# Information Theory Assignment

s1140740

## 1 Source Coding

### 1.1 Character Statistics

The entropy $H(X_n)$ is 4.168.

### 1.2 Bigram Statistics

We compute the probability of each two consecutive characters in the file. This results in $P(x_n, x_{n+1})$.

a. The joint entropy $H(X_n, X_{n+1})$ is 7.572.
b. Because $P(x_n, x_{n+1}) \neq P(x_n)P(x_{n+1})$. This means that two characters appearing together are not independent. In English, some characters appear together more than others. Thus, on average we expect to be surprised less when we see a letter appear with another letter than seeing those two letters appear independently.
c. The conditional entropy, computed as $H(X_{n+1} \mid X_n) = H(X_n, X_{n+1}) - H(X_n)$, is 3.404.

### 1.3 Compression with known distributions

We know that with arithmetic coding the message length is always within two bits of the Shannon information content and thus the maximum number of bits is computed as:

$$\lceil h(x) \rceil + 2 = \left\lceil -\sum_{n=1}^{N} \log_2 P(x_n) \right\rceil + 2$$

where $x_n$ is the $n$-th character in the file and $N$ is the length of the file. Probabilities from question 1 are used.

The maximum number of bits an arithmetic coder would use to represent `thesis.txt` assuming this model is therefore 1,433,836.

---

The maximum number of bits an arithmetic coder would use to represent `thesis.txt` assuming the more sophisticated model is 1,171,194. The length is computed as:

$$\left\lceil -\log_2 P(x_1) - \sum_{n=1}^{N} \log_2 P(x_{n+1}|x_n) \right\rceil + 2 = \left\lceil -\log_2 P(x_1) - \sum_{n=1}^{N} \log_2 \frac{P(x_{n+1}, x_n)}{P(x_n)} \right\rceil + 2$$

The joint probabilities from question 2 are combined together with probabilities from question 1 to compute the conditional probabilities which are used in chain rule.

### 1.4 Compression with limited precision header

Assuming we transmit the powers of 2 required to get back the original probabilities, we require 8 bits per character. Since our alphabet is of size 27 (`'a' - 'z'` and `' '`) we require $27 * 8 = 216$ bits for the header if we send the powers ordered alphabetically with space at the end. Thus the decoder knows where to locate the power of every probability in the header.

Furthermore, the probabilities are re-normalised before use (by both sender and receiver) to add up to 1 and then applied in the same fashion as in the previous question to compute the maximum number of bits required. Thus the maximum number of bits required to encode `thesis.txt` is 1,435,081. Hence, together with the header we require 1,435,297 bits with this scheme.

---

There are two ways to approach this problem. In one solution we could transmit information about the i.i.d. distribution and the joint distribution in the header. These distributions would then be combined to calculate conditional probabilities in the chain rule. Another solution would be to to transmit information about the i.i.d. distribution and the conditional probabilities directly. I chose to implement the second solution as it yielded better results.

Both solutions would require us to put $27 * 8 = 216$ bits about the i.i.d. distribution together with $27 * 27 * 8 = 5,832$ bits about the conditional or joint distribution in the header. Thus altogether we would require 6,048 bits for the header, assuming a 27 letter alphabet and 8 bits per character.

As for the body of the message, we need to re-normalise the conditional probabilities so that:

$$\sum_i P(x_{n+1} = a_i | x_n = a_j) = 1$$

after which the Shannon information content and hence the file size are computed in the same way as in the previous question. Hence, the maximum number of bits required to encode `thesis.txt` is 1,178,320. Thus, together with the header we would require 1,184,368 bits with this scheme.

### 1.5 Compression with adaptation

Using the Laplace prediction rule for the i.i.d. model the maximum number of bits required to encode `thesis.txt` is 1,434,027.

The rule is computed for every character of the file and these probabilities are then used to compute the Shannon information content. We then take the ceiling and add 2 as in the previous question.

---

Using the prediction rule for the bigram model the maximum number of bits required to encode `thesis.txt` is 1,175,382.

Again the rule is computed for every character of the file and the probabilities are then used to compute the Shannon information content. Again, we take the ceiling and add 2 as before.

## 2 Noisy Channel Coding

### 2.1 XOR-ing packets

The resulting string is `User: s5559183948`.

### 2.2 Decoding packets from a digital fountain

The algorithm takes as its input a list of packets and a list of sets containing identities of the source bytes that were used for each of the packets.

It loops through these lists zipped and only breaks out if all sets of identities are empty. Every time it encounters a set in position $i$ with only one identity number $k$ it pops $k$ from the set and saves the character represented by the $i$-th packet in ASCII in the $k$-th position of the result list.

After this, $k$ is removed from all the other sets that contain it. Furthermore, if any of these sets, say in position $j$, contains other numbers as well then the $i$-th packet is XOR'd with the packet in $j$-th position.

The source string is then re-constructed from the result list. The packets used to construct the string are also recorded and returned with the string.

The decoded string is `Password: X!3baA1z` and packets used are as follows: `16`, `23`, `2`, `21`, `22`, `20`, `8`, `10`, `17`, `19`, `6`, `7`, `9`, `11`, `14`, `15`, `12`, `13`.

## 2.3 Creating a code

I re-implemented the 2-dimensional parity-check code which is an $(N, K)$ block code. Even though my implementation is generic for any $N$ and $K$, in this report I am going to discuss the performance of an (8, 4) parity-check code. An $n$-dimensional parity-check code can correct $n/2$ errors. Thus this code can only correct bit sequences with 1 error.

### 2.3.1 Encoding

In the encoding step 4 parity check bits are added to the source sequence $x_1 \dots x_4$, computed as follows:

$$\begin{aligned} y_1 &= x_1 \oplus x_2 & z_1 &= x_1 \oplus x_3 \\ y_2 &= x_3 \oplus x_4 & z_2 &= x_2 \oplus x_4 \end{aligned} \tag{1}$$

Following is the message that is then transmitted over the channel:

$$x_1\ x_2\ y_1\ x_3\ x_4\ y_2\ z_1\ z_2$$

### 2.3.2 Decoding

The transmitted message can be represented as:

| | | |
|---|---|---|
| $x_1$ | $x_2$ | $y_1$ |
| $x_3$ | $x_4$ | $y_2$ |
| $z_1$ | $z_2$ | |

where every $y_i$ represents the result of applying XOR to the rest of the bits in row $i$ and every $z_j$ represents the result of applying XOR to the rest of the bits in column $j$.

Upon receiving the transmitted message

$$x'_1\ x'_2\ y'_1\ x'_3\ x'_4\ y'_2\ z'_1\ z'_2$$

$y_1, y_2, z_1$ and $z_2$ are re-computed as in equations 1 and compared to the corresponding values received. If $y_i \neq y'_i$ for exactly one $i$ and $z_j \neq z'_j$ for exactly one $j$ then there was an error in the body of the message in row $i$ and column $j$ and we can correct it.

In any other case, there is nothing we can do, since we can only correct one error and we do not care about single errors of the parity-check bits.

### 2.3.3 Rate

The rate under this code is:

$$R = \frac{\log_2 S}{N} = \frac{K}{N} = \frac{4}{8} = \frac{1}{2}$$

where S is the number of possible codewords. This is $2^4$ as we can vary the 4 source bits and parity check bits are determined from the source bits.

### 2.3.4 Bit error probability

I arrived at the following bit error probabilities for given values of $f$:

| $f$ | $p_B$ |
|---|---|
| 0.4 | 0.406528 |
| 0.1 | 0.059248 |
| 0.001 | 0.007976 |

The numbers were computed by generating a random 4-bit sequence and then generating all the possible noises and calculating the probability of a bit being flipped averaged over all bits. This can be seen in `real_bit_error` in `test.py`.

The numbers were also confirmed analytically, running 10,000 tests for 0.4 and 0.1 and 100,000 tests for 0.001. This can be seen in `analytic_bit_error` in `test.py`.

### 2.3.5 Example

As an example, consider source sequence 0100. We compute $y_1, y_2, z_1$ and $z_2$:

| | | |
|---|---|---|
| 0 | 1 | 1 |
| 0 | 0 | 0 |
| 0 | 1 | |

and hence transmit message 01100001.

Let us say that when the decoder receives the message bit in row 2 and column 1 is flipped.

| | | |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | |

Now, the decoder re-computes $y_1, y_2, z_1$ and $z_4$:

$$\begin{aligned} y_1 &= 1 = 1 = y_1' & z_1 &= 1 \neq 0 = z_1' \\ y_2 &= 1 \neq 0 = y_2' & z_2 &= 1 = 1 = z_2' \end{aligned} \tag{2}$$

and it can detect and correct the error as it now knows the error is in row 2 and column 1. Therefore it produces the original message 0100.

# 3 Code

## 3.1 count.py

Contains code for first 7 tasks.

```
import re
import math

from itertools import tee, chain
from collections import Counter, defaultdict


FILE = 'thesis.txt'
ALPHABET_SIZE = 27


def entropy(dist):
    """Computes the entropy of a distribution."""
    return sum(-prob * math.log(prob, 2) for prob in dist.values())


def information_content(dist):
    """Computes the information content of a distribution."""
    return sum(-math.log(prob, 2) for prob in dist.values())


def unigen(file_):
    """Takes a file and turns it into a character generator, ignoring \n."""
    prog = re.compile(r'[a-z ]')
    return (c.lower() for c in chain(*file_) if prog.match(c))


def bigen(file_):
    """Bigram generator from a file."""
    file1, file2 = [unigen(file_gen) for file_gen in tee(file_)]
    next(file2)  # drop first char
    return zip(file1, file2)


def unigram(file_):
    """Computes the bigram distribution of a file."""
    counter = Counter(unigen(file_))
    chars = sum(counter.values())
    return {key: (val / float(chars)) for key, val in counter.items()}


def bigram(file_):
    """Computes the bigram distribution of a file."""
    counter = Counter(bigen(file_))
    chars = sum(counter.values())
    return {key: (val / float(chars)) for key, val in counter.items()}


def iid_length(file_, dist):
    """Length of a file if its chars were i.i.d."""
    probs = (dist[c] for c in unigen(file_))
    file_len = -sum(math.log(prob, 2) for prob in probs)
    return int(math.ceil(file_len + 2))


def bi_length(file_, uni_dist, bi_dist):
    """Length of a file if its chars were bigramy."""
    chars = list(bigen(file_))
    char1 = chars[0][0]
    probs = [bi_dist[cs] / uni_dist[cs[0]] for cs in chars]
    probs.append(uni_dist[char1])
    file_len = -sum(math.log(prob, 2) for prob in probs)
    return int(math.ceil(file_len + 2))
```

```
def cond_dist(uni_dist, bi_dist):
    """Creates conditional distribution from i.i.d. and joint dists."""
    return {(c0, c1): bi_dist[c0, c1] / uni_dist[c0] for (c0, c1) in bi_dist}


def norm_dist(dist):
    """Normalise a distribution."""
    round_sum = sum(dist.values())
    return {k: p / round_sum for k, p in dist.items()}


def round_dist(dist, bits=8):
    """Round a distribution."""
    pow2 = pow(2, bits)
    return {k: math.ceil(pow2 * p) / pow2 for k, p in dist.items()}


def norm_cond_dist(dist):
    """Normalise a conditional distribution."""
    # turn into temporary form
    d = defaultdict(Counter)
    for (c0, c1), prob in dist.items():
        d[c0][c1] = prob

    for c0 in d:
        total = 0.
        # sum up conditional probs
        for c1, prob in d[c0].items():
            total += prob
        # normalise
        for c1 in d[c0]:
            d[c0][c1] /= total

    # turn back to original form
    v = Counter()
    for c0 in d:
        for c1 in d[c0]:
            v[c0, c1] = d[c0][c1]
    return v


def iid_round_length(file_, dist, bits=8):
    """How long the file would be if we used a rounding scheme."""
    rounded_dist = norm_dist(round_dist(dist))
    # each char needs bits in header
    header = ALPHABET_SIZE * bits
    data = iid_length(file_, rounded_dist)
    return {'header': header, 'data': data, 'total': header + data}


def bi_round_length(file_, uni_dist, bi_dist, bits=8):
    """How long the file would be if we used a rounding scheme."""
    uni_rounded_dist = norm_dist(round_dist(uni_dist))
    conditional_dist = cond_dist(uni_dist, bi_dist)
    cond_rounded_dist = round_dist(conditional_dist)
    cond_rounded_dist = norm_cond_dist(cond_rounded_dist)
    chars = list(bigen(file_))
    char1 = chars[0][0]
    probs = [cond_rounded_dist[cs] for cs in chars]
    probs.append(uni_rounded_dist[char1])
    file_len = -sum(math.log(prob, 2) for prob in probs)
    data = int(math.ceil(file_len + 2))
    # each pair of chars needs bits in header
    header = (1 + ALPHABET_SIZE) * ALPHABET_SIZE * bits
    return {'header': header, 'data': data, 'total': header + data}


def iid_adapt_length(f):
    """Compression with adaptation using Laplace prediction rule."""
```

```
    count = Counter()
    file_len = 0.0
    for n, char in enumerate(unigen(f), start=0):
        k_i = count.get(char, 0)
        prob = (k_i + 1.0) / (n + 27)
        file_len -= math.log(prob, 2)
        count[char] += 1

    return int(math.ceil(file_len + 2))


def bigram_adapt_length(f):
    """Compression with adaptation using prediction rule for bigram model."""
    k = Counter()
    n = Counter()
    file_len = 0.0
    prev = None
    for char in unigen(f):
        k_ij = k.get((prev, char), 0)
        n_j = n.get(prev, 0)
        prob = (k_ij + 1.0) / (n_j + 27)
        file_len -= math.log(prob, 2)
        k[(prev, char)] += 1
        n[prev] += 1
        prev = char

    return int(math.ceil(file_len + 2))


def nutritious_snacks(chars, nums):
    """XORs nutritious snacks with some numbers."""
    return ''.join(chr(ord(char) ^ num) for char, num in zip(chars, nums))


def digital_fountain(pkts, recvd):
    """Performs the digital fountain algorithm on packets."""
    rec_len = len(recvd)
    result = [None] * rec_len
    used_pkts = []
    while 1:
        empty_count = 0
        for idx, (rec, packet) in enumerate(zip(recvd, pkts), start=1):
            if len(packet) == 0:
                empty_count += 1
            elif len(packet) == 1:
                # we know what the char is
                num = packet.pop()
                result[num - 1] = chr(rec)  # -1 b/c zero indexing
                used_pkts.append(idx)
                for j, pkt in enumerate(pkts):
                    if num in pkt:
                        if len(pkt) == 1:
                            # packet redundant
                            pkt.discard(num)
                        elif len(pkt) > 1:
                            # XOR the char we know
                            pkt.discard(num)
                            recvd[j] ^= rec

        if empty_count == rec_len:
            break  # if all empty -> finish
    result = ''.join(res for res in result if res is not None)
    return result, used_pkts


def main():
    """Do the thing."""
    with open(FILE) as f:
        uni_dist = unigram(f)
        ent_x = entropy(uni_dist)
```

```
        print('H(X_n): {0:.4}'.format(ent_x))

        f.seek(0)
        bi_dist = bigram(f)
        ent_joint = entropy(bi_dist)
        print('H(X_n, X_n+1): {0:.4}'.format(ent_joint))
        ent_cond = ent_joint - ent_x
        print('H(X_n+1 | X_n): {:.4}'.format(ent_cond))

        f.seek(0)
        iid_len = iid_length(f, uni_dist)
        print('i.i.d. length: {}'.format(iid_len))

        f.seek(0)
        bi_len = bi_length(f, uni_dist, bi_dist)
        print('bigram length: {}'.format(bi_len))

        f.seek(0)
        iid_round_len = iid_round_length(f, uni_dist)
        print("""i.i.d. length rounded:
\theader: {header}
\tdata: {data}
\ttotal: {total}""".format(**iid_round_len))

        f.seek(0)
        bi_round_len = bi_round_length(f, uni_dist, bi_dist)
        print("""bigram length rounded:
\theader: {header}
\tdata: {data}
\ttotal: {total}""".format(**bi_round_len))

        f.seek(0)
        iid_adapt_len = iid_adapt_length(f)
        print('i.i.d. adaptation: {}'.format(iid_adapt_len))

        f.seek(0)
        bi_adapt_len = bigram_adapt_length(f)
        print('bigram adaptation: {}'.format(bi_adapt_len))

    snacks_xor = nutritious_snacks(
        'nutritious snacks',
        [59, 6, 17, 0, 83, 84, 26, 90, 64, 70, 25, 66, 86, 82, 90, 95, 75]
    )
    print('nutritious_snacks XOR: {}'.format(snacks_xor))

    with open('packets.txt') as pkts, open('received.txt') as recvd:
        pkts = [set(map(int, line.split())) for line in pkts]
        recvd = [int(line) for line in recvd]
        source_str, used_pkts = digital_fountain(pkts, recvd)
        print('Source string: {}'.format(source_str))
        print('Packets used: {}'.format(' '.join(map(str, used_pkts))))


if __name__ == '__main__':
    main()
```

## 3.2 code.py

Contains code for the encoder and decoder of my code.

```
import sys
import optparse
import numpy as np

K = 2


def encode(stream):
    """Encode the stream."""
```

```
    block = []
    arr = np.zeros((K + 1, K + 1), dtype=np.uint8)
    while 1:
        num = stream.read(1)
        if num in ['\n', '']:
            break
        num = int(num)
        block.append(num)
        if len(block) == K * K:
            # apply algoritm
            arr[:K, :K] = np.reshape(block, (K, K))
            arr[0:K, K] = np.bitwise_xor.reduce(arr[:K, :K], axis=1)
            arr[K, 0:K] = np.bitwise_xor.reduce(arr[:K, :K], axis=0)
            for out in arr.ravel()[:-1]:
                yield str(out)  # skip last char
            block = []


def decode(stream):
    """Decode the stream"""
    block = []
    while 1:
        num = stream.read(1)
        if num in ['\n', '']:
            break
        num = int(num)
        block.append(num)
        N = K * (K + 2)
        if len(block) == N:
            arr = np.resize(block, N + 1).reshape((K + 1, K + 1))
            # XOR columns
            col_xors = np.bitwise_xor.reduce(arr[:K, :K + 1], axis=1)
            nonzero_cols = col_xors.nonzero()[0]
            # XOR rows
            row_xors = np.bitwise_xor.reduce(arr[:K + 1, :K], axis=0)
            nonzero_rows = row_xors.nonzero()[0]
            if nonzero_cols.shape == nonzero_rows.shape == (0,):
                # no errors or undetected errors
                pass
            elif nonzero_cols.shape == (0,) and nonzero_rows.shape == (1,):
                # assuming parity bit error
                pass
            elif nonzero_rows.shape == (0,) and nonzero_cols.shape == (1,):
                # assuming parity bit error
                pass
            elif nonzero_cols.shape == nonzero_rows.shape == (1,):
                # flip that bit
                row = nonzero_cols[0]
                col = nonzero_rows[0]
                arr[row, col] = 1 - arr[row, col]
            else:
                # more than 1 errors
                pass

            for out in np.ravel(arr[:K, :K]):
                yield str(out)  # skip last char

            block = []


if __name__ == '__main__':
    parser = optparse.OptionParser()
    parser.add_option(
        '-e', '--encode',
        action='store_true', dest='encode', default=True,
        help='Encode the input stream.'
    )
    parser.add_option(
        '-d', '--decode',
        action='store_true', dest='decode', default=False,
```

```
        help='Decode the input stream.'
    )
    opts = parser.parse_args()[0]
    func = decode if opts.decode else encode

    for enc in func(sys.stdin):
        sys.stdout.write(enc)
```

## 3.3 flipper.py

Contains a function that adds noise to a sequence of bits.

```
import random
import sys


def noise(stream, f=0.4):
    """Adds noise to a stream with flip probability f."""
    while 1:
        num = stream.read(1)
        if num in ['\n', '']:
            break
        num = int(num)
        if random.random() < f:
            yield str(1 - num)
        else:
            yield str(num)


if __name__ == '__main__':
    for char in noise(sys.stdin, f=0.4):
        sys.stdout.write(char)
```

## 3.4 test.py

Unit tests for my code.

```
import unittest

from itertools import chain, combinations
from count import *
from flipper import *
from code import *


def generate_num(n):
    """Generates a random n-bit binary number."""
    pow2 = pow(2, n)
    return bin(int(round(random.random() * pow2))).lstrip('0b').zfill(n)


class MockStdIn(object):
    """Mock of stdin to test Encoder and Decoder."""

    def __init__(self, string):
        self.string = string + '\n'

    def read(self, n):
        ret = self.string[0]
        self.string = self.string[1:]
        return ret


class TestSourceCoding(unittest.TestCase):

    """Tests related to first part of the assignment."""
```

```
    def test_unigram(self):
        """Test that unigram distributions work."""
        a = 'abcdabcaba'
        dist = unigram(a)
        self.assertEqual(dist, {
            'a': 0.4,
            'b': 0.3,
            'c': 0.2,
            'd': 0.1,
        })

    def test_bigram(self):
        """Test that bigram distributions work."""
        a = ' abcdabcaba'
        dist = bigram(a)
        self.assertEqual(dist, {
            ('a', 'b'): 0.3,
            ('b', 'c'): 0.2,
            ('c', 'd'): 0.1,
            (' ', 'a'): 0.1,
            ('d', 'a'): 0.1,
            ('c', 'a'): 0.1,
            ('b', 'a'): 0.1,
        })

    def test_iid_length(self):
        """Test that i.i.d. length is reported correctly."""
        a = 'abcdabcaba'
        dist = unigram(a)
        iid_len = iid_length(a, dist)
        self.assertEqual(iid_len, 21)

    def test_bi_length(self):
        """Test that bigram length is reported correctly."""
        a = ' abcdabcaba'
        uni_dist = unigram(a)
        bi_dist = bigram(a)
        bi_len = bi_length(a, uni_dist, bi_dist)
        self.assertEqual(bi_len, 11)

    def test_round_dist(self):
        """Test rounding and renormalizing a distribution works."""
        a = 'abcdabcaba'
        dist = unigram(a)
        self.assertEqual(norm_dist(round_dist(dist)), {
            'a': 103 / 258.,
            'b': 77 / 258.,
            'c': 52 / 258.,
            'd': 26 / 258.,
        })

    def test_iid_round_length(self):
        """Test rounding scheme using i.i.d. works."""
        a = 'abcdabcaba'
        dist = unigram(a)
        self.assertEqual(iid_round_length(a, dist), {
            'header': 216,  # 27 chars, 8 bits each
            'data': 21,
            'total': 237,
        })

    def test_bi_round_length(self):
        """Test rounding scheme using bigrams works."""
        a = ' abcdabcaba'
        uni_dist = unigram(a)
        bi_dist = bigram(a)
        self.assertEqual(bi_round_length(a, uni_dist, bi_dist), {
            'header': 6048,  # 27 * 27 + 27 chars, 8 bits each
            'data': 11,
            'total': 6059,
```

```
        })


class TestNoisyChannel(unittest.TestCase):

    """Tests related to the second part of the assignment."""

    def test_nutritious_snacks(self):
        """Test nutritious_snacks function."""
        result = nutritious_snacks('abcd', [97, 2, 6, 21])
        self.assertEqual(result, '\x00`eq')

    def test_encode(self):
        """Test that encoding works correctly."""
        stream = MockStdIn('0100')
        result = ''.join(encode(stream))
        self.assertEqual(result, '01100001')

    def test_decode_no_error(self):
        """Test that code without errors gets decoded correctly."""
        stream = MockStdIn('01100001')
        result = ''.join(decode(stream))
        self.assertEqual(result, '0100')

    def test_decode_one_error(self):
        """Test that code with one error gets decoded correctly."""
        stream = MockStdIn('01110001')
        result = ''.join(decode(stream))
        self.assertEqual(result, '0100')

    def test_parity_bit_one_error(self):
        """Test that code with one parity bit error gets decoded correctly."""
        stream = MockStdIn('01100101')
        result = ''.join(decode(stream))
        self.assertEqual(result, '0100')

    def test_decode_two_errors_raises(self):
        """Test that code with two errors returns errors."""
        stream = MockStdIn('11110001')
        result = ''.join(decode(stream))
        self.assertEqual(result, '1110')


class TestBitError(unittest.TestCase):

    """Tests verifying that bit error computed analytically matches."""

    def analytic_bit_error(self, f, k):
        """Counts the average number of bit errors in k transmissions."""
        failure = 0.
        total = 0
        for i in range(k):
            number = generate_num(4)
            stream = MockStdIn(number)
            encoded = ''.join(encode(stream))
            noised = ''.join(noise(MockStdIn(encoded), f=f))
            decoded = ''.join(decode(MockStdIn(noised)))
            total += len(number)
            for enc, dec in zip(number, decoded):
                if enc != dec:
                    failure += 1
        return failure / total

    def real_bit_error(self, f):
        """Computes probability of bit error."""
        total_failure = 0.
        number = generate_num(4)
        stream = MockStdIn(number)
        encoded = ''.join(encode(stream))
        enc_len = len(encoded)
```

```
        flippy = chain(*(combinations(range(8), i) for i in range(enc_len + 1)))

        for flip in flippy:
            copy = [int(c) for c in encoded]
            prob = pow(f, len(flip)) * pow(1 - f, 8 - len(flip))
            for pos in flip:
                copy[pos] = 1 - copy[pos]  # flip it
            noised = ''.join(str(c) for c in copy)
            decoded = ''.join(decode(MockStdIn(noised)))
            failure = 0.
            for enc, dec in zip(number, decoded):
                if enc != dec:
                    failure += 1
            total_failure += prob * (failure / 4)
        return total_failure

    def analytic_bit_error_test_04(self):
        """Test with 0.4 probability."""
        real_failure_rate = self.real_bit_error(0.4)
        failure_rate = self.analytic_bit_error(0.4, 10000)
        self.assertAlmostEqual(real_failure_rate, failure_rate, places=1)

    def analytic_bit_error_test_01(self):
        """Test with 0.1 probability."""
        real_failure_rate = self.real_bit_error(0.1)
        failure_rate = self.analytic_bit_error(0.1, 10000)
        self.assertAlmostEqual(real_failure_rate, failure_rate, places=2)

    def analytic_bit_error_test_0001(self):
        """Test with 0.001 probability."""
        real_failure_rate = self.real_bit_error(0.001)
        failure_rate = self.analytic_bit_error(0.001, 100000)
        self.assertAlmostEqual(real_failure_rate, failure_rate, places=5)


if __name__ == '__main__':
    unittest.main()
```