TX Symbol Modulation, Filtering and PSD

Table of Contents

In this lab you will learn to:

- Create a sequence of QAM symbols
- Design an FIR interpolation filter
- Upsample and modulate the symbols with the designed filter
- Select the appropriate backoff for the TX
- Measure the TX and RX PSD
- Continuously measure the RX PSD to create a simple spectrum analyzer

Submissions: Run the lab with Pluto devices with either one or two hosts. Fill in all sections labeled `TODO` . Print and submit the PDF. Do not submit the source code.

While the lab is ideally run with one or more Pluto devices, I have pre-recorded data if you want to test your results.

```
% Set preRecorded = true if using without a Pluto and you want to run
% from pre-recorded data.  You can save new pre-recorded data with saveData
usePreRecorded = false;
saveData = true;

% Sample frequency
fsamp = 30.72e6;
```

Create Filtered QAM Symbols

First, create a set of QAM-symbols with random bits with the parameters below. Store the random bits in`b` and the QAM symbols in `s`.

```
nsym = 1024;        % number of symbols to transmit
nbitsPerSym = 4;    % number of bits per symbol
M = 2^nbitsPerSym;  % QAM order

% TODO:
%    nbits = ...
%    b = ...
%    s = qammod(b, ...);


% TODO:  Plot the TX constellation
```

Next, we create the TX interpolation filter. MATLAB has an excellent tool `firceqrip` which uses the Parks-Mcclellan algorithm for equiripple design. The function is of the form:

```
b = firceqrip(fillen, fp, [Rp Rst], 'passedge');
```

where `fillen` is the number of FIR filter taps, `fp` is the passband where `fp=1` corresponds to `pi;` and `Rp` and `Rst` are the passband and stopbband errors. The errors are specified in linear scale and can be related to the dB quantities by:

```
RstdB = mag2dB(-Rst);
RpdB =  mag2dB(2*Rp+1);
```

Invert these formulae to design the filter with the above specifications.

```
nov = 2;     % Oversampling ratio
fillen = 80; % Filter length
RpdB = 0.5;  % Max passband ripple in dB
RstdB = 40;  % Min stopband rejection in dB

% TODO
%    fp = ...
%    Rst = ...
%    Rp = ...
%    b = firceqrip(fillen, fp, [Rp Rst], 'passedge');


% TODO:  Plot the transfer function in dB vs. frequency in MHz using
% the freqz command with 1024 points.
%     npts = 1024;
%     [h,f] = freqz(...);
npts = 1024;
```

Compute and print the following:

- The desired passband edge, fpMHz, in MHz
- The stopband edge, fstMHz. This is the smallest frequency such that |`H(f)`| <= stopband target of -40 dB for all `f >= fstMHz`.
- The fraction excess bandwidth = (fst-fp)/fp

Re-plot the frequency response with lines on the passband and stopband edges.

```
% TODO:
%    fpMHz = ...
%    fstMHz = ...
%    excessBw = ...
%    plot(...);
```

Rescale the filter so that it has a DC gain = `nov`. Upsample and filter the symbols with the `filter` command to create the samples. Store the upsampled and filtered symbols in a vector `x`.

```
% TODO:  Rescale the filter
%    ptx = b* ...

% TODO:
%   s1 = upsample(...);
%   x = filter(...)
```

Next, use the `pwelch` command to compute the PSD of `x`. Store the PSD in a vector `Ptx and frequency in a vector fx.` Normally, the PSD is in units of dBW/Hz or dBm/Hz. That is, the power is relative to 1 W or 1 mW. However, in this case, we do not know the actual TX power. So, we will use the units relative to "full scale" or FS. When the digital signal, `|x(n)|=1,` we will say it is full scale. The units of the PSD will be dBFS / Hz meaning the PSD relative to the a digital signal average value of |x(n)|^2=1. Plot the PSD in dBFS/Hz vs. frequency in MHz. Label the axes.

```
% TODO:
%    [Ptx,fx] = pwelch(...);
%    PtxdB = ...
%    plot(...);
```

Setting the Backoff

The Pluto board clips signals such that `real(x)` and `imag(x)` are in the range `[-1,1].` This clipping prevents signals from over-flowing the DACs and other front-end components. It is important that you scale the signal before you send it to the Pluto so that it uses the available range well. Engineers typically use a term *backoff* defined as:

`backoff  = mag2db( xfs / xrms ),  xrms = sqrt(mean(abs(x).^2/2);`

where `xrms` is the mean energy per sample in the I or Q direction (hence the divsion by 2) and `xfs=1` and is the max full scale value. A typical value used is `backoff = 9` or 12 dB.

Complete the following code to see the effect of backoff on the PSDs. You should see that if you have a backoff that is not sufficiently high, the clipping will cause significant noise in the stopband.

```
% Values of backoff to test
backoffTest = [3,6,9];
nbackoff = length(backoffTest);


for i = 1:nbackoff

    backoff = backoffTest(i);
```

```
    % TODO:  Scale the signal so that it is the correct backoff
    %   xb = x* ...

    % TODO:  Clip the signal so that its real and imag components
    % are in [-1,1]
    %    xclip = ...

    % TODO:  Plot the PSD of the clipped signal.  Use a subplot so that the
    % different PSDs are on different windows.  Use the same ylim on all
    % the plots so you can see the effect.
    subplot(1,nbackoff, i);

end
```

For the remainder of the lab, we will use a backoff of 12 dB. Create the scaled signal with this backoff

```
backoff = 12;

% TODO:  Scale and clip the signal so that it is the correct backoff
%   xclip = ...
```

Create the TX and RX objects

We will now transmit and receive the signals with the Pluto devices. Following the initial lab, set the mode to the desired configuration (loopback, two hosts with single host, or two hosts with two hosts).

```
% TODO:  Set parameters
% Set to true for loopback, else set to false
loopback = true;

% Select to run TX and RX
runTx = true;
runRx = true;
```

Plug one or two Plutos into the USB ports of your computer. If it is correctly working, in each device the Ready light should be solid on and the LED1 light should be blinking. Next, run the following command that will detect the devices and create the TX and RX ob

```
% clear previous instances
clear tx rx

% add path to the common directory where the function is found
addpath('..\common');

% Parameters
nsampsFrame = length(xclip);

% Run the creation function.  Skip this if we are using pre-recorded
% samples
if ~usePreRecorded
```

```
    [tx, rx] = plutoCreateTxRx(createTx = runTx, createRx = runRx, loopback = loopback, ...
        nsampsFrame = nsampsFrame, sampleRate = fsamp);
    centerFreq = tx.CenterFrequency;
end
```

TX and RX Data via the Pluto Device

Now TX the clipped data `xclip` repreatedly through the Pluto device.

```
if runTx && ~usePreRecorded
    % TODO:  Use the tx.release and tx.transmitRelease commands to
    % continuously send xclip

end
% If not running the RX, stop the live script now
if ~runRx
    return;
end
```

On the RX Pluto device, receive the data

```
nbits = 12;               % number of ADC bits
fullScale = 2^(nbits-1);  % full scale

if ~usePreRecorded

    % TODO:  Capture data
    %   r = rx.capture(...)


    % TODO:  Scale to floating point
    %   r = ...

    % Save the data
    if saveData
        save symModSing r;
    end

else
    % Load pre-recorded data
    load symModSing;

end
```

Measuring the RX PSD

Using the `pwelch` command measure the PSD in `r`. Plot the PSD in dBW/Hz vs. frequency in MHz.

```
% TODO:  Create an initla plot as before
%    [Prx,fx] = pwelch(...);
```

```
%    plot(...);
```

Create a Simple Spectrum Analyzer

A spectrum analyzer is a device that continuously measures the spectrum. We can create a simple spectrum analyzer by simply capturing the samples and updating the plot. We can do this in a MATLAB live script by:

- Initially plot the data with a command of the form `p = plot(...).` This saves a handle to the plot `p`
- Set the axes of the plot with the `xlim` and `ylim` so they are fixed. This way the plot is not automatically rescaled on each new plot.
- Set pointers to the data with the `p.XDataSource` and `p.YDataSource` variables
- Get new data
- Updating the plot data and calling the `refreshdata` and `plotnow` commands.

Complete the following code to create a spectrum analyzer.

```
if runTx && ~usePreRecorded
    % TODO:  Use the tx.release and tx.transmitRelease commands to
    % continuously send x
end

% TODO:  Capture initial data

% TODO:  Create the inital plot as before
%    [Prx,fx] = pwelch(...);
%    fxMHz = fx/1e6;
%    p = plot(fxMHz, ...);


% TODO:  Fix the limits to some reasonable values
%   xlim([...]);
%   ylim([...]);


% Set pointers to X and Y data
p.XDataSource = 'fxMHz';
p.YDataSource = 'PrxdB';

% Number of loops before the spectrum analyzer stops
ncaptures = 100;

% Load pre-recorded data if required
if usePreRecorded
    load symModMult;
    ncaptures = size(rdat,2);
else
    rdat = zeros(nsampsFrame, ncaptures);
end

for t = 1:ncaptures
    if ~usePreRecorded
```

```
        % TODO:  Capture data
        %    r = rx.capture(...);

        % Save data if required
        if saveData
            rdat(:,t) = r;
        end
    else
        % Load pre-recorded
        r = rdat(:,t);
    end


    % TODO:  Re-compute the PSD
    %    PrxdB = ...

    % Redraw plot
    refreshdata
    drawnow

end

% Save data
if saveData
    save symModMult rdat;
end
```