COSC 5P01 - Project

# Code Channel Tester

#### An expandable utility for dynamic testing of error-correcting codes.

Michael Gircys

mg12vp - 5269550

COSC 5P01 – Coding Theory

Prof. Sheridan Houghten

2016-01-11

Table of Contents

[Code Channel Tester 1](#_Toc440261032)

[User Guide 3](#_Toc440261033)

[Program Data Flow 4](#_Toc440261034)

[Included Plugins 5](#_Toc440261035)

[Channel: Simple Noise (Plugin\_Channel\_SimpleNoise) 5](#_Toc440261036)

[Channel: Perfect Transmission (Plugin\_Channel\_Perfect) 6](#_Toc440261037)

[Channel: Manual Error Specification (Plugin\_Channel\_Manual) 6](#_Toc440261038)

[Code: Plaintext (Plugin\_Code\_Plaintext) 7](#_Toc440261039)

[Code: Repetition (Plugin\_Code\_Repetition) 7](#_Toc440261040)

[DataSource: All Values of Length n (Plugin\_DataSource\_Exhaustive) 7](#_Toc440261041)

[DataSource: Random Bytes (Plugin\_DataSource\_Random) 8](#_Toc440261042)

[DataSource: Samples from File (Plugin\_DataSource\_SamplesFromFile) 8](#_Toc440261043)

[SDK How-to 9](#_Toc440261044)

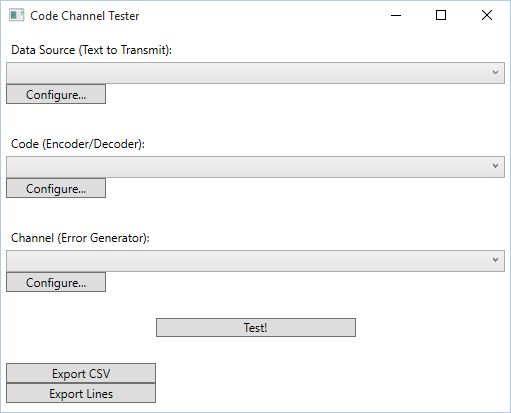
[Future Work 10](#_Toc440261045)

[References: 11](#_Toc440261046)

## User Guide

Code Channel Tester is an error-correcting code tester which permits for a selectable source of transmission strings, code, and transmission channel.

The configurable system will use the data source selected to produce an enumeration of messages (bursts). These messages are encoded and decoded using a selecting code. Before decoding, the encoded message is passed over a selected channel, which may produce noise and errors into the transmission stream.



Each selectable data source, code, or channel, may permit or require additional configuration. The ability to configure an object is performed by pressing the “Configure” button beneath the relevant object after a data source, code, or, channel is selected.

By pressing test on a configured system, a battery of tests (one for each of burst produced by the data source) will be performed, running a message through the full encoder, channel, decoder data flow. Successful transmission and coding rate can be determined through a stored test log.

When the tests have completed, an export of the testing logs can be produced in CSV form through the Export\_CSV button. An alternative is the Export\_Lines export, which will display each column of the testing log on a separate line (for easier comparison of data changes with manual inspection).

The system data flow is outlined on the next page.

## Program Data Flow

DataSource

Decoder

Encoder

Log

Channel

Code

## Included Plugins

### Channel: Simple Noise (Plugin\_Channel\_SimpleNoise)

This channel plugin induces a simple white Gaussian or uniform noise into the channel.

Gaussian Noise: The values of the noise are distributed approximately in line with a Gaussian distribution with the specified mean and variance.

Uniform Noise: The values of the noise are distributed uniformly centered about the mean.

The range of the noise is found from [ -Amplitude + Mean, Amplitude + Mean ].

If multiplicative noise is used, the values of the noise are absolute values.

|  |  |
| --- | --- |
|  | RNG Seed:  The number used to seed the random number generator. If left blank, the system clock will be used.  Noise Type:  The selection of either a Uniform or Gaussian distributed random number generator.  Noise Amplitude:  An additive noise needs to exceed 1.0 to have a hope of causing decode error. Increased amplitude widens the range of the distribution that may cause a notable error.  Noise Mean:  Mean should usually be 0.0, but we can skew the signal to have more false positives than negatives.  Noise Variance:  Generated numbers will be multiplied by the square root of this value before further calculation so as to ensure correct variance of the distribution. |
|  |  |

Noise Behaviour:

Additive noise recombination has the error and transmission streams added.

Multiplicative noise recombination has the transmission signal and absolute value of noise signal multiplied. With the current testing model, many of the defaults will not make good use of multiplicative testing, as a polar encoding is largely unaffected by multiplication of positive values, where multiplication of signed values often induces far too much variance.

### Channel: Perfect Transmission (Plugin\_Channel\_Perfect)

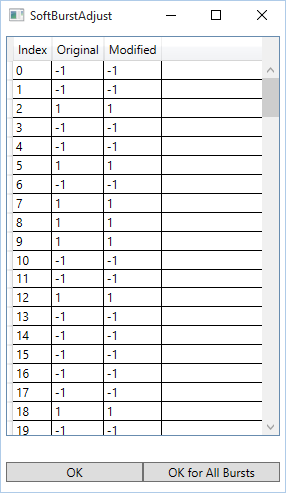
No errors are introduced from the channel. This may be useful for simply determining coding ratio, debugging purposes, or as sanity check.

### Channel: Manual Error Specification (Plugin\_Channel\_Manual)

With manual error specification, it is possible to specify the exact stream that is received by the decoder. For each burst, a window will appear with a data grid holding a copy of the bit at each offset of the stream.

The last column, “Modified”, can be adjusted, and will set the value of the bit for that index of the stream when the “OK” button is pressed. If it is desired that the current set of changes be applied for all further bursts, the “OK for All Bursts” option will apply the changes to the current stream, calculate the changes made for future use, and configure the plugin to use the stored error stream for future bursts.

When using the “OK for all Bursts” option with bursts of different length, any part of the error that didn't fit on the current burst will be ignored. Any part of the burst that exceeds the size of the error will remain unchanged. This plugin is likely more useful with fixed-length block codes and data sources.



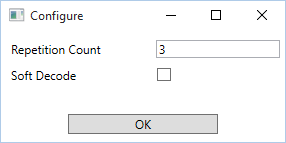
### Code: Plaintext (Plugin\_Code\_Plaintext)

No encoding is performed beyond a polar encoding. No decoding is performed beyond decoding from polar form (quantization to polar values and a max likelihood decode).

### Code: Repetition (Plugin\_Code\_Repetition)

A simplistic repetition code is provided, configurable with n repeated terms.

An option for soft decode is also present. If active, a soft decode will use the mean of signal samples corresponding to a digit. This is in contrast to the mode average used in hard decoding.



### DataSource: All Values of Length n (Plugin\_DataSource\_Exhaustive)

This exhaustive text generation produces all possible bursts of a given range of bit lengths.

|  |  |
| --- | --- |
|  | Alignment:  When a block, or other style, of code requires a set number of bits for transmission, an alignment value can be specified. When burst length has been exhausted, then next size to be tried will be ratchet up by the alignment value, instead of 1.  Min Length:  The initial length of binary burst data that will be exhaustively sent as data for transmission.  Max Length:  The maximum length of binary burst data that will be exhaustively sent as data for transmission. |

### DataSource: Random Bytes (Plugin\_DataSource\_Random)

The Random Bytes data source produces a random sampling of bursts to transmit with a given range of binary length.

|  |  |
| --- | --- |
|  | Sample Count:  The number of bursts to generate and transmit.  Alignment:  When a block, or other style, of code requires a set number of bits for transmission, an alignment value can be specified. Burst lengths are rounded up to ensure divisibility by this alignment value.  Average Length:  The mean length of a generated burst.  Length Deviation:  The length of a burst will be the mean +- this deviation range. |

Random Seed:

The number used to seed the random number generator. If left blank, the system clock will be used.

### DataSource: Samples from File (Plugin\_DataSource\_SamplesFromFile)

The samples from file data source reads in a specified file and treats each line as a transmission string.

|  |  |
| --- | --- |
|  | File:  The selected file from which each line will represent a possible transmission.  Sampling:  Each file line can be used once (in random order),  Or the lines can be sampled randomly.  Sample Count:  The number of lines of the file to sample. The value is not limited by the number of file lines found. Not used if sampling is “Every Item Once”.  Random Seed:  The number used to seed the random number generator. If left blank, the system clock will be used. |

## SDK How-to

Generation of a plugin for the Code Channel Tester system is rather straightforward.

Create a new project and reference the Plugin\_SDK.dll library. The project should set the application output type to Class Library.

Multiple plugin classes can be declared within a single project.

Create a new class, and have it inherit one of the Code, Channel, or DataSource classes from the CodeChannelTester namespace.

Every plugin with declare their Name, Description, and Version number as a property as below:

public override String Name { get { return "Plugin Visible Title"; } }

public override String Description { get { return "Plugin Description."; } }

public override Version Version { get { return new Version(1,0); } }

Plugins are also required to specify whether or not that can be configured by the user, and whether or not they require configuration from the user.

public override Boolean Configurable { get { return true; } }

public override Boolean Configured { get { return true; } }

All configurable plugins have the option of specifying custom implementations for

public override void Configure() {}

public override void Setup() {}

public override void Reset() {}

Configure is called to present users with configuration options. This is called when the plugin is selected for a Datasource/Code/Channel, and the user clicks the button to configure properties of the plugin. This should only return when the user has completed configuration.

Setup performs initialization based on user configuration. This will be called once, after the user requests testing to begin. This should initialize any state which is static or dependant on user configuration, but which may persist between the user requesting testing to begin, so long as the configuration has not changed. If the plugin was unselected, or reconfigured, this procedure will be called again the next time the user requests testing begins.

Reset resets any temporary state to initialized values. This is called every time the user requests testing begins.

## Future Work

While the system and plugins provided at the current time present a minimalist set of tools for dynamic testing of codes, the initial project proposal does contain a number of items that had been left unimplemented. From a retrospective, the original project scope was rather ambitious, and the completion of all items listed would be both largely useful and demanding in time.

Among the items considered, both new and old, a great amount of utility could be found from an increased library of codes and channels. The ability to use convolutional codes, by themselves or with some concatenation scheme would present a greater use for the soft signal representations used currently throughout the program, though this would require the implementation of one or more soft decoding methods.

It may be interesting to combine multiple channels / noise generators into a pipeline. While a custom channel could be developed to take a fixed set of noise types, it is quite possible and likely that a real-world channel could have multiple noise sources leaking into a channel’s band range.

Extending the possible channel plugins to include the use of audio files as noise samples, and/or tunable waveforms could also extend the utility and accuracy of the testing to be closer to what is found in real-world transmission channels.

## References:

Matthias Schulist. Evaluating orthogonal codes transmitted in a wireless communications net-work, January 26 2010. US Patent 7,653,021.

Elwyn R Berlekamp. Algebraic coding theory. McGraw-Hill, 1968.

Chris Heegard and Stephen B Wicker. Turbo coding, volume 476. Springer Science & Business Media, 2013.

W Cary Huffman and Vera Pless. Fundamentals of error-correcting codes. Cambridge university press, 2003.

George Marsaglia. Random number generation. John Wiley and Sons Ltd., 2003.

Matt Donadio. How to Generate White Gaussian Noise. http://dspguru.com/dsp/howtos/how-to-generate-white-gaussian-noise. [Online; accessed 2015-12-29].

Bioucas-Dias, José M., and Mário AT Figueiredo. Multiplicative noise removal using variable splitting and constrained optimization. Image Processing, IEEE Transactions on 19.7, 2010.