COSC264 Assignment 1

Dillon Thorsteinn George 38063248 Carl Kenny 93678486

Percentage Contribution

Both partners contributed equally to this assignment.

Question 1

The bit error rate was modeled using a Binomial distribution. As the probability of each bit error is independent of every other bit, and for each transmission the message length(n) and from this it follows that a Binomial distribution will accurately model the bit error rate. The number of 'successes' for this distribution are the number of errors occurring in a message of length n.

The python library Numpy was used to calculate the number of erroneous bits in each transmission. Numpy.random includes a binomial function, this function draws samples from a Binomial probability distribution.When n and p are passed into this function the return value simulated the number of errors in one transmission of a packet.

Question 2

The total number of bit errors follows a binomial distribution. Thus the probability of having m bit errors('successes') in a message of length n is given by:

$$P(X = m) = \binom{n}{m} p^m (1-p)^{n-m}$$

Where p is the probability of a bit error. And so the probability of having at least one bit error is:

$$\begin{aligned} P(X \geq 1) &= 1 - P(X = 0) \\ &= 1 - \binom{n}{0} p^0 (1-p)^n \\ &= 1 - (1-p)^n \end{aligned}$$

As required.

Question 3

In figure 1, there exists an extreme contrast between the two p values: 0.01 and 0.001. When the error-bit probability $p = 0.01$. The line for $p = 0.001$ is a smooth curve which approaches some asymptote(in this case the asymptote is u/v, which is the case where each packet transmitted only once), for $p = 0.01$ the line displays a 'sawtooth' pattern but still approaches the same asymptote albeit at a slower rate. This sawtooth behavior is due to the nature of the threshold value, each time the threshold increases a sudden 'leap' in efficiency is observed, and after this peak the efficiency gradually decreases until the next increase in the threshold. The gradual decrease is due to the message size increasing, but with the number of correctable errors staying the same. This sawtooth does not occur for $p = 0.001$ as the number of errors have a low probability of reaching thee threshold value for any given value of u.

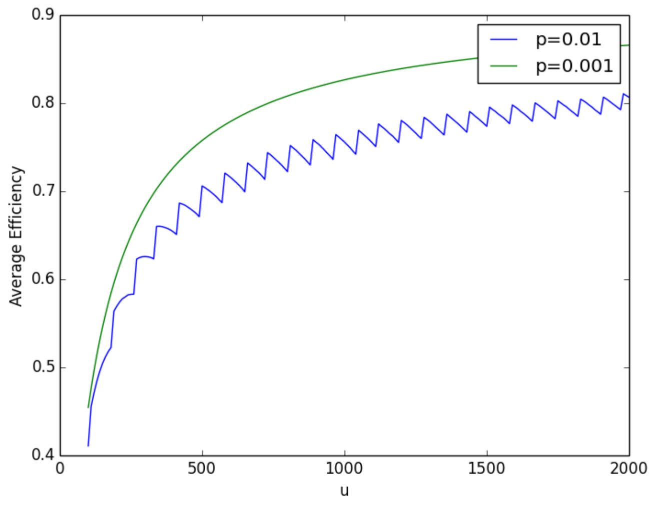

Figure 1: Average efficiency for $p = 0.01$, and $p = 0.001$ and varying u in the range $[100, 2000]$, where $n - k = \lceil 0.1 * (o + u) \rceil$

Question 4

Figure 2. below shows a relatively stable efficiency for p between 0.001 and around $0.004 - 0.005$, after which the efficiency sharply drops. This is due to the lower probabilities on average more bit errors over the threshold value, and the lower probabilities causing a number of bit errors under the threshold. This shows that there is a certain range of probabilities for which a binary symmetric channel (BSC) displays close to no errors, but when a certain threshold is reached the efficiency sharply decreases.

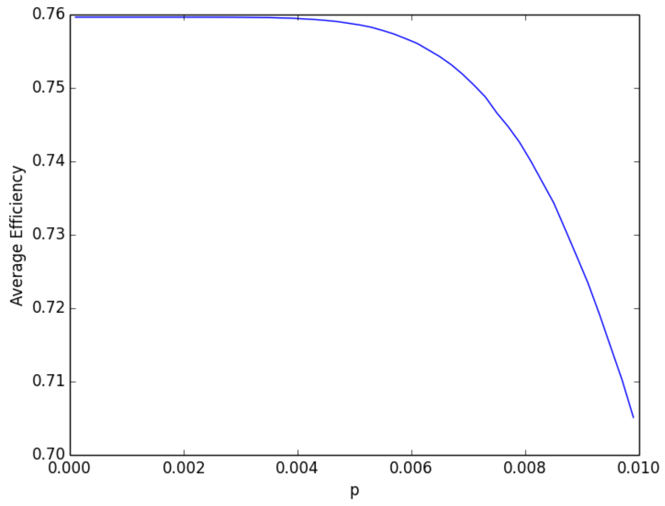

Figure 2: Average efficiency for p in the range $[0.001, 0.01]$ and fixed $u = 512$, where $n - k = \lceil 0.1 * (o + u) \rceil$

Question 5

Figure 3. shows the average efficiency of both a BSC channel and a two-state channel. Notably both these channels exhibit the same long term behaviour, at $n - k = 60$ both the curves overlap and continue to do so for all $n - k$ larger than that. This shows the bit error rate for the two-state channel averages out to 10^{-3} which is identical to that of the BSC channel, so for a significantly large number of redundant bits a BSC and a two-state channel will behave identically (if $p_g \approx p_{bsc}$ and $p_g << p_b$). So the long term behaviour of a two-state channel is determined by p_b in the case where $p_g << p_b$ as the number of retransmissions when the state is in a bad state will be much larger than when in a good state, and so the effect of the good state has a negligible effect on the average efficiency.

When $n-k$ is less than approx. 60 large jumps can be observed in the average efficiency. When $n - k = 0$ no bit errors can be corrected so the efficiency is low. Then when $n - k$ is around 20 the average efficiencies for both channels jumps up, this is due to the threshold being large enough to correct a large majority of the transmissions, this jump is slightly lower for the two-state due to $p_b < p_{bsc}$. And then both channels converge to the same error rate shortly after that.

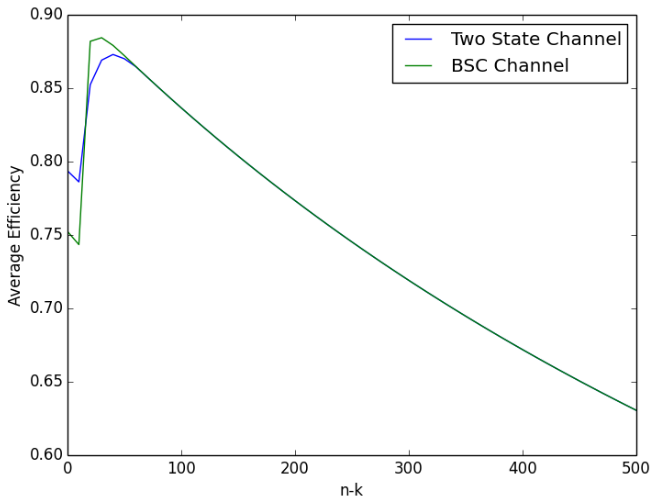

Figure 3: Average efficiency for varying $n - k$ in the range $[0, 500]$ and fixed $u = 1024$. For the BSC $p = 10^{-3}$. For the two-state channel $p_g = 10^{-5}$ and $p_b = 1.99 * 10^{-3}$, $p_{b,b} = p_{g,g} = 0.9$

Source Code

Note: Some errors may have occured when the code was appended to this file. See https://github.com/C-Kenny/264 for source files. To run simulations and plot the figures above run the plot.py file with the flag -all.

bsc_channel.py

```
'''
Simulates␣a␣BSC,␣modelled␣using␣a␣binomial␣distribution.
'''

import hamming
import distribution
from sys import argv
import numpy as np


def efficiency(data_size, num_transmissions, message_len):
    return data_size / (num_transmissions * message_len)
```

```
def average_efficiency(efficiencies):
    return sum(efficiencies) / len(efficiencies)


def simulate(num_simulations, error_rate, user_data, redundant_bits):
    efficiencies = []
    k = 100 + user_data
    message_len = k + redundant_bits
    total_submissions = 0
    num_transmissions = 0
    bound = hamming.hamming_bound(message_len, k)

    for _ in range(num_simulations):

        # Send one package
        num_errors = distribution.num_errors(message_len, error_rate)

        # While not successful
        num_transmissions = 1
        while num_errors > bound:
            num_errors = distribution.num_errors(message_len, error_rate)
            num_transmissions += 1
        eff = efficiency(user_data, num_transmissions, message_len)
        efficiencies.append(eff)

        total_submissions += num_transmissions

    return average_efficiency(efficiencies)


def prob_sim(prob_min, prob_max, prob_step, num_sims, user_data):
    '''
␣␣␣␣Perform␣multiple␣simulations.
␣␣␣␣'''
    avg_efficiencies = []
    error_rates = np.arange(prob_min, prob_max, prob_step)

    # Calculate redundant bits
    redundant_bits = int(np.ceil((user_data + 100) * 0.1))

    for error in error_rates:
        avg_efficiencies.append(simulate(num_sims, error,
                                         user_data, redundant_bits))


#     print(len(avg_efficiencies), len(error_rates))
```

```
    return error_rates, avg_efficiencies



if __name__ == "__main__":
    '''
    Command line parameters:
        -u data size u
        -p bit error probability
        -r number of redundant bits n-k
    '''
    overhead = 100
    num_simulations = 10 ** 6

    # Get simulation Parameters
    if len(argv) > 1:
        user_data = int(argv[argv.index("-u") + 1])
        error_rate = float(argv[argv.index("-p") + 1])
        redundant_bits = int(argv[argv.index("-r") + 1])

    else:   # Interactive Mode
        user_data = int(input("Data size (int): "))
        error_rate = float(input("Independent bit probability error (float): "))
        redundant_bits = int(input("Redundant bit size (int):"))

    message_len = user_data + overhead + redundant_bits

    average_efficiency = simulate(num_simulations, error_rate,
                                  user_data, message_len)

    print(average_efficiency)
```

two_state_simulation.py

```
'''
Simulates a 2-State channel, modelled using a binomial distribution.
'''

import hamming
import numpy    # for uniform random numbers
import distribution
from sys import argv


def efficiency(data_size, num_transmissions, message_len):
    return data_size / (num_transmissions * message_len)
```

```
def average_efficiency(efficiencies):
    return sum(efficiencies) / len(efficiencies)


def get_state(previous_state_good):

    new_state = None

    p_gg = 0.9
    p_bb = 0.9

    # Gen random number
    random_number = numpy.random.uniform()  # q


    if previous_state_good:
        if random_number >= p_gg:
            new_state = False

    elif not previous_state_good:
        if random_number >= p_bb:
            new_state = True

    if new_state is not None:
        return new_state
    else:
        return previous_state_good


def get_error_rate(current_state_good, error_rate_g, error_rate_b):

    if current_state_good:
        error_rate = error_rate_g
    else:
        error_rate = error_rate_b

    return error_rate


def simulate(num_simulations, error_rate_g, error_rate_b,
             user_data, redundant_bits):

    k = user_data + 100
    message_len = k + redundant_bits
```

```
efficiencies = []
total_submissions = 0

previous_state_good = False  # arbitarily defined, n-1 state
error_rate = error_rate_g
bound = hamming.hamming_bound(message_len, k)

run = 1
runs = []

for _ in range(num_simulations):

    # Send initial packet
    num_errors = distribution.num_errors(message_len, error_rate)

    # Update channel state
    current_state_good = get_state(previous_state_good)
    error_rate = get_error_rate(current_state_good,
                                error_rate_g, error_rate_b)
    if current_state_good == previous_state_good:
        run += 1
    else:
        runs.append(run)
        run = 1

    previous_state_good = current_state_good

    # Resend until all errors can be corrected
    num_transmissions = 1
    while num_errors > bound:
        num_errors = distribution.num_errors(message_len, error_rate)
        num_transmissions += 1

        # Update channel state
        current_state_good = get_state(previous_state_good)
        error_rate = get_error_rate(current_state_good,
                                    error_rate_g, error_rate_b)

        if current_state_good == previous_state_good:
            run += 1
        else:
            runs.append(run)
            run = 1

        previous_state_good = current_state_good
```

```
            efficiencies.append(efficiency(user_data,
                                           num_transmissions, message_len))
            total_submissions += num_transmissions

    print(numpy.average(runs))
#     print(total_submissions)
    return numpy.average(efficiencies)


if __name__ == "__main__":
    '''
    Command line parameters:
        -u data size u
        -pg bit error probability (good)
        -pb bit error probability (bad)
        -r number of redundant bits n-k
    '''
    overhead = 100
    num_simulations = 10
    # Get simulation Parameters
    if len(argv) > 1:
        user_data = int(argv[argv.index("-u") + 1])
        error_rate_g = float(argv[argv.index("-pg") + 1])
        error_rate_b = float(argv[argv.index("-pg") + 1])
        redundant_bits = int(argv[argv.index("-r") + 1])

        print(average_efficiency)
```

hamming.py

```
'''
Calculate the hamming bound of a k bit code.
'''
from scipy.special import binom
from math import log

binom_sums = {}


def hamming_bound(n, k):
    # Calculate the hamming bound of a k bit code, that after encoding is
    # n bits long.

    exp = 2 ** (n - k)  # n - k, number of redundant bits
#     print("exp = ", exp)
    binom_sum = 0
    i = 0
    while binom_sum < exp:
        if (n, i) not in binom_sums:
            binom_sums[(n, i)] = binom(n, i)

        if binom_sum + binom_sums[(n, i)] <= exp:
            binom_sum += binom_sums[(n, i)]
        else:
            break

        i += 1

    return i

if __name__ == '__main__':
    n = 2310
    k = 2100
    print(hamming_bound(n,k))
```

distribution.py

```
'''
Generate␣a␣binomially␣distributed␣random␣number.␣For␣the␣use␣of␣modelling␣the
bit␣errors␣of␣a␣BSC.
'''

from numpy.random import binomial


def num_errors(message_len: int, error_rate: float):
    return binomial(message_len, error_rate)
```

plot.py

```
'''
Plotting␣utility␣for␣COSC264␣Assignment.
Plots␣p␣on␣the␣x-axis␣and␣average␣efficency␣on␣the␣y-axis.
'''

import matplotlib.pyplot as plt
from numpy import ceil
from sys import argv


import bsc_simulation
import two_state_simulation

def plot_simulation(x1, y1, dest_name, xlabel, ylabel, x2=None, y2=None,
                    line1_label=None, line2_label=None):

    # Label axis
    plt.ylabel(ylabel)
    plt.xlabel(xlabel)



    if x2 is None and y2 is None:
        # Draw graph
        line1, = plt.plot(x1, y1)

    else:
        # Draw graph
        line1, = plt.plot(x1, y1)
        line2, = plt.plot(x2, y2)

    if line1_label is not None and line2_label is not None:
        plt.legend([line1, line2], [line1_label, line2_label])

    plt.savefig(dest_name)
    plt.close()
#     plt.show()


def question_three():
    print("in q 3")
    # Q3 | BSC, where p = 0.001 and p = 0.01
    p_ten = 0.01
    p_ten_container = []
    p_hund = 0.001
    p_hund_container = []

    user_data = range(100, 2010, 10)

    for data in user_data:
        # number of checkbits is 10% of packetsize
        n = data + 100
        check_bits = ceil(.1 * n)
        k = n + check_bits
#         print(check_bits, data, n, k)
        p_ten_container.append(
            bsc_simulation.simulate(10**6, p_ten, data, check_bits))
    for data in user_data:
        print("plotting 100s")
        n = data + 100
        check_bits = ceil(.1 * n)
        k = n + check_bits
        p_hund_container.append(
            bsc_simulation.simulate(10**6, p_hund, data, check_bits))

#     print(p_hund_container)
    plot_simulation(user_data, p_ten_container,
                    dest_name="img/q3.png", xlabel="u",
                    ylabel="Average Efficiency",
                    line1_label="p=0.01", line2_label="p=0.001",
                    x2=user_data, y2=p_hund_container)


def question_four():
    # Q4 | Plot BSC with varying effs
    u = 512
    p_min = 0.0001
    p_max = 0.01
    p_step = 0.0002
    num_sims = 10 ** 6

    (p, efficiencies) = bsc_simulation.prob_sim(p_min, p_max, p_step,
                                                num_sims, u)
    print(p, efficiencies)
    print("Finished BSC Simulation")
    plot_simulation(p, efficiencies, dest_name="img/q4.png",
                    xlabel='p', ylabel='Average Efficiency')

def question_five():
    # Q5 | Two State Simulation
    pg = (10)**(-5)
    pb = (1.99)*(10**-3)
    user_data = 1024
    two_state_efficiencies = []
    bsc_efficiencies = []

    bit_range = range(0, 510, 10)
    for check_bits in bit_range:
        two_state_efficiencies.append(two_state_simulation.simulate(10**6,
                                      pg, pb, user_data, check_bits))
        bsc_efficiencies.append(bsc_simulation.simulate(10**6, 10**-3,
                                                        user_data, check_bits))
    plot_simulation(bit_range, two_state_efficiencies, x2=bit_range,
                    y2=bsc_efficiencies, xlabel="n-k",
                    ylabel="Average Efficiency",
                    line1_label="Two State Channel",
                    line2_label="BSC Channel", dest_name="img/q5.png")

if __name__ == "__main__":
    '''
    Command line parameters:
        -q3
        -q4
        -q5
        -all
    '''

    if len(argv) > 1:
        question = argv[1]

        if question == "-q3":
            question_three()

        elif question == "-q4":
            question_four()

        elif question == "-q5":
            question_five()

        elif question == "-all":
            question_three()
            question_four()
            question_five()
```