COSC264 – Assignment

Andrew Dallow – ID: 56999204, Contribution: 50%

James Priddy – ID: 52456474, Contribution: 50%

Question One

The probability distribution used to generate the number of errors *s* in an encoded packet of *n* bits in length was a binomial probability distribution. This was used because a binomial distribution can represent the number of errors in a sequence of *n* independent binary numbers, each with a probability *p* of being erroneous. The probability density function is:

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

In order to generate a random number with a binomial probability distribution in Python, the function numpy.random.binomial(n, p) from the package NumPy was used with the total packet size for n and the probability of an error as p. This function would then return the number of errors s.

A similar step is that of the Poisson distribution method. However, such a technique is unsuitable for these circumstances. The Poisson method does not work with a fixed success rate, and because we work in this instance with a fixed success rate specified by the user this method would not be suitable.

Question Two

When finding the formula for the probability of a packet with n bits having a probability *p* that has at least one error, we must use algebra and similar probability formulas to determine the solution:

$$P(k\ successes \in n\ trials) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$P(at\ least\ one\ error) = 1 - P(noerrors)$$

$$P(at\ least\ no\ errors) = \binom{n}{0} p^0 (1-p)^{n-0}$$

$$P(at\ least\ one\ error) = 1 - \binom{n}{0} p^0 (1-p)^{n-0}$$

$$P = 1 - (1-p)^n$$

Question Three

From Figure 1 we can determine that as *p*, the bit error rate, increases, the average efficiency becomes less consistent. From this graph we can determine that where *p* = 0.01, represented in Figure 1 by the red line, the curve is a consistently increasing saw tooth line where an increased *p* value increases the variability of the results. When analysing the curve of *p* = 0.001, represented in Figure 1 as the green line, we can note that the curve is very smooth and suggests reduced variability of the results, thus containing a more consistent calculation of the average efficiency.

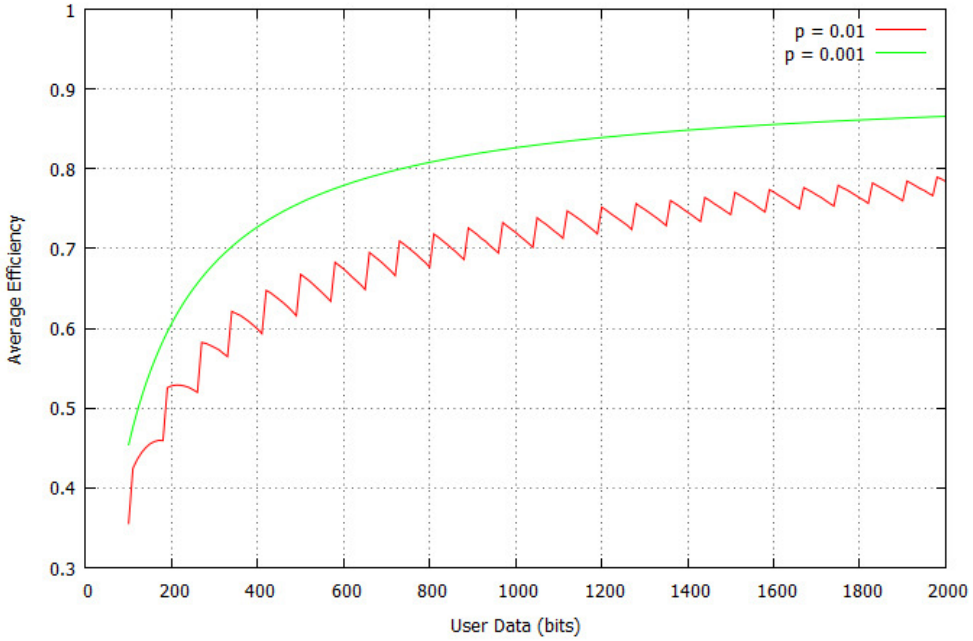

Figure 1 - The Efficiency of a BSC Channel

In both curves we can observe a sharp rise in the initial stage. However, in this particular model as the amount of user data, *u*, reaches the 1000 bits the growth of both curves slowly begins to flatten, trending towards the p value (Average efficiency – *p*), with a greatly reduced rate of increase.

Question Four

Figure 2 shows a plot of average efficiency over a range of bit error rates (*p*) for a constant amount of user data (512 bits) and constant number of redundant bits (51 bits). It can be seen that the average efficiency is constant at 0.76 over the bit error rates of 0.0001 – 0.005. After a bit error rate of 0.005 the average efficiency begins to decrease with increasing bit error rate.

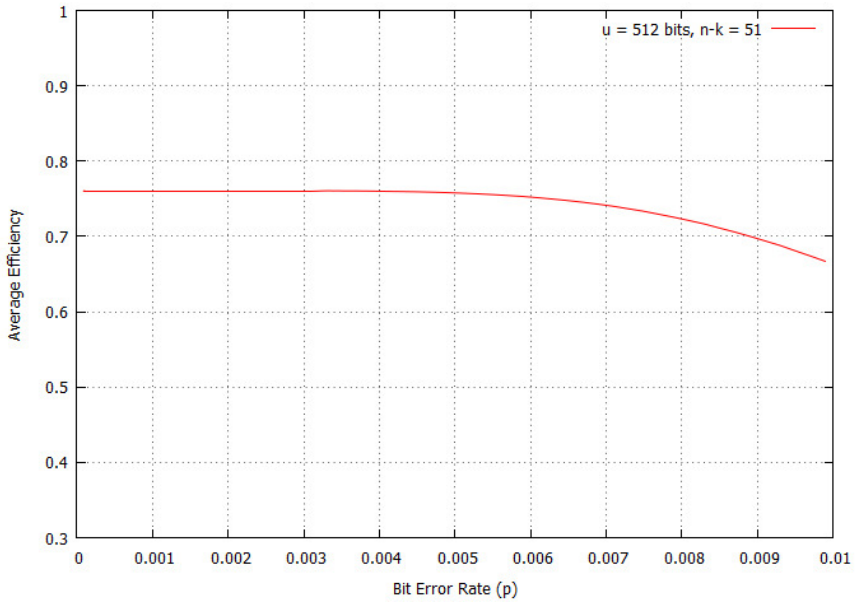

Figure 2 - The Efficiency of a BSC channel over a range of bit error rates, where u = 512 bits and n - k = 51 bits

It can be concluded that bit error rates below 0.005 do not produce enough errors to affect the average efficiency of the n = 100 + 512 + 51 = 663 bits. However, bit error rates above 0.005 do produce enough errors to gradually lower the average efficiency because the number of redundant bits is no longer sufficient to handle the number of errors. One can conclude for this particular packet that it works optimally in bit error rates less than 0.005.

Question Five

Figure 3 shows the average efficiency of a two-state BSC channel for 1024 bits over a range of the number of redundant bits. The one-state BSC channel was also plotted as a reference.

It can be seen that in the case of the one-state BSC channel the average efficiency starts at about 0.5 for 0 redundant bits and then rapidly increases to a peak of 0.87 at 40 redundant bits, after which it gradually decreases with increasing number of redundant bits.

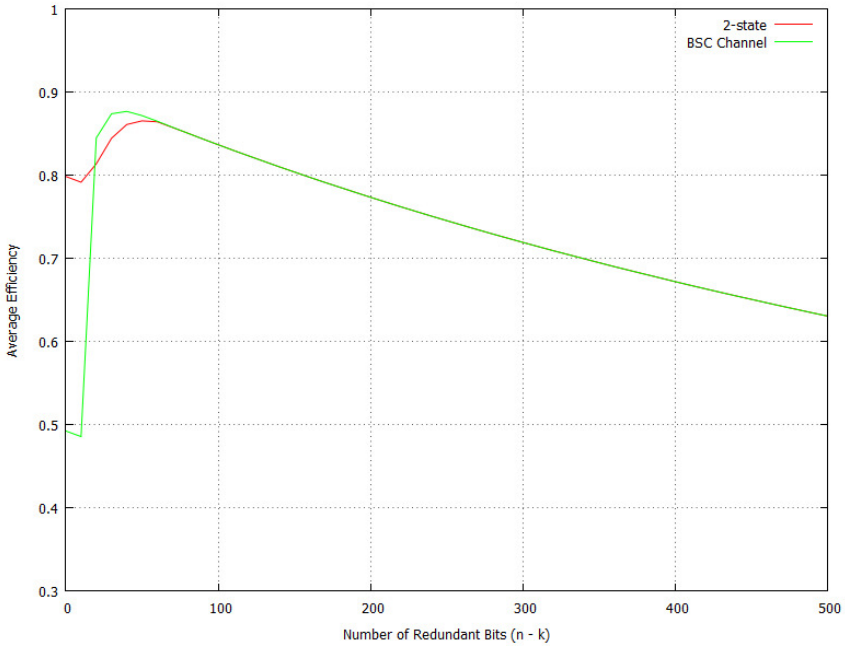

Figure 3 - Average Efficiency of a One-state and Two-state BSC Channel over a range of Redundant Bits, u = 1024 bits $p_b = 1.99 \times 10^{-3}, p_g = 10^{-5}$.

In the case of the two-state BSC channel we see a similar behaviour to the single state, except that it starts with a higher average efficiency of 0.8 at 0 redundant bits, rapidly increases to a peak of 0.86 at 50 redundant bits, after which it gradually decreases with increasing number of redundant bits in the same fashion as the single-state BSC channel.

Overall one can conclude that for both the one and two state BSC channels there exists an optimal number of redundant bits for a user data amount of 1024 bits. Therefore, having too little number of redundant bits results in not having enough bits to correct the random errors in the BSC channel, thus giving a very low average efficiency. However, having too many redundant bits results in sending more data than is needed to correct the random errors, thus also lowering the average efficiency.

The one-state BSC channel has a peak efficiency at 40 redundant bits, while the two-state BSC channel has a peak efficiency at 50 bits. This is because the bursty style transmissions encountered in the two-state BSC channel result in a larger number of retransmissions when the state randomly changes to a ‘bad’ state, thus a larger number of redundant bits are needed for the same amount of user data to get an optimal average efficiency.

BSC Channel Source Code (Python)

```
"""
   Program to simulate a Binary Symmetric Channel (BSC) and calculate the average
   efficiency given the user_data size (u > 0), the number of redundant bits
   (n-k > = 0) and the probability (0 < p < 1) for N = 1000000 packets sent by a
   transmitter.

   Authors:
        Andrew Dallow
        James Priddy
"""

from math import factorial
from numpy import random

def prompt_user():
    """Prompt the user for user data (u), the number of redundant bits (n-k)
    and the error probability (p) and return these values."""

    user_data = int(input("Enter the amount of User Data (u): "))
    while (user_data <= 0):
        print("Please Enter an amount greater than 0.")
        user_data = int(input("Enter the amount of User Data (u): "))

    redun_bits = int(input("Enter the number of redundant bits (n - k): "))
    while (redun_bits < 0):
        print("Redundant bits must be greater than or equal to 0.")
        redun_bits = int(input("Enter the number of redundant bits (n - k): "))

    error_prob = float(input("Enter the bit error rate (p): "))
    while (error_prob < 0 or error_prob > 1):
        print("Probability must be between 0 and 1.")
        error_prob = float(input("Enter the bit error rate (p): "))

    return user_data, redun_bits, error_prob


def get_Coefficient(num_bits, i):
    """Return the Binomial Coefficient given num_bits and i"""
    return factorial(num_bits) / (factorial(i) * factorial(num_bits - i))


def get_max_t(tot_data, num_redun_bits):
    """Return the error correction capability (t*), the maximum of wrong bits
    that can be corrected using the Hamming Bound"""

    ham_sum = 0
    num_errors = 0
    packet_size = num_redun_bits + tot_data
    ham_bound = 2**(num_redun_bits)

    # The Hamming Bound is achieved provided the relationship is fulfilled
    while (ham_sum <= ham_bound):
        ham_sum += get_Coefficient(packet_size, num_errors)
        num_errors += 1

    return num_errors - 2
```

```
def getEfficiency(user_data, packet_size, tStar, prob):
    """Calculate a random number of errors in a given packet via a binomial
    probability distrubultion and repeat until the number of errors is below
    the given threshold, then return the transmission efficiency"""
    num_transmissions = 1

    num_errors = random.binomial(packet_size, prob)
    while (num_errors > tStar):
        num_errors = random.binomial(packet_size, prob)
        num_transmissions += 1

    efficiency = user_data / (num_transmissions * packet_size)

    return efficiency


def run_sim(user_data, tStar, packet_size, prob):
    """Run a simulation with 1 mil trials and return the mean efficiency,
    given user_data (u), tStar (t*), packet_size (n), and bit error rate (prob)
    """

    transmit_eff = []
    num_trials = 1000000

    # Calculate the average efficiency over 1 million packets.
    for i in range(num_trials):
        efficiency = getEfficiency(user_data, packet_size, tStar, prob)
        transmit_eff.append(efficiency)

    mean_eff = sum(transmit_eff) / num_trials
    return mean_eff


def main():
    """Run the Program"""
    overhead = 100
    # Prompt the User
    user_data, num_redun_bits, error_prob = prompt_user()

    print("Running simulation for:")
    print("  User Data (u):", user_data)
    print("  Redundant Bits (n-k):", num_redun_bits)
    print("  Probability (p):", error_prob)
    # Calculate t*, the error correction capability
    packet_size = overhead + user_data + num_redun_bits
    tot_data = overhead + user_data
    error_threshold = get_max_t(tot_data, num_redun_bits)

    print("  Error Correction Capability: ", error_threshold)

    avg_eff = run_sim(user_data, error_threshold, packet_size, error_prob)
    print("Simulation Complete.")
    print("  The average effeciency of all packets is: {0:.4}".format(avg_eff))

main()
```

Two-State BSC Channel Source Code (Python)

```
"""
   Program to simulate a Binary Symmetric Channel (BSC) with two states.
   The channel can be in one of two states, "good" (pg) or "bad" (pb).
   Regardless of the state, the channel acts like a BSC, but the state changes
   the error bit rate of which the BSC operates. The channel switches its state
   in a random fashion.

   Authors:
        Andrew Dallow
        James Priddy
"""

from math import factorial
from numpy import random
state = 0

def prompt_user():
    """Prompt the user for user data (u), the number of redundant bits (n-k)
    and the error probability (p) and return these values."""

    user_data = int(input("Enter the amount of User Data (u): "))
    while (user_data <= 0):
        print("Please Enter an amount greater than 0.")
        user_data = int(input("Enter the amount of User Data (u): "))

    redun_bits = int(input("Enter the number of redundant bits (n - k): "))
    while (redun_bits < 0):
        print("Redundant bits must be greater than or equal to 0.")
        redun_bits = int(input("Enter the number of redundant bits (n - k): "))

    pg = float(input("Enter the 'Good' state bit error rate (pg): "))
    while (pg < 0 and pg > 1):
        print("Probability must be between 0 and 1.")
        pg = float(input("Enter the 'Good' state bit error rate (pg): "))

    pb = float(input("Enter the 'Bad' state bit error rate (pb): "))
    while (pb < 0 and pb > 1):
        print("Probability must be between 0 and 1.")
        pg = float(input("Enter the 'Bad' state bit error rate (pb): "))

    return user_data, redun_bits, pg, pb


def get_Coefficient(num_bits, i):
    """Return the Binomial Coefficient given num_bits and i"""

    return factorial(num_bits) / (factorial(i) * factorial(num_bits - i))


def get_max_t(tot_data, num_redun_bits):
    """Return the error correction capability (t*), the maximum of wrong bits
    that can be corrected using the Hamming Bound"""

    ham_sum = 0
    num_errors = 0
    packet_size = num_redun_bits + tot_data
    ham_bound = 2**(num_redun_bits)
```

```
    # The Hamming Bound is achieved provided the relationship is fulfilled
    while (ham_sum <= ham_bound):
        ham_sum += get_Coefficient(packet_size, num_errors)
        num_errors += 1

    return num_errors - 2


def get_num_errors(packet_size, pg, pb, pgg, pbb):
    """Return a random number of errors from a binomial probability distrubution
    based on the current state and last state of the channel which determine
    whether to use pg in a "Good" state or pb in a "Bad" state"""

    global state
    q = random.uniform()
    if state == 0: #If channel state is "Good"
        prob = pg
        if q >= pgg:
            state = 1
    elif state == 1: #If channel state is "Bad"
        prob = pb
        if q >= pbb:
            state = 0
    return random.binomial(packet_size, prob, 1)


def getEfficiency(user_data, packet_size, tStar, prob):
    """Calculate a random number of errors in a given packet via a binomial
    probability distribution and repeat until the number of errors is below
    the given threshold, then return the transmission efficiency"""

    num_transmissions = 1

    pg = prob[0]
    pb = prob[1]
    pgg = 0.9
    pbb = 0.9

    num_errors = get_num_errors(packet_size, pg, pb, pgg, pbb)
    while (num_errors > tStar):
        num_errors = get_num_errors(packet_size, pg, pb, pgg, pbb)
        num_transmissions += 1

    efficiency = user_data / (num_transmissions * packet_size)

    return efficiency


def run_sim(user_data, tStar, packet_size, prob):
    """Run a simulation with 1 million packets and return the average
    efficiency"""

    transmit_eff = []
    num_trials = 1000000

    # Calculate the average efficiency over 1 million packets.
    for i in range(num_trials):
        transmit_eff.append(getEfficiency(user_data, packet_size, tStar, prob))

    avg_efficiency = sum(transmit_eff) / num_trials

    return avg_efficiency
```

```
def main():
    """Run the Program"""

    overhead = 100
    # Prompt the User
    user_data, redun_bits, pg, pb = prompt_user()

    print("Running simulation for:")
    print("  User Data (u):", user_data)
    print("  Redundant Bits (n-k):", redun_bits)
    print("  Probabilities: pg = {}, pb = {}".format(pg, pb))

    packet_size = overhead + user_data + redun_bits
    tot_data = overhead + user_data

    # Calculate t*, the error correction capability
    tStar = get_max_t(tot_data, redun_bits)
    print("  Error Correction Capability: ", tStar)

    # Calculate the average efficiency
    avg_eff = run_sim(user_data, tStar, packet_size, [pg, pb])

    print("Simulation Complete.")
    print("  The average efficiency of all packets is: {0:.4}".format(avg_eff))

main()
```