How do I process CSI with MATLAB or Octave? [−]

A. Parsing the CSI trace file

Using MATLAB/Octave, change to the *matlab* directory in the CSI Tool supplementary material:

```
cd linux-80211n-csitool-supplementary/matlab
```

Now read in the CSI trace file. A sample file is included in the supplementary material, but you can also use the file that is generated by following the last step of the installation instructions.

```
csi_trace = read_bf_file('sample_data/log.all_csi.6.7.6');
```

Note that this uses a MEX-file compiled from `read_bfee.c` to unpack the binary CSI format. If this does not work, recompile this MEX-file using MATLAB/Octave and try again.

B. Inspecting the CSI

In our sample file, `csi_trace` is a 1x29 cell array, which holds 29 structs. This contains the CSI information for 29 received packets. Let's inspect one of the entries:

```
>> csi_entry = csi_trace{1}     (Note the curly-braces {}, not parentheses ().)

csi_entry =

    timestamp_low: 4            (In the sample trace, timestamp_low is invalid and always 4.)
       bfee_count: 72
              Nrx: 3
              Ntx: 1
           rssi_a: 33
           rssi_b: 37
           rssi_c: 41
            noise: -127
              agc: 38
             perm: [3 2 1]
             rate: 256
              csi: [1x3x30 double]
```

Let's break down this display:

- **`timestamp_low`** is the low 32 bits of the NIC's 1 MHz clock. It wraps about every 4300 seconds, or 72 minutes. This field was not yet recorded in the sample trace, so all values are arbitrary and always equal `4`.
- **`bfee_count`** is simply a count of the total number of beamforming measurements that have been recorded by the driver and sent to userspace. The `netlink` channel between the kernel and userspace is lossy, so these can be used to detect measurements that were dropped in this pipe.
- **`Nrx`** represents the number of antennas used to receive the packet by this NIC, and **`Ntx`** represents the number of space/time streams transmitted. In this case, the sender sent a single-stream packet and the receiver used all 3 antennas to receive it.
- **`rssi_a`**, **`rssi_b`**, and **`rssi_c`** correspond to RSSI measured by the receiving NIC at the input to each antenna port. This measurement is made during the packet preamble. This value is in `dB` relative to an internal reference; to get the received signal strength in `dBm` we must combine it with the Automatic Gain Control (AGC) setting (**`agc`**) in `dB` and also subtract off a magic constant. This process is explained below.
- **`perm`** tells us how the NIC permuted the signals from the 3 receive antennas into the 3 RF chains that process the measurements. The sample value of **`[3 2 1]`** implies that Antenna C was sent to RF Chain A, Antenna B to Chain B, and Antenna A to Chain C. This operation is performed by an antenna selection module in the NIC and generally corresponds to ordering the antennas in decreasing order of RSSI.

- **`rate`** is the rate at which the packet was sent, in the same format as the `rate_n_flags` defined above. Note that the antenna bits are omitted, as there is no way for the receiver to know which transmit antennas were used.
- **`csi`** is the CSI itself, normalized to an internal reference. It is a **`Ntx`**`×`**`Nrx`**`×30` 3-D matrix where the third dimension is across 30 subcarriers in the OFDM channel. For a 20 MHz-wide channel, these correspond to about half the OFDM subcarriers, and for a 40 MHz-wide channel, this is about one in every 4 subcarriers. Which subcarriers were measured is defined by the IEEE 802.11n-2009 standard (in Table 7-25f on page 50).

Now that we've described all the fields of this struct, we need to put them all together to compute the CSI in absolute units, rather than Intel's internal reference level. In particular, we need to combine the RSSI and AGC values together to get RSS in `dBm`, and include noise to get SNR. If there is no noise, as in the sample case, we instead use a hard-coded noise floor of `-92 dBm`. We use the script `get_scaled_csi.m` to do this:

```
    >> csi = get_scaled_csi(csi_entry);
```

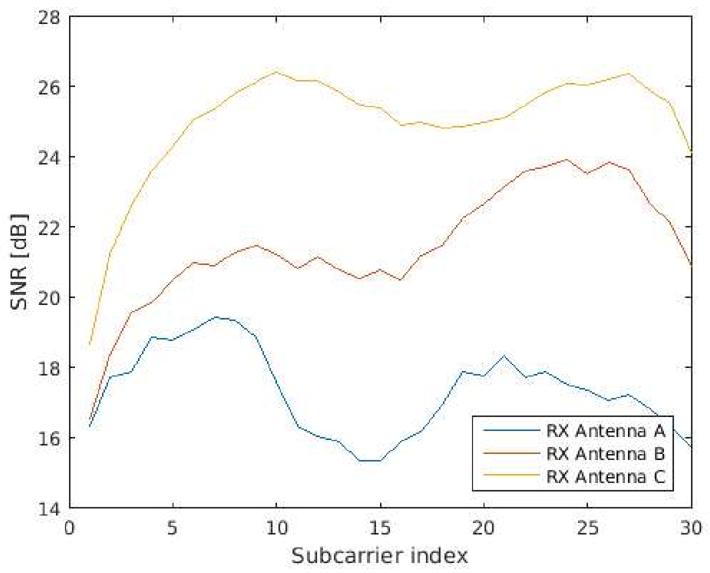
Finally, **`csi`** is a `1×3×30` matrix that represents the MIMO channel state for this link. It's units are in linear—i.e., not `dB`—voltage space. This is the format used in all textbooks I've seen, that is, we've normalized the CSI (in textbooks, usually called *H*) such that there is unit noise.

C. Plotting SNR

Let's look at the three different spatial paths on the `1×3` link we measured:

```
    >> plot(db(abs(squeeze(csi).')))
    >> legend('RX Antenna A', 'RX Antenna B', 'RX Antenna C', 'Location', 'SouthEast' );
    >> xlabel('Subcarrier index');
    >> ylabel('SNR [dB]');
```

In the plot command, `squeeze()` turns **`csi`** into a `3×30` matrix by removing the first singleton dimension. `db()` converts from linear (voltage) space into logarithmic (base-10, power) space. `abs` converts each complex number into its magnitude. Finally, the **`.'`** operator transposes the squeezed CSI from `3×30` matrix into a `30×3` matrix, and does not complement the complex numbers. Combined, we get the plot below.



We see that this is a mostly flat link, with relatively little frequency-selective fading (around `3 dB` for most antenna pairs). However, there is a fair (perhaps `8 dB`) difference between the best antenna C and the worst antenna A. This matches the difference between **`rssi_a`** and **`rssi_c`** (as we expect it should).

D. Computing effective SNR values

We'll conclude our discussion of the CSI by showing you how to compute the Effective SNR from our CSI matrices. To do so, we use the `get_eff_SNRs` script, which takes as input a CSI matrix and returns a `7×4` matrix of effective SNR values in linear (power) space. The 4 columns correspond to the effective SNR using the four 802.11 modulation schemes, namely BPSK/QPSK/16QAM/64QAM. The 7 rows correspond to the seven possible antenna selections with 3 antennas and 1, 2, or 3 spatial streams. In particular, the first 3 rows correspond to single-stream transmissions with antenna A, B, or C. The next 3 rows correspond to dual-stream transmissions with antennas AB, AC, or BC. The last row corresponds to a 3-stream transmission using all antennas.

```
>> db(get_eff_SNRs(csi), 'pow')

ans =

   22.1821   22.2698   22.9007   24.6297
 -156.5356 -156.5356 -156.5356 -156.5356
 -156.5356 -156.5356 -156.5356 -156.5356
 -156.5356 -156.5356 -156.5356 -156.5356
 -156.5356 -156.5356 -156.5356 -156.5356
 -156.5356 -156.5356 -156.5356 -156.5356
 -156.5356 -156.5356 -156.5356 -156.5356
```

Okay, that's pretty disappointing! What happened? Well, note that this is a `1×3` link, so the only valid antenna configuration is SIMO with the single transmit antenna we measured. The other 6 rows correspond to a very small SNR, i.e, a large, negative `dB`.

Let's look at a `3×3` matrix instead:

```
>> csi_entry = csi_trace{20}

csi_entry =

    timestamp_low: 4
       bfee_count: 91
              Nrx: 3
              Ntx: 3
           rssi_a: 34
           rssi_b: 39
           rssi_c: 39
            noise: -127
              agc: 40
             perm: [2 3 1]
             rate: 272
              csi: [3x3x30 double]

>> csi = get_scaled_csi(csi_entry);
>> db(get_eff_SNRs(csi), 'pow')

ans =

       Inf       Inf   32.3435   32.6069
       Inf       Inf   32.4238   32.6822
       Inf       Inf   32.2353   32.5051
   25.4763   25.5262   25.8974   26.8482
   24.6893   24.7490   25.1933   26.5660
   21.9185   22.0303   22.8060   24.6483
    6.5818    8.2321   12.4185   16.2016
```

Here, all 7 rows are valid because there are three transmit antennas. We see that all the SIMO streams are very likely to work; in fact, for BPSK and QPSK there are so few errors that MATLAB's error functions can't distinguish it from zero, and the SNR is effectively infinite. The MIMO2 rates are also likely to work, but only some of the MIMO3 rates will work. See our SIGCOMM 2010 paper for more details.