Lab 5: Line Coding and Decoding

Objectives:

1) To implement four different types of line coding in Simulink and use them to represent random bits;
2) To achieve more autonomy in creating Simulink models.

Introduction:

In order to transmit data across any sort of channel, it is necessary to encode the information bits into some electrical waveform. The relation of this waveform to the input bits is called *line coding*, and the choice of this waveform can have a substantial impact on the performance of the system. For example, the simplest line coding is called *unipolar NRZ,* which creates a waveform that goes to 0V when the input is 0, and some positive voltage when the input is 1.

Simple forms of line coding such as NRZ are sufficient for some applications, but they can cause problems in others. For instance, suppose your data includes a very long run of 0's or 1's. With no transitions of the voltage level, the receiver may lose sync with the data when it attempts to sample it, and this will cause unnecessary errors. More advanced line codes can embed the clock signal within the data, ensuring that enough transitions occur to maintain sync. Differential line codes will encode the data in transitions rather than levels, which helps in some applications involving phase modulation.

Unlike the previous labs, this lab will expect you to figure out how to construct most of the models on your own. Most of the line codes can be implemented using basic blocks such as gains, sums, and constants, but a couple of them require delay blocks, which will be explained.

Preliminary:

Review the following types of line coding and their power spectral density plots. If the total bandwidth available for the communication system is of 1 MHz, what is the bit rate that each of these line code can support?

Bipolar NRZ (NRZ change)
NRZI (NRZ mark)
Polar RZ (without AMI)
Manchester (Split phase)

Procedure:

Part A – Implementing NRZ (NRZ Change) Coding

To begin constructing line coding models, we must have a source of data bits. An ideal source would be one that returns 0's and 1's with equal probability one sample at a time. The best choice of block to do this is the *Bernoulli Binary Generator*. Add this block to your model and set it up to spit out a new bit every second (sample time = 1). Make sure to remember to set the simulation to use the discrete solver. Also, add a scope to visualize the results.

Between the generator and the scope, we need to create a system that takes in a single bit, either 0 or 1, and outputs a waveform that goes between -1 and 1 for NRZ. This can be done solely by using basic math blocks such as sums, gains, and constants. Once this system is finished, package it up into a submodel so that the top-level model looks clean. The data source and the scope will be shared by all line encoders, so do not include those in the submodel.

Part B – Implementing NRZ Mark (NRZI) Coding

NRZI line coding is a bit trickier, as the current output will depend on what the output was *in the previous sample*. Basically, if the current data bit is 0, the output voltage level remains constant, but if it is 1, the voltage level transitions to its opposite state. Of course, in order to keep the voltage level the same (or change it), the model must know what value it had to begin with. The block that will accomplish this is called the *unit delay*. The output of this block is simply whatever the input to it was in the previous sample. This creates a sort of memory in the system, which is necessary because it must "remember" what the last voltage level was in order to be able to keep it the same or change it.

To use this block effectively, its input must be *the final output of the line encoder*. That way, when the simulation advances by a sample, the previous output level of the line encoder will have shifted to the output of the unit delay block. By using this value along with the value of the current bit, math functions can be used to create the appropriate waveform.

An example of this sort of "feedback loop" is shown in Figure 1. Note that this is *not* the solution to the line coder; it is merely an example of how the blocks should be arranged. The example arrangement as it is shown is simply an integrator; the real solution will involve a *subtraction* rather than an addition, as well as other math blocks. When you finish this system, package it into a submodel as well.

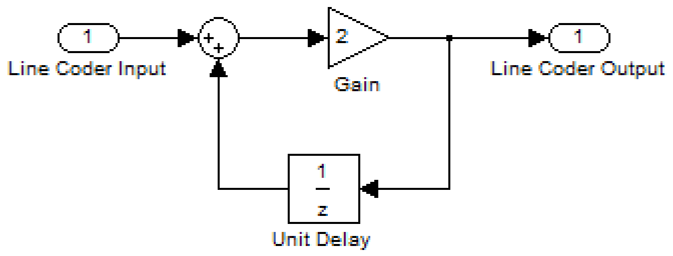

Figure 1 – Example arrangement for use of unit delay block

Part C – Implementing Polar RZ Coding

For polar RZ line coding, a clock is required in addition to the bit source. This clock will have the same period as the data rate, and its transitions are added to the line encoder output. To create this clock, add a *Pulse Generator* block. Set it to be sample based with sample time 0.5, period 2, and pulse width 1. This will make a clock that will alternate between 0 and 1 one time each for every data bit.

Using this clock as well as the data source, standard polar RZ line coding (*not* AMI) can be implemented using simple math blocks (no delays are required). You may implement the AMI version if you wish, but this will require a delay block to track the previous output polarity.

Part D – Implementing Differential Manchester Coding

The most difficult line code to implement is the Differential Manchester code. In ordinary Manchester code, for every data bit, the line encoder will use the clock to output a high-to-low transition on an input of 1 and a low-to-high transition on an input of 0. When this is made differential, the type of transition stays the same on a 0 and switches on a 1. You may refer to Figure 3 for a pictorial example of this.

When all encoders are completed and simulated, you should get a top-level model similar to Figure 2 and scope results similar to Figure 3. After confirming that the line codes are correct for different sequences of input bits, take a screenshot of your top-level model as well as the inside of every submodel and include them in your report. Include the scope output as well, and discuss in your report how these results compare to the theoretical results.

For extra credit (10 points each), you may design decoders for any of the line encoders created above. The output of any line decoder should give back the original bit sequence. Include scope results for these if you do them, and remember to submit all model files with the report.

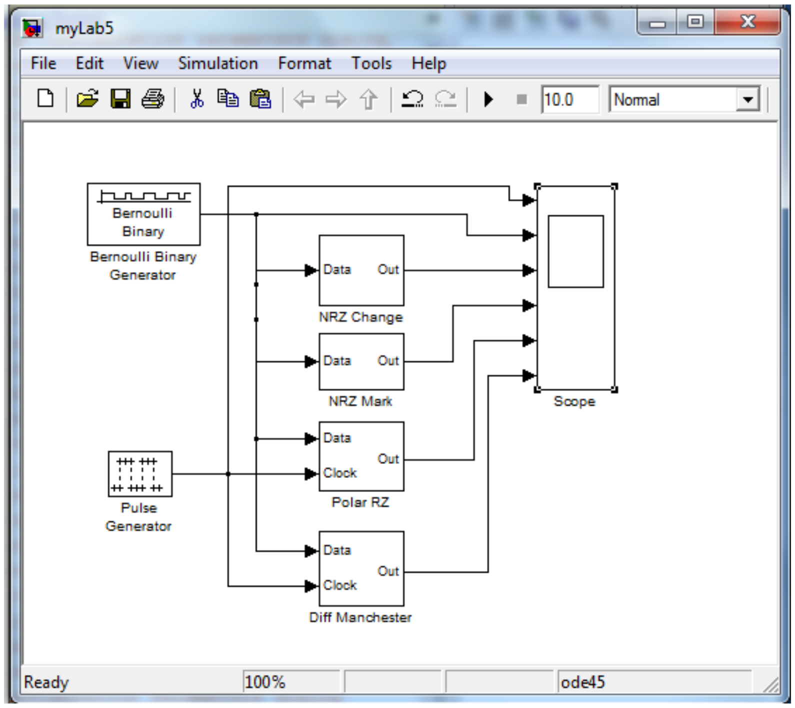

Figure 2 – Example of final top-level model

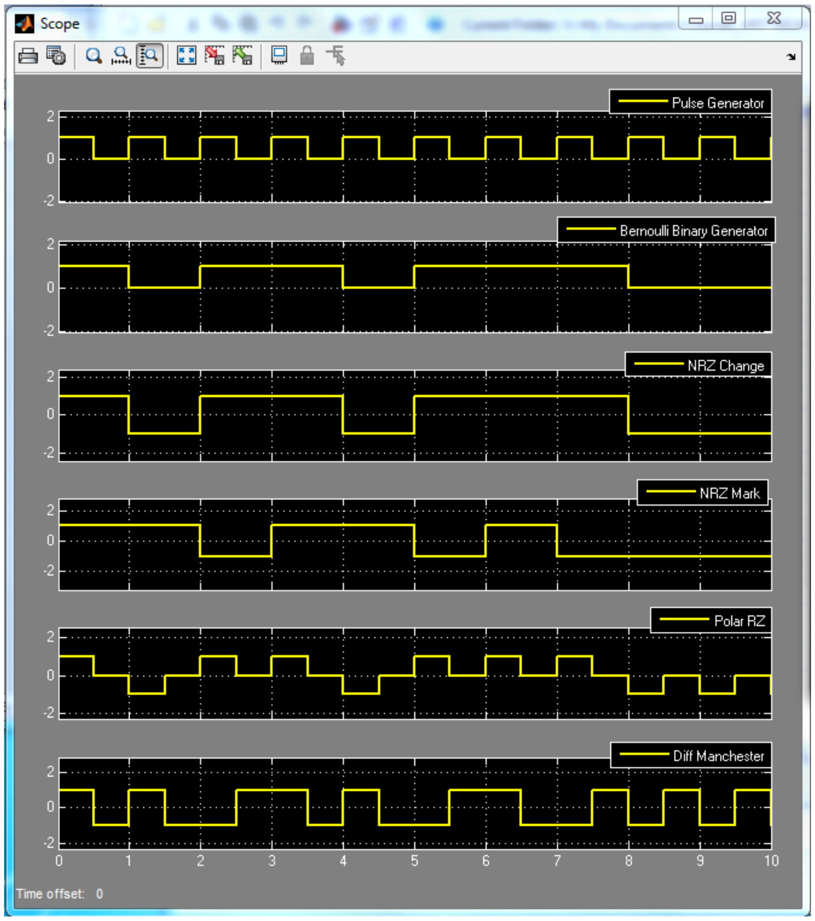

Figure 3 – Example of scope results for all line codes

Post-Lab Questions:

Q1. For the NRZ change line code, what is an advantage of using bipolarity as opposed to unipolarity? Why could it be better to have a negative voltage level as opposed to zero volts?

Q2. NRZ mark uses differential coding to encode the data in transitions. When the receiver is first turned on, having no previous information, can it immediately decipher the first bit received?

Q3. Alternate mark inversion (AMI) RZ leaves the voltage level at zero on a “0” bit and alternates between positive and negative voltages for successive “1” bits. What is a possible advantage or disadvantage of doing this?

Q4. Differential Manchester combines most of the advantages of the above line codes at the cost of high complexity and high bandwidth. To what applications might this line code be best suited?

Q5. If you are to plot the spectral density of the line codes, what changes have to be made to your model to plot the proper spectral graph? If you tried this part, send a separate model for this part in your submission.