ELG5131 Project 1

Sum/Max Product Decoding

Course: Graphical Models (ELG 5131)

Professor: Yongyi Mao

Teaching Assistant: Vivian Zhao

Date: October 20, 2017

Students:

Tahira Ghani (CUID #101086805)

Guillaume Perrault-Archambault (UOID #4106345)

Contents

[Project Setup 3](#_Toc496046642)

[Implementation Notes 5](#_Toc496046643)

[Running Instructions & Dependencies 7](#_Toc496046644)

[Results 7](#_Toc496046645)

# Project Setup

The goal of this assignment is to design a decoder that finds the mostly likely transmitted (*X1, X2, X3, X4, X5, X6, X7*) codeword given a set (*Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7)* of received values. We design a decoder that can perform this task using either of two algorithms: max product and sum product.

The transmitted codewords are produced by performing (7, 4) hamming encoding on four bit messages, and therefore the Xi values satisfy the following equations:

(X1+X3+X5+X7) mod 2 = 0

(X2+X3+X6+X7) mod 2 = 0

(X4+X5+X6+X7) mod 2 = 0

The received *Zi* values are formed by two sequential transformations of Xi. First the Xi bits are mapped from {0, 1} to {1, -1} such that

Then, zero-mean Gaussian noise with known variance is added to the resulting Yi value such that

where N(µ, σ2) is a random variable distributed as a Gaussian distribution centred at µ with variance σ2. In other words, if we let the Gaussian noise be a random variable N, we have

Where P(N) is the probability density function of N.

We note that the individual Yi values are assumed to be transmitted through independent and identically distributed Gaussian channels, such that given all Xi, the Zi values are independent, and all Zi have the same conditional distribution.

Figure 1 shows a factor graph model that represents the probabilistic relationships between the transmitted (*X1, X2, X3, X4, X5, X6, X7*) codewords and the received (*Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7)*, where we have:

* G = PZi|Xi(Zi, Xi): G(Zi, 0) = N(1, σ2) and G(Zi, 1) = N(-1, σ2)
* F1(X1, X3, X5, X7) = δ( (X4+X5+X6+X7) mod 2 = 0)
* F2(X2, X3, X6, X7) = δ((X2+X3+X6+X7) mod 2 = 0)
* F3(X4, X5, X6, X7) = δ((X4+X5+X6+X7) mod 2 = 0)

and where δ(p) = 1 if p is True, and δ(p) =0 if p is False.

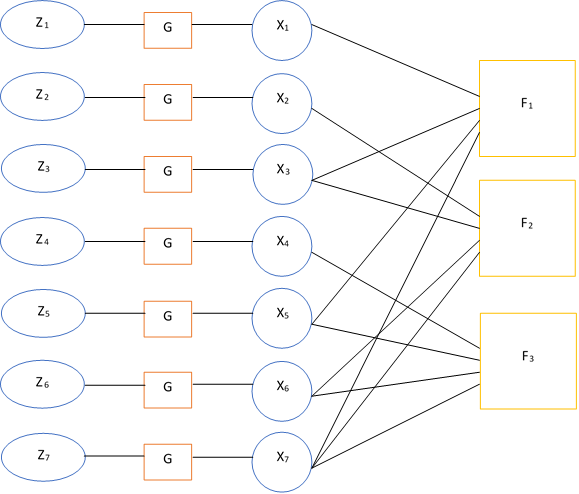


Figure 1

We prove that figure 1 is a factorization of the probability density function P:

(1)

(2)

(3)

(4)

In step (1) we make use of the definition of conditional probability. To go from step (1) to (2), we make use of the conditional independence of the Z*i* caused by the independence of the Gaussian channels, described earlier. To go from step (2) to (3) we simply note that given X*i*, Z*i* does not depend on any Xj where j!=i. And to go from step (3) to (4) we make use of our knowledge that there are 16 valid codewords, all of which are assumed equiprobable (and where invalid codewords have probability zero).

We see that (4) differs from the factor graph by only the scalar factor 1/16, and thus for the purposes of the Sum Product and Max Product algorithms (which finds the argmax of the relevant probability distribution, and not the max value itself), this factor graph is a complete probabilistic characterization of .

In this project, the Zi are observed variables and the Xi are the query variables.

We can therefore simplify the factor model diagram as follows:

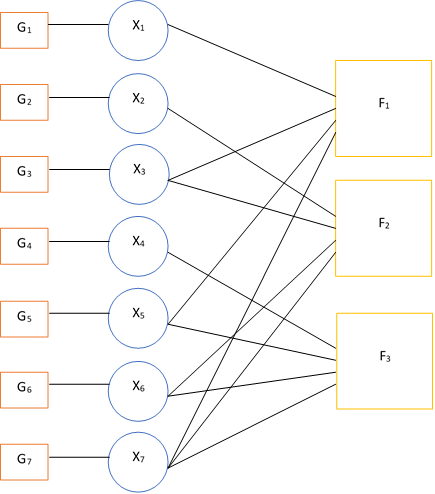


Figure 2

where we have Gi(Xi) = G(αi, Xi), and where αi is the observed value of Zi (Zi = αi).

# Implementation Notes

We implement our max product and sum product algorithms in python, and we leverage the numpy library for vector, matrix, and mathematical operations.

Our decoder is implemented in a parametrized class named ‘Decoder’. The Decoder class is initialized with a Hamming parity matrix (H), a Hamming decoder matrix (R), the number of iterations to use in the decoder, and a binary variable to select whether the original four bit message is retrieved. The Decoder class’ ‘decoder’ method accepts the received Z vector, the decoding algorithm (sum product vs max product), and the standard deviation, as input. The decoder method uses the message passing algorithm for max product and sum product algorithms to compute the most likely (*X1, X2, X3, X4, X5, X6, X7*). The message passing algorithm is based on the factor graph described in the setup, where the received Z = (α1, α2, α3, α4, α5, α6, α7) and standard deviation are used to compute the Gi functions. Optionally, the decoder uses the R matrix to retrieve the original four bit message.

The H matrix can be thought of as mapping the edges between the Xi variable nodes and the Fi function nodes:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | X1 | X2 | X3 | X4 | X5 | X6 | X7 |
| F1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| F2 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| F3 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

We note that each ‘1’ represents and edge between the row node and the column node, and a ‘0’ means the two nodes are not connected. Thus the Hamming matrix, the standard deviation, and the received Z vector form a full characterization of our factor graph (the Fi functions are implied from the H matrix).

We note that while our algorithm is designed to be general enough to work with any Hamming matrix, it was only tested with the above (7,4) Hamming decoder matrix.

The graph's messages can be partitioned as follows:

1. The messages from the Gi function nodes to variable (Xi) nodes -- these are cycle free and therefore are statically described in an array which we arbitrarily call m.
2. The messages from the variable (Xi) nodes to the function (fi) nodes, which may or may not be part of cycle (depending if they connect to 1 vs 2+ function nodes). Each variable can potentially send a message to any function node, and these messages change at each iteration of a cycle. We describe these messages in a sparse matrix V (for variable node) -- sparse because the entry for any unconnected Xi->fi node will be zero.
3. The messages from the function (fi) nodes to the variable (Xi) nodes. We describe these messages using a matrix F. Similarly to V (and for the same reasons), F is dynamic and sparse, and its non-zero entries may or may not belong to cycles.

At each iteration, our decoder first updates the F matrix, and then the V matrix. After a prescribed number of iterations (we chose 10, which we experimentally found to give sufficient performance), we take the summary message at each variable Xi node. The argmax of these summary messages is our best guess for the codeword (*X1, X2, X3, X4, X5, X6, X7*).

Finally, we note that because underflow was observed (small fractions multiplied many times become too small to be carried in floating point numbers) we take the log of all original messages, and perform all computations in the log domain.

In the log domain:

* Multiplication in linear domain becomes addition
* Addition becomes logsumexp (numpy has an efficient function for this, so we don’t use the algorithm suggested in the assignment description)
* Max remains max (since log(x) is a monotonically increasing function)

For performance testing of our decoder, we also implement Transmitter and Simulator classes. The Transmitter class is initialized with the G generator matrix, and a binary variable to select whether the encoder uses a 4 bit message as input instead of the Hamming codeword. The Transmitter class’ ‘transmit’ method takes the (*X1, X2, X3, X4, X5, X6, X7*) codeword and standard deviation as input, and transforms the X vector into the (*Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7)* transmitted values, as described in the setup. Optionally, the transmit method may also accept a 4 bit message, which it Hamming encodes before transforming into the (*Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7)* values.

The ‘Simulator’ class is initialized with a number of iterations, the H matrix, the R matrix, the G matrix, and the number of iterations (or samples) to use. It uses the H, R, and G matrices to initialize the Transmitter and Decoder classes. For each iteration, for each algorithm, and for a range of variances from 0.1 to 1, the Simulator class’ simulate method sets the Z vector, sets a range of variance values, and calls the transmit and decode methods and compares the output X with the Z vector. With the result of these operations, the simulate method computes a bit error rate for each variance, and then plots bit error rate against variance for each algorithm. Note that for simplicity (and as instructed in the assignment description) we set our Z vector to zero.

Finally, our ‘simulation.py’ script simply instantiates the Simulator class and runs the simulate method.

# Running Instructions & Dependencies

To run this system, you will need python 3.4 or higher (anything above 3.0 may work but we only tested 3.4).

You will also need the following python libraries:

* Numpy
* Scipy
* Matplotlib

To execute, run *simulation.py.*

# Results

Figure 3 shows the two plots computed and displayed by the simulate method, as described in Implementation Notes. The graph plots the bit error rate against variance of the Gaussian channel for the Max product algorithm (blue plot) and the Sum product algorithm (red plot). The variance values range from 0.1 to 1.0 in increments of 0.1. The The max product and sum product bit error rates range are shown in the table below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Variance -> | 0.1 | 0.2 | 0.3 | 0.4 | 0.5 | 0.6 | 0.7 | 0.8 | 0.9 | 1.0 |
| Max product | 2.14285714e-04 | 1.35714286e-03 | 9.35714286e-03 | 2.10714286e-02 | 3.67142857e-02 | 5.69285714e-02 | 7.40000000e-02 | 9.45000000e-02 | 1.16857143e-01 | 1.24642857e-01 |
| Sum product | 7.14285714e-05 | 2.07142857e-03 | 9.14285714e-03 | 2.00000000e-02 | 4.09285714e-02 | 5.55714286e-02 | 7.05714286e-02 | 8.87857143e-02 | 1.05928571e-01 | 1.22785714e-01 |

Each point in the figure 3 plots represents the bit error rate (probability of bit error) after transmission and decoding of 2000 codewords, for a particular variance.

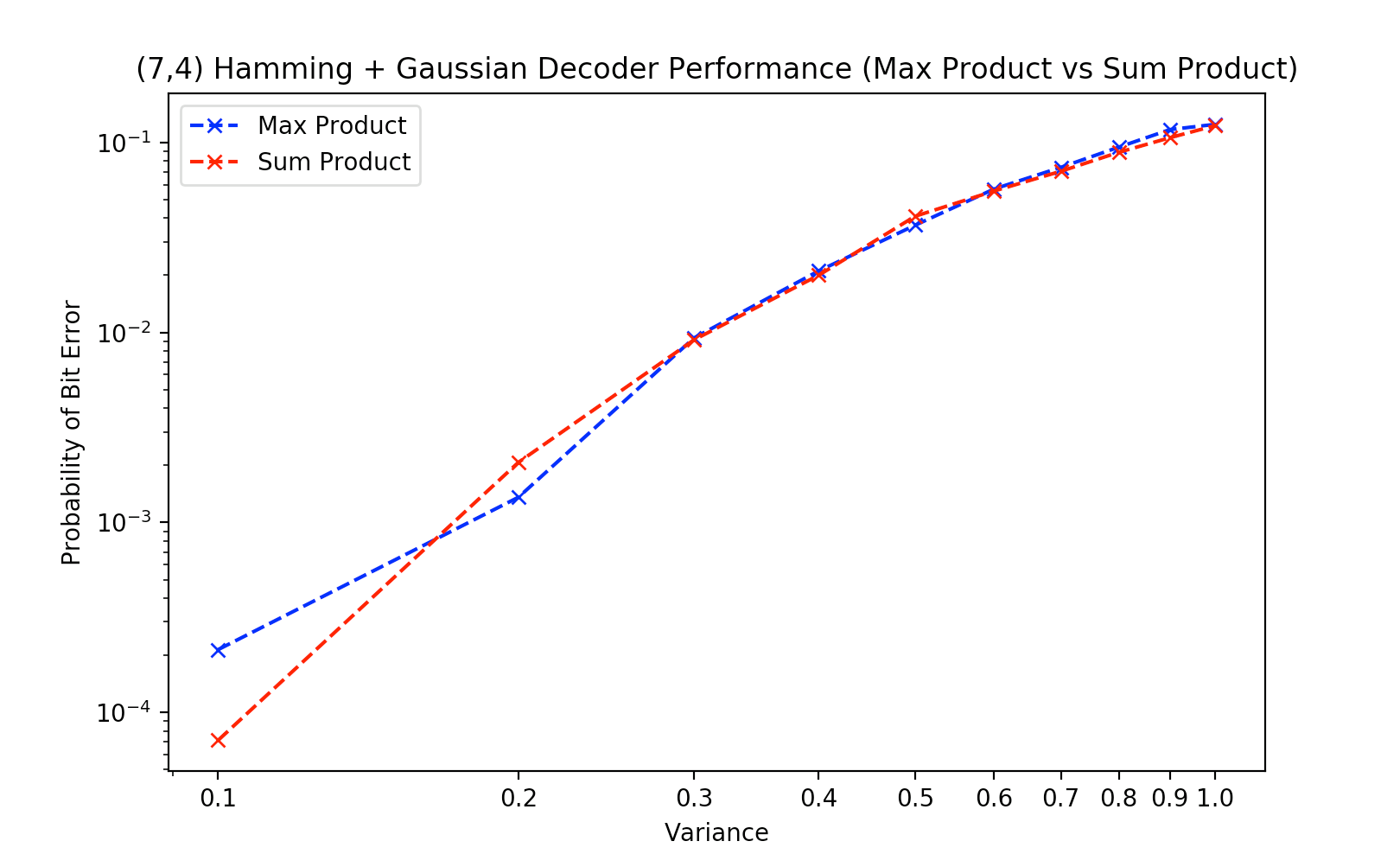


Figure 3

We notice for both plots that as variance increases, so does the probability of bit error. This is expected as higher noise variance increases likelihood that bit errors make the received codeword indistinguishable from an invalid codeword (or a valid codeword different than the transmitted codeword). We also notice that both algorithms perform almost identically at variances of 0.3 and greater, but have slightly different results at variances of 0.1 and 0.2. This is expected because at 2000 samples, at low variances bit errors are so uncommon that the sample size is too low to accurately represent the bit error probability. At larger variances, the similar performance of the algorithms is expected because there are sufficient errors to closely approach the true bit error probability. The true bit error probability of each algorithm is expected to be similar because while the max product is expected to have a greater portion of correct codewords, its errors are expected to be larger when it guesses wrong.