CommPy Documentation

Release 0.3.0

Veeresh Taranalli

May 24, 2019

Contents

CommPy is an open source package implementing digital communications algorithms in Python using NumPy, SciPy and Matplotlib.

CHAPTER 1

Available Features

1.1 Channel Coding

- Encoder for Convolutional Codes (Polynomial, Recursive Systematic). Supports all rates and puncture matrices.
- Viterbi Decoder for Convolutional Codes (Hard Decision Output).
- MAP Decoder for Convolutional Codes (Based on the BCJR algorithm).
- Encoder for a rate-1/3 systematic parallel concatenated Turbo Code.
- Turbo Decoder for a rate-1/3 systematic parallel concatenated turbo code (Based on the MAP decoder/BCJR algorithm).
- Binary Galois Field GF(2^m) with minimal polynomials and cyclotomic cosets.
- Create all possible generator polynomials for a (n,k) cyclic code.
- Random Interleavers and De-interleavers.

1.2 Channel Models

- SISO Channel with Rayleigh or Rician fading.
- MIMO Channel with Rayleigh or Rician fading.
- Binary Erasure Channel (BEC)
- Binary Symmetric Channel (BSC)
- Binary AWGN Channel (BAWGNC)

1.3 Filters

- Rectangular
- Raised Cosine (RC), Root Raised Cosine (RRC)
- Gaussian

1.4 Impairments

- Carrier Frequency Offset (CFO)

1.5 Modulation/Demodulation

- Phase Shift Keying (PSK)
- Quadrature Amplitude Modulation (QAM)
- OFDM Tx/Rx signal processing

1.6 Sequences

- PN Sequence
- Zadoff-Chu (ZC) Sequence

1.7 Utilities

- Decimal to bit-array, bit-array to decimal.
- Hamming distance, Euclidean distance.
- Upsample
- Power of a discrete-time signal

CHAPTER 2

Reference

2.1 Channel Coding (commpy.channelcoding)

2.1.1 Galois Fields

GF(x, m)	Defines a Binary Galois Field of order m, containing n, where n can be a single element or a list of elements within the field.

2.1.1.1 commpy.channelcoding.GF

class GF (*x*, *m*)

Defines a Binary Galois Field of order m, containing n, where n can be a single element or a list of elements within the field.

Parameters

- **n** (*int*) – Represents the Galois field element(s).
- **m** (*int*) – Specifies the order of the Galois Field.

Returns x – A Galois Field GF(2^m) object.

Return type int

Examples

```
>>> from numpy import arange
>>> from gfields import GF
>>> x = arange(16)
>>> m = 4
>>> x = GF(x, m)
```

(continues on next page)

(continued from previous page)

```
>>> print x.elements
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
>>> print x.prim_poly
19
```

__init__(*x*, *m*)

Methods

`__init__`(x, m)	
`cosets`()	Compute the cyclotomic cosets of the Galois field.
`minpolys`()	Compute the minimal polynomials for all elements of the Galois field.
`order`()	Compute the orders of the Galois field elements.
`power_to_tuple`()	Convert Galois field elements from power form to tuple form representation.
`tuple_to_power`()	Convert Galois field elements from tuple form to power form representation.

2.1.2 Algebraic Codes

`cyclic_code_genpoly`(n, k)	Generate all possible generator polynomials for a (n, k)-cyclic code.

2.1.2.1 commpy.channelcoding.cyclic_code_genpoly

cyclic_code_genpoly(*n*, *k*)

Generate all possible generator polynomials for a (n, k)-cyclic code.

Parameters

- **n** (*int*) – Code blocklength of the cyclic code.
- **k** (*int*) – Information blocklength of the cyclic code.

Returns **poly_list** – A list of generator polynomials (represented as integers) for the (n, k)-cyclic code.

Return type 1D ndarray of ints

2.1.3 Convolutional Codes

`Trellis`(memory, g_matrix[, feedback, code_type])	Class defining a Trellis corresponding to a k/n - rate convolutional code.
`conv_encode`(message_bits, trellis[, . . .])	Encode bits using a convolutional code.
`viterbi_decode`(coded_bits, trellis[, . . .])	Decodes a stream of convolutionally encoded bits using the Viterbi Algorithm

2.1.3.1 commpy.channelcoding.Trellis

class Trellis(*memory*, *g_matrix*, *feedback=0*, *code_type='default'*)

Class defining a Trellis corresponding to a k/n - rate convolutional code.

Parameters

- **memory** (*1D ndarray of ints*) – Number of memory elements per input of the convolutional encoder.
- **g_matrix** (*2D ndarray of ints (octal representation)*) – Generator matrix G(D) of the convolutional encoder. Each element of G(D) represents a polynomial.
- **feedback** (*int, optional*) – Feedback polynomial of the convolutional encoder. Default value is 00.
- **code_type** (*{'default', 'rsc'}, optional*) – Use 'rsc' to generate a recursive systematic convolutional code.

  If 'rsc' is specified, then the first 'k x k' sub-matrix of

  G(D) must represent a identity matrix along with a non-zero feedback polynomial.

k

Size of the smallest block of input bits that can be encoded using the convolutional code.

Type int

n

Size of the smallest block of output bits generated using the convolutional code.

Type int

total_memory

Total number of delay elements needed to implement the convolutional encoder.

Type int

number_states

Number of states in the convolutional code trellis.

Type int

number_inputs

Number of branches from each state in the convolutional code trellis.

Type int

next_state_table

Table representing the state transition matrix of the convolutional code trellis. Rows represent current states and columns represent current inputs in decimal. Elements represent the corresponding next states in decimal.

Type 2D ndarray of ints

output_table

Table representing the output matrix of the convolutional code trellis. Rows represent current states and columns represent current inputs in decimal. Elements represent corresponding outputs in decimal.

Type 2D ndarray of ints

Examples

```
>>> from numpy import array
>>> import commpy.channelcoding.convcode as cc
>>> memory = array([2])
>>> g_matrix = array([[0o5, 0o7]]) # G(D) = [1+D^2, 1+D+D^2]
>>> trellis = cc.Trellis(memory, g_matrix)
>>> print trellis.k
1
>>> print trellis.n
2
>>> print trellis.total_memory
2
>>> print trellis.number_states
4
>>> print trellis.number_inputs
2
>>> print trellis.next_state_table
[[0 2]
 [0 2]
 [1 3]
 [1 3]]
>>>print trellis.output_table
[[0 3]
 [3 0]
 [1 2]
 [2 1]]
```

__init__(*memory*, *g_matrix*, *feedback=0*, *code_type='default'*)

Methods

__init__(memory, g_matrix[, feedback, code_type])	
`visualize`([trellis_length, state_order, ...])	Plot the trellis diagram.

2.1.3.2 commpy.channelcoding.conv_encode

conv_encode(*message_bits*, *trellis*, *code_type='default'*, *puncture_matrix=None*)

Encode bits using a convolutional code.

Parameters

- **message_bits** (*1D ndarray containing {0, 1}*) – Stream of bits to be convolutionally encoded.
- **generator_matrix** (*2-D ndarray of ints*) – Generator matrix G(D) of the convolutional code using which the input bits are to be encoded.
- **M** (*1D ndarray of ints*) – Number of memory elements per input of the convolutional encoder.

Returns **coded_bits** – Encoded bit stream.

Return type 1D ndarray containing {0, 1}

2.1.3.3 commpy.channelcoding.viterbi_decode

viterbi_decode (*coded_bits*, *trellis*, *tb_depth=None*, *decoding_type='hard'*)

Decodes a stream of convolutionally encoded bits using the Viterbi Algorithm

Parameters

- **coded_bits** (*1D ndarray*) – Stream of convolutionally encoded bits which are to be decoded.
- **generator_matrix** (*2D ndarray of ints*) – Generator matrix G(D) of the convolutional code using which the input bits are to be decoded.
- **M** (*1D ndarray of ints*) – Number of memory elements per input of the convolutional encoder.
- **tb_length** (*int*) – Traceback depth (Typically set to 5*(M+1)).
- **decoding_type** (*str {'hard', 'unquantized'}*) – The type of decoding to be used. 'hard' option is used for hard inputs (bits) to the decoder, e.g., BSC channel. 'unquantized' option is used for soft inputs (real numbers) to the decoder, e.g., BAWGN channel.

Returns **decoded_bits** – Decoded bit stream.

Return type 1D ndarray

References

2.1.4 Turbo Codes

turbo_encode(msg_bits, trellis1, trellis2, . . .)	Turbo Encoder.
map_decode(sys_symbols, non_sys_symbols, . . .)	Maximum a-posteriori probability (MAP) decoder.
turbo_decode(sys_symbols, non_sys_symbols_1, . . .)	Turbo Decoder.

2.1.4.1 commpy.channelcoding.turbo_encode

turbo_encode (*msg_bits*, *trellis1*, *trellis2*, *interleaver*)

Turbo Encoder.

Encode Bits using a parallel concatenated rate-1/3 turbo code consisting of two rate-1/2 systematic convolutional component codes.

Parameters

- **msg_bits** (*1D ndarray containing {0, 1}*) – Stream of bits to be turbo encoded.
- **trellis1** (*Trellis object*) – Trellis representation of the first code in the parallel concatenation.
- **trellis2** (*Trellis object*) – Trellis representation of the second code in the parallel concatenation.
- **interleaver** (*Interleaver object*) – Interleaver used in the turbo code.

Returns

[sys_stream, non_sys_stream1, non_sys_stream2] – Encoded bit streams corresponding to the systematic output

and the two non-systematic outputs from the two component codes.

Return type list of 1D ndarrays

2.1.4.2 commpy.channelcoding.map_decode

map_decode (*sys_symbols*, *non_sys_symbols*, *trellis*, *noise_variance*, *L_int*, *mode='decode'*)

Maximum a-posteriori probability (MAP) decoder.

Decodes a stream of convolutionally encoded (rate 1/2) bits using the MAP algorithm.

Parameters

- **sys_symbols** (*1D ndarray*) – Received symbols corresponding to the systematic (first output) bits in the codeword.
- **non_sys_symbols** (*1D ndarray*) – Received symbols corresponding to the non-systematic (second output) bits in the codeword.
- **trellis** (*Trellis object*) – Trellis representation of the convolutional code.
- **noise_variance** (*float*) – Variance (power) of the AWGN channel.
- **L_int** (*1D ndarray*) – Array representing the initial intrinsic information for all received symbols.

  Typically all zeros, corresponding to equal prior probabilities of bits 0 and 1.
- **mode** (*str{'decode', 'compute'}, optional*) – The mode in which the MAP decoder is used. 'decode' mode returns the decoded bits

  along with the extrinsic information. 'compute' mode returns only the extrinsic information.

Returns **[L_ext, decoded_bits]** – The first element of the list is the extrinsic information. The second element of the list is the decoded bits.

Return type list of two 1D ndarrays

2.1.4.3 commpy.channelcoding.turbo_decode

turbo_decode (*sys_symbols*, *non_sys_symbols_1*, *non_sys_symbols_2*, *trellis*, *noise_variance*, *number_iterations*, *interleaver*, *L_int=None*)

Turbo Decoder.

Decodes a stream of convolutionally encoded (rate 1/3) bits using the BCJR algorithm.

Parameters

- **sys_symbols** (*1D ndarray*) – Received symbols corresponding to the systematic (first output) bits in the codeword.
- **non_sys_symbols_1** (*1D ndarray*) – Received symbols corresponding to the first parity bits in the codeword.
- **non_sys_symbols_2** (*1D ndarray*) – Received symbols corresponding to the second parity bits in the codeword.
- **trellis** (*Trellis object*) – Trellis representation of the convolutional codes used in the Turbo code.

- **noise_variance** (*float*) – Variance (power) of the AWGN channel.
- **number_iterations** (*int*) – Number of the iterations of the BCJR algorithm used in turbo decoding.
- **interleaver** (*Interleaver object.*) – Interleaver used in the turbo code.
- **L_int** (*1D ndarray*) – Array representing the initial intrinsic information for all received symbols.

  Typically all zeros, corresponding to equal prior probabilities of bits 0 and 1.

Returns **decoded_bits** – Decoded bit stream.

Return type 1D ndarray of ints containing {0, 1}

2.1.5 LDPC Codes

get_ldpc_code_params(ldpc_design_filename)	Extract parameters from LDPC code design file.
ldpc_bp_decode(llr_vec, ldpc_code_params, . . .)	LDPC Decoder using Belief Propagation (BP).

2.1.5.1 commpy.channelcoding.get_ldpc_code_params

get_ldpc_code_params (*ldpc_design_filename*)

Extract parameters from LDPC code design file.

Parameters **ldpc_design_filename** (*string*) – Filename of the LDPC code design file.

Returns **ldpc_code_params** – Parameters of the LDPC code.

Return type dictionary

2.1.5.2 commpy.channelcoding.ldpc_bp_decode

ldpc_bp_decode (*llr_vec*, *ldpc_code_params*, *decoder_algorithm*, *n_iters*)

LDPC Decoder using Belief Propagation (BP).

Parameters

- **llr_vec** (*1D array of float*) – Received codeword LLR values from the channel.
- **ldpc_code_params** (*dictionary*) – Parameters of the LDPC code.
- **decoder_algorithm** (*string*) – Specify the decoder algorithm type. SPA for Sum-Product Algorithm MSA for Min-Sum Algorithm
- **n_iters** (*int*) – Max. number of iterations of decoding to be done.

Returns

- **dec_word** (*1D array of 0's and 1's*) – The codeword after decoding.
- **out_llrs** (*1D array of float*) – LLR values corresponding to the decoded output.

2.1.6 Interleavers and De-interleavers

RandInterlv(length, seed)	Random Interleaver.

2.1.6.1 commpy.channelcoding.RandInterlv

class RandInterlv(*length*, *seed*)

Random Interleaver.

Parameters

- **length** (*int*) – Length of the interleaver.
- **seed** (*int*) – Seed to initialize the random number generator which generates the random permutation for interleaving.

Returns **random_interleaver** – A random interleaver object.

Return type RandInterlv object

Note: The random number generator is the RandomState object from NumPy, which uses the Mersenne Twister algorithm.

__init__(*length*, *seed*)

Methods

`__init__`(length, seed)	
`deinterlv`(in_array)	De-interleave input array using the specific interleaver.
`interlv`(in_array)	Interleave input array using the specific interleaver.

2.2 Channel Models (commpy.channels)

`SISOFlatChannel`([noise_std, fading_param])	Constructs a SISO channel with a flat fading.
`MIMOFlatChannel`(nb_tx, nb_rx[, noise_std, ...])	Constructs a MIMO channel with a flat fading based on the Kronecker model.
`bec`(input_bits, p_e)	Binary Erasure Channel.
`bsc`(input_bits, p_t)	Binary Symmetric Channel.
`awgn`(input_signal, snr_dB[, rate])	Addditive White Gaussian Noise (AWGN) Channel.

2.2.1 commpy.channels.SISOFlatChannel

class SISOFlatChannel(*noise_std=None*, *fading_param=(1, 0)*)

Constructs a SISO channel with a flat fading. The channel coefficient are normalized i.e. the mean magnitude is 1.

Parameters

- **noise_std** (*float, optional*) – Noise standard deviation. Default value is None and then the value must set later.
- **fading_param** (*tuple of 2 floats, optional*) – Parameters of the fading (see attribute for details). Default value is (1,0) i.e. no fading.

fading_param

Parameters of the fading. The complete tuple must be set each time. Raise ValueError when sets with value that would lead to a non-normalized channel.

- fading_param[0] refers to the mean of the channel gain (Line Of Sight component).
- fading_param[1] refers to the variance of the channel gain (Non Line Of Sight component).

Classical fadings:

- (1, 0): no fading.
- (0, 1): Rayleigh fading.
- Others: rician fading.

Type tuple of 2 floats

noise_std
Noise standard deviation. None is the value has not been set yet.

Type float

isComplex
True if the channel is complex, False if not. The value is set together with fading_param based on the type of fading_param[0].

Type Boolean, Read-only

k_factor
Fading k-factor, the power ratio between LOS and NLOS.

Type positive float, Read-only

nb_tx
Number of Tx antennas.

Type int = 1, Read-only

nb_rx
Number of Rx antennas.

Type int = 1, Read-only

noises
Last noise generated. None if no noise has been generated yet.

Type 1D ndarray

channel_gains
Last channels gains generated. None if no channels has been generated yet.

Type 1D ndarray

unnoisy_output
Last transmitted message without noise. None if no message has been propagated yet.

Type 1D ndarray

Raises `ValueError` – If the fading parameters would lead to a non-normalized channel. The condition is $|param[1]| + |param[0]|^2 = 1$

__init__(*noise_std=None*, *fading_param=(1, 0)*)
x.__init__(...) initializes x; see help(type(x)) for signature

Methods

__init__([noise_std, fading_param])	x.__init__(…) initializes x; see help(type(x)) for signature
generate_noises(dims)	Generates the white gaussian noise with the right standard deviation and saves it.
propagate(msg)	Propagates a message through the channel.
set_SNR_dB(SNR_dB[, code_rate, Es])	Sets the the noise standard deviation based on SNR expressed in dB.
set_SNR_lin(SNR_lin[, code_rate, Es])	Sets the the noise standard deviation based on SNR expressed in its linear form.

Attributes

fading_param	Parameters of the fading (see class attribute for details).
isComplex	Read-only - True if the channel is complex, False if not.
k_factor	Read-only - Fading k-factor, the power ratio between LOS and NLOS
nb_rx	Read-only - Number of Rx antennas, set to 1 for SISO channel.
nb_tx	Read-only - Number of Tx antennas, set to 1 for SISO channel.

2.2.2 commpy.channels.MIMOFlatChannel

class MIMOFlatChannel(*nb_tx*, *nb_rx*, *noise_std=None*, *fading_param=None*)

Constructs a MIMO channel with a flat fading based on the Kronecker model. The channel coefficient are normalized i.e. the mean magnitude is 1.

Parameters

- **nb_tx** (*int >= 1*) – Number of Tx antennas.
- **nb_rx** (*int >= 1*) – Number of Rx antennas.
- **noise_std** (*float, optional*) – Noise standard deviation. Default value is None and then the value must set later.
- **fading_param** (*tuple of 3 floats, optional*) – Parameters of the fading. The complete tuple must be set each time. Default value is (zeros((nb_rx, nb_tx)), identity(nb_tx), identity(nb_rx)) i.e. Rayleigh fading.

fading_param

Parameters of the fading. Raise ValueError when sets with value that would lead to a non-normalized channel.

- fading_param[0] refers to the mean of the channel gain (Line Of Sight component).
- fading_param[1] refers to the transmit-side spatial correlation matrix of the channel.
- fading_param[2] refers to the receive-side spatial correlation matrix of the channel.

Classical fadings:

- (zeros((nb_rx, nb_tx)), identity(nb_tx), identity(nb_rx)): Rayleigh fading.

- Others: rician fading.

  Type tuple of 2 floats

noise_std
Noise standard deviation. None is the value has not been set yet.

Type float

isComplex
True if the channel is complex, False if not. The value is set together with fading_param based on the type of fading_param[0].

Type Boolean, Read-only

k_factor
Fading k-factor, the power ratio between LOS and NLOS.

Type positive float, Read-only

nb_tx
Number of Tx antennas.

Type int

nb_rx
Number of Rx antennas.

Type int

noises
Last noise generated. None if no noise has been generated yet. noises[i] is the noise vector of size nb_rx for the i-th message vector.

Type 2D ndarray

channel_gains
Last channels gains generated. None if no channels has been generated yet. channel_gains[i] is the channel matrix of size (nb_rx x nb_tx) for the i-th message vector.

Type 2D ndarray

unnoisy_output
Last transmitted message without noise. None if no message has been propageted yet. unnoisy_output[i] is the transmitted message without noise of size nb_rx for the i-th message vector.

Type 1D ndarray

Raises `ValueError` – If the fading parameters would lead to a non-normalized channel. The condition is $NLOS + LOS = nb_{tx} * nb_{rx}$ where

- $NLOS = tr(param[1]^T \otimes param[2])$
- $LOS = \sum |param[0]|^2$

__init__(*nb_tx*, *nb_rx*, *noise_std=None*, *fading_param=None*)
x.__init__(. . .) initializes x; see help(type(x)) for signature

Methods

__init__(nb_tx, nb_rx[, noise_std, fading_param])	x.__init__(...) initializes x; see help(type(x)) for signature
generate_noises(dims)	Generates the white gaussian noise with the right standard deviation and saves it.
propagate(msg)	Propagates a message through the channel.
set_SNR_dB(SNR_dB[, code_rate, Es])	Sets the the noise standard deviation based on SNR expressed in dB.
set_SNR_lin(SNR_lin[, code_rate, Es])	Sets the the noise standard deviation based on SNR expressed in its linear form.

Attributes

fading_param	Parameters of the fading (see class attribute for details).
isComplex	Read-only - True if the channel is complex, False if not.
k_factor	Read-only - Fading k-factor, the power ratio between LOS and NLOS

2.2.3 commpy.channels.bec

bec (*input_bits*, *p_e*)

Binary Erasure Channel.

Parameters

- **input_bits** (*1D ndarray containing {0, 1}*) – Input array of bits to the channel.
- **p_e** (*float in [0, 1]*) – Erasure probability of the channel.

Returns **output_bits** – Output bits from the channel.

Return type 1D ndarray containing {0, 1}

2.2.4 commpy.channels.bsc

bsc (*input_bits*, *p_t*)

Binary Symmetric Channel.

Parameters

- **input_bits** (*1D ndarray containing {0, 1}*) – Input array of bits to the channel.
- **p_t** (*float in [0, 1]*) – Transition/Error probability of the channel.

Returns **output_bits** – Output bits from the channel.

Return type 1D ndarray containing {0, 1}

2.2.5 commpy.channels.awgn

awgn (*input_signal*, *snr_dB*, *rate=1.0*)

Addditive White Gaussian Noise (AWGN) Channel.

Parameters

- **input_signal** (*1D ndarray of floats*) – Input signal to the channel.
- **snr_dB** (*float*) – Output SNR required in dB.
- **rate** (*float*) – Rate of the a FEC code used if any, otherwise 1.

Returns **output_signal** – Output signal from the channel with the specified SNR.

Return type 1D ndarray of floats

2.3 Pulse Shaping Filters (commpy.filters)

rcosfilter(N, alpha, Ts, Fs)	Generates a raised cosine (RC) filter (FIR) impulse response.
rrcosfilter(N, alpha, Ts, Fs)	Generates a root raised cosine (RRC) filter (FIR) impulse response.
gaussianfilter(N, alpha, Ts, Fs)	Generates a gaussian filter (FIR) impulse response.
rectfilter(N, Ts, Fs)	Generates a rectangular filter (FIR) impulse response.

2.3.1 commpy.filters.rcosfilter

rcosfilter(*N*, *alpha*, *Ts*, *Fs*)

Generates a raised cosine (RC) filter (FIR) impulse response.

Parameters

- **N** (*int*) – Length of the filter in samples.
- **alpha** (*float*) – Roll off factor (Valid values are [0, 1]).
- **Ts** (*float*) – Symbol period in seconds.
- **Fs** (*float*) – Sampling Rate in Hz.

Returns

- **time_idx** (*1-D ndarray (float)*) – Array containing the time indices, in seconds, for the impulse response.
- **h_rc** (*1-D ndarray (float)*) – Impulse response of the raised cosine filter.

2.3.2 commpy.filters.rrcosfilter

rrcosfilter(*N*, *alpha*, *Ts*, *Fs*)

Generates a root raised cosine (RRC) filter (FIR) impulse response.

Parameters

- **N** (*int*) – Length of the filter in samples.
- **alpha** (*float*) – Roll off factor (Valid values are [0, 1]).
- **Ts** (*float*) – Symbol period in seconds.
- **Fs** (*float*) – Sampling Rate in Hz.

Returns

- **time_idx** (*1-D ndarray of floats*) – Array containing the time indices, in seconds, for the impulse response.
- **h_rrc** (*1-D ndarray of floats*) – Impulse response of the root raised cosine filter.

2.3.3 commpy.filters.gaussianfilter

gaussianfilter (*N*, *alpha*, *Ts*, *Fs*)

Generates a gaussian filter (FIR) impulse response.

Parameters

- **N** (*int*) – Length of the filter in samples.
- **alpha** (*float*) – Roll off factor (Valid values are [0, 1]).
- **Ts** (*float*) – Symbol period in seconds.
- **Fs** (*float*) – Sampling Rate in Hz.

Returns

- **time_index** (*1-D ndarray of floats*) – Array containing the time indices for the impulse response.
- **h_gaussian** (*1-D ndarray of floats*) – Impulse response of the gaussian filter.

2.3.4 commpy.filters.rectfilter

rectfilter (*N*, *Ts*, *Fs*)

Generates a rectangular filter (FIR) impulse response.

Parameters

- **N** (*int*) – Length of the filter in samples.
- **Ts** (*float*) – Symbol period in seconds.
- **Fs** (*float*) – Sampling Rate in Hz.

Returns

- **time_index** (*1-D ndarray of floats*) – Array containing the time indices for the impulse response.
- **h_rect** (*1-D ndarray of floats*) – Impulse response of the rectangular filter.

2.4 Impairments (commpy.impairments)

add_frequency_offset(waveform, Fs, delta_f)	Add frequency offset impairment to input signal.

2.4.1 commpy.impairments.add_frequency_offset

add_frequency_offset (*waveform*, *Fs*, *delta_f*)

Add frequency offset impairment to input signal.

Parameters

- **waveform** (*1D ndarray of floats*) – Input signal.
- **Fs** (*float*) – Sampling frequency (in Hz).
- **delta_f** (*float*) – Frequency offset (in Hz).

Returns **output_waveform** – Output signal with frequency offset.

Return type 1D ndarray of floats

2.5 Modulation Demodulation (commpy.modulation)

PSKModem(m)	Creates a Phase Shift Keying (PSK) Modem object.
QAMModem(m)	Creates a Quadrature Amplitude Modulation (QAM) Modem object.
mimo_ml(y, h, constellation)	MIMO ML Detection.

2.5.1 commpy.modulation.PSKModem

class PSKModem(*m*)

Creates a Phase Shift Keying (PSK) Modem object.

__init__(*m*)

Creates a Phase Shift Keying (PSK) Modem object.

Parameters **m** (*int*) – Size of the PSK constellation.

Methods

__init__(m)	Creates a Phase Shift Keying (PSK) Modem object.
demodulate(input_symbols, demod_type[, ...])	Demodulate (map) a set of constellation symbols to corresponding bits.
modulate(input_bits)	Modulate (map) an array of bits to constellation symbols.

2.5.2 commpy.modulation.QAMModem

class QAMModem(*m*)

Creates a Quadrature Amplitude Modulation (QAM) Modem object.

__init__(*m*)

Creates a Quadrature Amplitude Modulation (QAM) Modem object.

Parameters **m** (*int*) – Size of the QAM constellation.

Methods

__init__(m)	Creates a Quadrature Amplitude Modulation (QAM) Modem object.
demodulate(input_symbols, demod_type[, ...])	Demodulate (map) a set of constellation symbols to corresponding bits.

Continued on next page

Table 19 – continued from previous page

`modulate`(input_bits)	Modulate (map) an array of bits to constellation symbols.

2.5.3 commpy.modulation.mimo_ml

mimo_ml (*y*, *h*, *constellation*)
MIMO ML Detection.

Parameters

- **y** (*1D ndarray of complex floats*) – Received complex symbols (shape: num_receive_antennas x 1)
- **h** (*2D ndarray of complex floats*) – Channel Matrix (shape: num_receive_antennas x num_transmit_antennas)
- **constellation** (*1D ndarray of complex floats*) – Constellation used to modulate the symbols

class PSKModem (*m*)
Creates a Phase Shift Keying (PSK) Modem object.

class QAMModem (*m*)
Creates a Quadrature Amplitude Modulation (QAM) Modem object.

mimo_ml (*y*, *h*, *constellation*)
MIMO ML Detection.

Parameters

- **y** (*1D ndarray of complex floats*) – Received complex symbols (shape: num_receive_antennas x 1)
- **h** (*2D ndarray of complex floats*) – Channel Matrix (shape: num_receive_antennas x num_transmit_antennas)
- **constellation** (*1D ndarray of complex floats*) – Constellation used to modulate the symbols

2.6 Sequences (commpy.sequences)

pnsequence(pn_order, pn_seed, pn_mask, . . .)	Generate a PN (Pseudo-Noise) sequence using a Linear Feedback Shift Register (LFSR).
zcsequence(u, seq_length)	Generate a Zadoff-Chu (ZC) sequence.

2.6.1 commpy.sequences.pnsequence

pnsequence (*pn_order*, *pn_seed*, *pn_mask*, *seq_length*)
Generate a PN (Pseudo-Noise) sequence using a Linear Feedback Shift Register (LFSR).

Parameters

- **pn_order** (*int*) – Number of delay elements used in the LFSR.
- **pn_seed** (*string containing 0's and 1's*) – Seed for the initialization of the LFSR delay elements. The length of this string must be equal to 'pn_order'.

- **pn_mask** (*string containing 0's and 1's*) – Mask representing which delay elements contribute to the feedback in the LFSR. The length of this string must be equal to 'pn_order'.
- **seq_length** (*int*) – Length of the PN sequence to be generated. Usually (2^pn_order - 1)

Returns **pnseq** – PN sequence generated.

Return type 1D ndarray of ints

2.6.2 commpy.sequences.zcsequence

zcsequence (*u*, *seq_length*)

Generate a Zadoff-Chu (ZC) sequence.

Parameters

- **u** (*int*) – Root index of the the ZC sequence.
- **seq_length** (*int*) – Length of the sequence to be generated. Usually a prime number.

Returns **zcseq** – ZC sequence generated.

Return type 1D ndarray of complex floats

2.7 Utilities (commpy.utilities)

dec2bitarray(in_number, bit_width)	Converts a positive integer to NumPy array of the specified size containing bits (0 and 1).
bitarray2dec(in_bitarray)	Converts an input NumPy array of bits (0 and 1) to a decimal integer.
hamming_dist(in_bitarray_1, in_bitarray_2)	Computes the Hamming distance between two NumPy arrays of bits (0 and 1).
euclid_dist(in_array1, in_array2)	Computes the squared euclidean distance between two NumPy arrays
upsample(x, n)	Upsample the input array by a factor of n
signal_power(signal)	Compute the power of a discrete time signal.

2.7.1 commpy.utilities.dec2bitarray

dec2bitarray (*in_number*, *bit_width*)

Converts a positive integer to NumPy array of the specified size containing bits (0 and 1).

Parameters

- **in_number** (*int*) – Positive integer to be converted to a bit array.
- **bit_width** (*int*) – Size of the output bit array.

Returns **bitarray** – Array containing the binary representation of the input decimal.

Return type 1D ndarray of ints

2.7.2 commpy.utilities.bitarray2dec

bitarray2dec (*in_bitarray*)

Converts an input NumPy array of bits (0 and 1) to a decimal integer.

Parameters **in_bitarray** (*1D ndarray of ints*) – Input NumPy array of bits.

Returns **number** – Integer representation of input bit array.

Return type int

2.7.3 commpy.utilities.hamming_dist

hamming_dist (*in_bitarray_1*, *in_bitarray_2*)

Computes the Hamming distance between two NumPy arrays of bits (0 and 1).

Parameters

- **in_bit_array_1** (*1D ndarray of ints*) – NumPy array of bits.
- **in_bit_array_2** (*1D ndarray of ints*) – NumPy array of bits.

Returns **distance** – Hamming distance between input bit arrays.

Return type int

2.7.4 commpy.utilities.euclid_dist

euclid_dist (*in_array1*, *in_array2*)

Computes the squared euclidean distance between two NumPy arrays

Parameters

- **in_array1** (*1D ndarray of floats*) – NumPy array of real values.
- **in_array2** (*1D ndarray of floats*) – NumPy array of real values.

Returns **distance** – Squared Euclidean distance between two input arrays.

Return type float

2.7.5 commpy.utilities.upsample

upsample (*x*, *n*)

Upsample the input array by a factor of n

Adds n-1 zeros between consecutive samples of x

Parameters

- **x** (*1D ndarray*) – Input array.
- **n** (*int*) – Upsampling factor

Returns **y** – Output upsampled array.

Return type 1D ndarray

2.7.6 commpy.utilities.signal_power

signal_power (*signal*)

Compute the power of a discrete time signal.

Parameters **signal** (*1D ndarray*) – Input signal.

Returns **P** – Power of the input signal.

Return type float

Python Module Index

c

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

Z