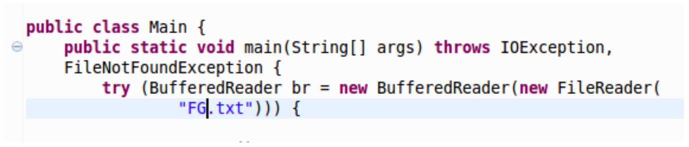
Project Report
Partially-generic Message-passing Decoding

University of Bergen
Hannah A. Hansen, ruq003@student.uib.no

Implementation

Factor graph text file Change the file name parameter of *new BufferedReader(new FileReader("FG.txt")))* in the class Main.java in package *Default package*, in order to change the input text file for the program.



```
public class Main {
    public static void main(String[] args) throws IOException,
    FileNotFoundException {
        try (BufferedReader br = new BufferedReader(new FileReader(
                "FG.txt"))) {
```

Figure 1: FG.txt - Factor graph, SPA-flooding

Data structures The data structures of the project are in the package *DataStructures*. The graph is represented by the Graph.java class. It contains information about the channel, the variable nodes, the factor nodes, and the AWGN nodes, and acts as a wrapper for all the information that is to be passed along to the object of the selected algorithm. Information about edges are kept in the nodes themselves and as an adjacency list of *edge* objects in the *graph* object. Each node knows of its neighbors and no other nodes. Factor nodes and all their functionality are represented by the class Factor.java, similarly for variable nodes and the class Variable.java, AWGN nodes and the class AWGN.java, and for state nodes and State.java class.

Algorithms The package *Algorithms* contains three classes which represent each variant of the problems A, B, and C given. They are all implemented as specified in the assignment text, except for one modification in problem C concerning the algorithms of decoding on a Trellis reperesentation. There are three algorithm options implemented for problem C and may be specified in the input text accordingly; C - BJCR, M - MinSum, P - MaxProduct.

Gaussian Distribution

For purposes of simulating an additive white gaussian channel, the class AWGN.java was implemented along with the methods *transmit()*, *boxMuller()* and *calcR()*. This class provides input to the given factor graph from a gaussian distribution which has as mean zero and variance equal to *noise/2*. In order to confirm the distribution to be gaussian, 50.000[1] samples was generated and plotted in Fig.2.

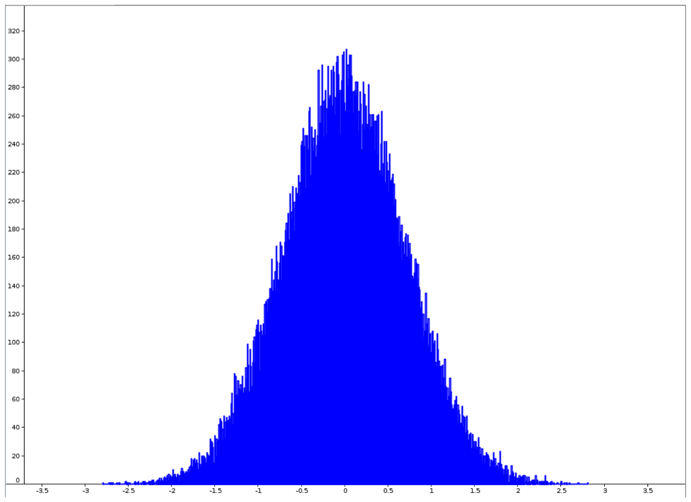


Figure 2: Gaussian distribution: Sample size 50 000

Figure 2. demonstrates with a solid degree of certainty that the distribution is indeed of a gaussian variety with mean = 0.

[1]A nicer plot may have been achieved by taking a larger sample of e.g. size 100.000, but this was not done as any attempt lead GeoGebra to stall the computer that was used in the plotting process.

Noise

In this section we shall examine the distribution of noise emitted by the AWGN channel. Here we see frequency distribution plots, concerning both $P(c_i = 1|r_i)$ and $P(c_i = 0|r_i)$ signals with varying SNR, each of which are of sample size 50.000.

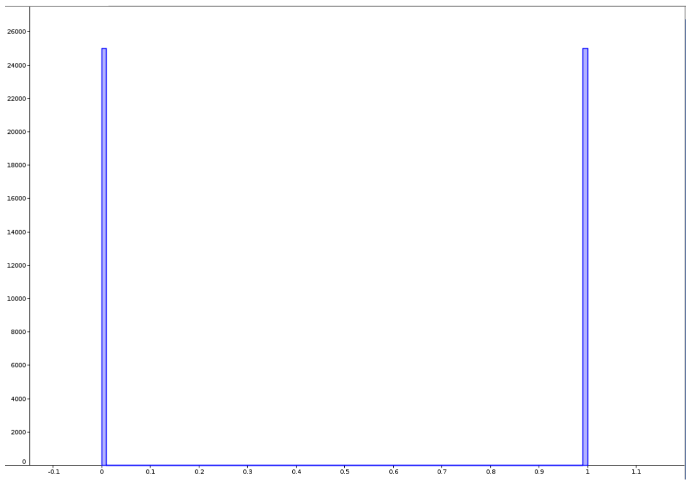


Figure 3: SNR: 10.0/0.05 Sample size: 50000

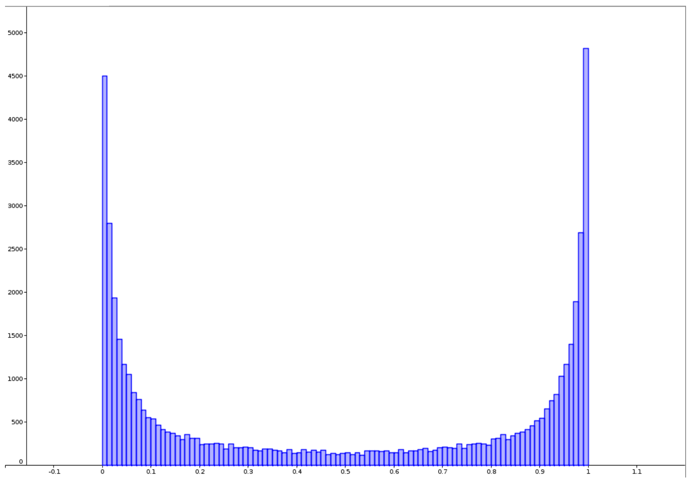


Figure 4: SNR: 10.0/10.0 Sample size: 50000

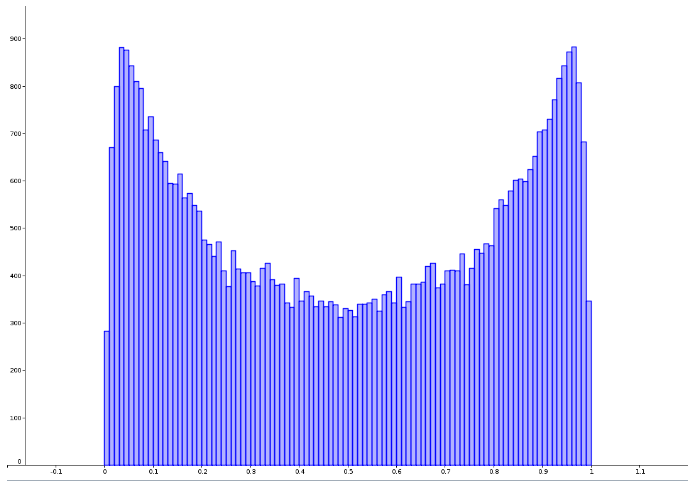


Figure 5: SNR: 10.0/20.0 Sample size: 50000

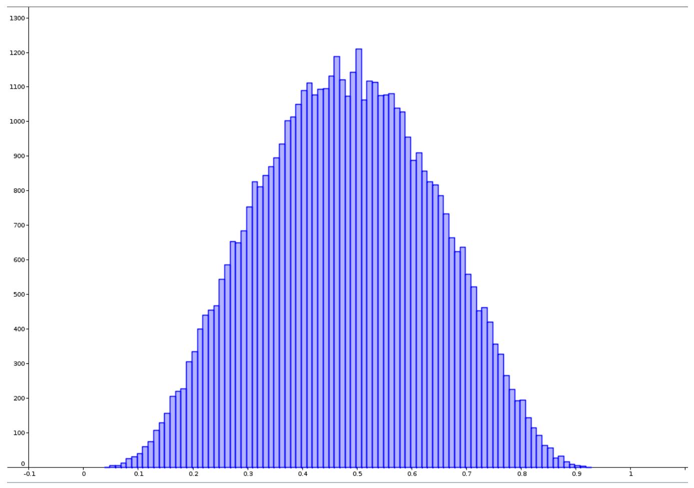


Figure 6: SNR: 10.0/100.0 Sample size: 50000

In these plots signals of either 0 or 1 are sent in equal amounts, as one sees in Fig.1 when the noise is very low the possibility of noise effecting these 1s and 0s is quite low - the output from the AWGN nodes has little to no effect on the transmission.

Transmitted bits can either be 0 or 1, so any noise added to it can only increase the likelihood of that particular bit being interpreted as its counterpart. Therefore, as the amount of noise increases we see a development that pushes the two previously mentioned tops towards eachother. At very high noise levels the frequency distribution becomes a gaussian curve. From Fig.6 we see that when there is a high amount of noise the chances of it effecting transmission bits are very high.

Results

Problem A

The algorithms for computating a single, or all, marginals of a factor graph containing no cycles is implemented in the class MargTree.java.

All marginals The method *allMarginals()* uses both *sendFromLeaves()* and *sendToLeaves()* in order to ensure that each edge has one and only one message going in each direction. It begins by taking a variable node in the middle of the list of variable nodes and recurses outwards until it reaches the leaves. Messages are then propogated back to the beginning of the recursion. Once that has taken place, the initial node now has the information necessary to initiate the propogation of messages back to the leaves. The marginals for each variable is then output printed out in the console. The below table gives an example of such output on the following graph.

A single marginal The method *marginalX()* uses only *sendFromLeaves()* to propogate messages from the leaves to the node chosen for marginalization.

5

0

5

[0]12

[1]03

$[0, 1, 3]$01121223

$[2, 3]$1201

$[4, 3]$2101

B

a

Marginals

X_0	30.0, 108.0
X_1	0.0, 138.0
X_2	30.0, 108.0
X_3	90.0, 48.0
X_4	114.0, 24.0

Problem B

The algorithm for decoding factor graphs with cycles was implemented in the SPAFlooding.java class and uses the Edge.class existing in the Graph.java class to iterate over all edges and send a message in each direction on it, and outputs the entire decoded codeword together with the BER and the WER. This process is repeated as specified in the input text file.

In relation to problem B, an examination of the different variables of the decoding process has been performed and the results plotted in the figures below. We began by examining the relationship between a varying SNR and the error rates, Fig.7. At each decrement of the SNR was plotted in relationship to a corresponding 1000 independent decodings computed with 100 iterations. Thereafter, we examined the error rates of decoding with a fixed SNR and a varying number of iterations.

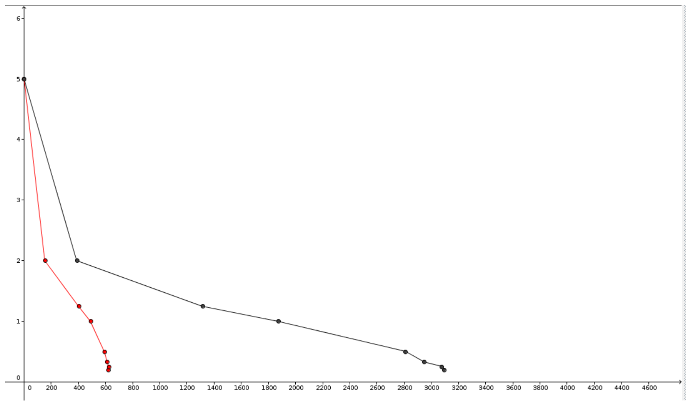


Figure 7: SNR variation: 100 iterations, 1000 ind. dec.

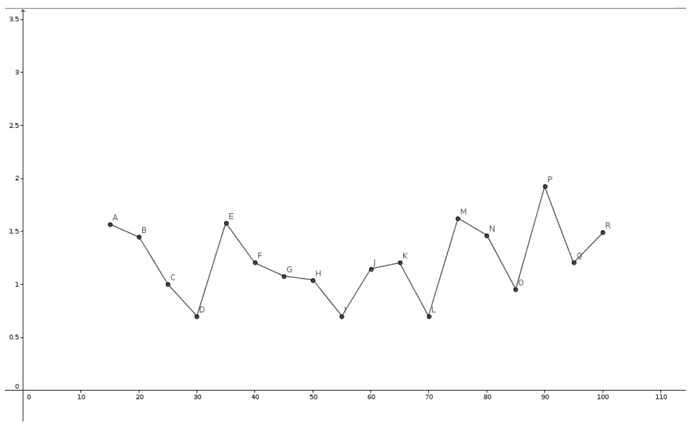


Figure 8: Number of iterations varyed: SNR = 10.0/100.0, BER

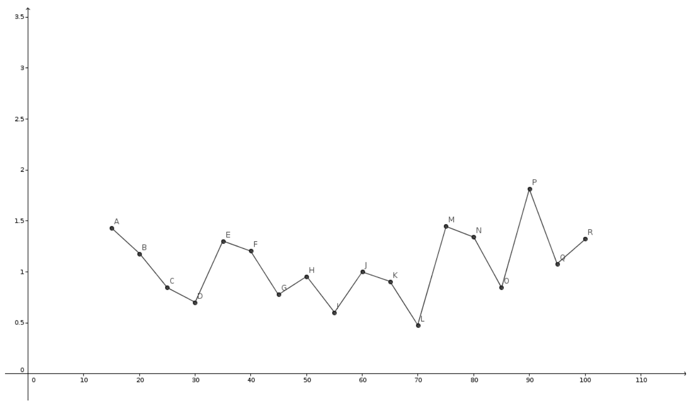
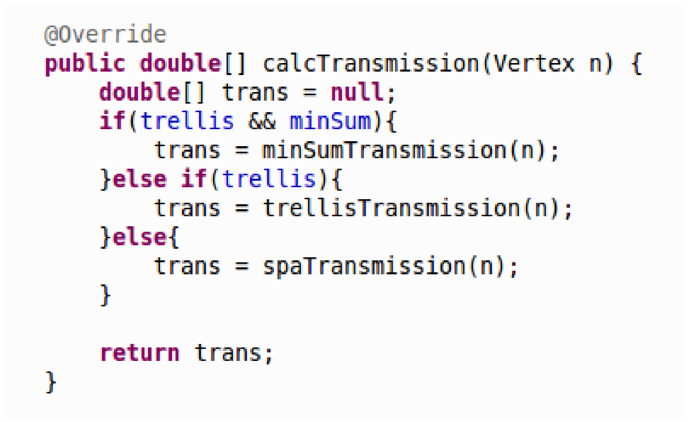


Figure 9: Number of iterations varyed: SNR = 10.0/100.0, WER

In Fig. 8.-9. we see that a bug exists somewhere that is disturbing the SPA-flooding algorithm, as the plot should resemble a curve illustrating that the error rates decrease as the number of iterations increase. In search for the error, the following components of the software have been varified to be sound; the method for computing messages, the passing of messages, the for-loop iteration over the edges, and the reseting of the graph information after an independent decoding. The author apologizes, as the bug has yet to be found.

Problem C

The algorithm for decoding on a Trellis have all been implemented in the class Trellis.java. The computational variants corresponding to the BJCR, min-sum, and the max-product reside in the class representing the factor nodes, as the procedure, or message-passing schedule, is the same, for all versions of decoding in assignment C. The method *calcTransmission()* of Factor.java call upon either *spaFlooding()*, *minSumTransmission()*, or *trellisTransmission()*, according to which algorithm has been specified in the input text file. The last method being the one responsible for calculating the max-product messages.



```java
@Override
public double[] calcTransmission(Vertex n) {
    double[] trans = null;
    if(trellis && minSum){
        trans = minSumTransmission(n);
    }else if(trellis){
        trans = trellisTransmission(n);
    }else{
        trans = spaTransmission(n);
    }

    return trans;
}
```

Figure 10: Method for calculating the transmission from a factor node

For problem C we have examined the relationship between the SNR and the error rates that may occur during decoding. The relationship between the SNR, the BER and the WER were then plotted in the figures below. These plots represent data from an iteratively decreasing SNR, at each iteration 1000 independent decodings were computed.

Figures 10.-11. demonstrate the influence the SNR has on error occuring on bits and entire codewords during decoding. As the noise decreases the SNR grows large, following this the error rates of decoding go down.

Additionally, Fig. 12. show that the increase of the BER during the decoding causes an increase in the WER some what proportianally to it. Although it is not indisputably represented in this figure, it is intuitive and the case that with a suffient amount of data one would be able to illustrate the proportional relationship between the BER and the WER.

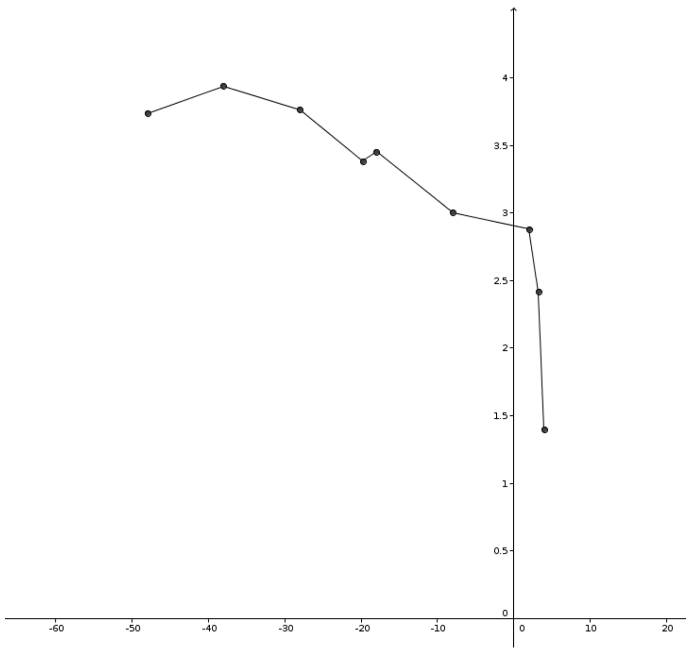


Figure 11: SNR variantions: BER

Conclusions

Results from the examination of the project confirm the correctness of the channel simulation, the computation of marginals on a factor graph containing no cycles, and the decoding on a Trellis representation. However, the results have also revealed that when varying the number of decoding iterations the error rates produced by running the SPA flooding algorithm grow in a chaotic and eradicate manner. This suggests that there is a bug unrelated to the SPA flooding algorithm somewhere in the code, as the only difference in the decoding of factor graphs with cycles and without cycles is the for-loop iteration in the SPAFlooding.java class.

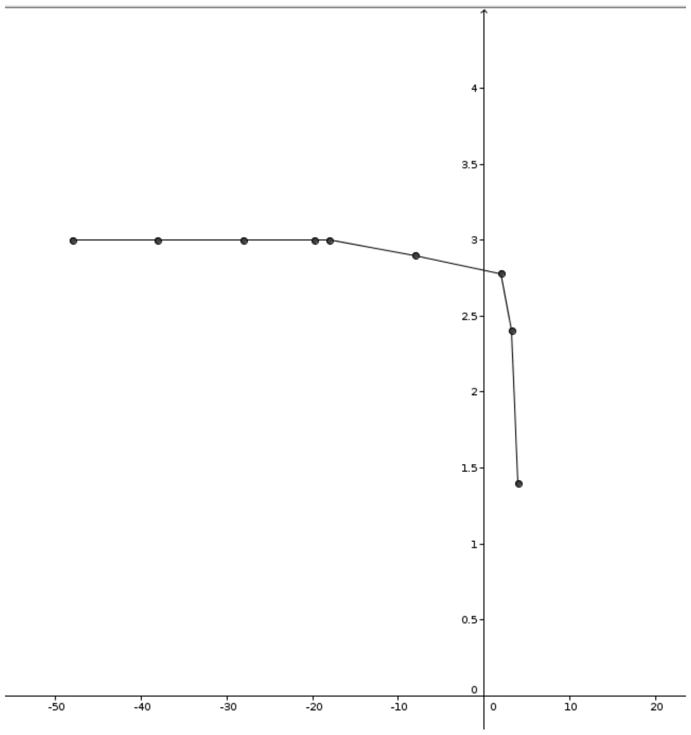


Figure 12: SNR variations: WER

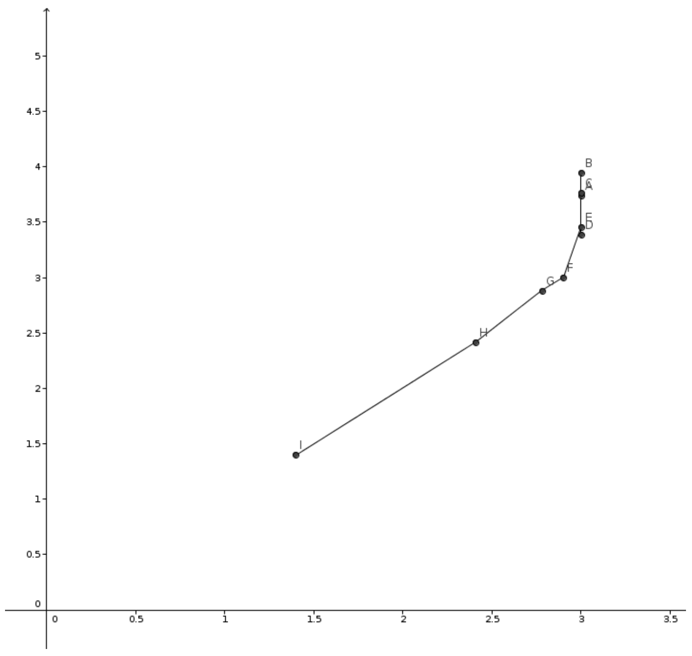


Figure 13: SNR variantions: (BER vs. WER)