ECE1508 Project Report
Spatially Coupled Codes and Decoders

James Bailey
May 3, 2015

Table of Contents

Figures

Introduction

This report discusses the work that I did for the Winter 2015 ECE1508 project. The topic of my project is an implementation of several decoders for a class of error correction codes known as spatially coupled low-density parity check codes (SC-LDPC). I also examine the performance of SC-LDPC with these decoders and compared my results to current literature.

SC-LDPC codes are a very promising class of error correction codes. They were first introduced in 1999 by Felstrom and Zigangirov in [1]. Unlike regular and irregular LDPC codes that suffer from comparatively poor performance in the waterfall and error-floor regions respectively, SC-LDPC codes perform well in both these regions. More recently, in [2] it was shown in that SC-LDPC codes universally achieve capacity in binary memoryless symmetric channels.

In section 1, I discuss a method for constructing these codes. The construction and structure of SC-LDPC codes lends itself to efficient decoder implementations, which I review in section 2. In section 3 I provide some information on the decoders I implemented, and the results of these implementations are discussed in section 4.

1 Construction Methods

There are many methods of constructing a spatially coupled code. The basic idea of a spatially coupled code is that:

1. Each parity check constraint can only apply to a spatially local set of variables,
2. There is overlap between these local variable sets such that *all* variables are (indirectly) constrained (or coupled) to each other

Several methods for constructing "good" SC-LDPC codes are shown in [3]. From this paper, I have used the JFZ technique of unwrapping an LDPC block code (LDPC-BC) into a SC-LDPC code.

The JFZ technique is as follows:

1. Start with a good LDPC-BC parity check matrix

$$H_{BC} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Figure 1 - (3,6) regular LDPC-BC

2. Cut the matrix along the diagonal, and shift the top half to the bottom

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \Longrightarrow \begin{array}{cccccccccc} 1 & 0 & & & & & & & & \\ 0 & 1 & 1 & 0 & & & & & & \\ 1 & 1 & 0 & 1 & 1 & 1 & & & & \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & & \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ & & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ & & & & 0 & 1 & 1 & 0 & 1 & 1 \\ & & & & & & 0 & 1 & 0 & 0 \\ & & & & & & & & 1 & 1 \end{array}$$

Figure 2 - LDPC-BC to SC-LDPC unwrapping

3. Extend this structure infinitely in both directions; this is the SC-LDPC parity check matrix.

$$H_{SC} = \left[\begin{array}{cccccccccccccccccccccc} & 1 & 0 & & & & & & & & & & & & & & & & & & & \\ \ddots & 0 & 1 & 1 & 0 & & & & & & & & & & & & & & & & & \\ & 1 & 1 & 0 & 1 & 1 & 1 & & & & & & & & & & & & & & & \\ & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & & & & & & & & & & & & & \\ & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & & & & & & & & & & & \\ & & & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & & & & & & & & & \\ & & & & & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & & & & & & & \\ & & & & & & & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & & & & & \\ & & & & & & & & & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & & & \\ & & & & & & & & & & & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & \\ & & & & & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & \\ & & & & & & & & & & & & & & & 0 & 1 & 1 & 0 & 1 & 1 & \\ & & & & & & & & & & & & & & & & & 0 & 1 & 0 & 0 & \ddots \\ & & & & & & & & & & & & & & & & & & & 1 & 1 & \end{array} \right]$$

Figure 3 - SC-LDPC parity check matrix

The tanner graph for this is shown in Figure 4.

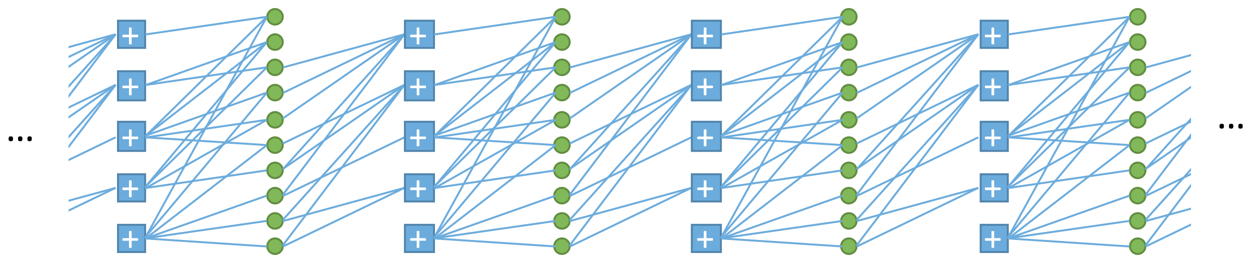

Figure 4 - SC-LDPC tanner graph

For practical reasons this parity check matrix can be terminated according to the amount of data being coded. This is shown in the next section.

2 Decoders

In this project I examined three types of SC-LDPC decoders. A brief description of each follows.

2.1 Standard Belief Propagation Decoder

As mentioned previously, we can terminate the SC-LDPC parity check matrix according to the amount of data that is being coded/decoded. A terminated *L=3* parity check matrix is shown in Figure 5.

$$H_{L3} = \left[\begin{array}{cccccccccccccccccccccccccccccc}
1&0\\
0&1&1&0\\
1&1&0&1&1&1\\
0&1&1&0&1&0&1&0\\
1&0&0&1&0&1&0&1&1&1\\
&&1&1&1&0&1&1&0&0&1&0\\
&&&&0&1&1&0&1&1&0&1&1&0\\
&&&&&&0&1&0&0&1&1&0&1&1&1\\
&&&&&&&&1&1&0&1&1&0&1&0&1&0\\
&&&&&&&&&&1&0&0&1&0&1&0&1&1&1\\
&&&&&&&&&&&&1&1&1&0&1&1&0&0&1&0\\
&&&&&&&&&&&&&&0&1&1&0&1&1&0&1&1&0\\
&&&&&&&&&&&&&&&&0&1&0&0&1&1&0&1&1&1\\
&&&&&&&&&&&&&&&&&&1&1&0&1&1&0&1&0&1&0\\
&&&&&&&&&&&&&&&&&&&&1&0&0&1&0&1&0&1&1&1\\
&&&&&&&&&&&&&&&&&&&&&&1&1&1&0&1&1&0&0\\
&&&&&&&&&&&&&&&&&&&&&&&&0&1&1&0&1&1\\
&&&&&&&&&&&&&&&&&&&&&&&&&&0&1&0&0\\
&&&&&&&&&&&&&&&&&&&&&&&&&&&&1&1
\end{array}\right]$$

Figure 5 - Terminated SC-LDPC parity check matrix

This code can be viewed as an asymptotically regular LDPC-BC (in *L*) and can be decoded using any standard belief propagation decoder, such as the one we discussed in class.

Note that both the number of variables and scale linearly with the termination length *L*.

2.2 Pipelined BP Decoder

Because of the repetitive structure of the SC-LDPC code it is possible to build a pipelined BP decoder (PD) that requires substantially fewer memory and processing elements than the standard decoder discussed in 2.1.

A pipelined decoder architecture was first introduced in [1]. I will give a brief description of the decoder below.

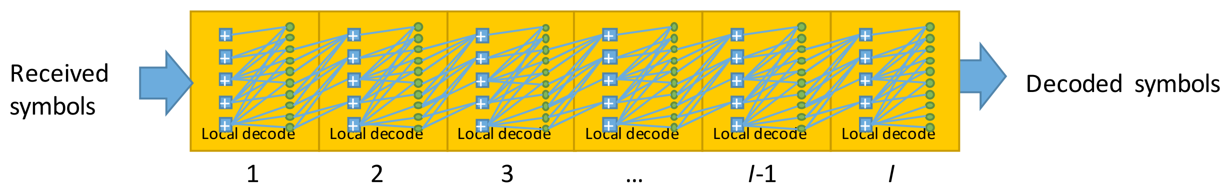

Figure 6 - Pipelined BP decoder graph

To construct the architecture, a decoding processor is built for a single unwrapping of the underlying LDPC-BC. This element copied multiple times corresponding to the number of decoding iterations I desired, and connected together to form a BP decoding graph such as the one shown in Figure 6.

Whenever a single stage of symbols is received they are shifted into the first decoding element, with each decoder stage also shifting its contents forward. A single iteration of BP decoding is then run. The output of the last stage is the decoded data; note that I iterations of BP decoding have been run on the output data by the time it is shifted out.

The memory for this decoder scales linearly in I, which in practise is much less than the L scaling of the traditional decoder.

The performance of the pipelined BP decoder is equivalent to the traditional BP decoder [1]. For this reason I only show one set of BP results in this report.

2.3 Window BP Decoder

A windowed BP decoder (WD) is defined in [4]. This window decoder has similar architecture to the pipelined decoder discussed previously; however it offers some attractive memory and latency advantages with potentially small BER performance impact.

I provide a brief description of the WD below.

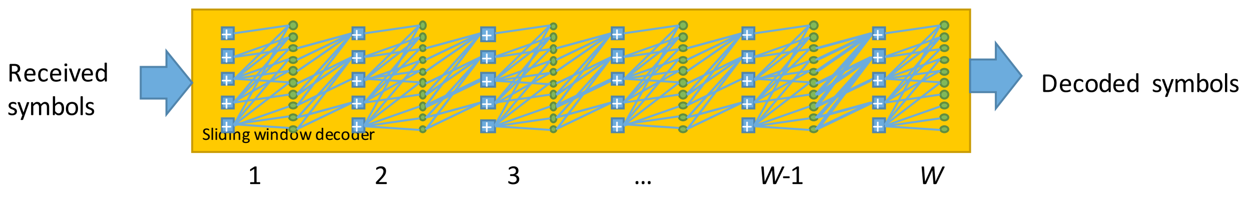

Figure 7 - Windowed BP decoder

The PD architecture from 2.2 is used for the WD, with the following differences:

- W instances of the decoding processor are used; where W is the window size of the decoder
- Multiple BP decoding iterations are run for each decoder shift, set to a fixed value or based on some stopping criteria

It is claimed that good BER performance can be achieved while choosing $W \ll I$. This means that the WD decoder has proportionately smaller variable memory and latency requirements compared to the pipelined decoder.

The WD was shown in [4] to approach standard BP performance exponentially in the size of W. The paper states that typical values for W in this paper are 4, 6, and 8 with local iteration counts less than 20 compared to total BP iteration count I was as high as 300.

3 Decoder Implementation

For this project I implemented a single configurable BP-based decoder that is capable of running the BP, PD, and WD decoders discussed above. The decoder is implemented and optimized for only the binary erasure channel. I started to add BSC and AWGN channel support however due to time constraints, and runtime constraints they are not included in this project. To give a sense of the runtime constraints, generating results for the BP decoder with a *L=64* terminated code took over 8 hours, therefore a BSC decoder would have taken several days if not longer.

The decoder takes an input file that defines the parity check matrix of the base code to be unwrapped. The format of the parity check file is the same as the one used in class. Based on the arguments passed to the decoder it will be unwrapped internally to the appropriate terminated SC-LDPC parity check matrix. The program then runs a sliding window decoder on the internally generated SC-LDPC code.

To achieve the standard BP decoder the window size should be set to the termination size of the code. This means there is only one valid window position, which is the entire code.

If the termination length is set to zero, the code is not unwrapped at all and decoding will be performed directly on the code inside the parity check file.

3.1 Build Procedure

I have included a `README` file in the source code package describing its contents.

The source code for my BP/PD/WD decoder is in:

- `window.cc`

The decoder is written in C++ and was tested on Ubuntu Linux. It requires:

- C++/11 compiler
- CMake build framework
- BOOST C++ libraries

To build the decoder, go to the source code directory and run:

```
1. cmake
2. make
```

This should create the decoder program: `window`

3.2 Running the Decoder

The program takes the following arguments.

```
Allowed options:
  -h [ --help ]            help message
  --parity-file arg        parity check matrix file
  --wsize arg (=1)         window size of decoder in multiples of the base code
  --iter arg (=100)        number decoding iterations per pipe stage
  --term arg (=100)        termination length of data
  --blocks arg (=100)      number block errors to test
  --start arg (=0.4)       start erasure probability
  --stop arg (=1)          stop erasure probability
  --num arg (=10)          number of erasure probabilities to test
```

Figure 8 - Decoder arguments

For standard BP decoding, the window size and termination size should be set the same value.

```
./window ../n1024_3-6_regular.txt \
          --wsize 64 --iter 300 --term 64 \
          --blocks 100 --start 0.5 --stop 0.4 --num 20
```

Figure 9 - BP decoder example command

For PD, the iteration count should be set 1 and window size to the desired pipeline length.

```
./window ../n1024_3-6_regular.txt \
          --wsize 20 --iter 1 --term 64 \
          --blocks 100 --start 0.5 --stop 0.4 --num 20
```

Figure 10 - PD decoder example command

For WD, the iteration count should be set to desired value and window size to desired value.

```
./window ../n1024_3-6_regular.txt \
          --wsize 8 --iter 30 --term 64 \
          --blocks 100 --start 0.5 --stop 0.4 --num 20
```

Figure 11 - WD decoder example command

4 Results

The decoder was used to examine the performance of a SC-LDPC compared to the corresponding base code. I also looked at the performance of the WD as a function of window size, as well as compared to the BP decoder.

To do this I generated a $(3,6)$-regular LDPC block code with $n = 1024$ to be used as a baseline for unwrapping to a SC-LDPC code. The performance of this code was tested against other generated codes in the same ensemble to verify that it was typical.

Performance of this SC-LDPC code compared to the base LDPC-BC is shown in Figure 12. Data to the SC-LDPC BP decoder was terminated after 64 blocks (65536 bits) with 300 decode iterations for each data block. All simulations were run until 100 block errors had occurred. Note that there is substantial BER performance gain with the SC-LDPC code in the waterfall region, and no error floor is observed.

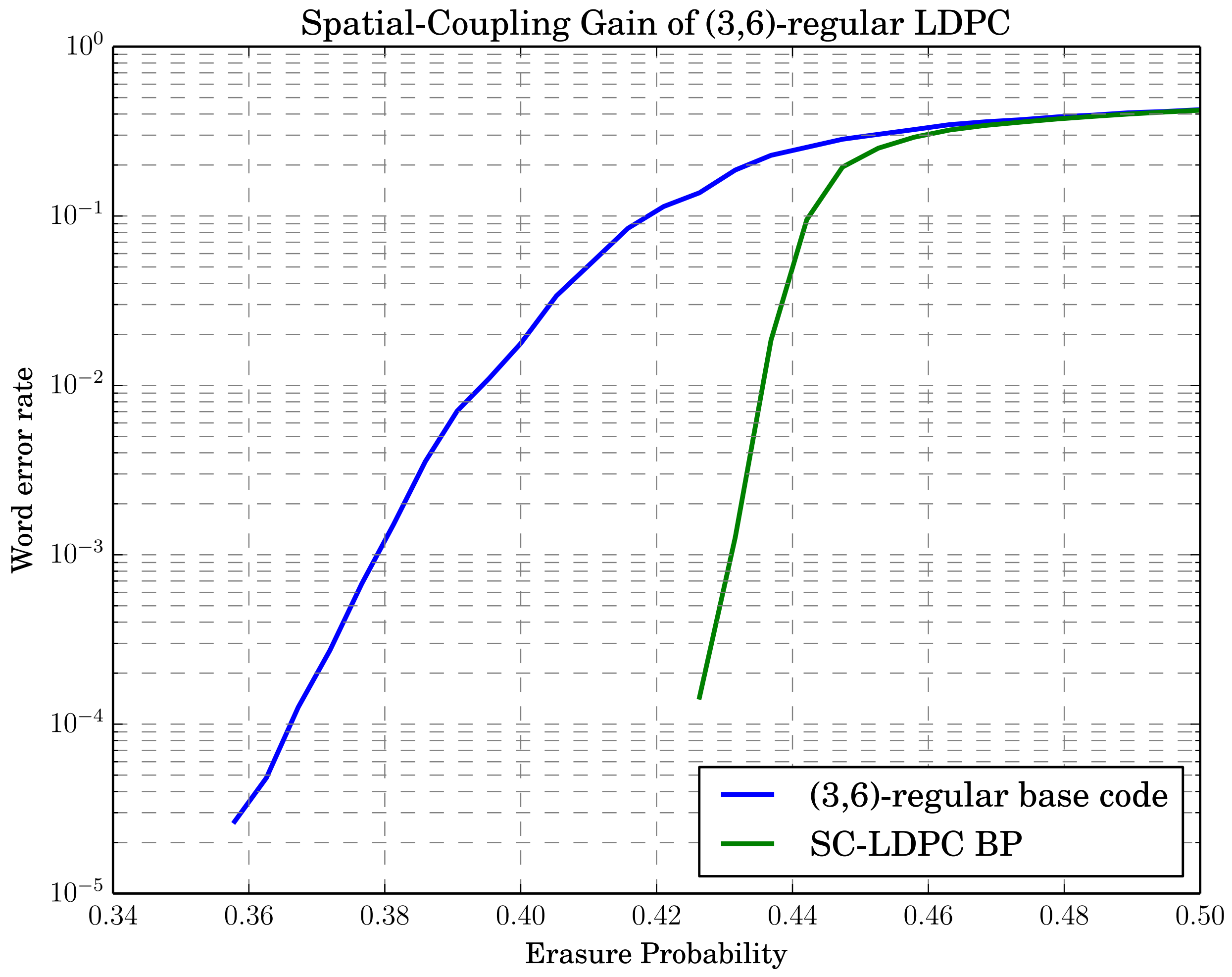

Figure 12 - Performance of SC-LDPC versus base code, data termination 64 blocks

The performance of the window decoder is shown in Figure 13. The WD decoders were run with window sizes of 1, 2, 4, and 8; each window position was run with 30, 60, 120, and 240 decoding iterations respectively. These iteration counts are higher than what was used in [4]; I chose them experimentally because I found they offered significant BER performance gain. Intuitively it makes sense that larger window size may require higher iteration counts, because it will take proportionally more iterations for information to travel from one end of the window to the other.

The performance of the SC-LDPC BP decoder is also shown in this plot. Note that BP decoder performance is the same as WD performance for window size 4 and 8; this agrees with the window size performance claims from [4].

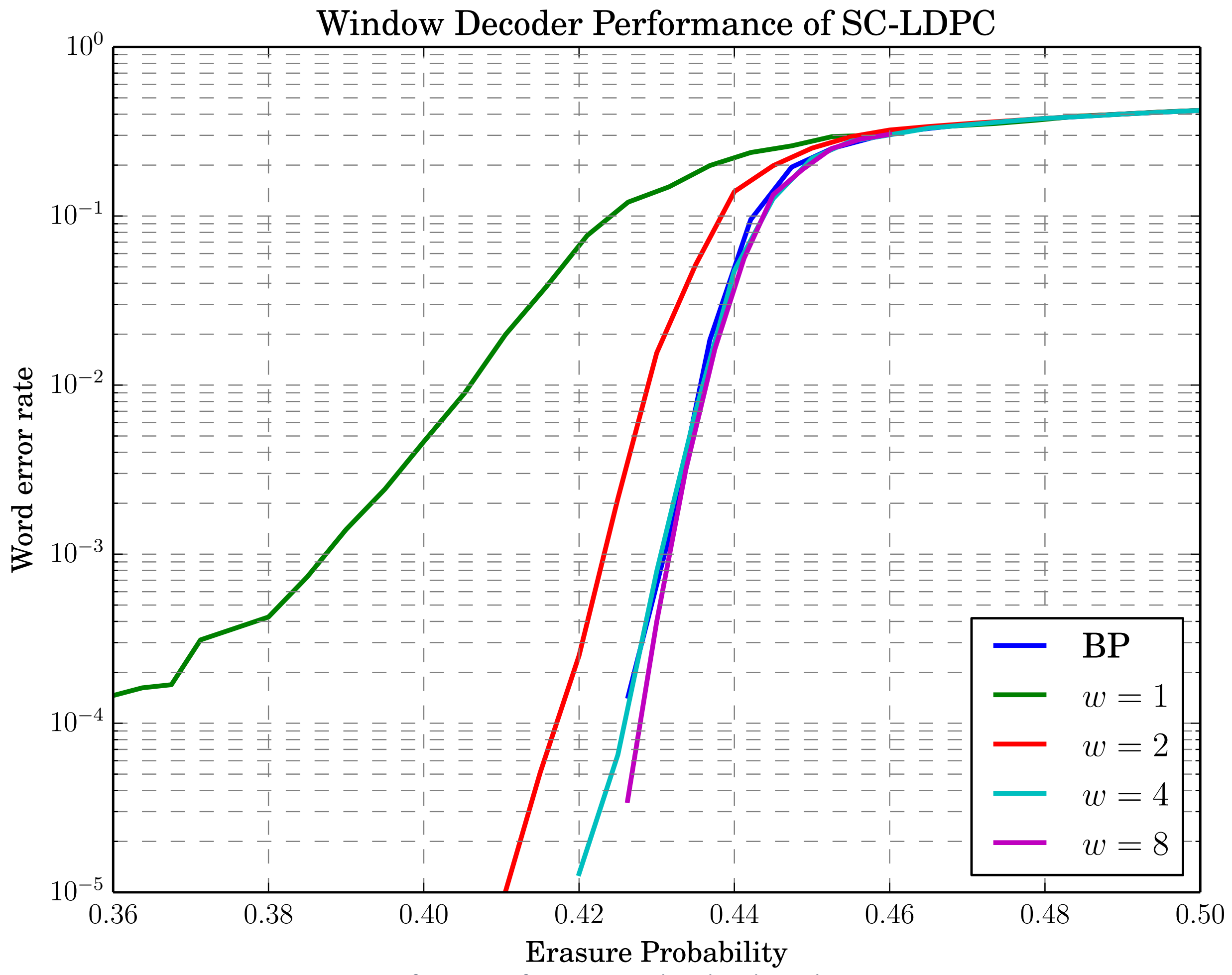

Figure 13 – Performance of SC-LDPC window decoder with n=1024, L=64

It is interesting to note that for $w = 1$ the window decoder has an error floor. I believe this is because for small window sizes the "working" code for each window position is increasingly irregular. This error floor is also seen in [4] for their lowest window size.

The window decoder experiment was also run for a SC-LDPC based on a larger (3,6) regular LDPC-BC with $n = 2500$. The results of this can be seen in Figure 14. In this scenario the WD performance does not seem to match the performance of BP, even for the higher window sizes.

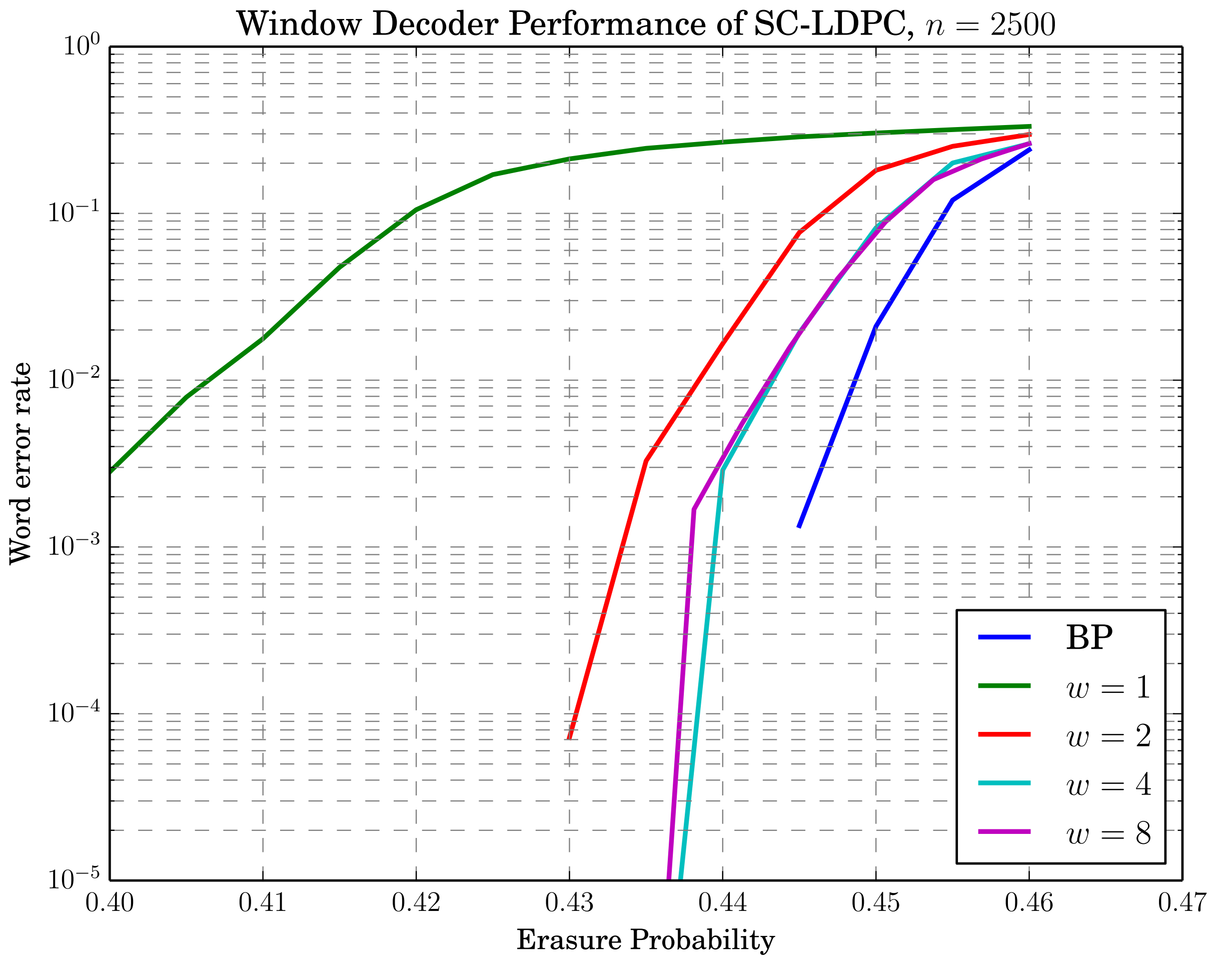

Figure 14 – Performance of SC-LDPC window decoder with n=2500, L=64

The commands used to generate these results are included in the source code package.

- `go_n1024`
- `go_n2500`

The data for these simulations is located in the source code package inside:

- `results/m1024/`
- `results/m2500/`

5 References

[1] Jimenez Felstrom, A.; Zigangirov, K.S., "Time-varying periodic convolutional codes with low-density parity-check matrix," Information Theory, IEEE Transactions on , vol.45, no.6, pp.2181,2191, Sep 1999

[2] Kudekar, S.; Richardson, T.; Urbanke, R.L., "Spatially Coupled Ensembles Universally Achieve Capacity Under Belief Propagation," Information Theory, IEEE Transactions on , vol.59, no.12, pp.7761,7813, Dec. 2013

[3] Pusane, A.E.; Smarandache, R.; Vontobel, P.O.; Costello, D.J., "Deriving Good LDPC Convolutional Codes from LDPC Block Codes," Information Theory, IEEE Transactions on , vol.57, no.2, pp.835,857, Feb. 2011

[4] Iyengar, A.R.; Siegel, P.H.; Urbanke, R.L.; Wolf, J.K., "Windowed decoding of spatially coupled codes," Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on , vol., no., pp.2552,2556, July 31 2011-Aug. 5 2011