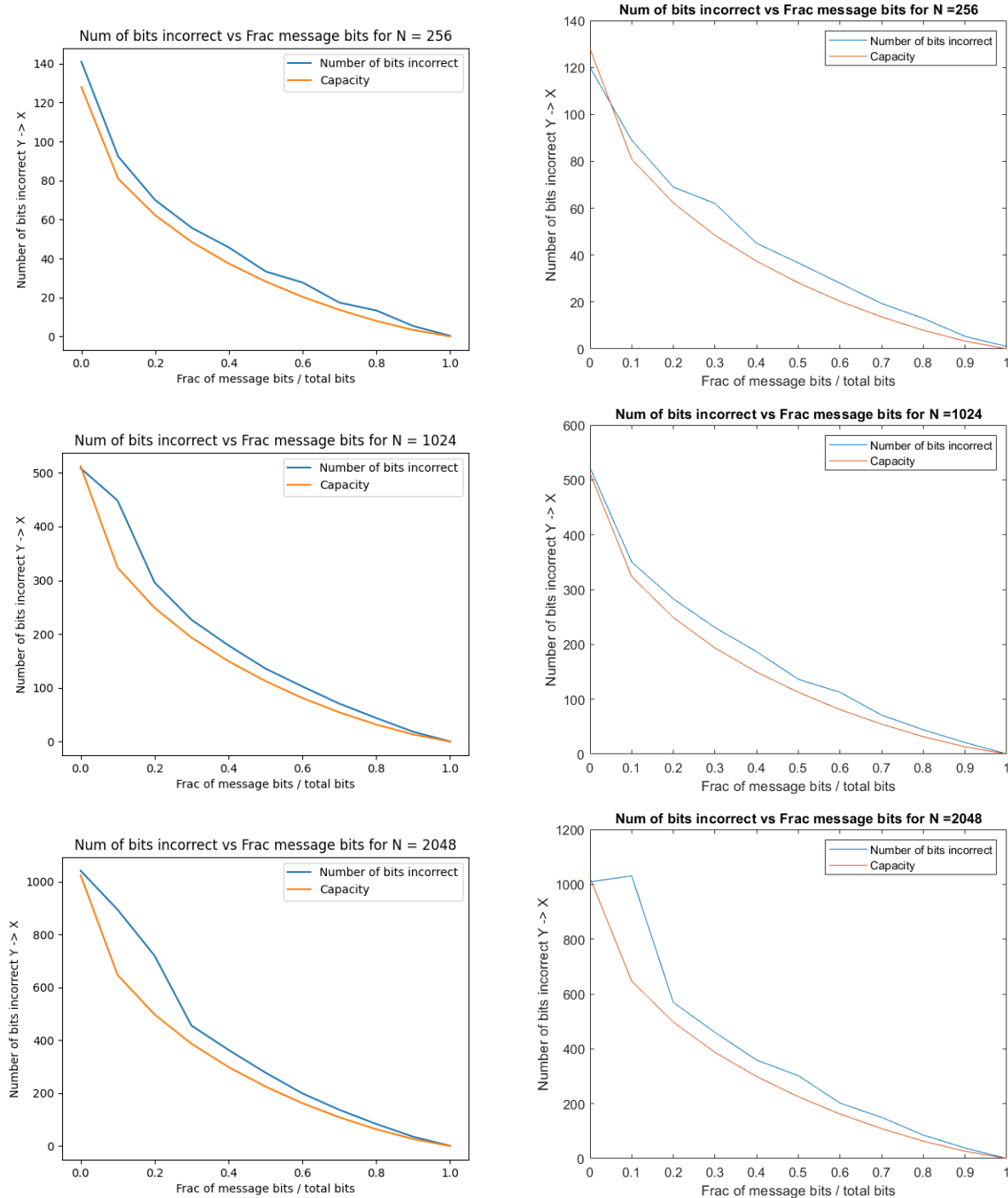
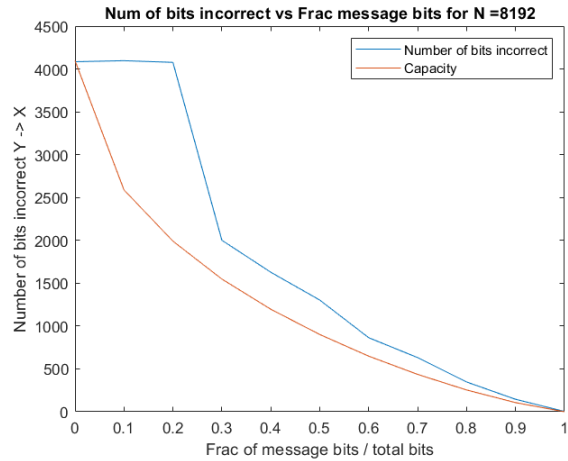
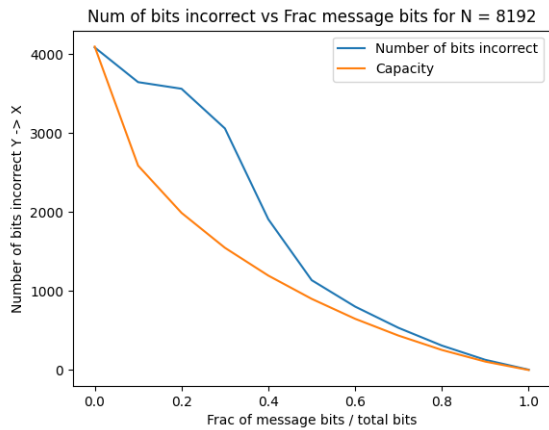
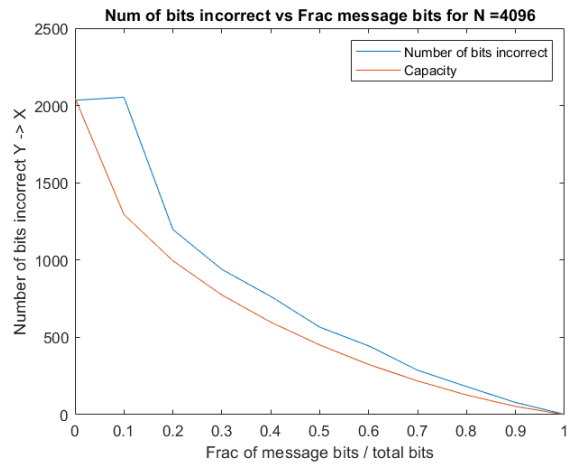
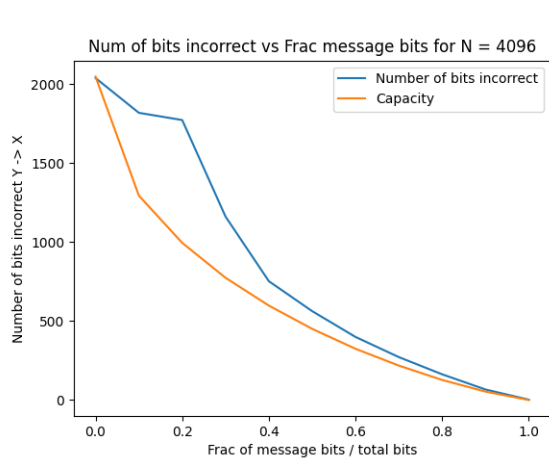
I corrected my code with this change, and I have the following plots, where the left column are my plots and the right side are Saurabh's results:

70

Num of bits incorrect vs Frac message bits for N = 4096

Num of bits incorrect vs Frac message bits for N =4096

Num of bits incorrect vs Frac message bits for N = 8192

Num of bits incorrect vs Frac message bits for N =8192

Comparing horizontally, the plots are very similar, seeing the same gradual decrease in number of bits incorrect instead of a decrease proportional to the capacity. This effect is shown to a further extent in the plots on the right, with the fraction of incorrect bits hovering around 0.5 until a later value of k, before decreasing towards the capacity plot. My plots have a slight decrease at first, and then remain around a smaller fraction before returning to the capacity plot. I will test below if adding error correcting bits and increasing list size has a big impact on these plots.

20.1.1 Variable List Size

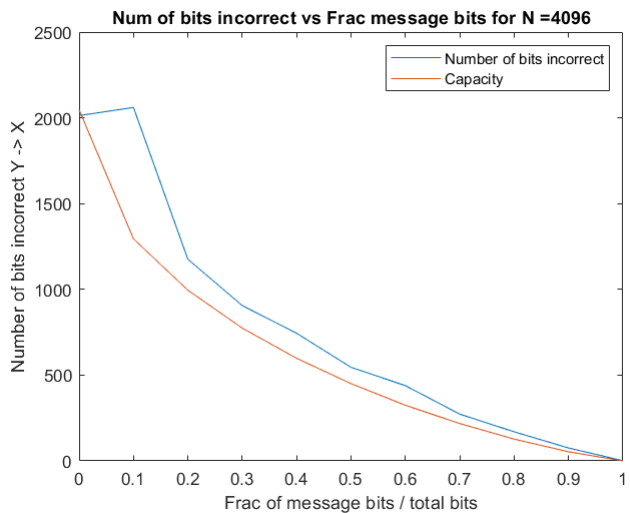
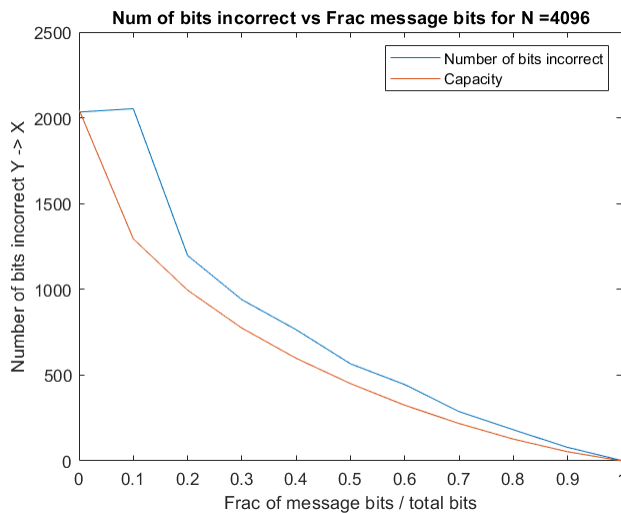
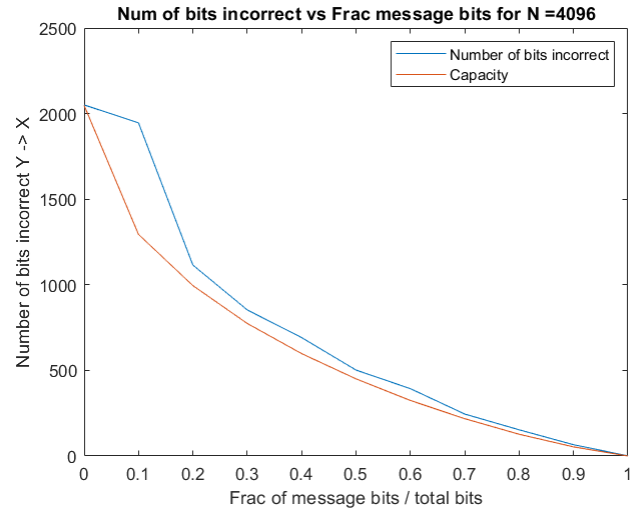
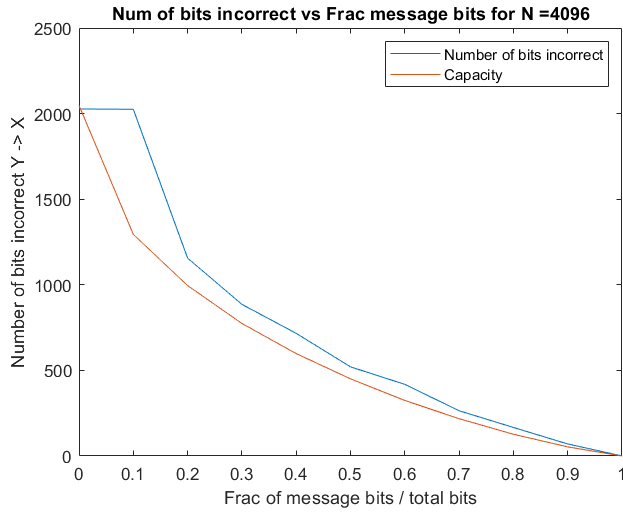
I varied the list size for Saurabh's list decoder where list size $= 1, 2, 5, 50$ for $N = 4096$:

Num of bits incorrect vs Frac message bits for N =4096

Num of bits incorrect vs Frac message bits for N =4096

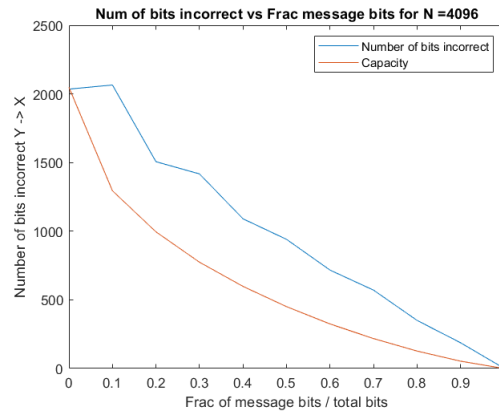
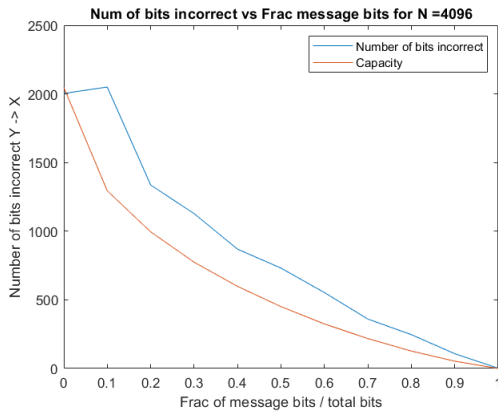
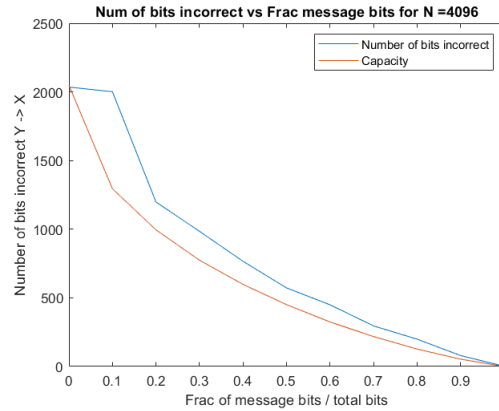
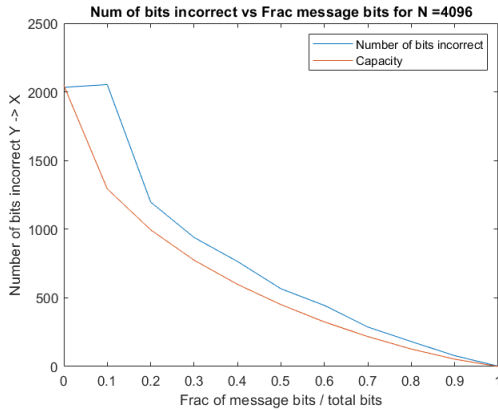
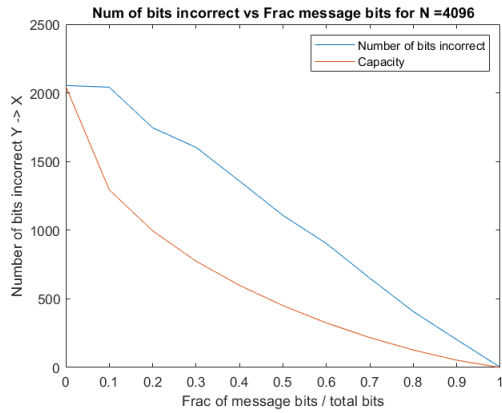
71

Num of bits incorrect vs Frac message bits for N =4096 (left)

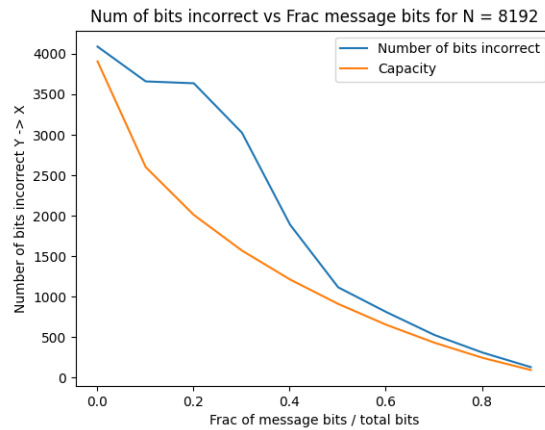
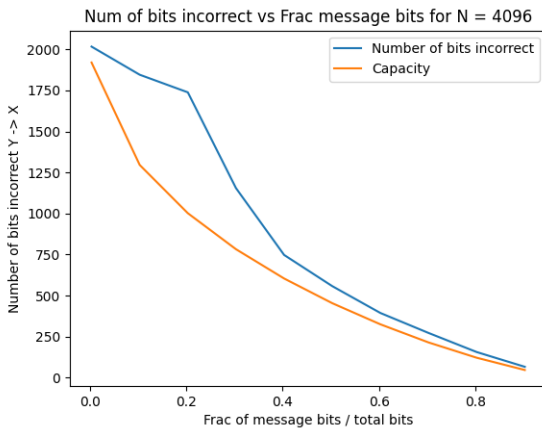
Num of bits incorrect vs Frac message bits for N =4096 (right)

The only difference I see in these plots is that the distance to the capacity plots decreases as list size increases, but only by a small bit.

20.1.2 Variable CRC bits

The following plots are for $N = 4096$ and # of error correcting bits $= 0, 1, 10, 100, 1000$

72

I don't believe that the simulation fraction of bits incorrect should increase with more error correcting bits, so I will continue to look at this..

20.2 Prefix Code

I obtained the capacity curves from my prefix code last week. The difference is in the x-axis where instead of plotting $\frac{\#\text{ message bits}}{N}$, now there are $\log_2 N +$ message bits sent to the decoder, so the horizontal axis is $\frac{\log_2 N + \# \text{ message bits}}{N}$. Below are the plots for $N = 4096, 8192$:

These are shifted versions of the previous capacity plots.

20.3 Subchannel Ranking Depends on ϵ?

I ran the degraded algorithm to check if changing epsilon alters the ranking of subchannels, and I found that indeed the subchannel ranking **DOES** change with ϵ. For example, here are the first few best subchannels for $N = 1024$ and varying ϵ:

$\epsilon = 0.01 : 1023, 1022, 1021, 991, 1020, 990, 831, 759, 507, 503$
$\epsilon = 0.05 : 991, 1015, 1007, 1014, 1011, 1006, 1005, 1003, 999, 990$
$\epsilon = 0.1 : 1023, 1022, 1019, 959, 895, 767, 1021, 1015, 1020, 1018$

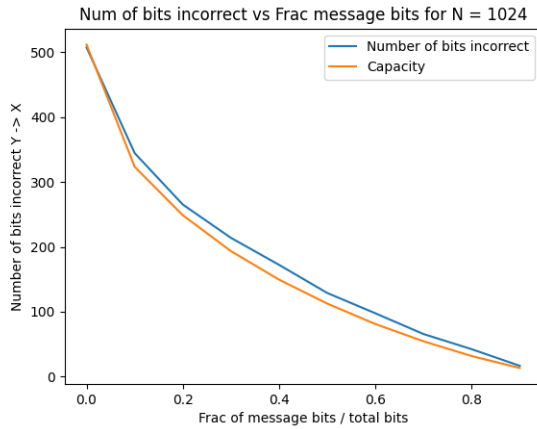
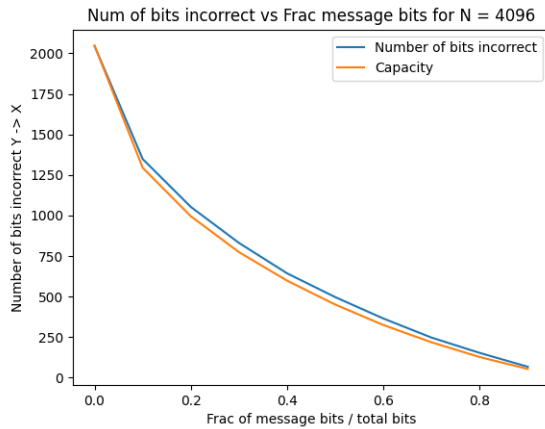
73

21 Week 21: March 16 - March 22

21.1 Fixed ϵ misconception

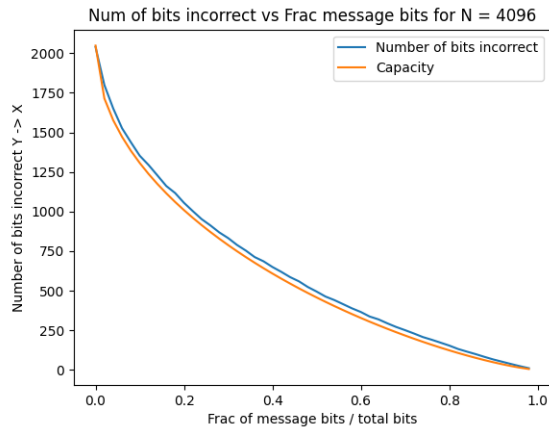
Last week, I found that the ordering of the subchannels is different when ϵ varies. This gave rise to a discussion during this week's meeting that perhaps the value of ϵ is not set prior to the simulation problem, but rather depends on the rate R, which is the number of message bits. Therefore, the value of epsilon will be the same as the capacity of the plot, which is:

$$\epsilon = H^{-1}(1 - R)$$

Prior, I was setting the value of $\epsilon = 0.01$ always, which is closer to the ϵ that results from larger R. By replacing the value of epsilon in my experiment with this value dependent on the rate. Here is the resulting plot with $N = 1024$:

This was a great result as the number of bits incorrect is almost on top of the capacity curve, meaning ϵ in $\Rightarrow \epsilon$ out. Here are the results with $N = 4096$:

The curves have less distance between then as N increases, which is expected. We see the results change very drastically as the value of ϵ is much larger than 0.01 for small rates R, so the subchannel rankings varying greatly. However, for large rates R, the value of ϵ is around 0.01, so the expected results are shown.

This increases the timing complexity of my code since the degraded algorithm must run each time a new ϵ is used for a new N, but the results are correct now. I ran this code for $N = 4096$ and at smaller increments of rate R.

74

Num of bits incorrect vs Frac message bits for N = 4096

At this point, I am still using an approximaion for inverse binary entropy, since there is no closed-form expression for this inverse function. This approximation is:

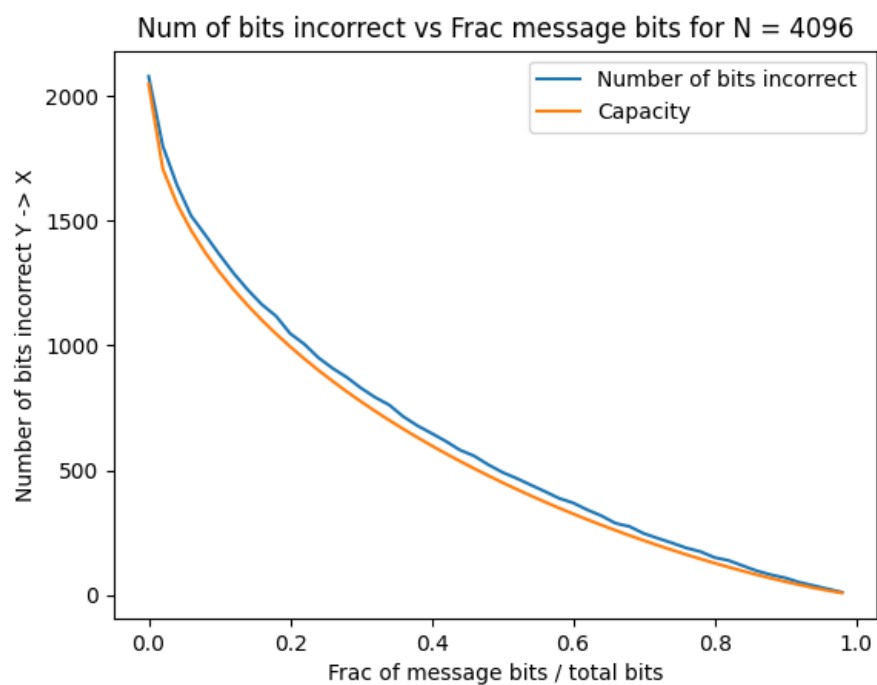
$$\frac{1}{2}\left(1 - \sqrt{1 - (1-R)^{4/3}}\right)$$

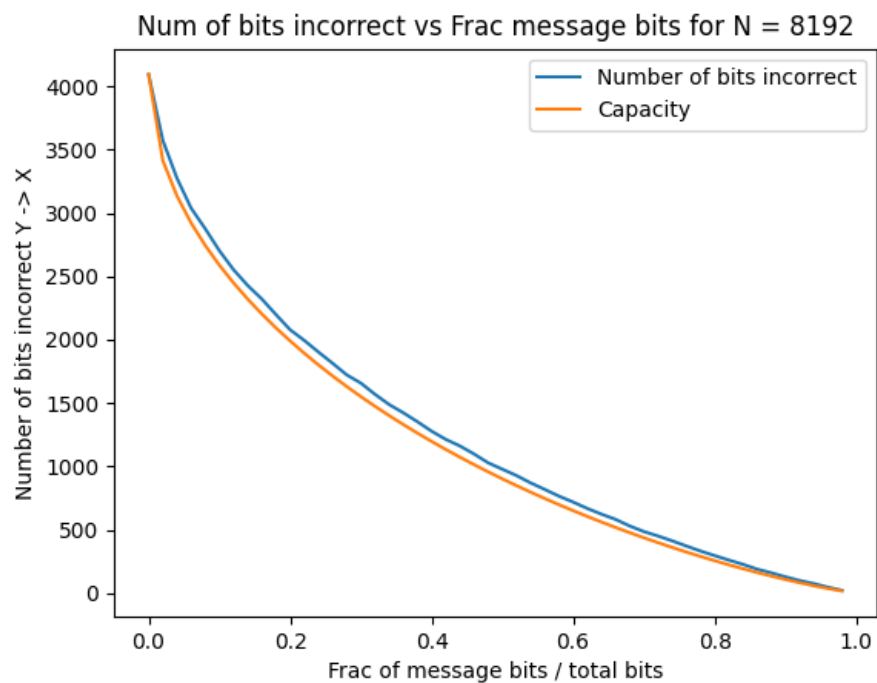
This is a pretty good approximation, but I see that it is off by around 0.01 for each point. To increase accuracy in my comparison plots above, I decided to utilize the *scipy.optimize* package, specifically the *minimize_scalar* function. I then used the following code:

```
def H(p):
    return -p * np.log2(p) - (1 - p) * np.log2(1 - p)

def inverse_H(arr):
    vals = []
    for val in arr:
        # Define a function that returns the absolute difference between H(p) and the desired val
        func = lambda p: abs(H(p) - val)
        # Use minimize_scalar to find the value of p that minimizes the absolute difference
        result = minimize_scalar(func, bounds=(1e-15, 1-1e-15), method='bounded')
        vals.append(result.x)
    return np.array(vals)
```

This code worked well, giving values for $H^{-1}(1 - R)$ that are much closer to the correct value. The only issue is that I wish to have an inverse value $= \epsilon$ that is < 0.5, as this corresponds to my plots. Some of the inverse values are greater than 0.5 since $H(x) = H(1 - x)$ by definition. I added one more line in my code that if $H^{-1} > 0.5$, just subtract it from 1 to get the proper inverse value that I am searching for. This may not be necessary for small N, but it is good for larger N to have greater accuracy to compare the results of the simulator to. This worked well, and here is the updated plot for $N = 4096$:

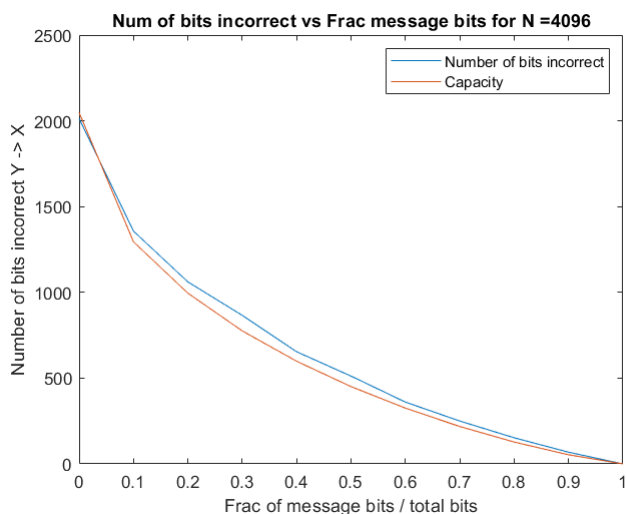
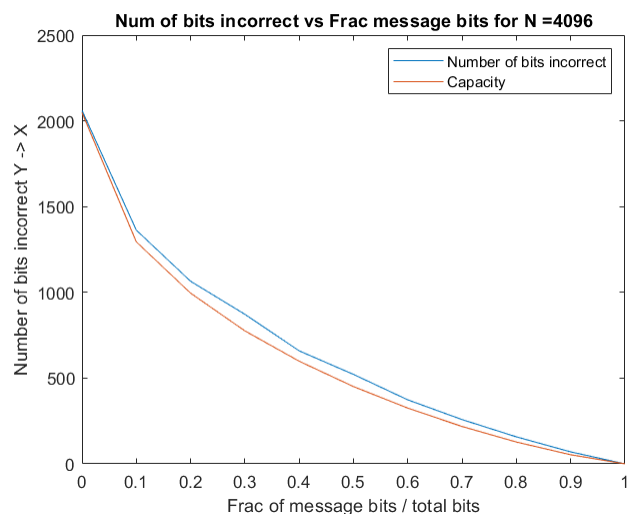
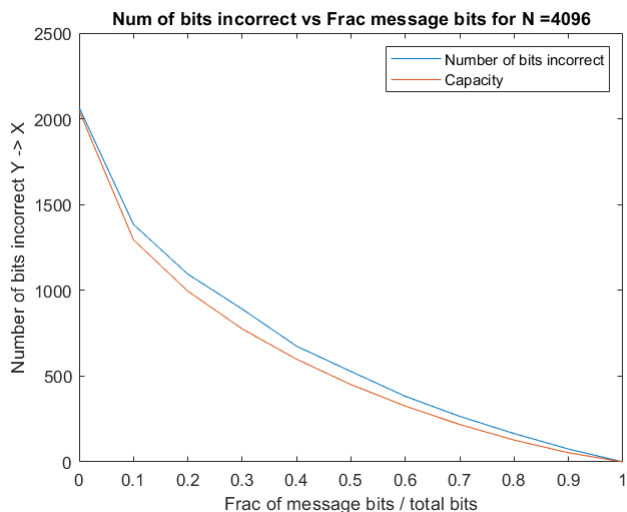
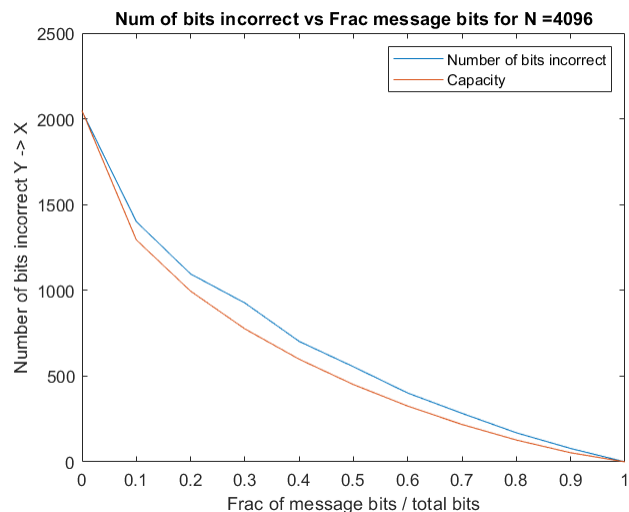
75

Num of bits incorrect vs Frac message bits for N = 4096

This curve is closer to the expected capacity curve.
I also ran this code for $N = 8192$:

Num of bits incorrect vs Frac message bits for N = 8192

The cuve looks a bit closer to the expected capacity curve than for smaller N, but surprisingly there is not a large difference between these two values of N.

76

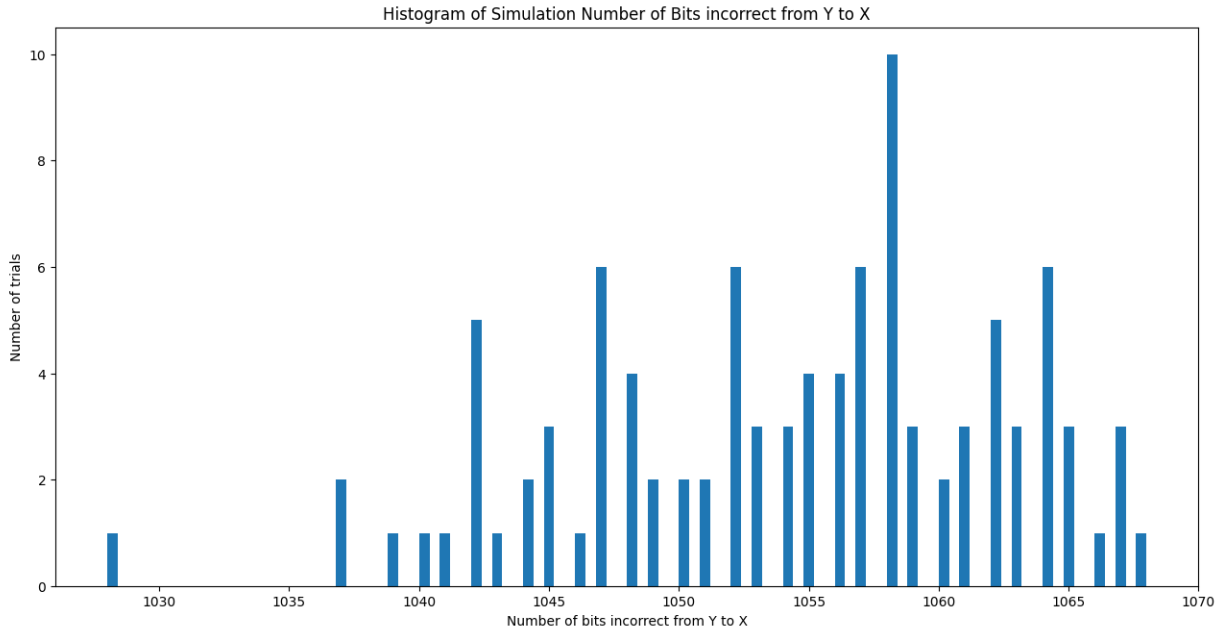
21.2 Fix on Saurabh's code

I implemented the same fix above on Saurabh's code with $N = 4096$ and for list sizes 1,2,5,10 below:

I see that increasing the list size improved simulator performance, but these blue plots are further from the orange curve than my blue plots to begin with.

77

21.3   Histogram

This is a histogram for $N = 4096$, $k = 0.2 \Rightarrow \epsilon = 0.24$:

Histogram of Simulation Number of Bits incorrect from Y to X

To compare this to the binomial distribution:

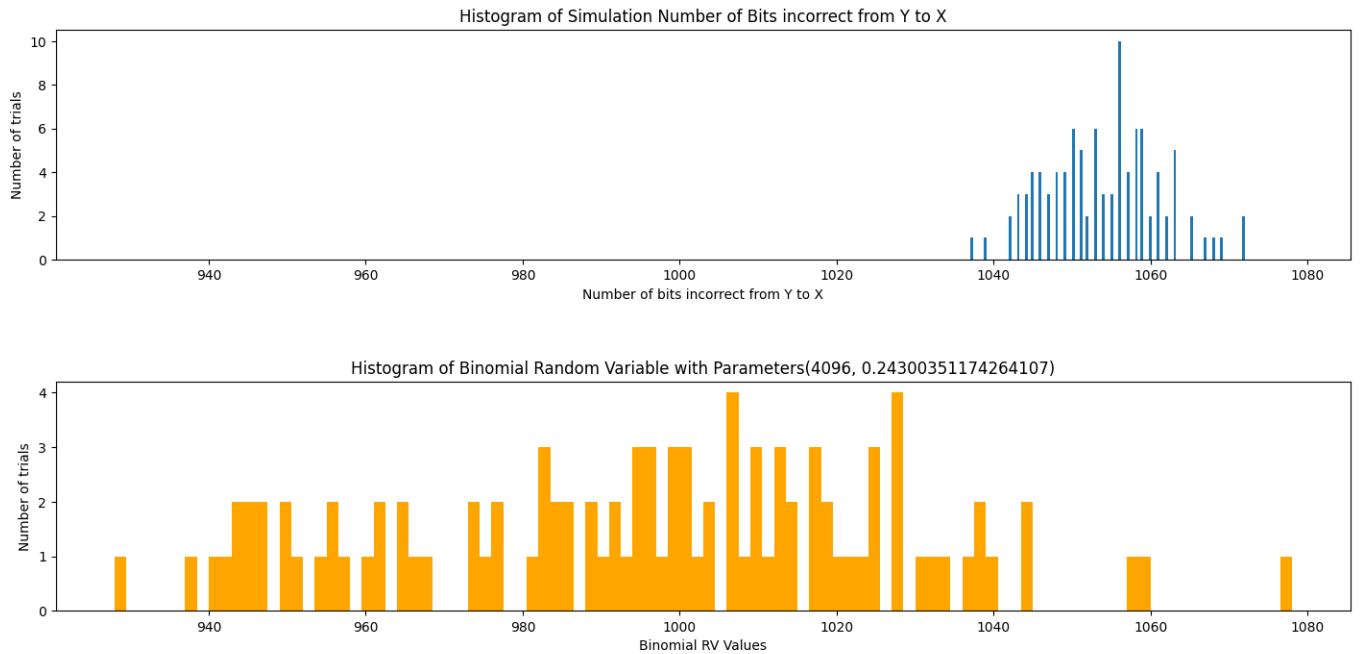
$$P(x) = \binom{n}{n} p^x q^{n-x}$$

This binomial random variable will have parameters $\text{Bin}(N, \epsilon)$

$$P(x) = \binom{n}{n} \epsilon^{N*\epsilon}(1 - \epsilon)^{n-x}$$

Here is a plot of the resulting histogram and binomial random variable outputs:

Histogram of Simulation Number of Bits incorrect from Y to X

Histogram of Binomial Random Variable with Parameters(4096, 0.24300351174264107)

The center of these values is slightly off at the moment since the simulator is currently for an approximate BSC. With an exact BSC simulator, which I will work on next week, and with more trials, these histograms should be the same.

78

22 Week 22: March 23 - March 28

22.1 Simulating an exact BSC

Currently, my progrma is simulating an approximate BSC, where the number of bits that are flipped are random but close to $\epsilon \times N$ variables . I instead want to simulate an exact BSC since this is a common practice in the field literature. The randomly generated y^N input string contains m ones and $N - m$ zeros, so I will generate random variables k_0, k_1 to represent the exact number of 0s and 1s that I want flipped. k_0, k_1 will be binomial random variables with parameters as follows:

$$k_0 \sim \text{Bin}(n - m, \epsilon)$$

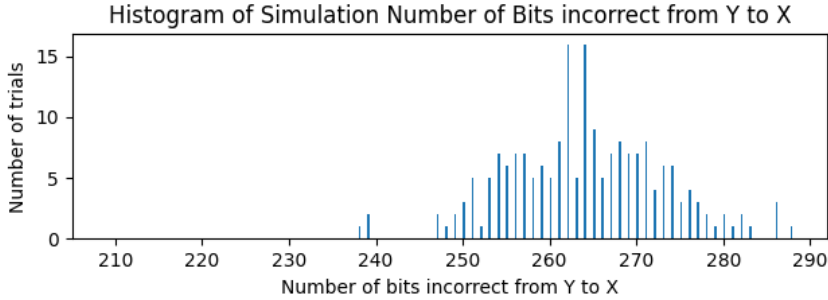
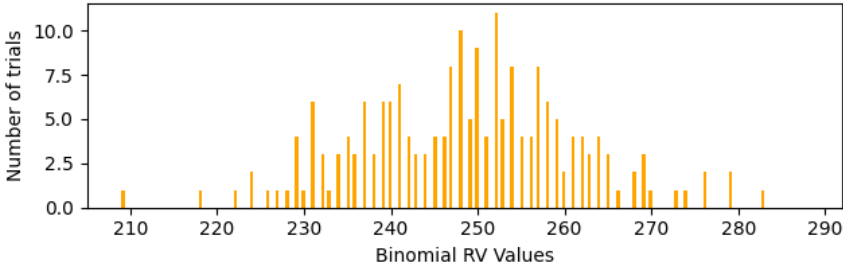
$$k_1 \sim \text{Bin}(m, \epsilon)$$

where $n =$ number of message bits.
The new flipping probability that will be used in the simulation will be

$$\delta = \frac{k_0 + k_1}{N}$$

This is so that the number of flipped bits for both 0,1 are close to k_0, k_1 to begin with so less time is spent on corrections. In the simulation decoder, I correct $\hat{x}^N$ so that the number of 0s flipped is exactly k_0 and the number of 1s flipped is exactly k_1. The value ℓ is the number of bits into $\hat{x}^N$ that must be iterated through until the number of flipped 0s $= k_0$ and the number of flipped 1s $= k_1$.
Some example outputs of num_flipped 1, 0 and k_0, k_1 and corresponding ℓ values for $N = 4096$, $\epsilon = 0.24$, $k = 0.2$ are:

num_flip0	k_0	num_flip1	k_1	ℓ
109	86	104	105	168
115	119	99	97	21
100	85	115	96	131
106	90	121	108	130
100	96	107	91	154

Here is a histogram for $N = 1024$, $\epsilon = 0.24$, and 200 trials for an exact BSC:

Here is a histogram for $N = 4096$, $\epsilon = 0.24$, and 200 trials for an exact BSC:

79

Histogram of Simulation Number of Bits incorrect from Y to X

togram of Binomial Random Variable with Parameters(4096, 0.243003511742

80