ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
School of Computer and Communication Sciences

Handout 3 Modern Digital Communications
Assignment 2 September 15, 2025 (due Sept 23, 2025)

Your Own Communication Toolbox

EXERCISE 1. In this exercise you will implement a set of functions that make up an elementary digital communication system. These functions are a subset of the functionality that the MATLAB *Communication Toolbox* (or the Python *CommPy* package) provides. Unless you are very fast, implementing every function in this assignment may be too much. For this reason, some functions are labeled as optional.

MATLAB Unless otherwise stated, any function that you implement in this exercise should rely only on *built-in* functions. These are functions that are located in *../toolbox/matlab/..*, as opposed to, say, functions that are in the communications toolbox (*../toolbox/comm/..*) or in the signal processing toolbox (*../toolbox/signal/..*). To test if a function is *built-in* or belongs to a toolbox you may type `which` followed by the function name. MATLAB will reply with the function location.

Python For brevity, the statements below will follow MATLAB's syntax. Most of the time, the corresponding Python commands are very similar. When that is not the case, we will appropriately comment and give specific instructions. Most of the functions you need to implement are grouped under `my_utilPDC.py`.

In order to let you test your implementation as thoroughly as you wish, we provide the solutions for each of the functions you are asked to write.

General guidelines: Implement your functions so that they behave as specified by their headers and as shown in the examples. Your functions must check the supplied arguments and display an error if the value of a supplied argument does not make sense. For example, if `my_qamMap` is called with an alphabet size `M` that does not correspond to a square QAM constellation, you might write

```
error('M must be in the form M = 2^(2K), where K is a positive integer.');
```

The general idea is to make your functions foolproof and user friendly.

1. Implement the function `my_qamMap`.
   Your function should use the modulation schemes as shown in Figure 1.

   Hint. You may want to check the help for function `meshgrid`/`numpy.meshgrid`.

   Example:

   ```
   >> my_qamMap(4)
   ans =

     -1.0000 + 1.0000i  -1.0000 - 1.0000i   1.0000 + 1.0000i   1.0000 - 1.0000i
   ```

   If the function is called with invalid arguments, for example if `M` is not of the form 2^{2m} for some positive integer m, an error must be displayed.

2. Implement the function `my_pskMap`.
   Your function should use the modulation schemes as shown in Figure 2.

   Example:

   ```
   >> my_pskMap(4)
   ans =

      1.0000 + 0.0000i   0.0000 + 1.0000i  -1.0000 + 0.0000i  -0.0000 - 1.0000i
   ```

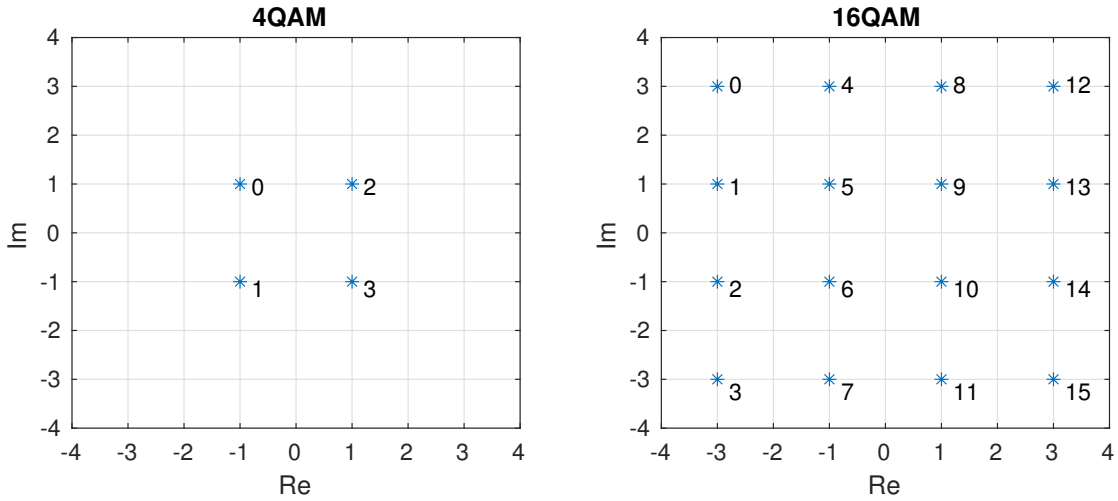

Figure 1: 4-QAM and 16-QAM constellations. The numbers show the correspondence between constellation points and message symbols.

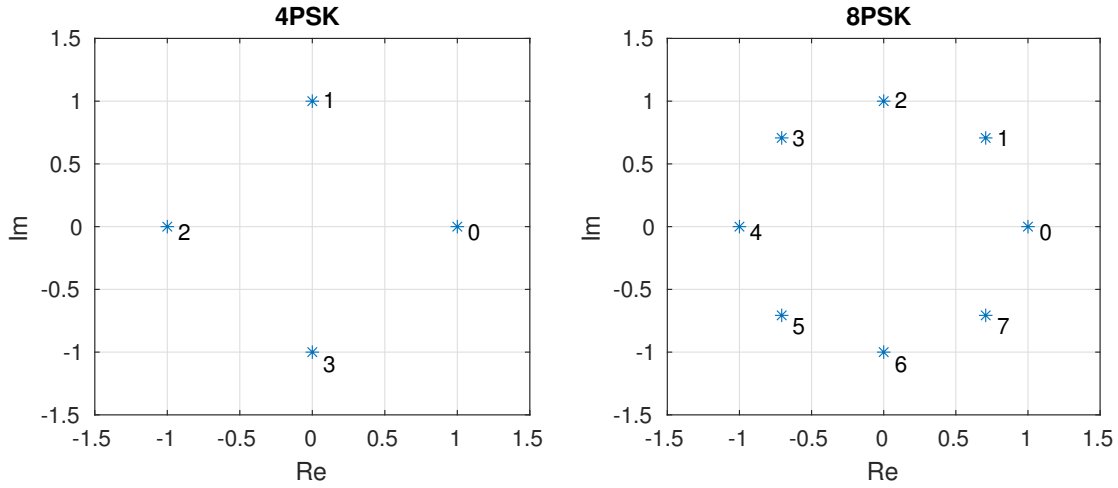

Figure 2: 4-PSK and 8-PSK constellations. The numbers show the correspondence between constellation points and message symbols.

If the function is called with invalid arguments, for example if `M` is not of the form 2^m for some positive integer m, an error must be displayed.

3. Implement the function `my_encoder`.

   If the function is called with invalid arguments, for example if any element of the input is not between 0 and `length(mapping)-1`, an error must be displayed.

   Example: Suppose that the information sequence is `[0 3 3 2 ...]`. The output of the encoder when we use a 4-QAM map can be achieved combining the use of the `my_qamMap` and `my_encoder` functions:

```
>> my_encoder([0 3 3 2 ...], my_qamMap(4))
ans =

  -1.0000 + 1.0000i   1.0000 - 1.0000i  1.0000 - 1.0000i   1.0000 + 1.0000i ...
```

4. Implement the function `my_decoder`.

   Example:

```
>> z = my_decoder([-3+3i, -3+0.7i, 1.1-0.9i, 2.8 + 1.7i], my_qamMap(16))
z =

     0     1    10    13
```

   Hint. Notice that the MATLAB function `min` can be called with two output arguments. In Python, you might want to check `numpy.argmin`.

   Hint. There are multiple ways to implement this function, but you may want to check the help for the command `repmat` or `meshgrid` (Python: `numpy.tile` or `numpy.meshgrid`).

5. Read the documentation of `rcosdesign`/`my_utilPDC.sol_rcosdesign` which will allow you to design a root-raised-cosine filter. We will use it as shaping filter (basic pulse) in the transmitter, and as matched filter in the receiver in the last exercise of this assignment. Notice that the impulse response of the root-raised-cosine filter has infinite length, so we will be using a truncated approximation. Observe also that, at this point, you need to fix the relationship between the symbol time and the sampling time, to which we shall refer as *upsampling factor*.

   - Get the impulse response of the root-raised-cosine filter for different values of the roll-off factor (for instance $\beta = 0, 0.5, 1$). Observe the effect of β both on the plots of the impulse response and of the frequency response.
   - Check that your pulse is normalized and verify that it is essentially orthogonal to its shift by appropriate amounts.

6. Implement the function `my_symbol2samples`.

   The sample-level signal is conveniently obtained by the discrete-time convolution of a modified symbol sequence and the matched filter. The modified symbol sequence is obtained from the actual symbol sequence by inserting a suitable number of zeros between each sample.

   Hint. The following functions may be useful: `upsample, conv, filter`.
   For Python: `numpy.kron, numpy.convolve`.

7. Implement the function `my_sufficientStatistics`.

   To put things into perspective, recall that there are two discrete-time channel models: the one that operates at the sample level and the one that operates at the symbol level. This function takes the sample-level channel output and delivers the symbol-level channel output. Conceptually, this function implements inner products, but it is more efficient to do a matched filter implementation.

   Note. Observe that the function should return a second output vector `y` that contains the matched filter output before downsampling. This vector is used for testing purposes. We will use it in the next assignment.

8. **Optional:** Implement the function `my_bi2de`.

   You should not use the pre-defined function `bi2de`.

   If the function is called with invalid arguments, for example if the data is not binary-valued, or if `MSBFLAG` is an invalid string, then an error must be displayed. For checking the entries of the input data, you might find useful the command `all` (`.all()` for Python).

   Examples:

```
>> my_bi2de([0 1 0 1])
ans =
    10

>> my_bi2de([0 1 0 1; 1 0 1 0])
ans =
    10
     5

>> my_bi2de([1 1 0; 1 0 1; 0 0 1], 'left-msb')
ans =
     6
     5
     1
```

9. **Optional:** Implement the function `my_de2bi`.

   You should not use the pre-defined function `de2bi`.

   Hint. You can use `bitshift` and `bitand` (`numpy.right_shift` and `numpy.bitwise_and`) to efficiently implement this function.

   Hint. You can implement this function without using any loops.

   Examples:

```
>> my_de2bi([12 5])
ans =
     0     0     1     1
     1     0     1     0

>> my_de2bi([12 5], 'right-msb', 5)
ans =
     0     0     1     1     0
     1     0     1     0     0

>> my_de2bi([12 5], 'left-msb')
ans =
     1     1     0     0
     0     1     0     1
```

   If the function is called with invalid arguments, then an error must be displayed.

EXERCISE 2. Using the functions you have written for the different parts of Exercise 1, write a script that simulates the transmission of a sequence of bits over the waveform channel with additive white Gaussian noise. The script must also implement the receiver side, and compute at the end the BER (Bit Error Rate) and SER (Symbol Error Rate) of the transmission.

To facilitate your task, we provide the file `my_errorRatesScript.m[py]` that lists the steps needed to get the job done.

For the shaping filter, design and use a root-raised-cosine filter as in Part 5 of Exercise 1, using a roll-off factor of $\beta = 0.22$.

If you have not completed the optional `my_bi2de` and `my_de2bi` functions, you can use those provided by MATLAB, `bi2de` and `de2bi`. For Python, you can use the solutions.

Hints

- To generate the random bit vector to be transmitted, you can use `randi` (MATLAB) or `numpy.random.randint` (Python).
- There are two obvious ways to add the channel noise. The more pedestrian one is to generate the real and the imaginary parts of the noise using `randn` (`numpy.random.randn`). For this, you need to find out the noise variance so that the resulting E_s/σ^2 at the matched filter output is what you want it to be. In MATLAB, the other obvious way is using the communication toolbox function `awgn`. To use this function you need to specify the symbol energy E_s. This can be computed using the `var` function applied to the symbol sequence to be transmitted.
- You can validate your implementation for 4-QAM, using the exact formulas $P_b = Q(x)$ and $P_s = 2Q(x) - Q(x)^2$ to compute respectively the BER and SER, where $x = \sqrt{E_s/\sigma^2}$. The Q-function $Q(x)$ can be evaluated using `qfunc`. For Python, you can use `scipy.special.erf` and the fact that `Q(x) = 0.5 - 0.5 * erf(x/sqrt(2))`.
- Plotting the constellation of received symbols at the output of the matched filter, after downsampling, can help to debug problems with your implementation. Use a fairly high value of E_s/σ^2.