# Bit-Error-Rate Simulation Using Matlab

James E. Gilley
Chief Scientist
Transcrypt International, Inc.
jgilley@transcrypt.com

August 19, 2003

## 1 Introduction

Matlab is an ideal tool for simulating digital communications systems, thanks to its easy scripting language and excellent data visualization capabilities. One of the most frequent simulation tasks in the field of digital communications is bit-error-rate testing of modems. The bit-error-rate performance of a receiver is a figure of merit that allows different designs to be compared in a fair manner. Performing bit-error-rate testing with Matlab is very simple, but does require some prerequisite knowledge.

## 2 Properties of Sampled Signals

In Matlab, we represent continuous-time signals with a sequence of numbers, or samples, which are generally stored in a vector or an array. Before we can perform a bit-error-rate test, we must precisely understand the meaning of these samples. We must know what aspect of the signal the value of these samples represents. We must also know the time interval between successive samples.

For communications simulations, the numeric value of the sample represents the amplitude of the continuous-time signal at a specific instant in time. We assume this amplitude is a measurement of voltage, though it could just as easily be a measurement of current.

The time between successive samples is, by definition, $T_s$. This tells us how often the continuous-time signal was sampled. Instead of specifying $T_s$, we usually specify the sampling frequency, $f_s$, which is the inverse of $T_s$.

For convenience, we will always associate a sample value of 1.0 with a voltage of exactly one volt. Furthermore, we will always assume a resistance of exactly one ohm. This allows us to dispense with the notion of resistance altogether. For our simulations, we will represent a continuous time signal as an array of samples, the numeric value of which is in units of volts, referenced to a resistance of one ohm. Usually, the sampling frequency is 8 KHz, but other sampling frequencies are also in common use, so the sampling frequency should always be specified.

## 2.1 Power

Suppose we have a signal $x(n)$, where $n$ is an index of the sample number. We define the instantaneous power of the signal as:

$$P_{ins} \equiv x^2(n).$$

In other words, the instantaneous power of a sample is just the value of that sample squared. Since the units of the sample are volts, the units of the power are watts.

A far more useful quantity is the average power, which is simply the average of the instantaneous power of every sample in the signal. For signal $x(n)$, of $N$ samples, we have:

$$P_{ave} \equiv \frac{1}{N} \sum_{n=1}^{N} x^2(n). \tag{1}$$

Note that this is simply the sum of the square of all samples, divided by the number of samples. One way to compute the average power, 'pav', of signal 'x', using Matlab is:

`pav = sum(x.^2)/length(x).`

If our signal has a mean of zero, or in other words, no DC component, we can find the average power of the signal by taking its variance. This works because:

$$\sigma(x) \equiv E[x^2] - (E[x])^2,$$

which states: the variance of a signal is the mean of its square, minus the square of its mean. If the mean is zero, the variance is just the mean of the square, exactly the same as the average power. Therefore, if a signal has no DC value, we can compute its average power by finding its variance.

We need to be careful using the variance to find the average power of a signal. This technique only works if the mean of the signal is zero. If the mean is not zero, we must use (1), which always works, regardless of whether the mean is zero or not.

## 2.2 Energy

By definition, power is the time derivative of energy; or equivalently, energy is the time integral of power. For sampled signals, integration reduces to a summation.

Since energy is the product of power and time, the total energy of a signal must be equal to its average power multiplied by its duration. Furthermore, the duration of a signal is its length in samples, divided by the sampling frequency, in samples per second. Therefore:

$$\begin{aligned} E_{tot} &= P_{ave} \cdot t, \\ &= \frac{1}{N} \sum_{n=1}^{N} x^2(n) \cdot \frac{N}{f_s}, \\ &= \frac{1}{f_s} \sum_{n=1}^{N} x^2(n). \end{aligned} \tag{2}$$

The Matlab command for finding the total energy, 'et', of signal 'x', that has sampling rate 'fs', is:

$$\texttt{et} = \texttt{sum(x.\^{}2)/fs}.$$

# 3 Simulation Overview

Bit-error-rate testing requires a transmitter, a receiver, and a channel. We begin by generating a long sequence of random bits, which we provide as input to the transmitter. The transmitter modulates these bits onto some form of digital signaling, which we will send though a simulated channel. We simulate the channel by adding a controlled amount of noise to the transmitted signal. This noisy signal then becomes the input to the receiver. The receiver demodulates the signal, producing a sequence of recovered bits. Finally, we compare the received bits to the transmitted bits, and tally up the errors.

Bit-error-rate performance is usually depicted on a two dimensional graph. The ordinate is the normalized signal-to-noise ratio (SNR) expressed as $E_b/N_0$: the energy-per-bit divided by the one-sided power spectral density of the noise, expressed in decibels (dB). The abscissa is the bit-error-rate, a dimensionless quantity, usually expressed in powers of ten.

To create a graph of bit-error-rate versus SNR, we plot a series of points. Each of these points requires us to run a simulation at a specific value of SNR. To obtain the bit-error-rate at a specific SNR, we follow the procedure given below.

# 4 Simulation Procedure

## 4.1 Run Transmitter

The first step in the simulation is to use the transmitter to create a digitally modulated signal from a sequence of pseudo-random bits. Once we have created this signal, $x(n)$, we need to make some measurements of it.

## 4.2 Establish SNR

The signal-to-noise-ratio (SNR), $E_b/N_0$, is usually expressed in decibels, but we must convert decibels to an ordinary ratio before we can make further use of the SNR. If we set the SNR to $m$ dB, then $E_b/N_0 = 10^{m/10}$.

Using Matlab, we find the ratio, 'ebn0', from the SNR in decibels, 'snrdb', as:

$$\texttt{ebn0} = \texttt{10\^{}(snrdb/10)}.$$

Note that $E_b/N_0$ is a dimensionless quantity.

## 4.3 Determine $E_b$

Energy-per-bit is the total energy of the signal, divided by the number of bits contained in the signal. We can also express energy-per-bit as the average signal power

multiplied by the duration of one bit. Either way, the expression for $E_b$ is:

$$E_b = \frac{1}{N \cdot f_{bit}} \sum_{n=1}^{N} x^2(n),$$

where $N$ is the total number of samples in the signal, and $f_{bit}$ is the bit rate in bits-per-second.

Using Matlab, we find the energy-per-bit, 'eb', of our transmitted signal, 'x', that has a bit rate 'fb', as:

`eb = sum(x.^2)/(length(x) * fb).`

Since our signal, $x(n)$, is in units of volts, the units of $E_b$ are Joules.

## 4.4 Calculate $N_0$

With the SNR and energy-per-bit now known, we are ready to calculate $N_0$, the one-sided power spectral density of the noise. All we have to do is divide $E_b$ by the SNR, providing we have converted the SNR from decibels to a ratio.

Using Matlab, we find the power spectral density of the noise, 'n0', given energy-per-bit 'eb', and SNR 'ebn0', as:

`n0 = eb/ebn0.`

The power spectral density of the noise has units of Watts per Hertz.

## 4.5 Calculate $\sigma_n$

The one-sided power spectral density of the noise, $N_0$, tells us how much noise power is present in a 1.0 Hz bandwidth of the signal. In order to find the variance, or average power, of the noise, we must know the noise bandwidth.

For a real signal, $x(n)$, sampled at $f_s$ Hz, the noise bandwidth will be half the sampling rate. Therefore, we find the average power of the noise by multiplying the power spectral density of the noise by the noise bandwidth:

$$\sigma_n = \frac{N_0 \cdot f_s}{2},$$

where $\sigma_n$ is the noise variance in W, and $N_0$ is the one-sided power spectral density of the noise in W/Hz.

Using Matlab, the average noise power, 'pn', of noise having power spectral density 'n0', and sampling frequency 'fs', is calculated as:

`pn = n0 * fs/2.`

The average noise power is in units of Watts.

## 4.6 Generate Noise

Although the communications toolbox of Matlab has functions to generate additive white Gaussian noise, we will use one of the standard built-in functions to generate AWGN. Since the noise has a zero mean, its power and its variance are identical. We need to generate a noise vector that is the same length as our signal vector $x(n)$, and this noise vector must have variance $\sigma_n$ W.

The Matlab function 'randn' generates normally distributed random numbers with a mean of zero and a variance of one. We must scale the output so the result has the desired variance, $\sigma_n$. To do this, we simply multiply the output of the 'randn' function by $\sqrt{\sigma_n}$. We can generate the noise vector 'n', as:

$$\texttt{n} = \texttt{sqrt(pn)} * \texttt{randn(1,length(x))};.$$

Like the signal vector, the samples of the noise vector have units of volts.

## 4.7 Add Noise

We create a noisy signal by adding the noise vector to the signal vector. If we are running a fixed-point simulation, we will need to scale the resulting sum by the reciprocal of the maximum absolute value, so the sum stays within amplitude limits of $\pm 1.0$. Otherwise, we can simply add the signal vector 'x' to the noise vector 'n' to obtain the noisy signal vector 'y' as:

$$\texttt{y} = \texttt{x} + \texttt{n};.$$

## 4.8 Run Receiver

Once we have created a noisy signal vector, we use the receiver to demodulate this signal. The receiver will produce a sequence of demodulated bits, which we must compare to the transmitted bits, in order to determine how many demodulated bits are in error.

## 4.9 Determine Offset

Due to filtering and other delay-inducing operations typical of most receivers, there will be an offset between the received bits and the transmitted bits. Before we can compare the two bit sequences to check for errors, we must first determine this offset. One way to do this is by correlating the two sequences, then searching for the correlation peak.

Suppose our transmitted bits are stored in vector 'tx', and our received bits are stored in vector 'rx'. The received vector should contain more bits than the transmitted vector, since the receiver will produce (meaningless) outputs while the filters are filling and flushing. If the length of the transmitted bit vector is $l_{tx}$, and the length of the received vector is $l_{rx}$, the range of possible offsets is between zero and $l_{rx} - l_{tx} - 1$. We can find the offset by performing a partial cross-correlation between the two vectors.

Using Matlab, we can create a partial cross-correlation, 'cor', from bit vectors 'tx' and 'rx', with the following loop:

$$\begin{aligned}&\texttt{for lag} = 1 : \texttt{length(rx)} - \texttt{length(tx)} - 1,\\ &\ \texttt{cor(lag)} = \texttt{tx} * \texttt{rx(lag : length(tx)} - 1 + \texttt{lag)}';\\ &\texttt{end}.\end{aligned}$$

The resulting vector, 'cor', is a partial cross-correlation of the transmitted and received bits, over the possible range of lags: $0 : l_{rx} - l_{tx} - 1$.

We need to find the location of the maximum value of 'cor', since this will tell us the offset between the bit vectors. Since Matlab numbers array elements as $1 : N$ instead of as $0 : N-1$, we need to subtract one from the index of the correlation peak.

Using Matlab, we find the correct bit offset, 'off', as:

$$\texttt{off} = \texttt{find(cor} == \texttt{max(cor))} - 1.$$

## 4.10 Create Error Vector

Once we know the offset between the transmitted and received bit vectors, we are ready to calculate the bit errors. For bit values of zero and one, a simple difference will reveal bit errors. Wherever there is a bit error, the difference between the bits will be $\pm 1$, and wherever there is not a bit error, the difference will be zero.

Using Matlab, we calculate the error vector, 'err', from the transmitted bit vector, 'tx', and the received bit vector, 'rx', having an offset of 'off', as:

$$\texttt{err} = \texttt{tx} - \texttt{rx(off} + 1 : \texttt{length(tx)} + \texttt{off)};.$$

## 4.11 Count Bit Errors

The error vector, 'err' contains non-zero elements in the locations where there were bit errors. We need to tally the number of non-zero elements, since this is the total number of bit errors in this simulation.

Using Matlab, we calculate the total number of bit errors, 'te', from the error vector 'err' as:

$$\texttt{te} = \texttt{sum(abs(err))}.$$

## 4.12 Calculate Bit-Error-Rate

Each time we run a bit-error-rate simulation, we transmit and receive a fixed number of bits. We determine how many of the received bits are in error, then compute the bit-error-rate as the number of bit errors divided by the total number of bits in the transmitted signal.

Using Matlab, we compute the bit-error-rate, 'ber', as:

$$\texttt{ber} = \texttt{te/length(tx)},$$

where 'te' is the total number of bit errors, and 'tx' is the transmitted bit vector.

# 5 Simulation Results

Performing a bit-error-rate simulation can be a lengthy process. We need to run individual simulations at each SNR of interest. We also need to make sure our results are statistically significant.

## 5.1 Statistical Validity

When the bit-error-rate is high, many bits will be in error. The worst-case bit-error-rate is 50 percent, at which point, the modem is essentially useless. Most communications systems require bit-error-rates several orders of magnitude lower than this. Even a bit-error-rate of one percent is considered quite high.

We usually want to plot a curve of the bit-error-rate as a function of the SNR, and include enough points to cover a wide range of bit-error-rates. At high SNRs, this can become difficult, since the bit-error-rate becomes very low. For example, a bit-error-rate of $10^{-6}$ means only one bit out of every million bits will be in error. If our test signal only contains 1000 bits, we will most likely not see an error at this bit-error-rate.

In order to be statistically significant, each simulation we run must generate some number of errors. If a simulation generates no errors, it does not mean the bit-error-rate is zero; it only means we did not have enough bits in our transmitted signal. As a rule of thumb, we need about 100 (or more) errors in each simulation, in order to have confidence that our bit-error-rate is statistically valid. At high SNRs, this can require a test signal containing millions, or even billions of bits.

## 5.2 Plotting

Once we perform enough simulations to obtain valid results at all SNRs of interest, we will plot the results. We begin by creating vectors for both axes. The X-axis vector will contain SNR values, while the Y-axis vector will contain bit-error-rates. The Y-axis should be plotted on a logarithmic scale, whereas the X-axis should be plotted on a linear scale.

Supposing our SNR values are in vector 'xx', and our corresponding bit-error-rate values are in vector 'yy', we use Matlab to plot:

```
semilogy(xx,yy,'o').
```

Fig. 1 shows an example of a plot of the results of a bit-error-rate simulation.

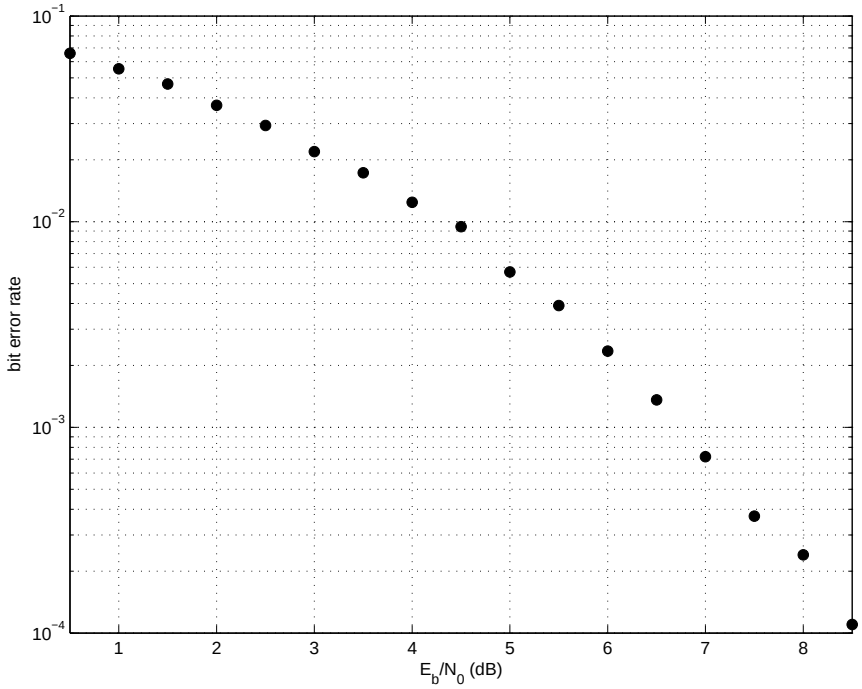


Figure 1: Typical Bit-Error-Rate Plot