MWW: Basic Wireless Tutorial Handout

C. Nicolas Barati, Rahman Doost-Mohammady,
Oscar Bejarano and Ashutosh Sabharwal

September 2019

This handout intends to guide you through a set of MATLAB® scripts covering basic concepts in Wireless Communications. The handout is divided into two parts. In Part I, we will use the Matlab-based simulation environment. In Part II, we will use the RENEW experimentation framework as deployed on POWDER framework. In both sections, you will be given a short description followed by a snapshot of incomplete code as your starting point for the exercises.

Preliminaries

You will need to run MATLAB remotely by doing `ssh` with graphics enabled to the POWDER machines. If you have Linux installed on your laptops, you just run `ssh -X usrnm@reomte_server` from a terminal. In case you are using Windows or Mac, here are some instructions:

Instructions for ssh -X on Windows

1. Install the Xming software.
2. Download PuTTY and install it.
3. Run Xming on your PC to start the X server.
4. Run PuTTY and set things up as follows:
   - Enter the remote server name in Host Name
   - Make sure the Connection type is set to SSH
   - Enable X11 forwarding (Connection > SSH > X11)
5. Log in to the remote server.
6. Once you are logged into the Linux system, you can run MATLAB.

Instructions for ssh -X on Mac

1. Install XQuartz on your Mac.
2. Run Applications > Utilities > XQuartz.app
3. Right click on the XQuartz icon in the dock and select Applications ¿ Terminal. This should bring up a new xterm terminal windows.
4. In this xterm windows, ssh into the linux system of your choice using the -X argument (secure X11 forwarding). `ssh -X usrnm@remote_server`
5. Once you are logged into the linux system, you can MATLAB and it will display on your Mac.

Simulation

1.1 QAM Modulation

As described in the presentation, in QAM modulation a set of bits is represented on a complex 2-D plane, where each axis is a sinusoid in 90^o phase difference. Each point on the complex plane consists of K bits while M is the modulation order.

Step I: Open the Matlab function `bits2syms.m`. The inputs are `bitstream` and `MOD_ORDER`. The latter can be one of 2, 4, 16 and 64. Before invoking the function, complete the following code by giving the number of bits corresponding to each symbol; this number is a function of `MOD_ORDER`. Also, check if the number of the given bits is divisible by the number of bits in a symbol.

```
% Do yourselves: derive the number of bits per symbol
nbits = length(bitstream);
sym_bits = ..;
if ..
    error("Length of bit stream has to be divisible by sym_bits");
end
```

Step II: After the above step, create a vector of random bits and call `bits2sym`. Test the four different modulation orders. Make sure your bit stream is large enough so in higher modulation orders you have a constellation point for every combination of bits. Note that the implementation of modulation is not unique. In fact, there are better (more intuitive) ways of mapping bits to complex symbols compared to what you see here.

1.2 QAM Demodulation

Here, you will demodulate a series of complex symbols received through an AWGN or Rayleigh channel. We assume perfect channel knowledge for now.

Step I: Open `mod_demod.m` and `syms2bits.m`. As you see, the first one is just a script calling `bits2syms` and `syms2bits` and plotting the modulated and demodulated symbols. The function `syms2bits` acts as both channel and demodulator. Complex baseband symbols that have undergone some distortion due to noise and the baseband channel are first equalized and then demodu-

lated to integer representations based on the decision regions defined for each modulation order. Again, this is not a unique way of mapping symbols/bits to integers/bits.

Step II: In `syms2bits` complete the following code to get the bits from the demodulated symbols:

```
%  Do your selves: Get bits from rx_data:
bits_matrix = .. ; % This is a matrix
bits = bits_matrix(:);
```

Then, in `mod_demod.m` calculate symbol and bit error rates.

```
% Do your selves: Calculate error rates:
sym_err = .. ;
bit_err = ..;
fprintf("Symbol Error Rate = %12.4e\t Bit Error Rate = ...
    %12.4e\n", sym_err, bit_err );
```

After executing your scripts, you will see plots depicting modulated and demodulated symbols.

Step III: Try different SNRs and change the channel to Rayleigh by setting the argument `awgn` to 0 for the same modulation order. Try different modulation orders. Observe how the plots and the error rates change based on these inputs.

1.3 OFDM SISO

Here we will run a simulation of an OFDM system. OFDM is widely used in modern wireless communications systems. We will leverage on the mod/demod functions we saw earlier and focus on frequency domain to time domain, and vice versa, symbol transition, equalization, and channel estimation. You will also test various channel models and observe their effect on the effective SNR and error rates. The script has several parts. They are, at the transmitter: modulation, sub-carrier assignment, time-domain conversion, CP and finally hand-over to the channel. At the receiver, we have synchronization through correlation, frequency domain conversion, per-sub-carrier equalization, and demodulation. Finally, we plot the results. Note that in simulation mode we loop over a range of SNRs `nt` times to assess the BER for a given SNR

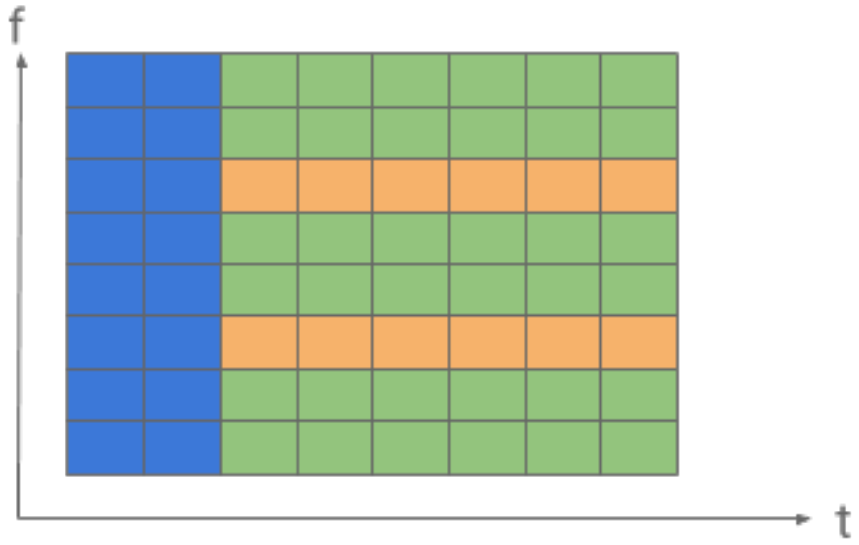

Figure 1: OFDM frame structure. Data frames in green, phase noise pilots in light orange and LTS used for synchronization and channel estimation in light blue.

Let us first look at the OFDM frame structure. It is shown in figure 1. The light blue portion is occupied by synchronization signals (LTS [1]) that will also be used for channel estimation. The light orange squares are for pilots used to mitigate frequency domain errors due to phase noise. The green ones are reserved for data.

Step I: Open `ofdm_siso.m`. Complete the following code to obtain time-domain samples. First, you need to assign data symbols and phase noise pilots to their respective sub-carriers. Then, transform frequency-domain symbols to time-domain samples.

```
% Do yourselves: construct the TD input matrix
fdd_mat = zeros(N_SC, N_OFDM_SYM);

% Insert the data and pilot values; other sub-carriers will ...
    remain at 0
fdd_mat(.., :)   = ..;
fdd_mat(.., :) = ..;

% Do yourselves: get TD samples:
tdd_tx_payload_mat = ..;
```

[1] LTS stands for long training symbols.

Step II: Now, add the CP to the time domain matrix. Remember, CP is essentially the last `CP_LEN` samples of each OFDM symbol copied and appended to the beginning the same OFDM symbol. When you are done, each OFDM symbol must have `N_SC+CP_LEN` samples.

```
if(CP_LEN > 0)
    % Do yourselves: Insert CP
    tx_cp = tdd_tx_payload_mat(.., :);
    tdd_tx_payload_mat = ..; % tdd_tx_payload_mat is CP and itself.
end
```

Step III: Upon reception, the receiver performs a correlation with the local replica of the known LTS to determine the start of the frame. Then, it will gather the samples belonging to the two LTS, bring them to frequency-domain and perform channel estimation. Complete the following code for the latter two steps.

```
% Received LTSs
% Do yourselves: take the FD pilots:
rx_lts1_f = ..;
rx_lts2_f = ..;

% Do yourselves: Calculate channel estimate from average of 2 ...
    training symbols.
% Remove the LTSs and average over the two columns.
rx_H_est = ..;
```

Step IV: Now that we have an estimate of the channel per each sub-carrier (`rx_H_est`), we can equalize, i.e., remove the effect of the channel from our data symbols. Complete the following code. Be careful with the dimensions of the matrices and vectors. You will need to repeat/expand one of them to match the other.

```
% Do yourselves: bring to frequency domain:
syms_f_mat = ..;

% Do yourselves: Equalize.
syms_eq_mat = .. ;
```

Step V: One last thing before moving on to the next section. As mentioned in the slides, we can have an estimate of the (effective) SNR based on how far the demodulated symbols are from the transmitted ones. The effective SNR depends on the square of their distances. This is especially useful in cases we don't directly know the SNR. Here in our simulations, we define the SNRs.

However, next when we will use the RENEW-POWDER equipment, we cannot set the SNR, and EVM is a nice way of SNR estimation. Complete the following code to calculate the EVM and the effective SNR.

```
% EVM & SNR
% Do yourselves. Calculate EVM and effective SNR:
evm_mat = ..;
aevms = ..; % needs to be a scalar
snr = ..; % calculate in dB scale.
```

1.4 Channel Models

Now that we have an OFDM system ready, we can start experimenting with various chanel models. We consider three simulated models here AWGN, smoothed Reyleigh and Multipath Clusetered (MPC). Rayleigh is smoothed and not i.i.d. because otherwise our channel estimation cannot keep up with such rapid changes that an i.i.d. channel would cause. In `getRXVec.m`, you can see the different channel implementations. Note that so far we have been using AWGN by default.

Step I: Take a look at the MPC portion of the code. Here are the parameters of the channel.

```
        nsub = 20;      % number of subpaths per cluster
        fmaxHz = 0;     % max Doppler spread in Hz. NOT USED.
        freqErrHz = 0;  % constant frequency error in Hz. KEEP ...
            at 0.
        dlyns=0;        % constant delay in ns
        angMotion=0;    % angle of motion. KEEP at 0.

        % Cluster parameters represented as a vector with one
        % component per cluster
        angcRx = [0 90]'*pi/180;        % RX center angle in radians
        angspdRx = 10*pi/180;           % RX angular spread in ...
            radians
        angcTx = 0;                     % TX center angle in radians
        angspdTx = 0;                   % TX angular spread in ...
            radians
        dlycns = [0 100]';              % excess delay of first ...
            path in cluster nsec
        dlyspdns = 30;                  % delay spread within ...
            cluster in nsec
        powcdB = [-3 -3]';              % gain power in dB 1/2 ...
            the power on each cluster
        fadec = [1 1]';                 % fadec(i)= 0 => cluster ...
            i is non-fading
```

Here we are simulating two clusters coming at 0^o and 90^o each. They have a certain angular spread and relative delays. Also, there are random delays between each ray of each cluster.

You may set the channel model in `ofdm_siso.m` by changing the variable `chan_type` in `chan_type = "rayleigh";` to either on of **`awgn`** (default), **`rayleigh`** or **`mpc`**.

Step II: Play a bit with various channel models and tweak the parameters of the MPC channel and see their effect on the BER and other metrics and plots in `ofdm_siso.m`. You can set the channel model to one of the options given. You can also tweak the noise/channel variance, and/or, in MPC, parameters such as power fraction on each cluster (dB), angles of arrival, delay spread between clusters and rays etc. Note that our MPC is static and hence anything related to Doppler and Doppler shift is kept to zero. Compare MPC with smoothed Rayleigh and AWGN.

1.5 OFDM SIMO

Now as our first multi-antenna exercise, we will look at the performance of SIMO. Here we use only two receivers because we wanted to illustrate the difference in performance between SIMO MRC and each branch individually, and plotting N different branches would be cumbersome.

Step I: Since we use the same OFDM system, most of the code is identical to the OFDM SISO case. You are only asked to do the MRC coherent combining by completing the following code.

```
% Equalize MRC
rx_H_est = reshape(rx_H_est_2d,N_SC,1,N_BS_NODE);       % Expand ...
    to a 3rd dimension to agree with the dimensions od syms_f_mat
% Do yourselves: normalization coefficient:
H_pow = ..;
H_pow = repmat(H_pow,1,N_OFDM_SYM);

% Do yourselves: MRC equalization:
syms_eq_mat_mrc =  ..; % MRC equalization: combine The two ...
    branches and equalize.
% Don't forget to normalize by the channel power!
```

The equation for MRC combinig is:

$$\hat{x} = \frac{\mathbf{h}^*}{||\mathbf{h}||^2}\mathbf{y}, \tag{1}$$

where $\hat{x}$ is the equalized frequency domain symbol (on a sub-carrier), $\mathbf{y}$ is the received symbol from two antennas, $\mathbf{h}$ is the channel and $(.)^*$ denotes conjugate transpose.

Step II: Try different channel models as you did in SISO and see what changes, and how much, in the plots and our metrics of interest (EVM, BER etc.)

Note: When implementing the MRC equation, be careful about the dimensions of the matrices you are about to manipulate.

1.6 OFDM MIMO

In this final section of simulations we will look into multiuser massive MIMO. You will implement Zero forcing and Conjugate beamforming. As with SIMO, we still simulate an OFDM system, and thus we can build on our previous exercises.

M-MIMO relies on pilot orthogonality. We have placed our pilots in different OFDM symbols as you see in the following excerpt of the code. Figure 2 depict this time orthogonality. Furthermore, to keep things simple we simulate only one channel model, Rayleigh.

```
% Arrange time-orthogonal pilots
preamble_common = [lts_t(33:64); repmat(lts_t,N_LTS_SYM,1)];
l_pre = length(preamble_common);
pre_z = zeros(size(preamble_common));
preamble = zeros(N_UE * l_pre, N_UE);
for jp = 1:N_UE
    preamble((jp-1)*l_pre + 1: (jp-1)*l_pre+l_pre,jp) = ...
        preamble_common;
end
```

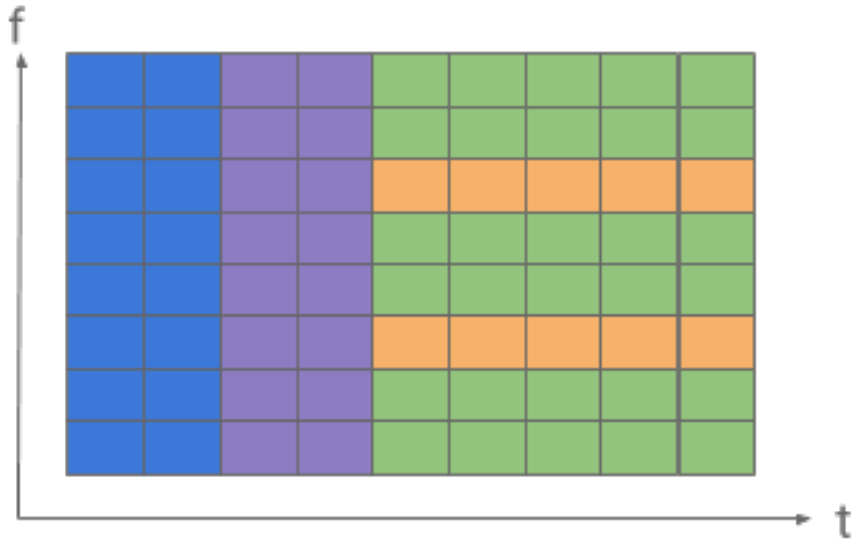

Figure 2: OFDM frame structure. Data frames in green, phase noise pilots in light orange and LTS used for synchronization and channel estimation in light blue for user 1, and light purple for user 2.

The equations for ZF is

$$\hat{\mathbf{x}} = \mathbf{H}^{\dagger}\mathbf{y} \tag{2}$$

and for conjugate BF

$$\hat{\mathbf{x}} = \frac{\mathbf{H}^{*}}{f(||\mathbf{H}_{[i,i]}||^2)}\mathbf{y}, \tag{3}$$

where where $\hat{\mathbf{x}}$ is the vector of equalized frequency domain symbols and contains one symbol $\hat{x}_i^j$ for each user i on each sub-carrier j, $\mathbf{y}$ is the received symbol vector from M antennas, $\mathbf{H}$ is the channel matrix, $\mathbf{H}^{\dagger} = (\mathbf{H}^{*}\mathbf{H})^{-1}\mathbf{H}^{*}$ is the pseudo inverse of $\mathbf{H}$, $f(.)$ is some normalizing function of the channel power that you need to find, and $(.)^{*}$ denotes conjugate transpose.

Step I:Complete the following code to implement ZF and conjugate BF.

```
for j=1:length(nz_sc)

    if(strcmp(MIMO_ALG,'ZF'))
       % Do yourselves: calculate pseudo-inverse and apply to ...
           the received symbols:
        HH_inv = ..;
        x = ..;
    else
        % Do yourselves: calculate the normalization coefficient ...
            and apply conjugate BF:
        % Normalization coeff:
        H_pow = ..;
        % Apply BF:
        x = ..;
    end


end
```

You may change the MIMO algorithm by replacing `"ZF"` in `MIMO_ALG = 'ZF';` to `"conj"` (or anything really!).

Step II: By running the script you will see various plots showing the performance of your MIMO implementation. Change the SNR and the number of BS antennas and see how the plots and the performance change. Note however, that as the number of nodes increases, the plots become more crowded.

2 RENEW-POWDER Platform

The experimentation platform's hardware consists of a massive MIMO BS and two clients. The two clients are two Iris boards and the BS is essentially a collection of 32 Iris boards, each with dual polarization, grouped in chains and connected through a hub. In this session, however, we will use only one chain of eight antennas. The supporting software with which you will interact today consists of these MATLAB scripts, a MATLAB driver and a python driver. The role of each was explained in the slides. Before running some experiments let us review some features of our massive MIMO platform.

TDD Frame Schedule: The system operates in TDD mode, i.e., the same bandwidth is used for both Downlink and Uplink but at different time slots.

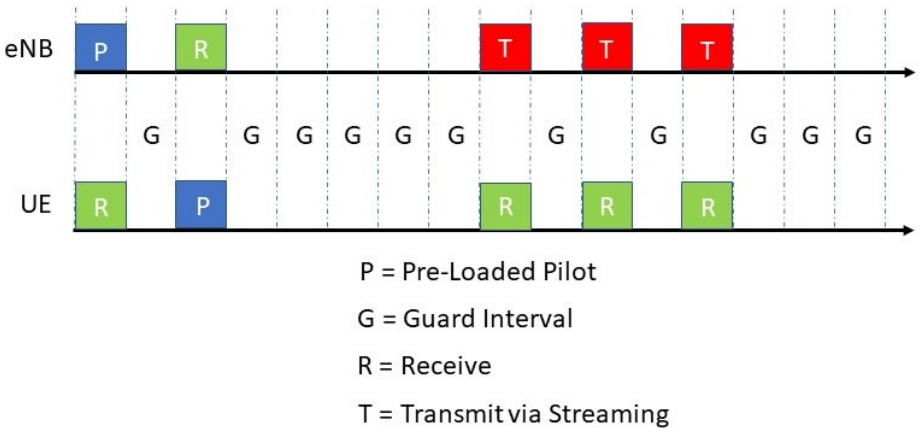

Figure 3: TDD Frame

The experimenter can design a frame schedule and give it as input to both the BS and the clients. An example of such a frame design is shown in figure 3. Each slot can be in one of four modes: R, for reception, G, for guard-band, T for transmission and P for pilot. The notion of P may be a bit confusing, since we will be using this mode for transmitting not only our pilots (e.g, LTS) but also our data, and will not use T. P essentially denotes transmission of data stored onto the boards' FPGA RAM block.

Block RAM Transmission: While the system provides a streaming functionality, we use Block RAM TX. This way, during P slots the boards transmit the content stored on their RAM. This can be, as said above, either some sort of pilot or data. Since throughout this tutorial we only look into Uplink transmission, only the clients will transmit data. They will do so every time there is a P slot in their schedule. Note that the amount of samples that can be transmitted at each P slot is upper-bounded by the size of the RAM, which is 4096. Therefore, the total number of (sub-carriers + CP) $\times$ OFDM symbols should be below 4096.

The communication starts as follows. Both the clients and the BS boards are given their respective schedules. When there is a P slot on the BS side, a beacon sequence is broadcast through only one of the available boards. Upon detecting this beacon through correlation, the clients start counting slots and frames and are synchronized with the base station. They know that the beacon slot was the first slot of the frame and they make out the rest of the schedule relative to that. Now, since we use *Block RAM TX*, during the P slots of the clients, they transmit what they have in their RAM. The implementation of the whole

synchronization and beacon TX/RX is done under the hood and the MATLAB user does not need to bother. The only thing that is defined in MATLAB is the schedule for the clients and the BS boards.

2.1 Setting up the RENEW-POWDER MATLAB environment

In this section, we look into how to set up the experimentation platform first through a basic SISO OFDM.

Observe the following code in `ofdm_siso.m`.

```
% Create two Iris node objects:
    b_ids(end+1) = "0339";
    ue_ids(end+1) = "RF3C000045";

    b_prim_sched = "PGGGGGRG";           % BS primary noede's ...
        schedule: Send Beacon only from one Iris board
    ue_sched = "GGGGGGPG";               % UE schedule

    b_scheds =  b_prim_sched;

    ue_scheds = string.empty();
    for iu = 1:N_UE
        ue_scheds(iu,:) = ue_sched;
    end

    n_samp = length(tx_vec_iris);

    % Iris nodes' parameters
    sdr_params = struct(...
        'id', b_ids, ...
        'n_chain',N_BS_NODE, ...
        'txfreq', TX_FRQ, ...
        'rxfreq', RX_FRQ, ...
        'txgain', TX_GN, ...
        'rxgain', RX_GN, ...
        'sample_rate', SMPL_RT, ...
        'n_samp', n_samp, ...          % number of samples per ...
            frame time.
        'tdd_sched', b_scheds, ...     % number of zero-paddes ...
            samples
        'n_zpad_samp', (N_ZPAD_PRE + N_ZPAD_POST) ...
        );

    sdr_params(2) = sdr_params(1);
    sdr_params(2).id =  ue_ids(1);
    sdr_params(2).n_chain = 1;
    sdr_params(2).txfreq = TX_FRQ;
```

```
35      sdr_params(2).rxfreq = RX_FRQ;
36      sdr_params(2).tdd_sched = ue_scheds(1);
37
38      rx_vec_iris = getRxVec(tx_vec_iris, N_BS_NODE, N_UE, ...
            chan_type, [], sdr_params(1), sdr_params(2));
```

We begin by defining the ids of the boards we will be using. The ones in the example above are different than what we will use today.

Next, we will define the TDD frames' schedule for both the BS and the client (or, UE as noted in the script). Then, we create structs holding the parameters of the nodes. As you can see these parameters are things such as schedule, ID, sample rate, TX/RX frequency number of samples per transmission slot etc. Finally, we call `getRxVec()` with our two structs as input. Before running, make sure that `SIM_MOD` is set to 0 and `chan_type` is now `"iris"`.

Now open `getRXVec.m` to see the data flow before handing over to the MATLAB driver. Notice that this file can support up to two clients, which is fine since this is the maximum number of clients at our disposal.

```
1  elseif chan_type == "iris"
2  % Real HW:
3      N_ZPAD_PRE = 90;
4      n_samp = bs_param.n_samp;
5      node_bs = iris_py(bs_param);        % initialize BS
6      node_ue1 = iris_py(ue_param(1));    % initialize UE
7      if n_ue  >1
8          node_ue2 = iris_py(ue_param(2));
9      end
10
11     node_ue1.sdr_txgainctrl();
12     if n_ue  >1
13         node_ue2.sdr_txgainctrl();      % gain control
14     end
15
16     node_bs.sdrsync(1);                 % synchronize delays ...
           only for BS
17
18     node_ue1.sdrsync(0);
19     if n_ue  >1
20         node_ue2.sdrsync(0);
21     end
22     node_ue1.sdrrxsetup();             % set up reading stream
23     if n_ue  >1
24         node_ue2.sdrrxsetup();
25     end
26     node_bs.sdrrxsetup();
27
28     chained_mode = 0;
29     node_bs.set_config(chained_mode,1); % configure the BS: ...
```

```
            schedule etc.
30
31      node_ue1.set_config(chained_mode,0); % configure the UE: ...
            schedule etc.
32      if n_ue  >1
33          node_ue2.set_config(chained_mode,0);
34      end
35
36      node_bs.sdr_txbeacon(N_ZPAD_PRE);   % Burn beacon to the ...
            BS(1) RAM
37
38      node_ue1.sdrtx(tx_data(:,1));       % Burn data to the UE RAM
39      if n_ue  >1
40          node_ue2.sdrtx(tx_data(:,2));
41      end
42      node_bs.sdr_activate_rx();          % activate reading stream
43
44      node_ue1.sdr_setcorr()              % activate correlator
45      if n_ue  >1
46          node_ue2.sdr_setcorr()
47      end
48
49      % Iris Rx
50      % Only UL data:
51
52      [y, data0_len] = node_bs.sdrrx(n_samp); % read data
53
54      node_bs.sdr_close();                % close streams and exit ...
            gracefully.
55      node_ue1.sdr_close();
56      if n_ue  >1
57          node_ue2.sdr_close();
58      end
59      fprintf('Length of the received vector from HW: \tUE:%d\n', ...
            data0_len);
```

The above portion of the code calls several functions of the MATLAB driver. The driver in return will "translate" these calls into Python and call the respective functions in the Python driver. It begins by calling a function that instantiates two containers of Python objects. One with all the boards of the BS, in case we use more than one board, and one for the client. Next, various internal functionalities such as gain control and delay synchronization between the BS boards are set. After that, reading streams from the HW boards are set up and will be activated a few calls down. Next, the BS and UEs are configured and beacon and data are stored onto the BS and client's respective RAM. On the UE the correlator is set up to expect the beacon and through a trigger the BS starts the frame count. The latter step is pushed to the MATLAB driver. In each P slot, it will send the beacon and during each R slot, it will read data. Finally, when all the slots of the TDD frame schedule are parsed, the nodes are shut down.

2.2 Experimentation

We have covered all the material on basic wireless comm. and MIMO in the simulation section. Here you are asked to repeat the same process but on data obtained via a real channel instead of simulation. The BS is a common resource and unfortunately, there is no way of parallelization. Each group is asked to run their experiments one after the other. Also, to make things smoother and faster we ask that only one person from each group run an experiment. After a successful run, the group representative can save the data returned by `getRXVec()` and share it with her group members.

If you want to run the scripts on stored data, first set `SIM_MOD` to 1. Then, set `chan_type` to `"past_run"`. Finally, in `getRXVec.m`

```
elseif chan_type == "past_run"
    % Past data
    old_data = load(past_data_fname);
    y = old_data.rx_vec_old_data;
```

replace `past_data_fname` and `rx_vec_old_data` with the names you used when storing your data. After these steps, you are ready to run your scripts on stored data.

2.2.1 SISO Experiments

If you have set up everything according to Sec. 2.1, you can go ahead and run your script. If the run is successful, store your data. You may have to run the script several times to get sound data.

Compare what you see with simulation.

2.2.2 SIMO Experiments

Not much changes here. You just need to provide two valid IDs for the BS nodes in `ofdm_simo.m`. Consult `POWDER_arrays.txt` and use the board next to the one you used for SISO.

Save your data and compare against simulation.

2.2.3 MIMO Experiments

Finally, run `ofdm_mimo.m`. Here too, you need to provide valid BS node IDs. Consult the ID txt file we have given you.

Upon a successful run, save your data. If you used ZF, when collecting data, run the script in simulation mode with conjugate BF this time, and vice versa. As in other cases, compare your results with what you get in simulation.