**University of Science and Technology – Zewail City**

**Communications and Information Engineering Program**

**CIE 337: Communication Theory**

**Project 2**

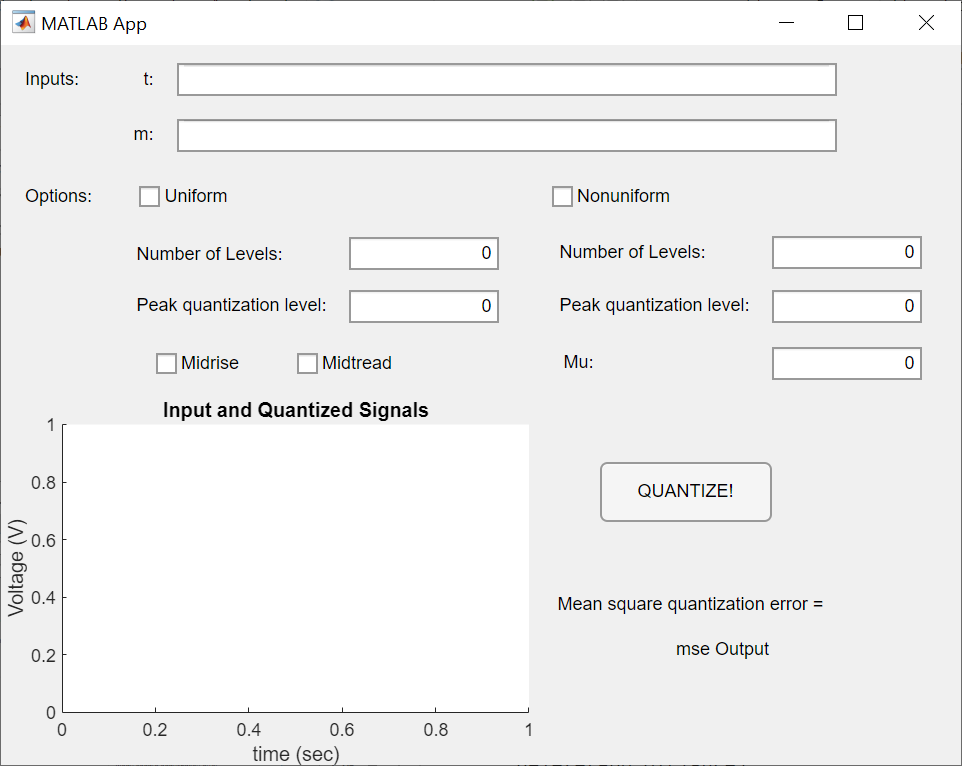
**Alaa Alajmy 201700095**

**Moemen Gaafar 201700873**

**1. Part A**

In this part of the project, we build a signal quantizer application using Matlab and build its GUI using Matlab’s AppDesigner.

The application interface:

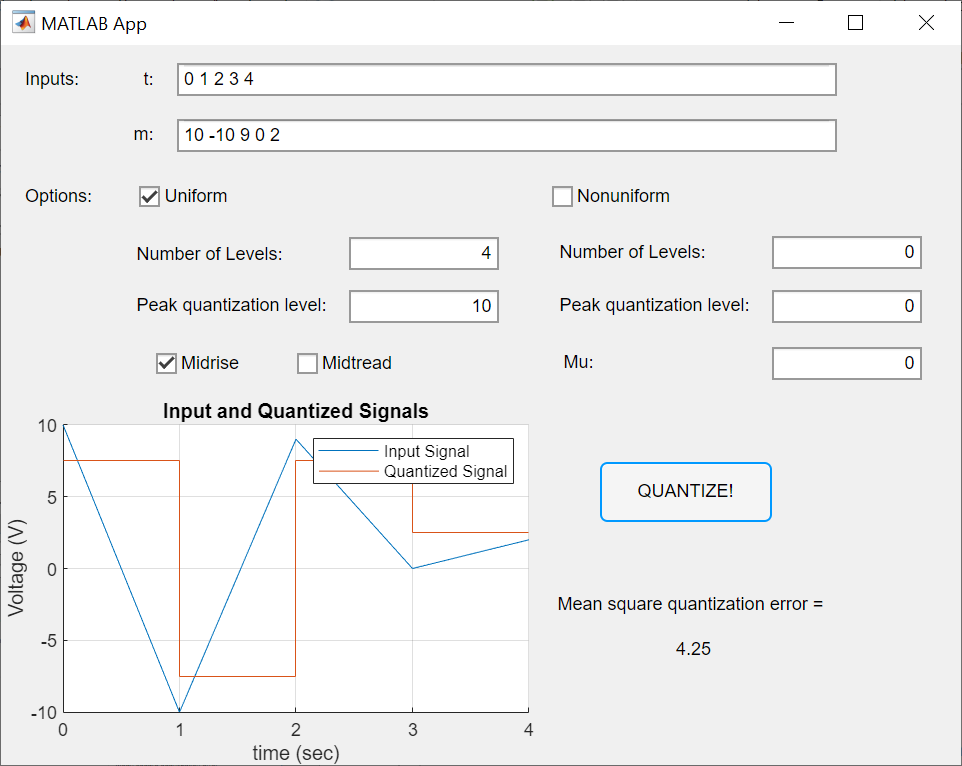


To quantize a signal m, the user must enter the signal and its corresponding time steps as arrays of doubles. The user then selects whether to perform uniform or non-uniform quantization and inputs each quantization type’s parameters. Finally, the user presses the “”QUANTIZE!” button and the application plots the input signal m and its quantized output on the “Input and Quantized Signals” graph and prints the resulting mean square error (MSE). The application also returns an array of the quantization steps, the quantized signal, and the MSE to the base Matlab workspace.

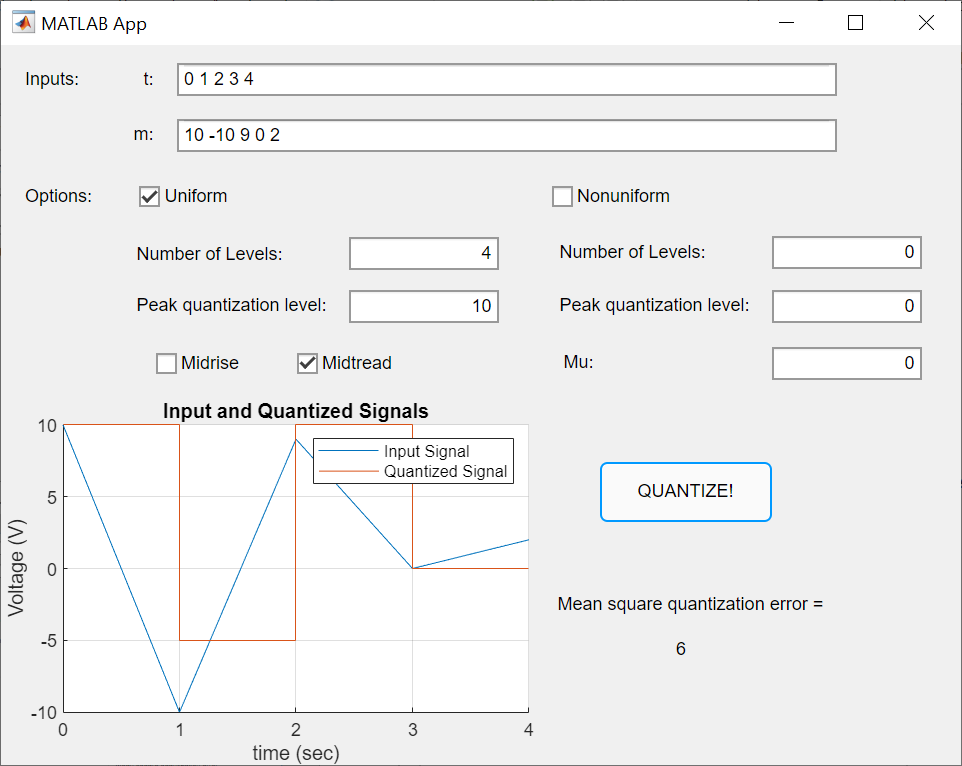
1. **Uniform Quantization**

To perform uniform quantization, the user has to input the number of quantization levels and the peak quantization level then select midrise or mid-tread quantization.

A simple example of midrise quantization:



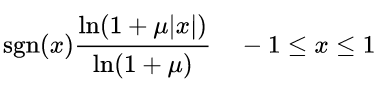
A simple example of mid-tread quantization:



1. **Non-Uniform Quantization**

To perform non-uniform quantization, the user must input a value for mu, the number of quantization levels, and the peak quantization level. The application perform mu-law non-uniform quantization (mid-tread quantization where quantization steps are non-uniformly spaced).

First, the application normalizes the input signal and compresses it using the mu-law:



Then, the application performs uniform quantization on the compressed signal with a peak value of 1. Finally, the quantized signal is expanded using the inverse mu law:

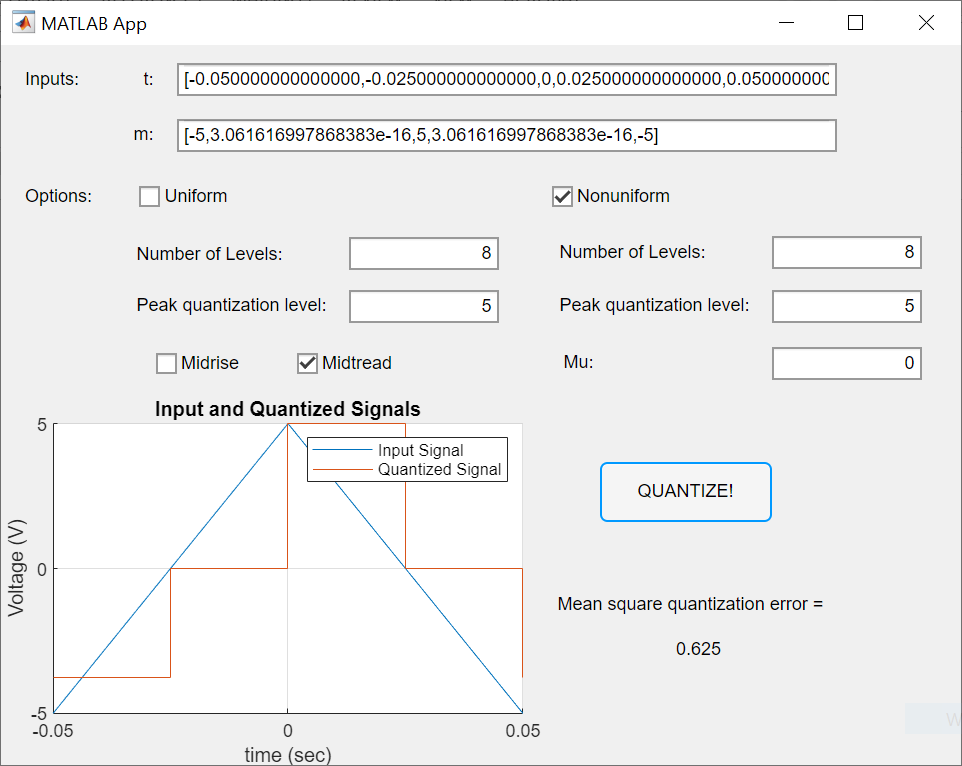


The resultant is denormalized and plotted.

1. **Test Examples**

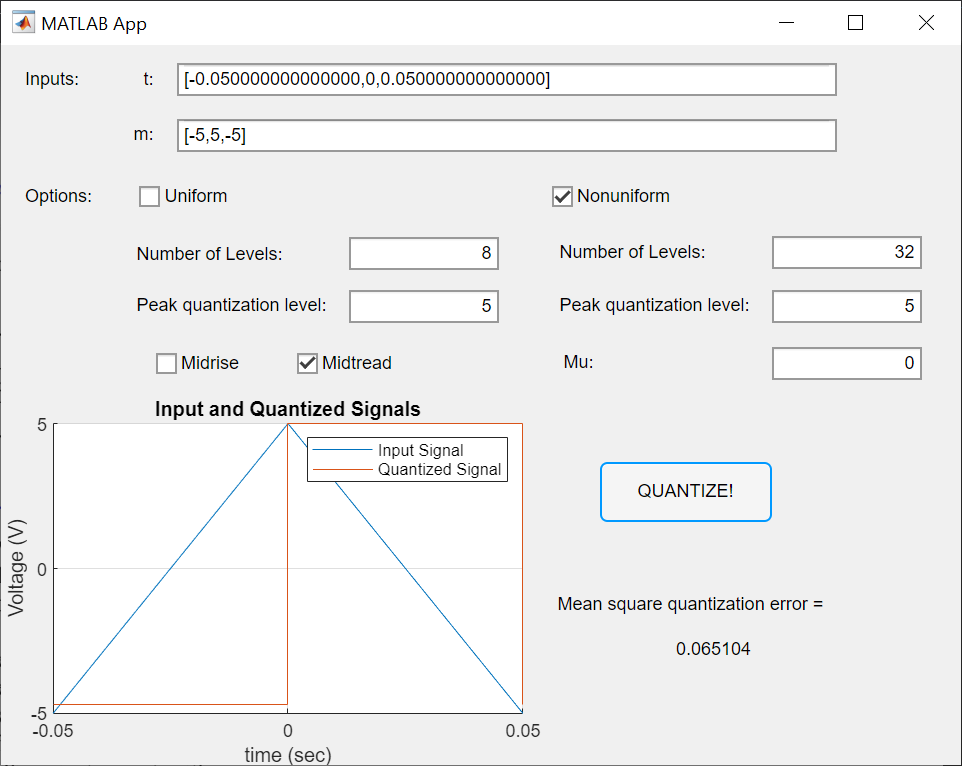
Note: while the project description requires we take just one cycle of the signal, stopping at exactly 1/20 seconds usually leads us to take 1 extra point from the next cycle. Although in some cases this hugely affects the resultant MSE, we assume that’s fine because the sampling frequency is too low and we can’t really give up this one point.

* Sampling frequency = 40 Hz, mu = 0, L = 8, mp = 5:



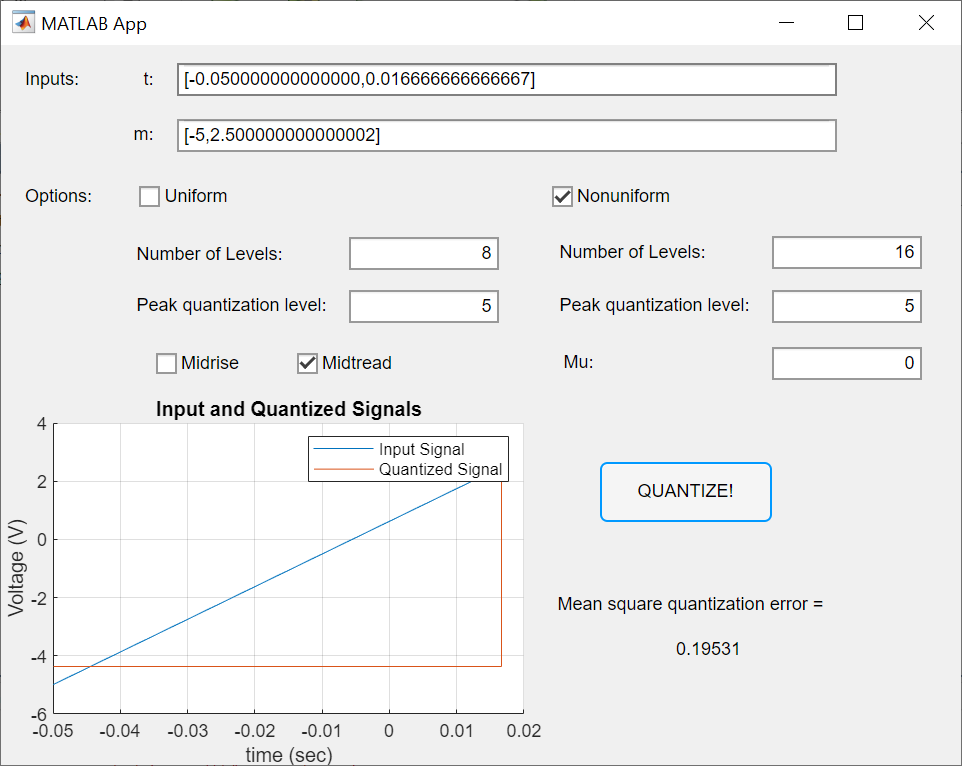
With mu = 0, this case applied typical uniform mid-tread quantization. Since we have an even number of steps, the points with value = -mp did not have a step exactly equal their value and thus were quantized to a larger number leading to a significant error.

* Sampling frequency = 20 Hz, mu = 0, L = 32, mp = 5:



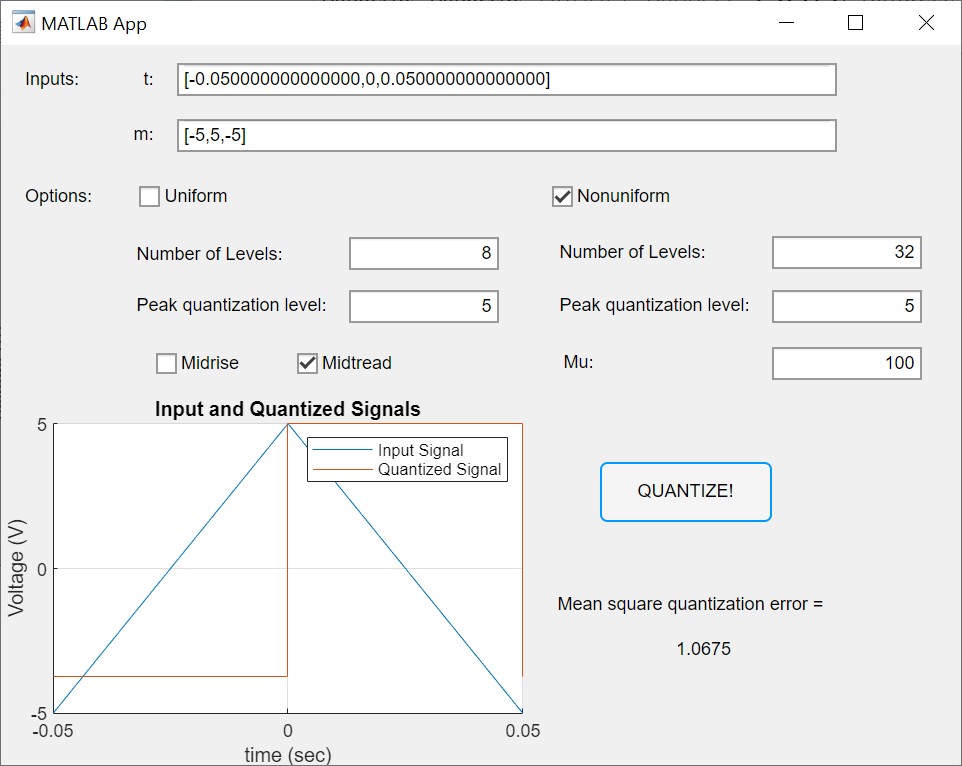
Again, this is an application of uniform mid-tread quantization. The large number of steps made the quantization of points at -5 almost accurate leading to a small error.

* Sampling frequency = 15 Hz, mu = 0, L = 16, mp = 5:



Yet another uniform mid-tread quantization! Because of the extremely low sampling frequency, only two points from the cycle were considered. Again, the error results from the point at -5 V. This error lies between the errors of the two previous cases, since the number of levels 16 is larger than the number of levels in case 1 and smaller that in case 2.

* Sampling frequency = 20 Hz, mu = 100, L = 32, mp = 5:

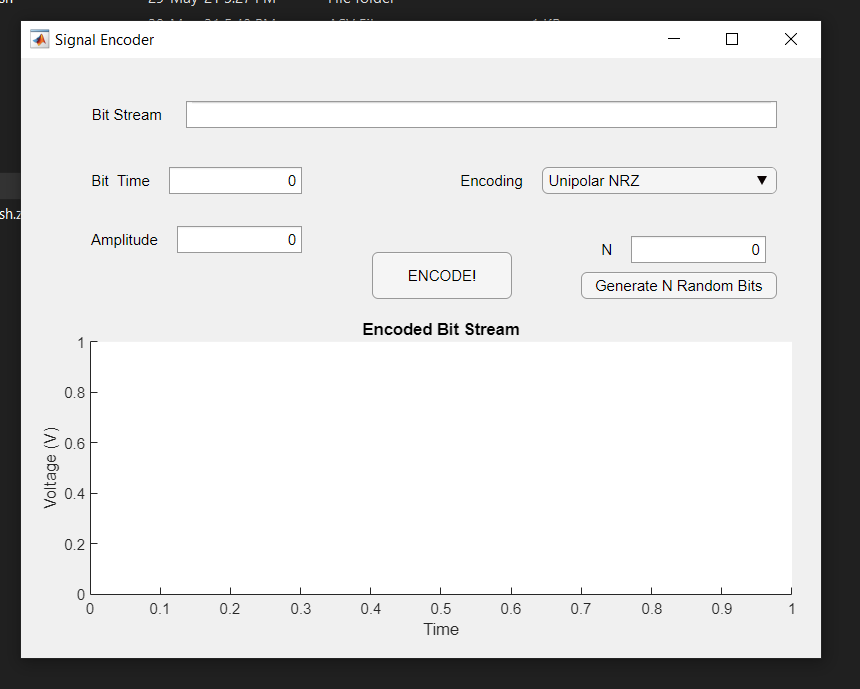


Here we do use a nonzero value of mu! Yet, the error is incredibly huge because the since all three values of this case are equal negative or positive the peak value, where quantization steps are most sparse.

**2. PART B**

In this part of the project, we build a signal encoder application using Matlab and build its GUI using Matlab’s AppDesigner.

The application interface:



To encode a bit stream, the user can either enter a bit stream or randomly generate N bits. The user then enters the bit time and amplitude and picks the encoding type from a drop-down list. Finally, the user presses the “ENCODE!” button and the application plots the encoded bit stream. The application also returns an array of the original and the encoded bit streams to the base Matlab workspace.

1. **Return-To-Zero Codes**

In return-to-zero codes, like Polar RZ and Alternate Mark Inversion codes, the bit takes its appropriate amplitude for 25% of the bit’s duration and returns to zero for the rest of it.

1. **Manchester Code**

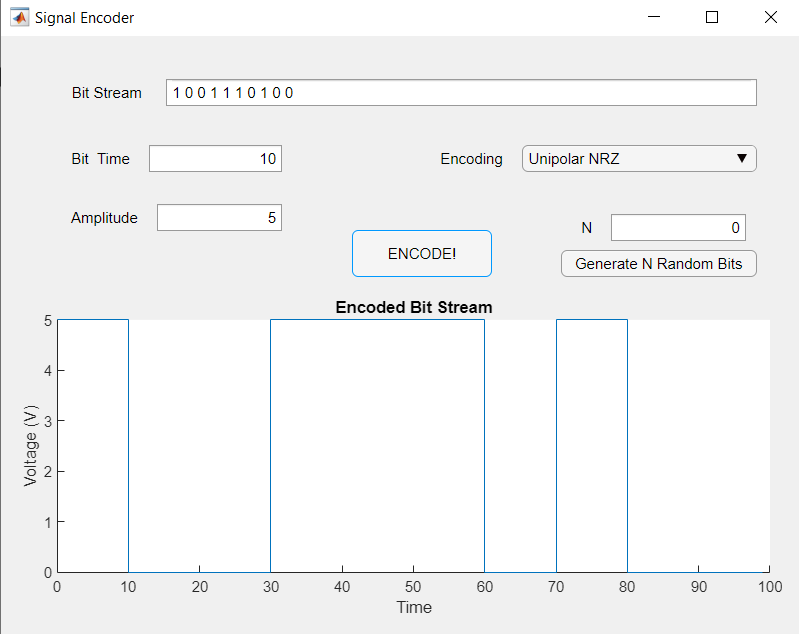
In Manchester code, the bit reverses its amplitude after 50% of the bit’s duration.

1. **Test Examples**

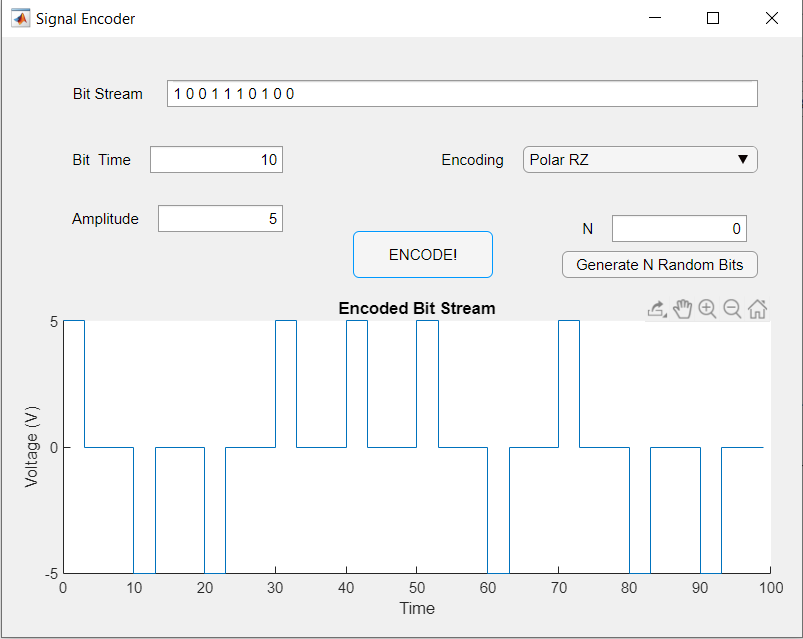
We generate a random 10-bit stream and apply the four available encodings on it. The generated bit stream is:

We use a bit time of 10 time steps and an amplitude of 5 volts.

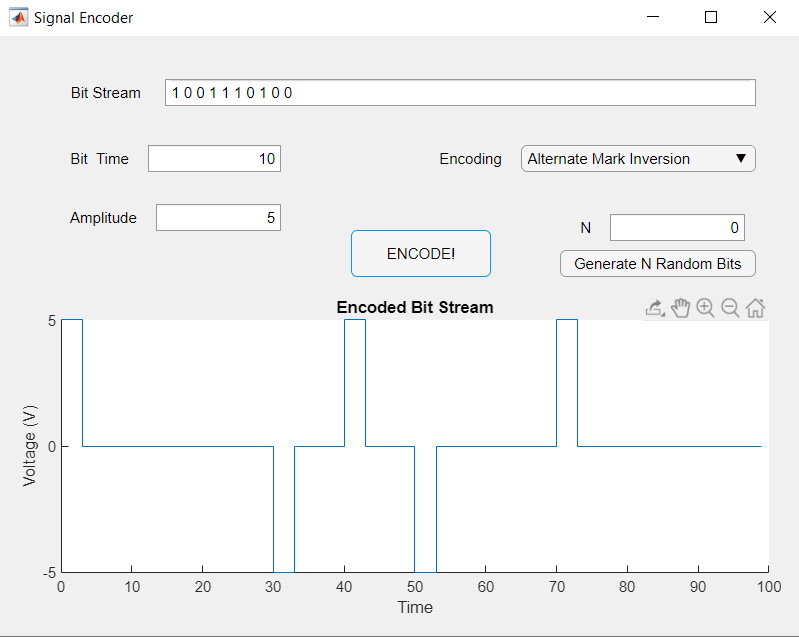
* Unipolar NRZ



* Polar RZ



* Alternate Mark Inversion



* Manchester Code

