16 QAM Communication Toolbox in Python

Release 1.0

Arthur Cruz Morbach, Jonas Dandanel de Castro, Sandro Binsfeld

Feb 26, 2021
Low intermediate frequency (low-IF) receivers are the most used architectures in modern wireless communications, for instance in the growing market of Internet of Things (IoT) applications. The QAM (Quadrature Amplitude Modulation) is used in several communication systems, such as digital TV, digital radio, high-speed internet services, and other systems with a high data rate requirement. The main reason for its adoption is its high spectral efficiency. This documentation covers the main blocks used in a 16-QAM communication system.
Module containing all the functions used in the modulation process of a 16 QAM communication system.

1.1 Data Generator

modulation.data_gen(N, data_sync=0)
Generates an array of data. If the synchronization bits are not informed a default sequence will be used.

Parameters
- **N** *(int)* – Number of bits.
- **data_sync** *(1D array of ints.)* – Synchronization bits.

Returns **data** – Pseudo randomic data with synchronization bits.

Return type 1D array of ints

```python
import numpy as np

def data_gen(N, data_sync=0):
    if data_sync == 0:
        data_sync_osc = []
        for i in range(176):
            data_sync_osc.append(1)
        data_sync_symb = [0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1]
        data_sync = np.concatenate((data_sync_osc, data_sync_symb), axis=None)
        data_r = np.random.rand(N - len(data_sync))
        data_r[np.where(data_r >= 0.5)] = 1
        data_r[np.where(data_r < 0.5)] = 0
        data = np.concatenate((data_sync, data_r), axis=None)
    return(data)
```
1.2 Slicer

`modulation.slicer(data)`

It separates the even bits in the In-phase (I) vector, and the odd bits in the Quadrature (Q) vector.

Parameters `data (1D array of ints)` – Array that will be divided.

Returns

- `dataI (1D array of ints)` – Array with the even position bits.
- `dataQ (1D array of ints)` – Array with the odd position bits.

```python
def slicer(data):
    dataI = data[slice(0, len(data), 2)]
    dataQ = data[slice(1, len(data), 2)]
    return (dataI, dataQ)
```
1.3 Mapper

modulation.mapper_16QAM(QAM16, data)

Uses the input data to index the array with the 16QAM amplitudes and phases.

Parameters

- **QAM16 (1D array of floats)** – Array that will be indexed.
- **data (1D array of ints)** – Data that will index the QAM16 array.

Returns **dataMapped** – Array with the mapped data.

Return type 1D array of floats

```python
def mapper_16QAM(QAM16, data):
    map0 = 2*data[slice(0, len(data), 2)] + data[slice(1, len(data), 2)]
    map0 = list(map(int, map0))
    dataMapped = []
    for i in range(len(map0)):
        dataMapped.append(QAM16[map0[i]])
    return dataMapped
```
1.4 Upsampler

modulation.upsampler(Ns, K, symbols)

Increases the symbol samples by adding zeros between each value.

Parameters

- Ns (int) – Number of symbols.
- K (int) – Up-sampler factor.
- symbols (1D array of floats) – Symbols array.

Returns up – Array with the upsampled data.

Return type 1D array of floats

```python
import numpy as np

def upsampler(Ns, K, symbols):
    up = np.zeros(Ns*K)
    up[slice(0, len(up), K)] = symbols
    return(up)
```
1.5 Shaping Filter

The `shaping_filter` function is used to shape the symbols. It convolves the upsampled symbols with the impulse response of a square root raised cosine filter. This filter also arranges the information in a defined frequency spectrum and reduces intersymbol interference.

**Parameters**

- `upsampler (int)` – Upsampled symbols.
- `Ns (int)` – Number of symbols.
- `Fif (float)` – Intermediary frequency.
- `Fs (float)` – Sampling frequency

**Returns**

- `shaped_signal (1D array of floats)` – Signal convoluted with a SRRC filter impulse response.
- `x_axis (1D array of floats)` – Data domain.
- `y_response (1D array of floats)` – Array with the amplitudes varying regarding the domain.
```python
import numpy as np
import commpy as cp

def shaping_filter(upsampler, Ns, alpha, Fif, Fs):
    [x_axis, y_response] = cp.rrcosfilter(Ns, alpha, 2/Fif, Fs)
    shaped_signal = np.convolve(upsampler, y_response, 'full')
    return (shaped_signal, x_axis, y_response)
```

1.5. Shaping Filter
1.6 Oscillator

modulation.\texttt{oscillator}(\texttt{start, stop, step, frequency, phase=0})

Generates the carrier signal.

\textbf{Parameters}

- \texttt{start (number)} – Start of interval. The interval includes this value.
- \texttt{stop (number)} – End of interval. The interval does not include this value.
- \texttt{step (number)} – The distance between two adjacent values.
- \texttt{frequency (float)} – Frequency of oscillation in Hz.
- \texttt{phase (float, optional)} – Phase of the sinusoidal wave. The default phase is 0.

\textbf{Returns}

- \texttt{Osc (1D array of floats)} – Amplitude values of the sinusoidal wave.
- \texttt{time (1D array of floats)} – Data domain.

\begin{verbatim}
import numpy as np
from math import pi

def oscillator(start, stop, step, frequency, phase=0):
    import numpy as np
    from math import pi
    \end{verbatim}
1.7 Mixer

modulation.\texttt{mixer} \begin{syntax} \texttt{(signal, carrier)} \end{syntax}

It is a pointwise product function. In this application it mixes a signal with the carrier, shifting the frequency spectrum to a defined intermediary frequency.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{signal (1D array of floats)} – Signal that will be mixed.
  \item \texttt{carrier (1D array of floats)} – Carrier signal.
\end{itemize}

\textbf{Returns} \texttt{mix} – Mixed signal.

\textbf{Return type} 1D array of floats

\begin{verbatim}
def mixer(signal, carrier):
    mix = []
    for i in range(len(signal)):
        mix.append(signal[i] * carrier[i])
    return mix
\end{verbatim}
1.8 Combiner

`modulation.combiner(signal_I, signal_Q)`

It’s a pointwise sum, combining the modulated signals in quadrature.

Parameters

- `signal_I` (*1D array of floats*) – In-phase symbols modulated.
- `signal_Q` (*1D array of floats*) – Quadrature symbols modulated.

Returns `combined_sig` – Quadrature signal.

Return type 1D array of floats

```python
def combiner(signal_I, signal_Q):
    combined_sig = []
    for i in range(len(signal_I)):
        combined_sig.append(signal_I[i] + signal_Q[i])
    return(combined_sig)
```
Module containing the functions that cover the channel characteristics.

### 2.1 Additive White Gaussian Noise

channel.\texttt{AWGN} (\texttt{IFsig}, \texttt{SNR})

Adds noise to the IF signal based on the proposed SNR.

**Parameters**

- \texttt{IFsig} (\textit{1D array of floats}) – IF modulated signal.
- \texttt{SNR} (\textit{float}) – Signal to noise ratio in dB.

**Returns** \texttt{IF\_n} – IF signal with noise based on the SNR.

**Return type** 1D array of floats

```python
import numpy as np

def AWGN(IFsig, SNR):
    dP = np.zeros(len(IFsig))
    P = 0

    for i in range(len(IFsig)):
        dP[i] = abs(IFsig[i])**2
        P = P + dP[i]

    P = P/len(IFsig)
    gamma = 10**(SNR/10)
    N0 = P/gamma
    n = ((N0/2)**(0.5))*np.random.standard_normal(len(IFsig))
    IF_n = np.zeros(len(IFsig))

    for i in range(len(IFsig)):
        IF_n[i] = IFsig[i] + n[i]

    return(IF_n)
```
3.1 Phase Locked Loop

def PLL(input_signal, Fs, lenght, N):
    zeta = .707  # damping factor
    k = 1
    Bn = 0.01*Fs  # Noise Bandwidth
    K_0 = 1  # NCO gain
    K_d = 1/2  # Phase Detector gain
    K_p = (1/(K_d*K_0))*((4*zeta)/(zeta+(1/(4*zeta)))) * \
           (Bn/Fs)  # Proportional gain
    K_i = (1/(K_d*K_0))*((4*(zeta+(1/(4*zeta)))**2)) * \
           (Bn/Fs)**2  # Integrator gain
    integrator_out = 0
    phase_estimate = np.zeros(lenght)
    e_D = []  # phase-error output
    e_F = []  # loop filter output
    sin_out_n = np.zeros(lenght)
cos_out_n = np.ones(lenght)
for n in range(lenght-1):
    # phase detector
    try:
        e_D.append(
            math.atan(input_signal[n] * (cos_out_n[n] + sin_out_n[n])))
    except IndexError:
        e_D.append(0)
    # loop filter
    integrator_out += K_i * e_D[n]
    e_F.append(K_p * e_D[n] + integrator_out)
    # NCO
    try:
        phase_estimate[n+1] = phase_estimate[n] + K_0 * e_F[n]
    except IndexError:
        phase_estimate[n+1] = K_0 * e_F[n]
    sin_out_n[n+1] = -np.sin(2*np.pi*(k/N)*(n+1) + phase_estimate[n])
    cos_out_n[n+1] = np.cos(2*np.pi*(k/N)*(n+1) + phase_estimate[n])

sin_out_n = -sin_out_n
cos_out = cos_out_n[280:400]
sin_out = sin_out_n[280:400]

for i in range(18):
    cos_out = np.concatenate(
        (cos_out, cos_out_n[280:400], cos_out_n[280:400]), axis=None)
    sin_out = np.concatenate(
        (sin_out, sin_out_n[280:400], sin_out_n[280:400]), axis=None)
return(cos_out, sin_out)
3.2 Mixer

After the synchronization, the output of the PLL will mix the IF signal. Let’s say \( s(t) \) is the IF signal and will be mixed by a cosine wave already synchronized.

\[
d_l(t) = s(t) \ast \cos(2\pi f_c t) + n(t)
\]

\( s(t) \) is composed of the symbols in-phase (\( a_i \)) mixed with cosine and the symbols in quadrature (\( a_q \)) mixed with a sine. Expanding \( s(t) \):

\[
d_l(t) = [a_i(t) \cos(2\pi f_c t) + a_Q(t) \sin(2\pi f_c t)] \ast \cos(2\pi f_c t) + n(t)
\]

The mixing process will result in a baseband signal with high-frequency components:

\[
d_l(t) = \frac{a_i(t)}{2} [1 + \cos(4\pi f_c t)] + \frac{a_Q(t)}{2} [\sin(4\pi f_c t)] + n(t)
\]
3.3 Low Pass Filter

Since the only thing that’s important at this point is the baseband signal, an LPF will filter the high-frequency components.

```python
import numpy as np
import scipy.signal as sig
from math import pi
```

```python
demodulation.LPF(signal, fc, Fs)
Low pass filter, Butterworth approximation.

Parameters

- **signal** *(1D array of floats)* – Signal to be filtered.
- **fc** *(float)* – Cutt-off frequency.
- **Fs** *(float)* – Sampling frequency.

Returns

- **signal_filt** *(1D array of floats)* – Filtered signal.
- **W** *(1D array of floats)* – The frequencies at which ‘h’ was computed, in Hz.
- **h** *(complex)* – The frequency response.
```
def LPF(signal, fc, Fs):
    o = 5  # order of the filter
    fc = np.array([fc])
    wn = 2*fc/Fs

    [b, a] = sig.butter(o, wn, btype='lowpass')
    [W, h] = sig.freqz(b, a, worN=1024)

    W = Fs*W/(2*pi)
    signal_filt = sig.lfilter(b, a, signal)
    return(signal_filt, W, h)

3.4 Matched Filter

demodulation.matched_filter(signal, template)

Convolutes the baseband signal with the template of the impulse response used in the modulator (Square Root Raised Cosine) to increase the SNR.

Parameters

- signal (1D array of floats) - Baseband signal to be filtered.
• **template** *(1D array of floats)* – Impulse response of the filter used at the signal shaping block

**Returns** `signal_filt` – Filtered signal.

**Return type** 1D array of floats

```python
import numpy as np
def matched_filter(signal, template):
    signal_filt = np.convolve(signal, template, 'full')
    return signal_filt
```

### 3.5 Downsampler

**demodulation.downsampler**(signal, packet_s, upsampler_f)

The algorithm analyzes the synchronization symbols and tries to find the sample where the value of the symbol is maximum. After that, is possible to estimate in which sample the information begins to appear on the signal (i.e. detects the delay)

**Parameters**

- **signal** *(1D array of floats)* – Baseband signal.
- **packet_s** *(int)* – Number of bits in the transmitted packet.
- **upsampler_f** *(int)* – Upsampler factor used at the modulator.
Returns symbols – The sampled symbols.

Return type 1D array of floats

def downsampler(signal, packet_s, upsampler_f):
    e = 0
    gardner_e = []
    peak_sample = 0
    peak_sample_acc = []
    low_point = 0
    threshold = 4
    for i in range(len(signal)):
        if signal[low_point] < -threshold:
            if signal[i] > threshold:
                e = (abs(signal[(i+1)]) - 
                     abs(signal[i-1])) * abs(signal[i])
                gardner_e.append(e)
                if e > 0.8:
                    peak_sample = peak_sample + 1
                    peak_sample_acc.append(peak_sample)
                elif e < -0.8:
                    peak_sample = peak_sample - 1
                    peak_sample_acc.append(peak_sample)
            else:
                break
        else:
            peak_sample = peak_sample + 1
            peak_sample_acc.append(peak_sample)
        else:
            low_point = low_point + 1
            peak_sample = peak_sample + 1
            peak_sample_acc.append(peak_sample)

    # 450 is the number of samples before the convergence symbol of the algorithm.
    cut_i = peak_sample - 450
    cut_f = cut_i + int((packet_s/4)*upsampler_f)
    print("Cut_i = ", cut_i)
    print("Cut_f = ", cut_f)

    # For the code to still work, even when there is a big BER, this section is required.
    if cut_i > 730:
        signal = signal[261:2306+510]
    elif cut_i < 690:
        signal = signal[261:2306+510]
    else:
        signal = signal[cut_i:cut_f]

    symbols = signal[slice(0, len(signal), upsampler_f)]
    return(symbols)
3.6 Demapper

demodulation.demapper(symbols_I, symbols_Q, packetSize, threshold=3.0)

Generates an array of bits using the values based on the 16QAM indexing vector.

- If the symbol amplitude is between 0 and the threshold, it corresponds to the bits 10, if it’s greater than the threshold, it corresponds to the sequence 11.
- If the symbol amplitude is between 0 and -threshold, it corresponds to the bits 01, if it’s lower than -threshold, it corresponds to the sequence 00.

After the recovery of the bits, both vectors (I and Q) are merged, generating the output bitstream.

Parameters

- symbols_I (1D array of floats) – Downsampling in-phase symbols.
- symbols_Q (1D array of floats) – Downsampling quadrature symbols.
- packetSize (int) – Number of bits in the transmitted packet.
- threshold (float, optional) – The limit between two symbols in the 16QAM constellation. The default value is 3.

Returns bitstream – Bits transmitted.

Return type 1D array of ints
import numpy as np

def demapper(symbols_I, symbols_Q, packetSize, threshold = 3.0):
    Ns = int(packetSize/4)
    bits_I = []
    bits_Q = []
    for i in range(Ns):
        if symbols_I[i] >= 0 and symbols_I[i] <= threshold:
            bits_I.append(1)
            bits_I.append(0)
        elif symbols_I[i] > threshold:
            bits_I.append(1)
            bits_I.append(1)
        elif symbols_I[i] < 0 and symbols_I[i] >= -threshold:
            bits_I.append(0)
            bits_I.append(1)
        elif symbols_I[i] < -threshold:
            bits_I.append(0)
            bits_I.append(0)
    bits_I = list(map(int, bits_I))
    bits_Q = list(map(int, bits_Q))
    bitStream = np.zeros(packetSize)
    for i in range(len(bits_I)):
        bitStream[2*i] = bits_I[i]
        bitStream[2*i-1] = bits_Q[i-1]
    return (bitStream)
This example demonstrates the implementation of a 16-QAM communication system. Link: https://youtu.be/vZQu4YDwEKc

The code follows the block diagram below:

```python
import qamfunctions.modulation
import qamfunctions.demodulation
import qamfunctions.channel
import qamfunctions as qf
import matplotlib.pyplot as plt
import numpy as np
from math import pi
import scipy.fftpack as sf

plt.close('all')
```
def QAMsys(SNR, plot=1):
    """16QAM system.

    Args:
        SNR (float): Signal to Noise Ratio (dB)
        plot (int, optional): If it's set to 1 the graphics will be plotted, if it's set to 0 there will be no plots. Default: 1.

    output (float): Bit error rate (BER).
    ""
    # Upsampler Factor
    K = 10

    # Number of symbols
    Ns = 256

    # Roll-off factor
    alpha = 0.3

    # Bits per symbol
    Bs = 4

    # 16QAM Constellation vector
    QAM16 = [-1, -0.333, 0.333, 1]

    # Intermediary frequency
    Fif = 2e6

    # Sampling Rate
    Fs = Fif*K/2

    # Pseudo-Rand Generator with synchronization bits

    data = qf.modulation.data_gen(Ns*Bs)
    if plot == 1:
        plt.figure(0)
        plt.stem(data)
        plt.title('Data')
        plt.grid()

    # Slicer

    (dataI, dataQ) = qf.modulation.slicer(data)

    # Mapper

    mapI = qf.modulation.mapper_16QAM(QAM16, dataI)
    mapQ = qf.modulation.mapper_16QAM(QAM16, dataQ)

    if plot == 1:
        plt.figure(1)
        plt.subplot(2, 1, 1)
        plt.stem(mapI)
        plt.title('Mapper I')
        plt.grid()
plt.subplot(2, 1, 2)
plt.stem(mapQ)
plt.title('Mapper Q')
plt.tight_layout()
plt.grid()

plt.figure(2)
plt.scatter(mapI, mapQ)
plt.title('Constelation IQ out mapper')
plt.xlabel('In-Phase')
plt.ylabel('Quadrature')
plt.tight_layout()
plt.grid()

# Up-sampler
upI = qf.modulation.upsampler(Ns, K, mapI)
upQ = qf.modulation.upsampler(Ns, K, mapQ)

if plot == 1:
    plt.figure(3)
    plt.subplot(2, 1, 1)
    plt.stem(upI)
    plt.grid()
    plt.title('Up-Sampler I')
    plt.subplot(2, 1, 2)
    plt.stem(upQ)
    plt.title('Up-Sampler Q')
    plt.tight_layout()
    plt.grid()

# Shaping filter
[shape_I, a, b] = qf.modulation.shaping_filter(upI, Ns, alpha, Fif, Fs)
[shape_Q, a, b] = qf.modulation.shaping_filter(upQ, Ns, alpha, Fif, Fs)

template = b

if plot == 1:
    plt.figure(4)
    plt.plot(a, b)
    plt.title('SRRC Filter Impulse Response')
    plt.grid()
    plt.figure(5)
    plt.subplot(3, 1, 1)
    plt.plot(shape_I)
    plt.title('Raised Cosine Filter Convolution I')
    plt.grid()
    plt.subplot(3, 1, 2)
    plt.plot(shape_Q)
    plt.title('Raised Cosine Filter Convolution Q')
    plt.grid()
    plt.subplot(3, 1, 3)
    plt.plot(shape_I, shape_Q)
    plt.title('Constalation IQ filter output')
    plt.tight_layout()
    plt.grid()
# Comparison between the FFT of the Upsampler output and Shaping Filter output.

```python
if plot == 1:
    X_f = abs(sf.fft(upI))
    l = np.size(upI)
    fr = (Fs/2)*np.linspace(0, 1, int(l/2))
    xl_m = (2/l)*abs(X_f[0:np.size(fr)])

    plt.figure(6)
    plt.subplot(2, 1, 1)
    plt.plot(fr/1e6, 20*np.log10(xl_m))
    plt.title('Upsampler Output Spectrum')
    plt.xlabel('Frequency(MHz)')
    plt.ylabel('Magnitude(dB)')
    plt.grid()
    plt.tight_layout()

    X_f2 = abs(sf.fft(shape_I))
    l2 = np.size(shape_I)
    fr2 = (Fs/2)*np.linspace(0, 1, int(l2/2))
    xl_m2 = (2/l2)*abs(X_f2[0:np.size(fr2)])

    plt.subplot(2, 1, 2)
    plt.plot(fr2/1e6, 20*np.log10(xl_m2))
    plt.title('Shaping Filter Output Spectrum')
    plt.xlabel('Frequency(MHz)')
    plt.ylabel('Magnitude(dB)')
    plt.grid()
    plt.tight_layout()

# Oscillator

delta_phase = np.random.normal(0, pi/3, 1)
delta_freq = np.random.normal(0, 20, 1)

(loCos_TX, t) = qf.modulation.oscillator(0, 4e-4, 2 / (Fif*K), Fif + delta_freq, delta_phase + pi/2)
(loSin_TX, t) = qf.modulation.oscillator(0, 4e-4, 2 / (Fif*K), Fif + delta_freq, delta_phase)
```

```python
if plot == 1:
    X_f_1 = abs(sf.fft(loCos_TX))
    l_1 = np.size(loCos_TX)
    fr_1 = (Fs/2)*np.linspace(0, 1, int(l_1/2))
    xl_m_1 = (2/l_1)*abs(X_f_1[0:np.size(fr_1)])

    plt.figure(7)
    plt.subplot(2, 1, 1)
    plt.plot(fr_1/1e6, 20*np.log10(xl_m_1))
    plt.title('Spectrum of local oscilator')
    plt.xlabel('Frequency(MHz)')
    plt.ylabel('Magnitude(dB)')
    plt.grid()

    plt.subplot(2, 1, 2)
    plt.plot(t, loCos_TX)
```

(continues on next page)
```python
plt.title('Local Cos')
plt.xlabel('t(s)')
plt.ylabel('Amplitude')
plt.tight_layout()
plt.grid()

# Mixers
mixI = qf.modulation.mixer(shape_I, loCos_TX)
mixQ = qf.modulation.mixer(shape_Q, loSin_TX)

if plot == 1:
    plt.figure(8)
    plt.subplot(3, 1, 1)
    plt.plot(mixI)
    plt.title('Mix I')
    plt.ylabel('Amplitude')
    plt.tight_layout()
    plt.grid()
    plt.subplot(3, 1, 2)
    plt.plot(mixQ)
    plt.title('Mix Q')
    plt.ylabel('Amplitude')
    plt.tight_layout()
    plt.grid()

# Combiner
IF = qf.modulation.combiner(mixI, mixQ)

# Noise
IF_n = qf.channel.AWGN(IF, SNR)

if plot == 1:
    plt.subplot(3, 1, 3)
    plt.plot(IF_n)
    plt.title('Mix IQ')
    plt.ylabel('Amplitude')
    plt.tight_layout()
    plt.grid()

X_f1 = abs(sf.fft(IF_n))
l1 = np.size(IF_n)
fr1 = (Fs/2)*np.linspace(0, 1, int(l1/2))
xl_m1 = (2/l1)*abs(X_f1[0:int(l1/2)])

plt.figure(9)
plt.plot(fr1/1e6, 20*np.log10(xl_m1))
plt.title('IF Spectrum')
plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.tight_layout()
plt.grid()

# Synchronization
```
(loCos_RX, loSin_RX) = qf.demodulation.PLL(IF_n, Fs, len(loCos_TX), K/2)

if plot == 1:
    plt.figure(10)
    plt.plot(IF_n)
    plt.title('Mixed Signal')
    plt.xlabel('Samples')
    plt.ylabel('Amplitude')

plt.figure(11)
plt.subplot(2, 1, 1)
plt.plot(loCos_RX)
plt.plot(loCos_TX)
plt.title("TX and RX Cos")
plt.subplot(2, 1, 2)
plt.plot(loSin_RX)
plt.plot(loSin_TX)
plt.title("TX and RX Sin")

# Mixer
shape_I_demod = qf.modulation.mixer(IF_n, loCos_RX)
shape_Q_demod = qf.modulation.mixer(IF_n, loSin_RX)

if plot == 1:
    plt.figure(12)
    plt.subplot(2, 1, 1)
    plt.plot(shape_I_demod)
    plt.title('Demodulator Mix I')
    plt.ylabel('Amplitude')
    plt.tight_layout()
    plt.grid()

    plt.subplot(2, 1, 2)
    plt.plot(shape_Q_demod)
    plt.title('Demodulator Mix Q')
    plt.ylabel('Amplitude')
    plt.tight_layout()
    plt.grid()

X_f3 = abs(sf.fft(shape_I_demod))
l3 = np.size(shape_I_demod)
fr3 = (Fs/2)*np.linspace(0, 1, int(l3/2))
xl_m3 = (2/l3)*abs(X_f3[0:int(np.size(fr3))])

plt.figure(13)
plt.plot(fr3/1e6, 20*np.log10(xl_m3))
plt.title('Demodulator Mixer Output')
plt.xlabel('Frequency (MHz)')
plt.ylabel('Magnitude (dB)')
plt.grid()

# Low Pass Filter (Butterworth)

fc = 1e6

(continues on next page)
[shape_I_demod_filt, W, h] = qf.demodulation.LPF(shape_I_demod, fc, Fs)
[shape_Q_demod_filt, W, h] = qf.demodulation.LPF(shape_Q_demod, fc, Fs)

if plot == 1:
    plt.figure(14)
    plt.subplot(3, 1, 1)
    plt.plot(W, 20*np.log10(h))
    plt.title('Filter Freq. Response')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Magnitude (dB)')
    plt.grid()

    plt.subplot(3, 1, 2)
    plt.plot(shape_I_demod_filt)
    plt.title('Filtered Signal I')
    plt.ylabel('Amplitude')
    plt.grid()

    plt.subplot(3, 1, 3)
    plt.plot(shape_Q_demod_filt)
    plt.title('Filtered Signal Q')
    plt.ylabel('Amplitude')
    plt.grid()

# Matched Filter

signal_I = qf.demodulation.matched_filter(shape_I_demod_filt, template)
signal_Q = qf.demodulation.matched_filter(shape_Q_demod_filt, template)

if plot == 1:
    plt.figure(15)
    plt.subplot(3, 1, 1)
    plt.plot(a,template)
    plt.title('Template for Matched filter')
    plt.ylabel('Amplitude')
    plt.grid()

    plt.subplot(3, 1, 2)
    plt.plot(signal_I)
    plt.title('Signal I')
    plt.ylabel('Amplitude')
    plt.grid()

    plt.subplot(3, 1, 3)
    plt.plot(signal_Q)
    plt.title('Signal Q')
    plt.ylabel('Amplitude')
    plt.grid()

# Sampling - Gardner Algorithm

symbols_I = qf.demodulation.downsampler(signal_I, len(data), K)
symbols_Q = qf.demodulation.downsampler(signal_Q, len(data), K)
if plot == 1:
    plt.figure(16)
    plt.subplot(2, 1, 1)
    plt.stem(symbols_I)
    plt.title('Demodulated Symbols I')
    plt.grid()
    plt.subplot(2, 1, 2)
    plt.stem(symbols_Q)
    plt.title('Demodulated Symbols Q')
    plt.grid()
    plt.tight_layout()

plt.figure(17)
plt.scatter(mapI, mapQ)
plt.title('Constellation TX')
plt.grid()
plt.figure(18)
plt.scatter(symbols_I, symbols_Q)
plt.title('Constellation RX')
plt.grid()
plt.tight_layout()

# Demapper

data_demod = qf.demodulation.demapper(symbols_I, symbols_Q, len(data))

error = 0.0

for i in range(len(data)):
    if data_demod[i] != data[i]:
        error = error + 1.0

BER = (error*100)/(len(data))

print("\n\n\n")

print("BER:", BER, "%")

print("Phase Offset Through AWGN Channel:", delta_phase/pi, " pi rad")
print("Frequency Offset AWGN Channel:", delta_freq, "Hz")

print("\n\n\n")

return(BER/100)
Analyses the BER to each SNR value.

```python
import matplotlib.pyplot as plt
import numpy as np
import QAMsys as qam

def SNR_BER_analysis():
    plt.close('all')

    SNR = 0
    n_tests_per_snr = 50
    max_snr = 20
    BER_mean_acc = []

    for i in range(max_snr):
        SNR = SNR + 1
        BER = np.zeros(n_tests_per_snr)
        BER_acc = 0

        for i in range(len(BER)):
            BER[i] = qam.QAMsys(SNR,0)
            BER_acc = BER_acc + BER[i]

        BER_mean = BER_acc/len(BER)
        BER_mean_acc.append(BER_mean)

    plt.figure(1)
    plt.scatter(SNR, BER_mean)
    plt.xscale('linear')
    plt.yscale('log')
    plt.xlabel('SNR (dB)')
    plt.ylabel('BER')
    plt.grid()
```

CHAPTER

FIVE

BER ANALYSIS
Chapter 5. BER Analysis
Copyright 2021 Arthur Cruz Morbach, Jonas Dandanel de Castro and Sandro Binsfeld Ferreira

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

6.1 Contact

Questions? Please contact arthur.morbach@hotmail.com
If you are having any trouble please email arthur.morbach@hotmail.com
CHAPTER
EIGHT

INDICES AND TABLES

• genindex
• modindex
• search