ELG5131 Project 1

Sum/Max Product Decoding

Course: Graphical Models (ELG 5131)

Professor: Yongyi Mao

Teaching Assistant: Vivian Zhao

Date: October 16, 2017

Students:

Tahira Ghani (CUID #101086805)

Guillaume Perrault-Archambault (UOID #4106345)

# 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.

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

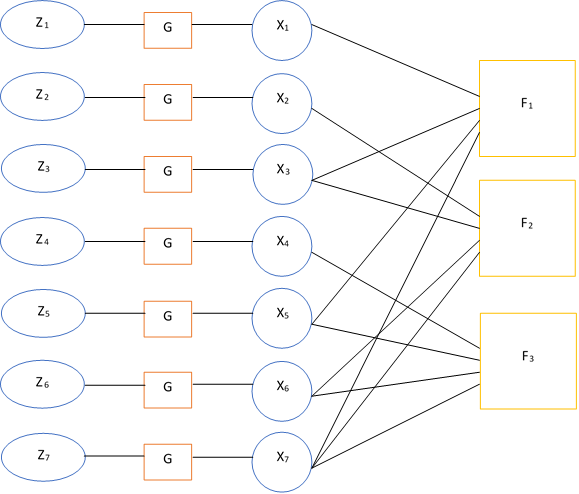
(X4+X5+X6+X7) mod 2 = 0

The received *Zi* values are formed from the Xi in two steps. First the X*i* bits are mapped from (0, 1) to (1, -1) such that Yi = 1 if Xi = 0, and Yi = -1 if Xi = 1.

Then, zero-mean Gaussian noise with known variance is added to the resulting Yi value value such that Zi = Yi + N(0, σ2).

where N(µ, σ2) is a Gaussian distribution centred at µ with variance σ2. 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 have the same conditional distribution.

The factor graph model below 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.

The above graph is a factorization of the probability density function P(, which we prove:

(1)

(2)

(3)

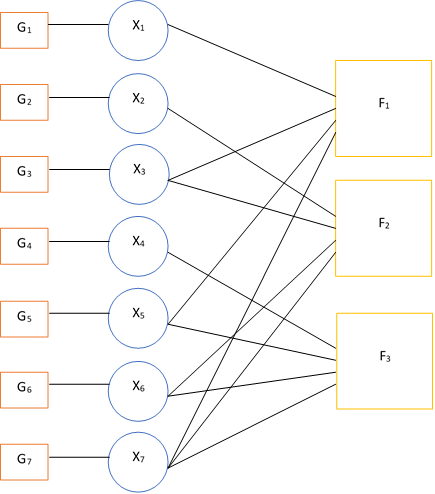
(4)

In step (1) we made use of the definition of conditional probability. To go from step (1) to (2), we made use of the conditional independence of the Z*i* mentioned earlier. To go from step (3) to (4) we simply notice that given X*i*, Z*i* does not depend on X*j* where j!=i. And to go from step (3) to (4) we of our knowledge that there are 16 valid codewords, all of which are assumed equiprobable.

We see that (4) differs from the factor graph by one the scalar factor 1/16, and thus for the purposes of the Sum Product and Max Product algorithms (which care about the argmax, and not the max value itself), this factor graph is a complete characterization of P(.

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:



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

# Implementation Notes

The system setup has been divided into three main components (i.e. classes), namely the transmitter, the decoder, and the simulator.

The simulator initializes constant variables including the parity, decoding and code generator matrix, as well as the number of iteration and list of standard deviations for testing. It calculates and stores bit errors as it iterates through the variances and repeatedly calls transmitter and decoder.

The transmitter is responsible for transferring the message through the channel which includes (optional) Hamming encoding, extension of range through bit inversion, and addition of noise (*Ni*) from a Gaussian distribution. The input is *m*, which based on the optional Hamming encryption is either the original message or the Hamming encoded message. The output is *z*, which is the observed code word for which the decoder will try and find the most likely possibility of *X*.

The decoder takes the following as input:

* *z*, a 7-entry array: (z1, z2, z3, z4, z5, z6, z7) as described in the setup
* *use\_maxproduct* to specify the algorithm used for decoding (1 for max product, 0 for sum product)
* *std\_deviation* is the known standard deviation of the channel Ni, also described in the setup

The graph's messages can be partitioned as follows:

1. The messages from the Pz|x 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.

The returned value 'x' is the decoder's best guess for the original code word sent by the transmitter. To compute x, we take the summary message (multiplication of all incoming messages at the variable (xi) nodes), after a prescribed number of iterations.

Finally, 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.

# 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).

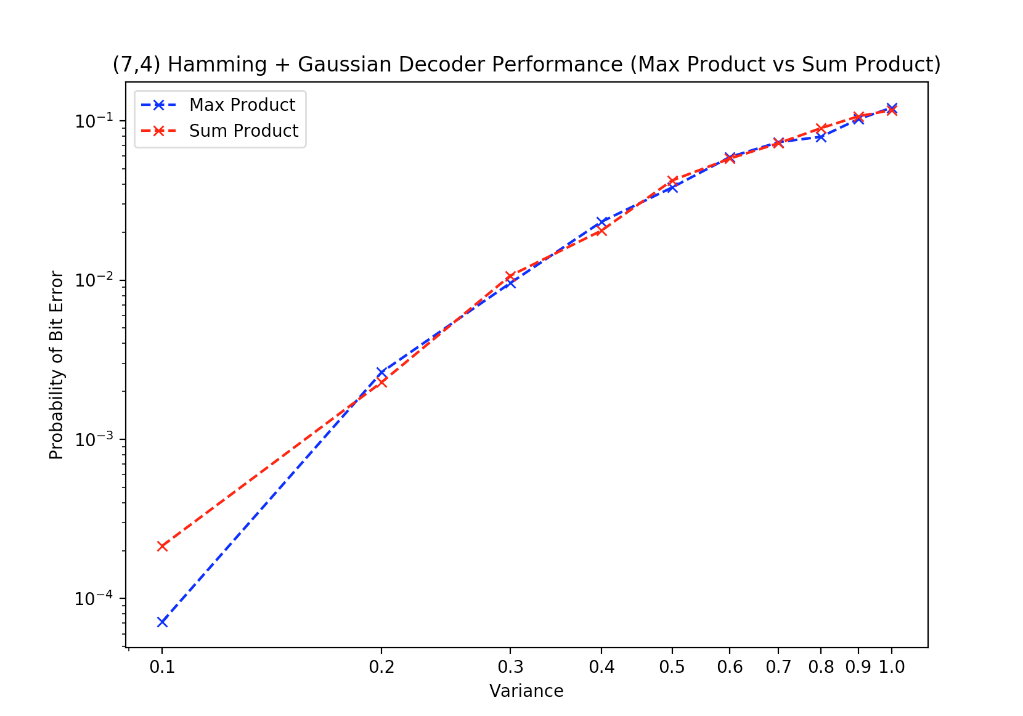
You will also need the following python libraries:

* Numpy
* Scipy
* Matplotlib

Execute *simulation.py.*

# Results

The graph displays the relation between probability of bit error and variance for an execution of the system with 2000 code words and 10 iterations per decode run.



The graph shows that as variance increases, so does the probability of bit error. However, the results follow through with our expectations that max product generally performs better than sum-product.