ULB

WIRELESS COMMUNICATIONS
RESEARCH GROUP
OPERA.ULB.AC.BE

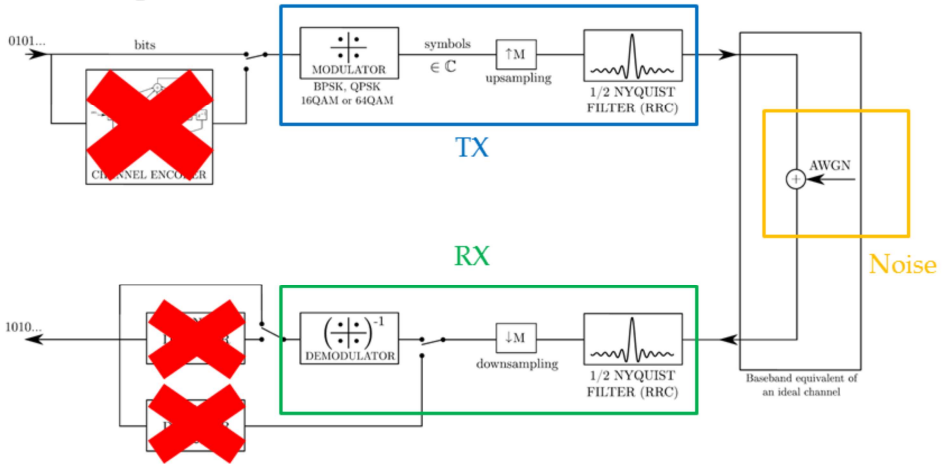
Modulation & Coding: Lab 1

Jean-François Determe (jdeterme@ulb.ac.be)
Trung Hien Nguyen (trung-hien.nguyen@ulb.ac.be)

ULB

Goal of the first lab

- **Goal:** Implement in Matlab: TX, RX, and noise addition.

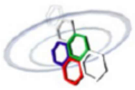


2

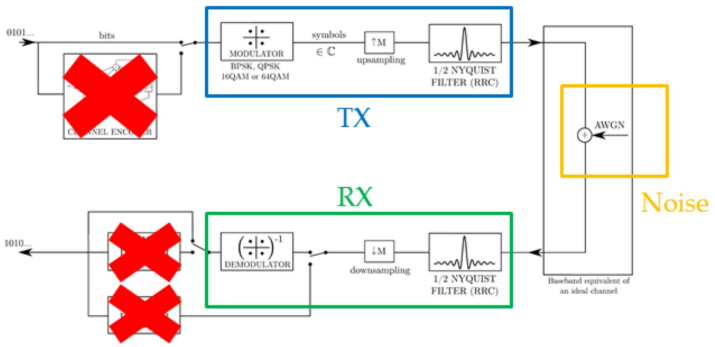
For the first lab., we only look at building the Tx, Rx and additive noise.
The encoder/ decoder is built later on.

ULB

Outline

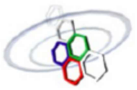


- How to add the AWG noise
- How to design the root-raised-cosine filters
- How to perform the convolutions
- Miscellaneous tips

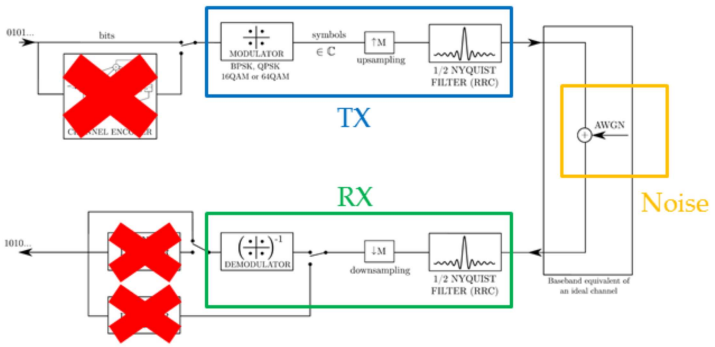


ULB

Outline



- **How to add the AWG noise**
- How to design the root-raised-cosine filters
- How to perform the convolutions
- Miscellaneous tips



ULB

Additive white Gaussian noise (AWGN)

- Noise affects each transmitted sample independently => `signal_noise = signal + noise`
- Additive zero-mean white Gaussian noise: `randn`
- Noise generation

```
sqrt(NoisePower/2) * ( randn(1, length(noise)) +
                       1i*randn(1, length(noise)) )
```

?

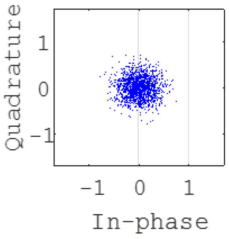
5

We assume that Eb/N0 is known (make it a variable in your code and fix it before doing simulations)
Eb can be easily computed (see next slides)
N0 is easily obtained (see next slides)

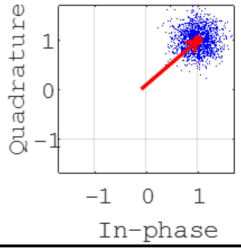
Additive white Gaussian noise (AWGN)

- Bit energy
  - `Eb = SignalEnergy / NbEncodedBit`
  - `SignalEnergy = (trapz(abs(signal).^2))*(1/Fsampling)`
  - Note: `Eb = Eb/2` (Slide 32 of signal representation)
- Noise power
  - `N0 = Eb/EbN0`
  - `NoisePower = 2*N0*Fsampling`
- Example EbN0 = 5 dB

Noise



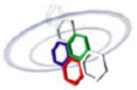
Signal (1+1i) + Noise



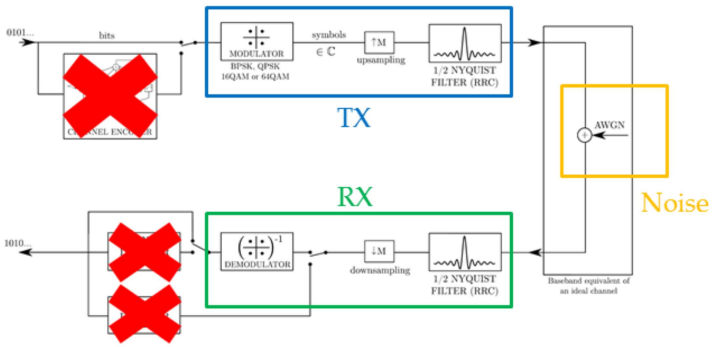
6

ULB

Outline



- How to add the AWG noise
- **How to design the root-raised-cosine filters**
- How to perform the convolutions
- Miscellaneous tips



ULB

Root-raised-cosine (RRC) filter

- Time and frequency domain representation

$$H_{\mathrm{RC}}(f) = \begin{cases} T, & |f| \leq \frac{1-\beta}{2T} \\ \frac{T}{2}\left[1+\cos\left(\frac{\pi T}{\beta}\left[|f|-\frac{1-\beta}{2T}\right]\right)\right], & \frac{1-\beta}{2T} < |f| \leq \frac{1+\beta}{2T} \\ 0, & \text{otherwise} \end{cases}$$

$$H_{\mathrm{RRC}}(f) = \sqrt{H_{\mathrm{RC}}(f)}$$

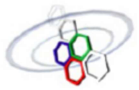
- Idea: designing the filter in frequency domain; impulse response obtained by IFFT

- Parameters for the filter design
  - Roll-off factor (β)
  - Symbol frequency ($1/T$); sampling frequency (*Fsampling*)
  - # taps of the filter: time extension & frequency resolution

8

ULB

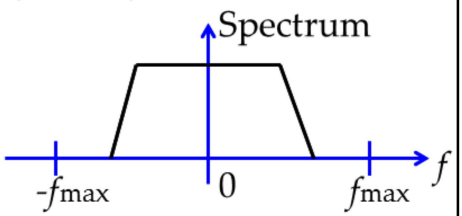
RRC filter
Frequency domain generation



$$H_{\mathrm{RC}}(f) = \begin{cases} T, & |f| \le \frac{1-\beta}{2T} \\ \frac{T}{2}\left[1+\cos\left(\frac{\pi T}{\beta}\left[|f| - \frac{1-\beta}{2T}\right]\right)\right], & \frac{1-\beta}{2T} < |f| \le \frac{1+\beta}{2T} \\ 0, & \text{otherwise} \end{cases}$$

$$H_{\mathrm{RRC}}(f) = \sqrt{H_{\mathrm{RC}}(f)}$$

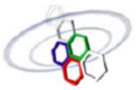
- Frequency grid definition – f variable (odd # taps only)
  - *stepOffset = (1/RRCTaps)*fs;*
  - *highestFreq = stepOffset*(RRCTaps-1)/2;*
  - *freqGrid = linspace(-highestFreq, highestFreq, RRCTaps);*



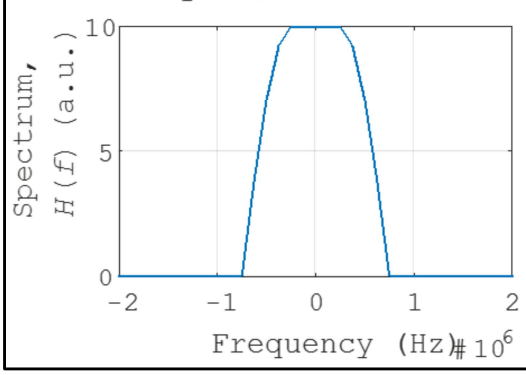
- Note:
  - You may need to normalize $h(t)$ so that it is equal to 1 @ $t = 0$

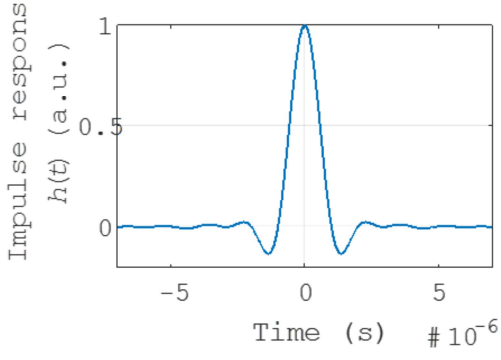
ULB

RRC filter
Time domain generation



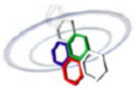
- Time axis (odd number of taps)
  - Time resolution: *Delta_t* = 1/*Fsampling*
  - *t* = (-(*RRCTaps*-1)/2 : (*RRCTaps*-1)/2) * *Delta_t*
- Impulse response: use the *ifft* and *ifftshift* functions
- Example: β = 0.5; *Tsymb* = 1e-6; *Fsampling* = 8e8



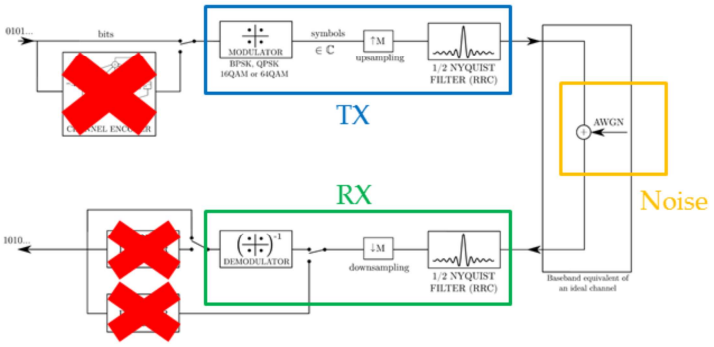


ULB

Outline



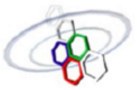
- How to add the AWG noise
- How to design the root-raised-cosine filters
- **How to perform the convolutions**
- Miscellaneous tips



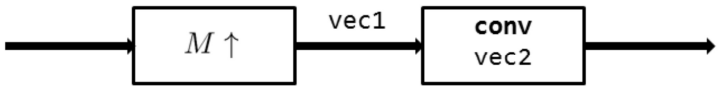
11

ULB

Convolution (1)



- Function to be used: `conv(vec1, vec2)`

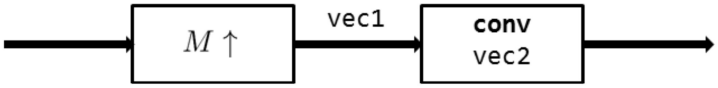


- `vec1` = upsampled symbols
- `vec2` = impulse reponse of the root raised cosine filter
- `RRCTaps` = # elems in `vec2`

12

ULB

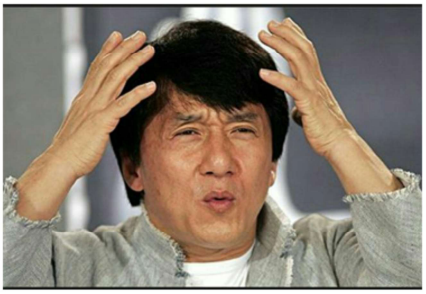
Convolution (2)




```
numel(conv(vec1,vec2)) == numel(vec1) + numel(vec2) - 1
instead of numel(conv(vec1,vec2)) == numel(vec1)
```

- Explanation in the next slides with

1 BPSK symbol (1+0*i)
Oversampling factor M = 4
RRCTaps = 33
Symbol frequency = 1 MHz
Roll-off = 0.3

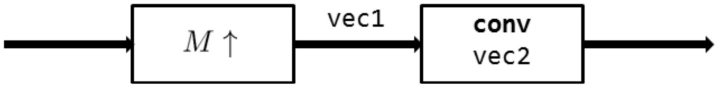


13

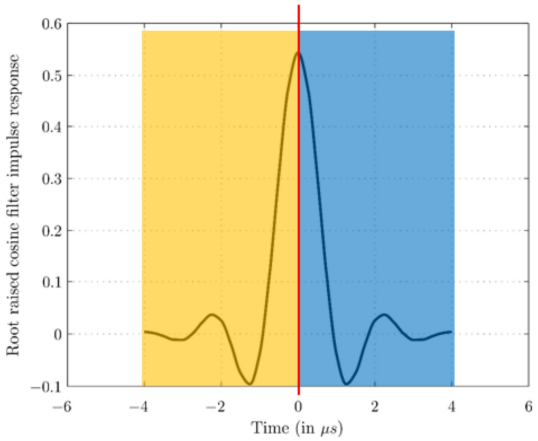
Matlab function numel = number of elements

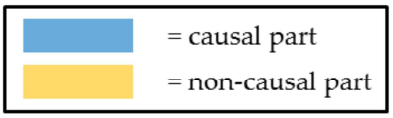
ULB

Convolution (3)



- How the continuous time RRC filter looks like in the time domain:

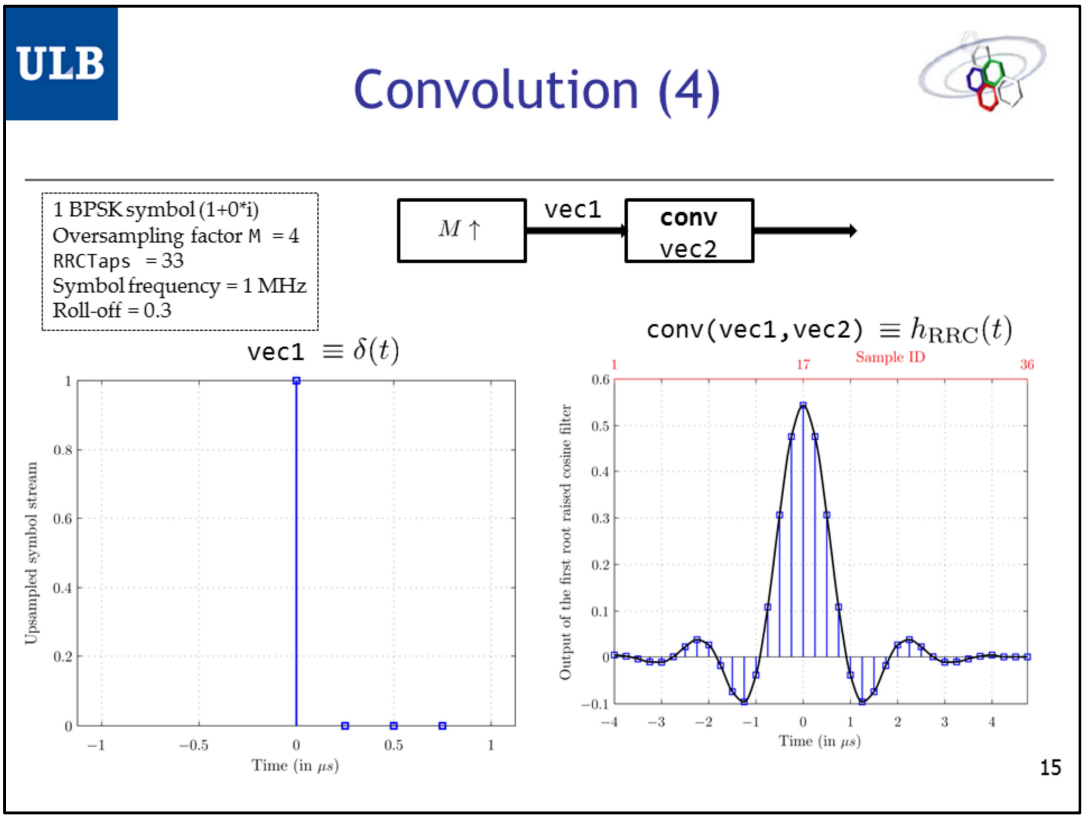




Oversampling factor M = 4
RRCTaps = 33
Symbol frequency = 1 MHz
Roll-off = 0.3

14

Time support is limited to `0.5*(RRCTaps-1)*Ts = 16*(1/4 MHz) = 4 µs` where `Ts` is the time delay separating two consecutive samples at the simulation rate `fs = M*fSymbol = 4 MHz`.
Note that an odd number of taps implies a symmetrical RRC filter.



Left figure:

- The equivalent continuous time signal is the Dirac pulse \delta(t)

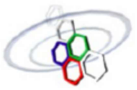
Right figure:

- Dark continuous curve = continuous time signal
- Blue signal = discrete time signal, i.e., it is the continuous time signal sampled at a rate equal to M*fSymbol

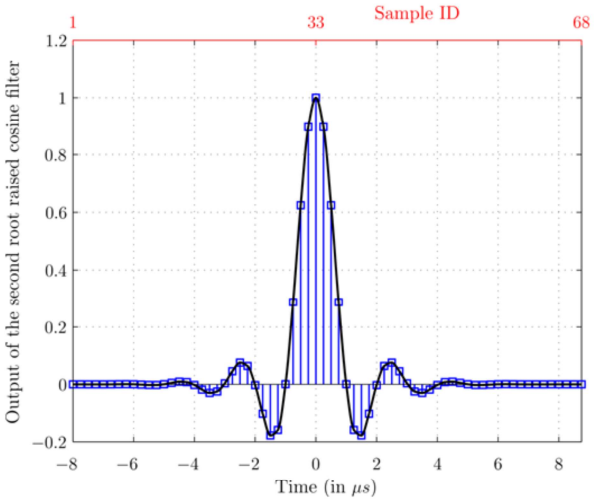
The theory in the continuous time domain predicts that the signal becomes non-zero for negative times (-> non-causal part of the filter).
Similarly, the causal part of the filter extends the duration of the signal.
It is fairly natural to expect that this effect also occurs when doing discrete time simulations.
In practice: physical time t = 0 µs now corresponds to sample `0.5*(RRCTaps-1)+1= 17. Indeed, by discarding the sample corresponding to the physical time t = 0 µs, there are RRCTaps-1 taps left. RRCTaps-1 is an even number in this case which means that the number of taps for the « left » and « right » part of the filter are both equal to (RRCTaps-1)/2.`

ULB

Convolution (5)



- After the convolution at the RX (without noise):
- **In practice:** physical time t = 0 µs now corresponds to sample `2*0.5*(RRCTaps-1)+1 = RRCTaps = 33.`
- Never discard the additional samples in between the two convolutions -> it creates inter symbol interference (ISI)
- Discarding the first `RRCTaps-1` samples at the beginning of the RX signal basically ensures that 1st sample corresponds to physical time t = 0 µs.
- Usually, you want to discard the additional samples at the end of the signal as well.

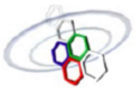


16

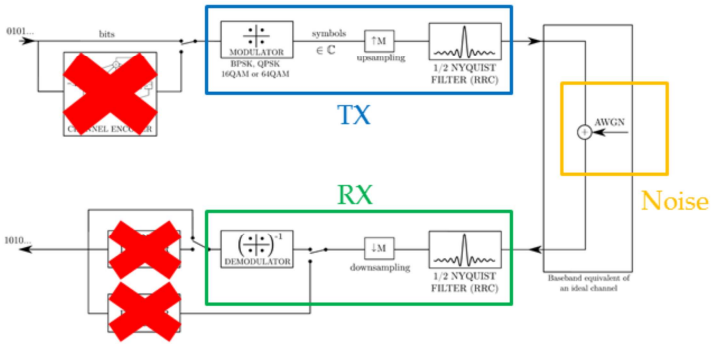
- Black continuous curve = continuous time signal
- Blue signal = discrete time signal, i.e., it is the continuous time signal sampled at a rate equal to M*fSymbol

ULB

Outline



- How to add the AWG noise
- How to design the root-raised-cosine filters
- How to perform the convolutions
- **Miscellaneous tips**



ULB

Tips (1)

- Useful functions/quantities (use `doc [Function name]` for help)

Name	Purpose
`1i`	Variable representing the complex number 0+1*i
`randi`	Generates random integers
`length`	Returns the length of a vector (largest dimension for 2D arrays)
`numel`	Returns the # of elems of a vector or a matrix
`size`	Returns the size of a matrix
`reshape`	Reshapes a matrix/vector to another one of prescribed dimensions -> useful to convert matrices to vectors and inversely
`circshift`	Circular permutation of the columns of a matrix (if vector, ensure that it is a column vector)
`rng`	Fixes the seed of the random number generator

18

- Use 1i instead of i or j in your simulations!
- Numel, size, reshape, and circshift are probably useless for the first lab

ULB

Tips (2)

- Functions `mapping` and `demapping` for converting bits to symbols and symbols to bits, respectively.
- The functions are provided by us and should be included in the folder from which the simulations are run

```
if (bitsPerSymbol > 1)
    txSymbols = mapping(txBits.', bitsPerSymbol, 'qam'); % Symbols at the tx
else % BPSK case
    txSymbols = mapping(txBits.', bitsPerSymbol, 'pam');
end
```

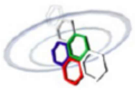
```
if (bitsPerSymbol > 1)
    rxBits = demapping(rxSymbols, bitsPerSymbol, 'qam').';
else % BPSK case
    rxBits = demapping(real(rxSymbols), bitsPerSymbol, 'pam').';
end
```

19

- The input vectors for both functions should be column vectors.
- In the BPSK case, the demapping function requires a real column vector. Taking the real part of the symbol vectors (rxSymbols) basically boils down to projecting the received complex symbols onto the real axis.

ULB

Tips (3)



- Last tips:
  - Never manually inject numbers depending on simulation parameters into your code, use variables instead (e.g. `nbBits, RRCTaps, bitsPerSymbol, EbN0`). Otherwise:
    - Difficult/Time-consuming to change the parameters afterwards in all the functions
    - Debugging made easier if parameters can be easily changed
  - `variable.'` = transpose while `variable'` = Hermitian transpose
  - Use `semilogy` instead of `plot` to plot the BER as a function of Eb/N0.
  - In your reports, explicitly answer the questions of the project statement
- Feel free to use Python (with numpy/scipy) instead of Matlab (if you want)