The Multi-Antenna, Multi-Node, Multi-Band, Multi-Cell (four_multi) framework: A User Guide

Per Zetterberg
June 20, 2013

Abstract

The four_multi framework is a software and hardware solution based on a particular configuration of USRPs. The system allows rapid testing of new multi-antenna schemes including interference alignment, network MIMO and massive MIMO in semi-realtime. An important aspect of the framework is that it allows simulation of the system prior to the semi-realtime runs. The framework is encapsulated into a library called libfour_multi.

Contents

1 Introduction 2

2 Hardware setup 3
  2.1 Original basic setup .......................................................... 3
  2.2 Modified setup (autumn 2011) .............................................. 4

3 Samples and buffers 6
  3.1 Amplitude levels .............................................................. 6
  3.2 OFDM modulation and oversampling ..................................... 10
  3.3 Time synchronization issues ............................................... 10
    3.3.1 System with common GPS splitter and 10MHz clock .......... 10
    3.3.2 System with independent GPS splitter and 10MHz clock ..... 11
  3.4 Organizing the buffer ...................................................... 11
  3.5 Simulation considerations ............................................... 11

4 The software four_multi: fundamentals 12
  4.1 The four_multi_node class .................................................. 12
  4.2 Installing ........................................................................... 14
  4.3 Learning IT++ .................................................................... 16
  4.4 Implementing a new scheme ................................................. 16
    4.4.1 Necessary files ............................................................ 16
    4.4.2 Compiling .................................................................. 17
    4.4.3 The executables .......................................................... 17
  4.5 Simulation, debugging and transferring performance metrics .... 20
  4.6 Working with real signals .................................................. 21
    4.6.1 Matlab only ................................................................. 22
  4.7 Inter-node communication using the backbone ....................... 23
    4.7.1 Using UDP in the backbone ......................................... 23
  4.8 Thread safety .................................................................... 24
1 Introduction

This document is a user-guide and introduction to the four_multi software and hardware framework. The framework is intimately connected to a hardware setup, which is also carefully described. In short, the hardware setup is based on extensive use of USRP2 and/or USRPN210 from Ettus research, www.ettus.com, and some additional hardware made in-house at the signal processing group at KTH. The details are found in Section 2 below. If there is interest from other groups, inside or outside KTH, to replicate the hardware, we can help you to do so, please contact the author. The software is called Multi-Antenna,Multi-Node,Multi-Band,Multi-Cell or four_multi for short. The software consists of a library and utilities for writing software and controlling the system. The library is documented on a class by class manner at http://www.ee.kth.se/~perz/usrp/four_multi. However, the necessary background is only obtained in this document. The main component of four_multi is the class four_multi_node. New implementations are enabled by making new descendant classes of this class. You then override the node_init, node_process and end_of_run methods. Within these function you can access the input and output buffers of samples in addition to backbone transfer of data between nodes (if connected by Ethernet) and many other parameters. The four_multi software framework also contains examples, a toolbox of coding and modulation (AMC and OFDM1), and utilities for e.g. grabbing samples. Details and explanations on the issues are found below. The framework also provides a function simulate which allows you to simulate the system.

In the text below, the term user will be used to denote a person who is implementing a new scheme using the facilities provided by four_multi, i.e. it does not denote a node class such as a “user-equipment” used in LTE literature.

By semi-realtime we mean a system operating on synchronized frames but where idle time has been inserted between the frames to allow for processing time in the nodes (and thus save substantial development time). In the following we will refer to this as “realtime” for simplicity.
2 Hardware setup

2.1 Original basic setup

The most straightforward configuration is illustrated in Figure 1. This is how the system was originally planned. The system in Figure 1 consists of only two nodes $d_{\text{node}\_ix=0}$ and $d_{\text{node}\_ix=1}$, this is just for the purposes of illustration and already the first system was planned to have six nodes. A node consists of a PC and multiple USRPs, one for each antenna (and, in case of a multi-band implementation, one for each band). The USRPs are of type USRP2 or USRPN210 and have XCVR2450 daughterboards, see www.ettus.com. The base-band processing is performed in the PCs. Each USRP has a dedicated Ethernet interface to the PC. The IP addresses of the USRPs in a node with $n$ antennas are given by $192.168.10.2$ to $192.168.(10*n).2$. On the host side the corresponding addresses are manually configured to $192.168.10.1$ to $192.168.(10*n).1$. Sometimes the USRPs need to be power-cycled and the Ethernet cables re-connected to have the connection between the PC and the USRPs work properly. You can issue the command `uhd_find_devices` (with firewall turned off) to check which USRPs are properly connected. However, as it turn out, this is only a necessary condition (and not sufficient) for proper operation.

The nodes are synchronized by using three different signals, the pulse-per-second PPS, the 10MHz reference clock, and the NMEA. The PPS is a pulse which goes high for around one microsecond every second. The pulse is active high (5 volts). The 10MHz clock is in our case a sinusoid with 10MHz frequency generated by a signal generator such as the HP 8648A illustrated in Figure 2. The NMEA is an ASCII protocol which provides position and hour-minute-second time. We use a EM-406A GPS module to generate the NMEA and PPS. We have designed a circuit board which contains the EM-406A and drivers to distribute the PPS and NMEA signals on BNC connectors over coaxial cables. We mount the circuit board in a box and distribute both the PPS and NMEA on six BNC connectors each. Each connector has it’s own driver but the signals are highly aligned and can be considered identical. The signals from the GPS splitter are inverted and thus are active low. We call the box “GPS splitter” see Figure 3. Before reaching the nodes the outputs of the GPS splitter are split once again. This is accomplished by a box we refer to as the “USB splitter” see Figure 4. This box splits the PPS signal into ten copies. The copies are inverted back to an active high pulse. The NMEA signal is also converted from an RS232-like format into a RS232 over USB format in the “USB splitter” (by a UM232R USB module). The ASCII stream is typically accessed in a Linux computer as “/dev/ttyUSB0”. There are six USB outputs on each USB splitter. Thus six computers can be connected with identical NMEA information being received.

The NMEA stream of characters is parsed by the PC to find the time stamps. These time stamps are then used to coarsely synchronize the nodes.

The 10MHz and PPS clocks are connected to the corresponding inputs on the USRPs. The USRP uses the 10MHz clock to fine synchronize it’s 100MHz clock which drives the ADC and DACs and the FPGA. The 10MHz clock is also used as a reference for the RF generation in the XCVR2450 daughterboard (which is mounted on the USRP motherboard and located in the same box). The 10MHz clock input need be in the range 5-15dBm for USRP2 and 0-15dBm for N210. To achieve this we use amplifiers, splitters and filters using mini-circuits components (see www.minicircuits.com). In particular we use the SBP-10.7 filter to reject interfering signals and limit undesired side-effects. These components are located both close to the 10MHz generator and close to the nodes.

The 100MHz clock is used to define the time inside each USRP. However, when switched on - every USRP will start in a random state. To align all clocks, the UHD driver command `set_time_next_pps` is called for each USRP in the system (this is done for you inside the four_muli code). Upon receiving this command, the USRP sets the time to zero when receiving the next PPS pulse. Thus, it is important that the command is called by all PCs within the same PPS interval. The GPS synchronization using the command `gps_wait_until` has proved to be sufficient for this purpose.

The phase of the 100MHz clock on the USRPs will always be different. Thus the actual timing of two USRPs may differ up to 10ns, and all receiver processing algorithms must be capable of handling this uncertainty. For OFDM modulations this is straightforward due to the cyclic prefix which will convert the delay into a phase-shift. Note that the offset will be fixed for the duration of the run. The phase offset can therefore be readily compensated for.

Since all XCVR2450 are driven by the same 10MHz clock, no phase-drifts exist in the system (apart
from phase-noise). The system is operated at 2490MHz since this frequency is unoccupied where the KTH testbed is located.

A fixed wired backbone is provided via the router, see Figure 1.

### 2.2 Modified setup (autumn 2011)

A modified setup was implemented autumn 2011 and is still in use as of today. This setup is illustrated in Figure 5. The setup in Figure 5 differs from the one in Figure 1 in that several logical nodes shares a PC and that some of the nodes no longer have XCVR2450 boards. Instead these nodes use a basic daughterboards, see www.ettus.com, which receives signals at a carrier frequency of 70MHz. An external receiver has been connected to downconvert the received signal from the 2490MHz carrier to 70MHz. These receiver boards are similar to the one described in [1], with increased filter bandwidths and the last downconverter stage removed. Multiple nodes per PC were implemented in order to reduce the number of PCs needed, primarily due to space limitations. The receiver boards were replaced due to the poor performance we have experience with XCVR2450 receivers (noise figure and dynamic range). The receiver local oscillator of the external receiver boards are no longer locked to the 10MHz reference. However, the two receiver boards of one node share a common local oscillator and are thus perfectly phase-locked.

The transmitter USRPs were connected to external amplifiers of type mini-circuits ZRL-2400LN in order to further improve performance.

Thus in the current state of the testbed, three of the nodes are transmit only, and three of the nodes are receive only. According to the software design, see Section 4.4, each node is implemented as a **four_multi_node** object. This object has a “run” method which is used when running in real-time. In order to be able to run multiple nodes on one PC the object was modified such that the IP addresses of the USRPs of one node could be 192.168.(1+s)*10.2 to 192.168.10*(n+s).2, where the parameter s (in the API the parameter s is called **skip_ant**) can be used to address a different range of antennas for each node. The idea was originally to run each node in it’s own process on the PC. As it turned out this did not work well. Therefore, the concept of a **super_node** was introduced. A supernode (object) encompasses a number of nodes (i.e. **four_multi_node** objects) but handles the communication with the USRPs for them. It may seem from Figure 5 that the system has no mobility given the spider-web of cables between the different entities. However, this is not the case. The physical layout of the cables is according to a star topology where the two PCs the 10MHz generator and the GPS splitter is co-located in center of the star while the USRPs and their antennas (corresponding to a node i.e. a **four_multi_node** object) is placed on the edges. The cables to one such node (Ethernet, power, PPS, 10MHz) are strapped together to ease the handling. The physical layout of the system is illustrated in Figure 6, while a receiver node and the the strapped-together cables are shown in Figure 7.
Figure 2: HP 8648A used to generated 10MHz reference

Figure 3: GPS splitter
Since the local oscillators of the receivers are no longer locked against the 10MHz there will be a frequency offset. This frequency offset can be reduced by calibration. To do so we let all six transmitters transmit a CW. This is done by starting matlab in super-user (root) mode on both computers and changing directory to the \texttt{utils} directory of \texttt{four\_multi}. Then we issue the command \texttt{tx\_cisoid} on the transmitter PC. The receiver PC may then estimate the frequency offset using the call

\[
[\text{df}_0, \text{df}_1, \text{df}_2] = \text{est\_freq\_offset};
\]

Upon completing this command the frequency offset of node0, node1 and node2, will reside in \text{df}_0, \text{df}_1 and \text{df}_2. These values i.e. \text{df}_0, \text{df}_1 and \text{df}_2 can subsequently be used as input to scripts which run the system and thereby reduce the frequency offset down to to a level where receiver algorithms do not need to take into account. The frequencies used should be 70e6-\text{df}_0, 70e6-\text{df}_1, and 70e6-\text{df}_2, i.e. around 70MHz (the analog board downconverts from 2490MHz to 90MHz).

A new setup is expected during 2013. This setup is likely to have unlocked oscillators in all nodes, and all nodes will be capable of both transmission and reception. The new setup will be capable of tuning in the 4.9GHz to 6GHz range, where 5.8GHz is a likely candidate for experiments. This will call for updated frequency offset estimation routines.

3 \hspace{1em} \textbf{Samples and buffers}

3.1 \hspace{1em} \textbf{Amplitude levels}

In theoretical simulations one rarely have to pay much attention to the amplitude levels - only the signal to noise ratio is of interest. However, this is not the case when working with real hardware. In fact, every point in the transmitter and receiver chains has to operate at a point within it’s dynamic range. Fortunately, we only have to consider the base-band levels and the transceiver gain setting when programming.

The format we use for data transmitted to and received from the USRP is \texttt{std::complex<int16t>} this means that the real and imaginary part of the data are 16-bit signed integers. The samples are accessed by the vector of buffers \texttt{std::vector<void *> d\_tx\_buffers} and
Figure 5: Configuration since autumn 2011. Blue, red and green lines transfer 10MHz clock, NMEA (GPS information) and the pulse-per-second clock, respectively, all using coaxial cables. The magenta colored line is NMEA over USB, and the yellow line is the Ethernet backhaul over cat-5 cable.
The testbed

Figure 6: Illustration of the physical layout of the system
std::vector<void *> d_rx_buffers. These buffers are accessible inside the node_init and node_process functions of your implementation. Each entry of d_tx_buffers and d_rxBuffers contains a pointer to an array of std::complex<int16_t>. The length of the buffer vectors are equal to the number of USRPs in the node. Thus each pointer correspond to the signal of one antenna. In order to transmit signals on antenna n, we may execute the following code

```c++
std::complex<int16_t> *p1;
p1 = (std::complex<int16_t> *) d_tx_buffers[n];
for (uint32_t i1=0; i1<d_buffer_length; i1++) {
    p1[i1]= waveform_tx(i1);
}
```

where we assume the signal to be transmitted is contained in the IT++ cvec vector called waveform (see Section 4.3). The values transmitted are of type std::complex<int16_t>. Thus the real and imaginary part can take values in the range from $-2^{15}$ to $2^{15} - 1$. We have found experimentally that an RMS value of 5000 samples or less seems to be a good value for an OFDM modulated signal in the transmitter, This can be combined with setting the transmitter gain gain_tx in the frame_settings structure returned by your node_init and node_process implementations. The gain is nominally expressed in dB but the gain actually achieved may not correspond to that set. For instance, increasing the gain setting from 10 to 20 may not increase the signal 10dB but could be 8dB. This could possibly be compensated for by means of a measured look-up table. With the external amplifiers ZRL-2400LN only gains 0-3 are advisable. This gives an output power of around +15dBm. When using the XCVR2450 without ZRL-2400LN we can use gains up to 10dB which gives us an output power of around -12dBm an EVM of 2-3% (higher gains reduce the EVM drastically). If a greater range is desired (when using ZRL-2450) an attenuator could be mounted between the XCVR2450 and the amplifier. However, a better solution is probably to vary the base-band signal level. Starting with RMS values 5000 we can probably back down to an RMS value of 50 (i.e. 40dB reduction) without having significant problems with DAC converter noise. However, the
DC subcarrier leakage could become substantial under these conditions. In order to receive signals from the nth USRP the signals are accessed as

```c
std::complex<int16_t> *p1;
p1=(std::complex<int16_t> *) d_rx_buffers[n];
for (uint32_t i1=0;i1<d_buffer_length;i1++)
    waveform_rx(i1)=p1[i1];
```

where the data is loaded into the IT++ cvec vector waveform_rx for further processing. The receiver gain of XCVR2450 can be varied from 0 to 90dB. However, experience shows that only 0,15 and 30dB gains should be used, see [2]. This is probably due to a bad receiver circuit design. For this and other reasons we use a custom receiver board (as already mentioned). This board is similar to the one described in [1]. The difference being that the narrow crystal filters are removed and replaced by an additional SBP-70 filter. In addition an extra low-noise amplifier is connected at the front-end. The output is then sampled directly at the 70MHz intermediate frequency using a basic daughterboard, see www.ettus.com. With this design the gain_rx parameter in the frame_settings struct is actually not used. The receiver noise figure is around 10dB (as compared to at least 20dB for the XCVR2450). The noise factor is set by the front end amplifier. The standard deviation of the noise is approximately 11.7 when there is no antenna connected When the antenna is connected this value rises to around 21.7. (or 15.4 on real and imaginary part). This noise rise occur when the receiver is picking up various interfering signals. The values presented are averages over all receivers. The individual receivers should not vary more than 1.5 dB from these values (the variation is less with no antenna connected). When using the OFDM1 modulation described in Section 5.2, the noise variance per subcarrier becomes around 1794.4 with antennas connected. In addition to these

At RMS amplitudes higher than 5000 (or 3500 on the real and imaginary part) reduced performance due to saturation start to occur, especially if interfering signals account for the saturation. During the measurement campaign in [3] a 10dB attenuator was sometimes inserted between the antenna and the first amplifier. This clearly solved the saturation problem, but the noise level remained unchanged. If instead the first amplifier had been removed, then the noise variance would have been reduced. We chose not to do so since the system was balanced so that the noise level in every receiver was near the above mentioned value. This made the system analysis much simpler (constant noise power).

During the spring of 2013 a new transceiver is planned to be introduced. This will likely change the values described above.

### 3.2 OFDM modulation and oversampling

The above mentioned signal levels describes the time-domain samples as received and transmitted in the buffers. Note that due to the bandwidth used in the interpolation and decimation implemented in the FPGA of the USRP N210, we recommend a maximum data bandwidth of 12MHz for OFDM modulations and 6.25MHz for single-carrier modulations. For OFDM this means that the frequency of the maximum and minimum subcarrier should be at +6MHz and -6MHz, respectively. The center subcarrier with frequency 0Hz should not be used since the XCVR2450, as many direct upconversion transmitters, blocks signals close to 0Hz. This presents a problem also for single-carrier modulations. Contact the author for advice on how to solve this problem in single-carrier systems.

The OFDM modulation **OFDM1** of the **four_multi** software (see Section 5), is tailored to the above requirements. It is similar to IEEE802.11a but uses a smaller subcarrier spacing. The `modulate` method of **OFDM1** contains an output parameter `rms_power` which can be used to measure and scale the output signal properly.

### 3.3 Time synchronization issues

#### 3.3.1 System with common GPS splitter and 10MHz clock

When does the received signal appear in the receive buffer? When simulating using the `simulator` function and the `block_fading` channel model (see Section 5), it will appear exactly at the same place where it was placed by the transmitter (possibly a bit of smearing towards later samples if `num_of_taps` is larger than one). When running in real-time, the signal will be received around 55 samples later. So
far - only OFDM modulations have been used for which the synchronization is not very critical. Thus if single-carrier modulations are employed, the above value of 55 samples may need to be modified. Why is there a delay at all? The reason is due to the processing delay in the DSP pipeline of the transmitter and receiver FPGA plus a little bit of delay in the analog circuitry. Propagation delay can add one or possibly two additional samples. Future changes in the FPGA implementation may change the delay. In your receiver implementation in node_process you need to select the start sample for start of demodulation depending on whether you are processing real- or simulated-data. It may be necessary to test different demodulation positions to have good performance in your implementation - that doesn’t mean synchronization algorithms - but verification of the chosen demodulation position.

3.3.2 System with independent GPS splitter and 10MHz clock

So far we have assumed that sample-clocks of all nodes are fully synchronized through the use of common 10MHz and PPS reference clocks. This is possible thanks to the intricate wiring as shown in Figure 5. This however limits the maximum distance between the transmitter and receiver. One may however use independent GPS splitters and 10MHz generators at different nodes. This leads to some uncertainty in the synchronization i.e. where in the buffer the signal arrives. The GPS difference may result in an error of up to one microsecond or 25 samples. The first buffer is sent after one second following the triggering using PPS. During this second the timing of the transmitter and receiver may slip. If the 10MHz clocks of the nodes differ by one part per million (ppm) then a delay of an additional 25 samples will occur. The receiver algorithm will have to cope with this difference, if not compensated for at the transmitter. In order to compensate at the transmitter one could develop a scheme were a few extra frames (and a lot of processing) are used to align all nodes. Each node can then compensate itself by setting the time file of the frame_settings returned from node_process, see Section 4.1.

3.4 Organizing the buffer

The purpose of four_multi is experimentation. We therefore propose to use the samples a bit wastefully - in order to facilitate additional research and debugging possibilities. The size of the buffer is set during the instantiation of the four_multi_node object, which is described further in Section 4.1 below. To start with - we propose that the first 1000 samples are not used. This is mainly due to the fact that the USRP seems not to be meet it’s best performance during these samples. We may then leave another 1000 samples empty. These samples can then be used to estimate the thermal noise level. Finally, we propose to transmit some different replicas of the burst. For example, if the frame length of our system is 5000 samples, we may use samples 2000-7000, 8000-13000 and 14000-19000 to transmit three different replicas of the frame. This can be used for research and debugging purposes. For instance, say two links are simultaneously operating on the same frequency. We may then have both links transmitting in samples 2000-7000, link number one in 8000-13000 and link number two in 14000-19000. The performance we are most interested in is the samples 2000-7000. However, we can use the other two “sub bursts” to provide reference performance estimates that are useful for debugging, benchmarking and other purposes.

3.5 Simulation considerations

The channel model block_fading has the elements scaling_signal and scaling_noise. The output level from the transmitter should have an RMS value in the range 50-5000 as mentioned in section 3.1 above. The receiver should receive signals up to rms-levels of 5000 as also mentioned above. The block_fading exhibits Rayleigh fading. Assuming that the transmitter gives a maximum RMS output value of 5000 a good value for scaling_signal would be 0.45 which is also the default value. The scaling_noise gives the standard deviation of the noise. Given the values described in Section 3.1 a realistic value is 21.7 which is therefore used as default. These settings produces a model of the actual testbed performance. However, for debugging and research purposes it may be useful to set the noise variance to zero. Note that scaling_signal=10 and scaling_noise=1 may give different results from scaling_signal=100 and scaling_noise=10. This is because the signals are rounded to integers before being received into the receive buffer.
4 The software four\_multi: fundamentals

As already mentioned we use the four\_multi framework for building applications. We have already touched upon several features of the software when describing the hardware setup. Below we describe how to install and use the framework. However, before going to these details, let us just outline the contents of four\_multi. The framework is downloaded from SourceForge using the command `git clone git://code.ettus.com/ettus/uhd.git`. Once downloaded the project consists of a main directory called four\_multi and four sub-directories called core, examples, libfour\_multi, modem, and utils. The core directory contains, the definition and declaration of the four\_multi_node object in the files four\_multi.cpp and four\_multi.hpp, the super_node object in super\_node.cpp and super\_node.hpp, and the simulator function in simulator.cpp and simulator.hpp. The modem directory contains functions for adaptive coding and modulation contained in the objects AMC and OFDM1. These components does not need to accessed directly. They are instead glued together into a library called libfour\_multi. The Makefile for this compilation is located in the subdirectory libfour\_multi. The contents are then subsequently installed under /usr/local/lib/ and /usr/local/include/four\_multi/. The directory examples contains an example SISO AMC OFDM which illustrates how to use the framework functionality. The utils directory, contains small stand-alone pieces of software that helps develop applications and general use of the testbed. These utilities are installed in the /usr/local/share/four\_multi/utils/ directory.

4.1 The four\_multi\_node class

The main work of implementing a new approach is to define a descendant class to four\_multi\_node. Once the functionality has been encapsulated into the descendant class, the framework should help you make realtime runs and simulations. Let’s say we want implement a new approach which we call my\_approach. Below is an outline of how the definition of the descendant class my\_approach\_node could look.

```cpp
class my\_approach\_node : public four\_multi\_node
{
public:
    my\_approach\_node(int node\_ix, int number\_of\_taps, int number\_of\_frames...);

private:
    virtual frame\_settings node\_init(void);
    virtual frame\_settings node\_process(void);
    virtual void end\_of\_run(void);

    /* Example states */
    int frame\_number;
    cmat channel\_estimate;
    int ARQ\_state;

    /* Example outputs*/
    public:
    cvec BER\_results;
    double SNR\_estimate;
..
};
```

The parameters of the node are set through the constructor and public members, in the example above such members are `node\_ix`, `number\_of\_taps` and `number\_of\_frames`. Likewise, our main way of retrieving the results from the node will be through it’s public members such as `BER\_results` and `SNR\_estimate` above. All the variables needed to hold the state of node has to be defined as members such as `frame\_number`, `channel\_estimate` and `ARQ\_state` in the example above. These issues are further described in Section 4.4 below.
The processing performed by the node is defined by the constructor and the methods `node_init`, `node_process` and `end_of_run`. Typically, the constructor would set the size of arrays and matrices used by the node. In our example, `number_of_frames` and `number_of_taps` may be used to set the size of BER results and channel estimate, respectively. In `node_init` we will typically reset variables to prepare for another run. In our example, `frame_number` and `ARQ_state` may be set to zero. In `node_process`, which can be seen as the most important method, the received and transmitted samples are processed. We access them through the vectors of buffers `d_rx_buffers` and `d_tx_buffers`, as described in Section 3. Finally, we can do post processing of measurement results in `end_of_run`.

The functions `node_init`, `node_process`, and `end_of_run` are overriding the empty definition in the base object `four_multi_node` The object is brought to life by calling the `run` method of `four_multi_node` for real-time evaluations or by proving it as an input parameter to the `simulate` function, as described in Section 4.4.3. When this is done the functions `node_init`, `node_process`, and `end_of_run` will be called implicitly (i.e. as callbacks).

Your implementation of `my_approach_node` interacts with its environment through the processing of incoming and outgoing buffers (i.e. `d_rx_buffers` and `d_tx_buffers`, see Section 3). However, equally important, it also interacts by the return value of `node_init` and `node_process` which is of type `frame_settings`. The definition of this type is shown below.

```cpp
struct frame_settings {
    std::vector<double> frequency;
    std::vector<double> gain_rx;
    std::vector<double> gain_tx;
    uhd::time_spec_t time;
    burst_t what;
};
```

The `frame_settings` type is used to define the next frame (or maybe we should say buffer). It does so by specifying the start of the frame in `time` (referenced from 0.0 when the run starts), the frequency and gains to be employed by the USRP daughterboards (in `frequency`, `gain_rx`, `gain_tx`), and the type of the burst (in `what`). The properties `frequency`, `gain_rx`, `gain_tx` are vectors with one element corresponding to each USRP. In case these vectors are shorter than the number of USRPs, the last value will be repeated to achieve the desired length. The recommended setting of the gain and frequencies were discussed in Section 3. The burst type is defined as

```cpp
enum burst_t {TX=0, RX=1, START=2, END=3,
               CALIB_TXRX=4, CALIB_RXTX=5, CALIB_RXRX=6, CALIB_TXTX=7};
```

Typically, your method would return one of TX, RX or END. The START type is only for internal use. The remaining four burst types are for advanced schemes that involve calibration frames that are sent before any other frames are transmitted for the purpose of calibration e.g. TDD reciprocity calibration as described in the paper [4]. We recommend you contact the author of this document if you want to get into the details of this. For now, we assume that these bursts types are not used.

Obviously, you return TX if you like to transmit in the next frame, RX if you want to receive and END if you want to exit. If you return TX then the contents according to `d_tx_buffers` will be transmitted at the time specified by the `time` field. This leads us to another detail we have not mentioned so far: when `node_init` returns TX it must also fill the buffer `d_tx_buffers` with the appropriate contents.

If the last call returned RX then `node_process` is responsible for processing its contents.

Inside all the methods (constructor, `node_init`, `node_process` and `end_of_run`) you can make use of the functionality for coding and modulation, see Section 5. The inter-node communication using the backbone Ethernet (described in Section 4.7), can only be used inside the method `node_process`. When implementing the constructor in C++, the code must also call the instantiator of the base object. The definition of constructor may look like this

```cpp
my_approach_node::my_approach_node(
    int node_ix, int number_of_taps, int number_of_frames,
    std::vector<std::string> IP_addresses, simulate,
    uint32_t no_ant, bool simulate) :
```
The part after the colon is the instantiation of the `four_multi_node` object. The declaration of this constructor is as follows:

```cpp
four_multi_node(uint32_t buffer_length, uint32_t no_ant1, uint32_t no_ant2,
                bool use_external_10MHz, bool same_antenna,
                uint32_t node_ix, std::vector<std::string> IP_addresses,
                bool simulate, uint32_t skip_ant);
```

In our constructor we pass arguments as required from the parameter list of `my_approach_node` to `four_multi_node`. The first one, `buffer_length`, defines the number of samples per buffer (per USRP). There are two parameters for setting the number of antennas. This split is related to the CALIB frames mentioned earlier. Unless you are going to use any of these frame types you may simply set `no_ant1` to your number of USRPs and `no_ant2` to zero. The Boolean `use_external_10MHz` should be set to `true` since we plan to use a 10MHz reference clock for the foreseeable future. The parameter `same_antenna` is only used when the XCVR2450 daughterboard is employed. When it is set to `true`, transmission and reception is both performed on the antenna connector marked “J1”. When set to `false`, transmission is performed on the antenna marked “J1” and reception on the antenna connector marked “J2”. The `node_ix` sets the internal variable `d_node_ix` which is important variable to be used for defining the behavior of the node. This internal variable were already mentioned in Section 2. The parameter `IP_addresses` is a list of the backbone addresses of all nodes in the system, as described in more detail in Section 4.7. The Boolean `simulate` must be set to `true` when simulating the system as described in Section 4.5 and `false` when doing real-time runs and when instantiating the constituent nodes of the `super_node` as described in Section 4.4.3. The value of this parameter is stored in the `four_multi_node` member `d_simulate`. The value is used to determine wether USRPs should be initialized and controlled, and wether backbone transfers should actually be performed over internet or just inside the simulation environment. A special case is when using `super_node`. In this case, the `super_node` takes care of the control of the USRPs while the backbone transfers actually uses the internet (also within one computer). This is solved so that the user sets `simulate` to `true`. The inner workings of `super_node` will then set it to `false` before calling `node_process`. The final parameter, `skip_ant` was described already in Section 2.2.

### 4.2 Installing

The software both for simulation and realtime operation runs on normal PCs with Ubuntu operating system the version currently used is the 64-bit variant (x86_64) of 11.10 and 12.04. Instructions for installing a realtime patched kernel in 12.04 are provided in Section 9 below. For offline simulations this patch should be unnecessary. The UHD driver, [http://files.ettus.com/uhd_docs/manual/html/](http://files.ettus.com/uhd_docs/manual/html/), need to be installed even on a computer used only for offline simulations. This software is available at [http://ettus-apps.sourcerepo.com/redmine/ettus/projects/uhd/wiki/UHD_Build](http://ettus-apps.sourcerepo.com/redmine/ettus/projects/uhd/wiki/UHD_Build). As of writing we are using version 3.3.2.

The following sequence of instructions should download build and install the UHD, but may need to be updated and modified.

1. `sudo apt-get install libboost-all-dev libusb-1.0-0-dev \` python-cheetah doxygen python-docutils
2. `git clone git://code.ettus.com/ettus/uhd.git`
3. `cd uhd`
4. `git checkout release_003_003_002`
5. cd host
6. mkdir build
7. cd build
8. cmake ../
9. make all
10. sudo make install
11. sudo ldconfig

We extensively use the IT++ library which is a library of C++ functions for linear algebra, coding and modulation. We use version 4.0.7 although there exist a newer version. However, this version does not seem to work for our purposes. The problems we have seen may only occur on 64-bit architectures which is our intended platform. The library can be downloaded from http://sourceforge.net/projects/itpp/files/itpp/4.0.7/. The following sequence of instructions should install IT++

1. tar jxvf itpp-4.0.7.tar.bz2
2. cd itpp-4.0.7
3. sudo apt-get install libblas3gf
4. sudo apt-get install liblapack3gf
5. sudo apt-get install libfftw3-3
6. sudo apt-get install libfftw3-dev
7. ./configure --with-blas=/usr/lib/libblas.so.3gf \  
   --with-lapack=/usr/lib/liblapack.so.3gf
8. make
9. sudo make install
10. sudo ldconfig,

make sure there are now warnings after running configure. To build libfour_multi do the following

1. Download the four_multi framework using: git clone git://git.code.sf.net/p/fourmulti/code four_multi.
2. Enter directory libfour_multi.
3. make all,
4. sudo make install
5. sudo ldconfig.
6. cd ../examples
7. make all
8. cd ../utils
9. make all.
10. sudo make install
The source code for the libfour_multi is located under core and modem. If you decide to make a change to this code, you must first do sudo make clean from the directory libfour_multi before rebuilding and installing the library in order to make shore the changes will take effect. Under examples you can find an implementation of a SISO system which uses OFDM and adaptive modulation and coding (AMC). To build the simulator of this approach do make SISO_AMC_OFDM_simulator. You can run it by doing sudo ./SISO_AMC_OFDM_simulator. We recommend you test your installation by verifying that you can run this program.

4.3 Learning IT++

As mentioned above four_multi is based on IT++. We recommend you acquaint yourself with IT++ by visiting the documentation homepage at http://itpp.sourceforge.net/stable/. Below is a minimal IT++ program that you can start with. If you save the file as test.cpp you should be able to compile it using g++ test.cpp -o test /usr/local/lib/libitpp.so /usr/lib/libblas.so.3gf /usr/lib/liblapack.so.3gf and run it as ./test.

```cpp
#include <itpp/itbase.h>
#include <itpp/itcomm.h>
#include <complex>
using namespace itpp;

int main( int argc, char *argv[]) {
    cvec v1;
    cvec v2;
    cvec v3;

    v1="(5,3),(-13,-4),(3,36)"; // (real,imag)
    v2="(28,-13),(30,7),(-1,7)"; // (real,imag)

    v3=v1+v2;
    std::cout << "v3=" << v3 << std::endl;
}
```

One thing to be aware of is that many functions in IT++, and four_multi, take vectors and/or matrices as output, and assumes that they are large enough to hold the result of the computations. For instance, the function insert_pilot which is one of the members of class OFDM1 which takes as one of this parameters the vector waveform. This function places a pilot OFDM symbol into waveform at a position defined by the other input parameters. The waveform vector need to be large enough to have these positions. For instance, say the waveform input was defined to be of length 10, e.g. using waveform.set_size(10). When the function insert_pilot tries to write beyond that location, an error will occur (segmentation fault likely).

4.4 Implementing a new scheme

4.4.1 Necessary files

Lets assume we want to implement a new approach called my_approach. The first step would typically be to create a directory called my_approach with the following files

1. my_approach_node.hpp
2. my_approach_node.cpp
3. my_approach_simulator.cpp
4. my_approach_realtime.cpp.
5. Makefile

The first file contains the definition of a new class called `my_approach_node`. This class is derived from `four_multi_node` i.e. it will contain the following type of declarations

```cpp
class my_approach_node : public four_multi_node { .... },
```

as described in Section 4.1. The `my_approach_node` class should contain all the members you need to keep the state of your implementation, as well as tweaking parameters. In scientific investigations there is generally a need to have a reference scheme to benchmark against. By providing this reference by means of parameters selection generally helps to make the comparison relevant i.e. keeping as much as possible identical and concentrate the comparison into an isolated set of components.

The files `my_approach_simulator.cpp` and `my_approach_realtime.cpp` are used to generate executables with the same names (but without extension) which are used for simulations and real-world experiments, respectively. More details on them follows in the next section.

4.4.2 Compiling

An example Makefile is shown below

```cpp
CXXFLAGS = -o3 -Wall -DNDEBUG
LDLIBS = -lboost_system -lboost_program_options -luhd /usr/local/lib/libitpp.so
/usr/lib/libblas.so.3gf /usr/lib/liblapack.so.3gf
CXX = gcc-4.4

my_approach_simulator: my_approach_simulator.o my_approach_node.o my_approach_node.hpp
   $(CXX) -o $@ $^ -lfour_multi $(LDLIBS)

my_approach_realtime: my_approach_realtime.o my_approach_node.o my_approach_node.hpp
   $(CXX) -o $@ $^ -lfour_multi $(LDLIBS)
```

By saving these instructions in a file called `Makefile`, it is sufficient to type `make my_approach_simulator` and `make my_approach_realtime` to create the corresponding executables. For more information on how this works search for help on “GNU make” in e.g. google. The Makefile may have to be adapted to your system. For instance, in the example above we had to use an old compiler `gcc-4.4` because of compatibility problems with the `boost` library. The latest compiler should typically be used. This compiler is selected by replacing `gcc-4.4` with `g++`.

4.4.3 The executables

Passing parameters The files `my_approach_simulator.cpp` and `my_approach_realtime.cpp` are used to create executables and will thus contain a `main` entry function. The executables will accept command line arguments which are used to control the functionality. We typically use matlab scripts which call the executables using the `system(str)` call where `str` is string containing the command name, in our example case `my_approach_simulator` or `my_approach_realtime`. Command line arguments can be added by adding parameters in the string `str`. For instance let us show how to pass the parameter `node_ix` parameter from matlab to the application. The beginning of the main function in `my_approach_realtime.cpp` may then look like this

```cpp
#include <uhd/utils/thread_priority.hpp>
#include <uhd/utils/safe_main.hpp>
#include <uhd/usrp/multi_usrp.hpp>
#include <boost/program_options.hpp>
#include <boost/format.hpp>

namespace po = boost::program_options;
```
using namespace boost;

int UHD_SAFE_MAIN(int argc, char *argv[]){
    po::options_description desc("Allowed options");
    desc.add_options()
        ("help", "help message")
        ("node_ix",po::value<uint32_t>(&node_ix)->default_value(0),)
        ;

    po::variables_map vm;
    po::store(po::parse_command_line(argc, argv, desc), vm);
    po::notify(vm);
    if (vm.count("help")){
        std::cout << boost::format("my_approach_realtime %s") % desc <<
        std::endl;
        return ~0;
    }
    
    
    }

Here we use the program_options module of the boost library to catch the command line parameters (see www.boost.org for more details). To set node_ix to one from the command line, we would call the program as

sudo ./my_approach_realtime --node_ix=1

To make this call from matlab we could utilize the following code

str=['sudo ./my_approach_realtime --d_node_ix=',num2str(node_ix)];
system(str);

Thus we use matlab as a scripting environment to save important variables, make sequences of runs and present and store results. More ways of communicating between matlab and the C++ program is covered in Section 4.5 on debugging. Note that the main function which is the entry point of typical C/C++ programs is replaced by UHD_SAFE_MAIN. This is a macro from the UHD driver.

Instantiation of four_multi_node objects  Both executables my_approach_simulator.cpp and my_approach_realtime.cpp would use the same implementation of the my_approach_node class. This also helps debugging and making scientific comparisons of “theory” and “practice”. In the file my_approach_simulator.cpp you will instantiate a vector of my_approach_node objects. For example, in a relaying application there may be a source, a relay and a destination. After stacking them in a vector of type std::vector<four_multi_node*> you are able to simulate the system by means of the simulator function. The role of the node is typically defined by the d_node_ix member of four_multi_node. Thus in your processing you will use this property to define whether the node should act as a source, relay or destination.

The file my_approach_realtime.cpp may either instantiate one my_approach_node or more my_approach_node objects. In case only one is instantiated, one PC will be needed for each node. Each node will then perform the run method of four_multi_node. The following code snippet is taken from SISO_AMC_OFDM_realtime.cpp:

SISO_AMC_OFDM_node node(gain_rx,gain_tx,node_ix,frequency,false,
    modulation_order,false,codec_ix);

node.run(dev_name.c_str(),start_time);
The first line instantiates an object of type SISO_AMC_OFDM_node. Since SISO_AMC_OFDM_node is a descendant of four_multi_node, it has inherited the method run. The second line makes the node run in real-time. The two arguments of run are both for the sake of synchronization: they define the device name for the GPS (typically /dev/ttyUSB0) and the start time. The node will wait until the time is due according to GPS is time. The fine synchronization is then subsequently performed as discussed in Section 2.1. The time defined by start_time is simply an unsigned int defined as

\[ \text{start\_time} = \text{hour} \times 3600 + \text{minute} \times 60 + \text{second}. \]

When there are less than one PC per node, the code used is illustrated by the following code snippet which is taken from IA1_super_realtime.cpp (see Section 7 below).

```cpp
#include <four_multi/super_node.hpp>
std::vector<four_multi_node*> all_my_nodes1;
IA1_node node0_IA(2,0,0,0,freq,true,IP_addresses,true,60,start_time,
    false,1);
IA1_node node1_IA(2,0,0,1,freq,true,IP_addresses,true,60,start_time,
    false,1);
IA1_node node2_IA(2,0,0,2,freq,true,IP_addresses,true,60,start_time,
    false,1);
all_my_nodes1.push_back(&node0_IA);
all_my_nodes1.push_back(&node1_IA);
all_my_nodes1.push_back(&node2_IA);
super_node my_super_node(6,0,true,false,IP_addresses,skip_ant,
    &all_my_nodes1);
my_super_node.run(dev_name.c_str(),start_time);
```

The code snippet above is similar to the from SISO_AMC_OFDM_realtime.cpp given earlier. However, this time the node which “run” is my_super_node. This node has three constituent nodes node0_IA, node1_IA and node2_IA. These three nodes use two antennas. The my_super_node use six antennas. These six antennas are distributed among the three nodes such that the first two antennas are given to node0_IA, the next two to node1_IA and so on. As already described earlier in the document, each node returns the structure frame_settings from it’s init() and process functions. This structure sets the start time of the next frame, the direction (i.e. transmit or receive), the frequency, and the gain of receiver and transmitter amplifiers. The constituent nodes of one super_node must all use the same frame timings and directions. However, the transmit and receive gains and the frequencies can change arbitrarily. Nevertheless, there is an option to override the gain and frequency settings of the constituent nodes using the optional override parameters in the instantiation of the super_node. This however, means that only a fixed set of gains and frequencies can be used. As it turns out this is the option that has been mostly used to date. The super_node has a method called set_nodes. This method can be used to replace the nodes of the supernode. This is very handy when performing multiple runs with different set of parameters in a tight sequence. We may think of this as replacing the DNA of the super_node. Another detail to notice, is that the constituent nodes has to be instantiated in simulation mode, i.e. the simulate parameter of the base class has to be set to true, see Section 4.1.

The constituent nodes of the super_node are run in different threads. This causes an issue with functions which are not thread safe. Example of non-thread safe functions are the IT++ random number generator, and FFT. The problem with these functions is that they employ static variables. When two thread calls functions which access this data simultaneously, problems arise. To resolve the two first problems we have introduces the method randb_thread_safe to the AMC object (thus you need to instantiate one AMC object to be able to use these functions). This method provide the same functionality as the randb function of IT++ which is typically used to generate random bits for transmission. Moreover, the OFDM object uses it’s own thread safe FFT implementation (pz_fft). The randb_thread_safe method also uses, as one of it’s arguments, the seed of the random number generator. This makes the handling of seeds very simple and explicit. How to handle non thread safe code section is described in Section 4.8, however, we recommend that non thread safe code is avoided, if possible.
**Realtime priority** To obtain the highest performance (lowest execution time). We may include the following code in our executables

```cpp
if (!(uhd::set_thread_priority_safe(1,true))) {
    std::cout << "Problem setting thread priority" << std::endl;
    return 1;
};
```

this sets the process priority to the highest level i.e. realtime. This implies that the executable must be started with root privileges e.g. using `sudo`. Since we call our executables from `matlab`, we must run it also with root privileges. Realtime runs should nearly always be done with realtime priority. When testing code using simulations, it may be useful to test execution time with realtime priority. However, during code development, it advisable not to use it, since, erroneous code will often hang the PC.

### 4.5 Simulation, debugging and transferring performance metrics

The first steps taken to develop and debug a new approach would be to make an implementation of a new node object in `my_approach_node.cpp` and `my_approach_node.hpp`. In the course of doing this, there may be a preceding Matlab implementation. Parts of the C++ code is also likely be developed in small programs outside of the `four_multi` framework. However, at some point one would start to create the implementation in `my_approach_node.cpp` and `my_approach_node.hpp`. The next step would be to create `my_approach_simulator.cpp` in order to start simulations. This includes choosing a channel model. At present there is only one channel model available namely `block_fading`, unless we also count `from_file` as a channel model (the use of this channel model is described in Section 4.6 below). If need arises, additional channel models will be implemented. The following code snippet illustrates the use of simulator

```cpp
std::vector<four_multi_node*> all_my_nodes;
block_fading ch_model;

// Create vector of nodes.
my_approach_node node0(...);
my_approach_node node1(...);
my_approach_node node2(...);
all_my_nodes.push_back(&node0);
all_my_nodes.push_back(&node1);
all_my_nodes.push_back(&node2);

// Set channel model parameters
ch_model.forgetting_factor=1;
ch_model.num_of_taps=2;
ch_model.is_awgn=false;
ch_model.is_diagonal=false;

// Run the simulation
simulate(all_my_nodes,ch_model);
```

During debugging it is frequently needed to observe the values of variables. One way of doing this may be to use the GNU debugger `gdb`. The author of this document has never used it and leaves to reader to find out how to use it. Another, very straightforward way of debugging is to use printouts using the standard C++ `cout` facility. For instance, to observe the state of the variable `rate` we may use the code line

```cpp
std::cout << "rate" << rate << std::endl;
```

this will write out the value of this parameter on it’s own line. This also works very well for IT++ variables. This illustrated by the following example
cvec x(10);
...
std::cout << "x" << x << ";" << std::endl;

The code above will print-out \textit{x} in a format which can be easily copy-pasted into Matlab for further processing and plotting. There is often a desire to terminate processing under a certain condition, and then store some variables for further processing in Matlab. This is technique exemplified by the following code.

\begin{verbatim}
cmat waveform_rx;
cvec impulse_response, x;
vec soft_outp:
...
if ((d_node_ix==1) && (Sync>0.9)) {
    std::cout << "Sync condition detected \n";
    std::cout << "x" << x << ";" << std::endl;
    itpp::it_file d1;
    d1.open("debug.it");
    d1<< itpp::Name("waveform_rx") << waveform_rx;
    d1<< itpp::Name("impulse_response") << impulse_response;
    d1<< itpp::Name("soft_outp") << soft_outp;
    d1.close();
    exit(1);
}
\end{verbatim}

Here you may notice that in addition to display the contents of the vector \textit{x} as a printout, it will also store the contents of the IT++ variables \textit{waveform\_rx, impulse\_response} and \textit{soft\_outp} in the file \texttt{debug.it} under the condition (\texttt{d\_node\_ix==1} \&\& (\texttt{Sync>0.9}). This condition could for example mean that the node has detected the presence of a communication signal. We may be interested to debug whether the soft values and channel estimate are correct. We then load them into Matlab using the Matlab function \texttt{itload}. This function is found in the IT++ library but is also located in the \texttt{utils} directory of \texttt{four\_multi}. Conversely, it is also possible to save a variable from Matlab into a *.it file and load it into C++/IT++ program.

Sometimes you may want to see what happens inside \texttt{libfour\_multi}. The way to this is to insert printouts as suggested above in e.g. \texttt{simulator.cpp}. In order to have the change make effect you need to rebuild \texttt{libfour\_multi}. This is done by entering the directory \texttt{libfour\_multi} and issue the commands \texttt{make all} and \texttt{sudo make install}.

Note that for both simulation and real time runs, we generally want to observe the results - for instance the frame error rates. This is done by assigning these results as public members of the node class \texttt{my\_approach\_node}. These members can then be treated in the main function of \texttt{my\_approach\_simulator.cpp} and \texttt{my\_approach\_realtime.cpp} as desired. Obviously we can also save these result in *.it files.

### 4.6 Working with real signals

Once your implementation work in simulations, the next step is to run live with real signals. In fact, this is probably the main cause for your effort to start with. One way of debugging the system is to log all the signals on harddisc for subsequent offline processing. This is done by calling the method \texttt{save\_raw\_input\_on\_file} on your \texttt{four\_multi\_node} instances before running. This method has the following declaration

\begin{verbatim}
void save\_raw\_input\_on\_file(uint32\_t max\_no\_frames\_to\_save,
std::string directory\_name="/usr/local/meas\_data/");
\end{verbatim}

By doing so the buffer contents of frame number \textit{y} of the node with index \textit{x} will be saved in the file \texttt{frame\_node=x\_frame=y.dat}. By default these files will be saved in the directory /usr/local/meas\_data/. However, another directory can be specified by using the second parameter \texttt{directory\_name}. 
The parameter directory_name can also be used to include a prefix to the filename e.g. 
directory_name=/home/arnold/sunday_. This will save all the results in the directory /home/arnold/
and all files will be prefixed sunday_. This is very useful when handling measurements from several
measurement campaigns. The data stored in the files can be loaded into Matlab using the following code:

```matlab
number_of_antennas = 2;
fileInfo = dir(filename);
fileSize = fileInfo.bytes;
Nsamples = fileInfo.bytes/(4*number_of_antennas);
fid = fopen(filename,'r');
temp = fread(fid,inf,'int16');
fclose(fid);
X = zeros(number_of_antennas,Nsamples);
for i1=1:number_of_antennas
    ix_start = 2*Nsamples*(i1-1)+1;
    ix_start1 = ix_start+1;
    ix_stop = 2*Nsamples*(i1-1)+2*Nsamples-1;
    ix_stop1 = ix_stop+1;
    X(i1,:) = temp(ix_start:2:ix_stop) - j*temp(ix_start1:2:ix_stop1);
end;
```

Unfortunately, the user need to know the number of receiver antennas as evident from the above code.
Another way of using the measurement data is to use the simulate function and the from_file channel
model. Assuming the files were stored with directory_name="/home/arnold/sunday_" we would define
the channel model as

```matlab
from_file chan_model("/home/arnold/sunday_");
```

By doing so the received signals in each node will be read from harddisc. This enables debugging of the
receiver functionality. It may also be used for post-processing of measurements. For instance, in the
IA_CoMP implementation described in Section 7, the decoder is not run in when running the system live
(only raw BER and FER is calculated). The decoder is subsequently run in the post-processing of
the measurement data using the from_file channel model as just described.

### 4.6.1 Matlab only

Another way of working with real signals is to both generate and sample signals from matlab without
making any C++ code at all. For this purpose we have devised the matlab functions txm and rxm,
which will be installed under /usr/local/share/four_multi/utils. These functions can be used to
transmit and receive a number of samples to and from Matlab, respectively. The parameters of the two
functions are documented and can be read by doing help in Matlab. However, we would like to mention
the gps_time_combined parameter which is present in both functions. If this parameter is set larger than
zero, then the action is delayed until the GPS time has passed the time indicated by
gps_time_combined and the following pulse-per-second (PPS) pulse. The gps_time_combined is formatted as

```
gps_time_combined = hour*3600+minute*60+second.
```

The GPS time is defined as GMT without daylight saving. The txm has a parameter called no_times.
This parameter specifies how many times the transmit signal should be repeated. If this parameter is set
to zero, the signal will be repeated indefinitely (to stop the transmitter do CTRL+C). This functionality
is useful when there is no working PPS signal - just let the transmitter run - and capture the received
signal asynchronously at the receiver (i.e. pps_trigger=0 in both transmitter and receiver). Another
application is when performing measurements with a spectrum analyzer. Note that the rxm function
may present the received signal conjugated compared to what you expect.
4.7 Inter-node communication using the backbone

When creating a node e.g. my\_approach\_node (i.e. an four\_multi\_node ancestor object) one of the arguments that must be provided to the base class is IP\_addresses which is of type std::vector<std::string>. This should be a vector of length equal to the number of nodes in the system where first element correspond to the backbone IP address of the node with d\_node\_ix=0, second element to node d\_node\_ix=1 and so on. If a node is not connected to any backbone then the string corresponding to that node may contain anything e.g. “xxx”. The system will still work as long as that node does not attempt to send information to any other node and no other node transmits information to it. Let’s say node number 1 (i.e. d\_node\_ix=1) is going to transmit to node number 2. Node number 1 would the call x2\_tx with the parameter other\_node\_ix set to 2. Reciprocally, node number 2 need to call x2\_rx with other\_node\_ix set to 1. If node number 1 calls first, it will be hung on the call until node number 2 calls x2\_rx and vice versa. The port number used for the transaction is given by 2000+100*tx+rx, where tx is the node index of the transmitting node and rx the index of the receiving node. The data to be transmitted is a memory segment of size bytes pointed to by the void * pointer pointer\_to\_data. The receiver need to know beforehand the size of the data to receive. Thus it is up to the user to do the required type casting. A convenient way to transfer data is often to transfer the raw samples directly.

It is possible to make multiple transfers during a single node\_process call. Typically, these calls are done after processing any received data and before generating any base-band signals to be transmitted. Several backhaul transfers can be performed during one call of node\_process. However, some combinations of transfer can lead to deadlock situations. These situations are few and arise and the reader can probably figure out when they will occur based on the descriptions above.

When doing simulation using the simulate function the backbone transfers are facilitated by buffer passing. This functionality is implemented inside the four\_multi framework and the user accesses it using the x2\_tx and x2\_rx calls, just as in the real-time execution. In other words the same code should run both in simulation and measurements. - which is one of the main intentions of the four\_multi framework. Contrary to what one would think, the simulations are actually more restrictive than the realtime runs. For instance, if a node calls x2\_rx to receive data from another node , the data need to be available i.e. transmitted from that node (using x2\_tx) before the node tries to read it using x2\_rx. To ensure this it may be useful to know how the simulate function works which is indicated by the following list:

1. Call init function for all nodes in the system in the order of the cnodes argument to the simulate call.
2. Generate the received signals in frame i.
3. Call node\_process for all nodes for all nodes which are receiving in frame i in the order of cnodes.
4. Call node\_process for all nodes which were transmitting in frame i in the order given by cnodes.
5. Set i = i + 1 and go to step 2.
6. Comment: Any node which reports END as the frame type of the next node is eliminated from the processing. When all nodes have reported END the simulation ends.

As a last resort, the user may distinguish between simulation and realtime runs using the d\_simulate member which is set true by four\_multi when doing simulation and false during realtime runs. For instance, extra frames could be introduces only in simulation to help transfer data through the backbone.

4.7.1 Using UDP in the backbone

The x2 functionality above uses TCP/IP for transferring data between the nodes by default. By setting the four\_multi\_node member variable use\_udp to true UDP/IP is used instead (this can be set e.g. in node\_init()). This improves the processing speed of the system. However, it is then necessary that the receiving end of a connection is ready before the transmitter. This turns out to be a problem when several nodes are simultaneously trying to transmit to the same node over the backbone. To solve this problem, we have implemented a version of the x2\_rx with the following declaration
void x2_rx(std::vector<uint32_t> other_node_ixs, std::vector<void*> pointers_to_data, std::vector<uint32_t> this_node_ixs)

this will in effect receive data from a vector of nodes in parallel (the data may arrive in any order). The call returns when all buffers have been received. The call currently supports three simultaneous receptions, but this number could be increased if need arises.

To switch between TCP and UDP on a per call basis, the calls x2.tx.tcp, x2.rx.tcp, x2.tx.udp, and x2_rx.udp can be used.

### 4.8 Thread safety

When running multiple nodes inside a super_node object, a problem arises with thread safety, since the constituent nodes work as separate threads in the same memory context. The user should use a mutex or more precisely a mutex of type boost::signals2::mutex to eliminate this problem. Thus the user could add a public member m to its my_approach_node class as

```cpp
... include <boost/signals2/mutex.hpp>
...
class my_approach_node : public four_multi_node {
...
public:
boost::signals2::mutex *m

In the my_approach_realtime.cpp program the user would then initialize the mutex as

```cpp
# include <boost/signals2/mutex.hpp>
...
boost::signals2::mutex m;
int which=0;
my_approach_node node0(...);
my_approach_node node1(...);
my_approach_node node2(...);
...
node0.m=&m;
node1.m=&m;
node2.m=&m;
node0.which=&which;
node1.which=&which;
node2.which=&which;
...

all_my_nodes.push_back(&node0);
all_my_nodes.push_back(&node1);
all_my_nodes.push_back(&node2);
```

Then it’s possible to protect sensitive areas of code by surrounding it by a m->lock(); and ending with a m->unlock();. This means that only one thread will enter the sensitive code at a time.

### 5 The software four_multi: coding and modulation

The four_multi software contains a component for adaptive modulation and coding which is implemented in the object AMC, and a component for multi-carrier multiplexing called OFDM1. The object OFDM1 also has functionalities for decoding including multi-antenna reception and combining. The user can decide to use these functionalities or implement the functionality themselves. The author recommends that AMC is used unless implementation of new coding schemes or following a particular standard
is an important component of the users project. The multicarrier multiplexing OFDM has many details to grasp and thus the gain for the user of using this module rather and making her/his own is more questionable.

5.1 Adaptive modulation and coding (AMC)

To use AMC you need to declare and initialize it as indicated by the code below

```cpp
#include <four_multi/AMC.hpp>
...
AMC amc_instance;
...
amc.init(code_word_size, load_from_file);
...
```

The adaptive modulation and coding AMC module has four coding rates 1/4, 1/2, 5/8 and 3/4. The four codes all have the same code word size, i.e. the size after encoding is the same independent of the code rate. The message length of input to the encoder is however, dependent on the code rate and is of size \( \text{code_word_size}/4 \), \( \text{code_word_size}/2 \), \( \text{code_word_size}*5/8 \) and \( \text{code_word_size}*3/4 \), respectively (rounded downwards). The actual length is returned by the method \text{codec_message_size}().

If \text{load_from_file} is set to \text{false}, then a new code is generated and saved on files called \text{code1_data1.it}, \text{code1_data2.it}, \text{code1_data3.it} and \text{code1_data4.it}. Generating the codes takes some time. In order to save this time the \text{load_from_file} may be set to \text{true} because then code parameters will be read from four files. However, it is up to the user to keep track of what \text{code_word_size} was used when the files were generated or otherwise the behavior will be undefined. The same code will always be generated for a given \text{code_word_size}.

The main methods of AMC are \text{modulate} and \text{demodulate}. The \text{modulate} takes a sequence of bits, encodes them using the \text{codec_ix} defined by the user. The encoded data is stored in the vector \text{transmitted} of type \text{bvec}. This vector need to be defined long enough to store the result (this assumed for all vectors which are used to store output results in \text{four_multi}). The encoded bits are also modulated using a QAM modulation of order \text{modulation_order} (one of 4, 16, 64 or 256) and stored in complex vector \text{symbols} of type \text{cvec}. The mean power of the symbols are one. A parameter \text{skip_symbols} can be used to move the location of where the complex symbols are written into \text{symbols}.

The \text{demodulate} method does the opposite of \text{modulate}, i.e., it takes noisy samples of the symbols (contained in the \text{cvec} vector \text{input_symbols}) and writes the output result into the two \text{bvec} vectors \text{message_hat} and \text{transmitted_hat}. Note that \text{input_symbols} needs to be equalized first i.e. \text{input_symbols} should look something like illustrated in Figure 8. The \text{message_hat} output is an estimate of the message bits obtained after iterative decoding of the LDPC code. The \text{transmitted_hat} output is an estimate of the encoded bits, which can be used to estimate the raw bit error rate. The demodulation needs an estimate of the noise variance. This can either be taken care of by the user, or the \text{noise_variance} member can be used. The latter is based on knowledge of the modulation constellation and estimates the noise variance by first doing hard detection and then estimate the noise power from the residual (the performance penalty is negligible for a code word size of 1520).

The demodulation is rather processor heavy operation. It is therefore often advisable to leave the decoding to after the realtime measurements, if possible. In order to easily disable decoding, the AMC object has a public \text{bool} member called \text{only_raw_bits} which can be set to \text{true} (the default value is \text{false}) to disable decoding when calling \text{demodulate}. If we save the received signals using the \text{save_raw_input_on_file} functionality, as described in Section 4.6, we can easily do the decoding afterwards. This procedure works as long as there are no realtime control loops which depend on frame errors. However, with a bit of cheating such scenarios can often be emulated by post-processing, see e.g. Section 5.2.1.

It is of course important to know the SNR needed to achieve a certain frame error rate (FER). This is illustrated in Figure 9 for the case of a code word size of 1520 (\text{code_word_size}=1520). A plot like this can be achieved by calling the method \text{run_through_all_mcs(void)} of AMC (for instance de-comment this call in \text{SISO_AMC_OFDM_node}, see Section 6). This function stores the result in a file called \text{results.it}. In the directory \text{utils} there is an Matlab script called \text{plot_waterfall} which creates a plot such as in Figure 9 based on \text{results.it}.  

25
Figure 8: Illustration of error vector

Figure 9: Throughput performance over an AWGN channel for individual coding and modulation settings without HARQ
5.2 Orthogonal Frequency Multi-Carrier Multiplexing (OFDM1)

The class OFDM1 helps the user to implement an OFDM modulation, which has the same subcarrier spacing as the IEEE 802.11 standards, but with fewer subcarriers. The modulation is tailored to fit the USRP using 25MHz sample frequency. The name used, OFDM1, indicate that there likely will be more versions of OFDM implemented later (OFDM2, OFDM3,...). The class also provides support for channel estimation and multi-antenna receivers. The division of work between AMC and OFDM1 is that AMC converts between bits and modulation symbols, and OFDM1 interfaces between modulation symbols and time domain samples. Obviously, the user can defer from using OFDM1 and implement her or his own scheme instead.

The OFDM1 need to be declared and initialized similar to AMC, see the following pseudo-code.

```cpp
#include<four_multi/modem_OFDM1.hpp>

OFDM1 my_ofdm;
my_ofdm.init(use_pilot_carriers, prepend_training, prefix_length, Ns, known_pos, re_order);
my_ofdm.init_multi_antenna(no_ant, buffer_size, interferer_pos);
```

In the code above we do two initializations. The first one is necessary for any use of the object, while the second is needed only when the object is to be used for multi-antenna reception. The modulation is based on a DFT of length Nfft=80. With the sample rate of 25MHz, this gives the desired subcarrier spacing of 312.5kHz. The transmitted signal, \( x(n) \), can therefore be written as

\[
x(n) = \frac{1}{N_{fft}} \sum_{k} s_k \exp(-j2\pi kn/N_{fft}),
\]

where \( s_k \) are the subcarrier symbols. The discrete-time frequency of the \( k \)th frequency is thus \( 2\pi k/N_{fft} \).

Due to aliasing frequencies \( \nu \) above \( \pi \) folds to \( \nu - 2\pi \). Thus (1) can be rewritten as

\[
x(n) = \frac{1}{N_{fft}} \sum_{k \leq N_{fft}/2} s_k \exp(-j2\pi nk/N_{fft}) + \frac{1}{N_{fft}} \sum_{k > N_{fft}/2} s_k \exp(-j2\pi n(k - N_{fft})/N_{fft})
\]

Of the \( N_{fft} = 80 \) subcarriers we use 38. The OFDM1 class has a member `ix_all`. Each element of this `ivec` holds the frequency bin number of a used subcarrier. (i.e. \( k \) in equation (2) above). The relationship between the frequency of the subcarriers and the `ix_all` vector is illustrated in Figure 10. Two of the subcarriers in Figure 10 are marked with red. These two subcarriers constantly transmit the known symbol \( \sqrt{0.5} + j\sqrt{0.5} \), if the parameter `use_pilot_carriers` is set to `true` in the initialization. The frequency bins of these two subcarriers are stored in the public members `pilot_carrier1` and `pilot_carrier2` which are therefore equal to 10 and 71, respectively. The two members `pilot_carrier1_sub` and `pilot_carrier2_sub` points to these subcarrier via `ix_all`, i.e. `ix_all(pilot_carrier1_sub)=pilot_carrier1` and `ix_all(pilot_carrier2_sub)=pilot_carrier2`. The `ivec` vector member `ix` is similar to `ix_all` but points only towards information bearing subcarriers. When `use_pilot_carriers` is set to `false` then `ix` and `ix_all` are identical. When `use_pilot_carriers` is set to `true` then `ix` is defined as shown in Figure 11.

The number of OFDM symbols to be transmitted in one burst is set by the `Ns` parameter. This number includes both payload and pilot symbols. The position(s) of the pilot symbols within the burst is set by the `ivec` vector `known_pos`. The number of payload OFDM symbols in one burst is given by `Ns-known_pos.size()`. The values in `known_pos` are numbers between 0 and `Ns-1`. The cyclic prefix length is specified by the `prefix_length` parameter and is in unit of samples (see any book on OFDM for a definition of the cyclic prefix). The total length of an OFDM symbol is thus `Ns+prefix_length` samples.

The vector `re_order` is a surprisingly useful vector that can be used to relocate the positions of the OFDM symbols, this will be discussed more below. If the flag `prepend_training` is set to `true` then the time-domain training sequence given by the member `train` will be added to the beginning of the frame.

```cpp
my_ofdm.init(use_pilot_carriers, prepend_training, prefix_length, Ns, known_pos, re_order);
...
We will assume that the system is synchronized using cables as described in Section 2 and we will thus assume that `prepend_training` is set to `false` in the following.

The method `modulate` formats an OFDM burst according to the parameters selected during initialization. The result is written into the parameter `waveform` which is of type `cvec`. The modulation symbols are taken from the input vector `symbols_in` which is typically taken from the output of `AMC::modulate`. Figure 5.2 below illustrates the burst formatting using the `OFDM1::modulate` method for the case $N_s=5$, $\text{known_pos}=2$, $\text{reorder}=10:4:1$, $\text{offset}=20$, and $\text{prefix_length}=10$. The position of the $n$th symbols is given by $\text{offset}+\text{reorder}(n)\ast(N_s+\text{prefix_length})$. Thus the $\text{reorder}$ vector can be used to relocate symbols within the burst. This is illustrated in Figure 5.2 where $\text{reorder}=\{0,1,4,6,7\}’$.

The `OFDM1` implementation has only a single pilot symbol. This means that multiple antennas and users need to time multiplex their transmission of the known pilot symbol to eliminate interference (this is where the $\text{reorder}$ parameter comes particularly handy). The frequency domain representation of this pilot symbol can be obtained by the `pilot_pattern` method which returns a length $N_{\text{FFT}}$ complex `cvec` vector. This `pilot_pattern` method has a second parameter `precoder` which is also a parameter of `modulate`. This is a `cvec` vector of length $\text{length\_ix\_all}$. In terms of equation (2) the value $s_k$ will be given by $\text{precoder}(p)\ast\text{symbols\_in}(p)$ where $k=\text{ix\_all}(p)$. The function of `precoder(p)` is, generally, used to implement linear pre-coding (also know as beamforming). However, the method `modulate` only creates the waveform for a single antenna, and thus the user need to iterate the `modulate` method with different recorders corresponding to the different antennas.

As mentioned in Section 3.1 the power level of the transmitted waveform need to be adjusted to a suitable range for the hardware. In order to facilitate this the `modulate` method provides an output parameter called `rms_power`. We suggest that this parameter is used during simulations in order to find suitable scaling factors to be applied on the `waveform` output. Ones such a scaling factor has been found it can be statically applied. To disable the calculation of `rms_power`, it can be set -10 when calling `modulate` (this eliminates the computational load associated with the power calculation).

In the receiver, the `demodulate` method may be applied to a `cvec` vector `waveform` of received data. Note that the receiver needs to know the start position of the of the first symbol i.e. the `start_sample` input parameter. This position will be different for simulated and real data as noted in Section 3.3.1

Another input parameter is the `position_of_known_symbols` (typically only one). The primary output of the `demodulate` method of `OFDM1` is `symbols_out`. If the reception is successful then these symbols should look like indicated in Figure 8 i.e. they should be noisy versions of the transmitted constellation symbols which can be used as input to the `demodulate` input of `AMC`.

The described `demodulate` function operates on single antenna signal. For multi-antenna reception `OFDM1` provides a method `demodulate_multi_antenna`. The `OFDM1` object need to be initialized also with with the method `init_multi_antenna` in order to use `demodulate_multi_antenna`. This initialization requires knowledge of the number of receive antennas, `no_ant`, the buffer size `buffer_size`, and the positions where the pilot symbols of the interferers are located, `interferer_pos`. When doing the actual multi-antenna reception, the method `extract_multi_antenna` is used. This function has as it’s first parameter the complex `cmat` matrix `waveform`. This should be a matrix with the number of rows and columns equal to the number of antennas and number of columns equal to `buffer_size`. Each row correspond to the signal received by one antenna. The `buffer_size` here used doesn’t necessarily have to be the same as the buffer size used over the air, as in Section 3.4, but should obviously cover the entire length of the considered signal. The parameter `interferer_pos` contains the positions (measured in OFDM symbol periods), where the interfering users are transmitting the known pilot symbols. Thus the $n$th interfering signal is assumed be located at $\text{interferer_pos}(n)\ast(N_s\ast N_{\text{fft}})+\text{start\_sample}$.

Based on the priory information thus supplied to the `OFDM1` object, the method combines the signals from the receive antennas based on MMSE weighting. The combining is done independently - subcarrier by subcarrier. The result is then equalized and the resulting symbol estimates placed in the vector `symbols_out` which (as in the single antenna case) represents noisy samples of the transmitted symbols. The noisy samples should subsequently be feed to to the `demodulate` method of `AMC` in order to extract the transmitted bits.
Figure 10: Illustration of all used subcarriers, their frequency and the use of the \texttt{ix\_all} member.

Figure 11: Illustration of all used subcarriers, their frequency and the use of the \texttt{ix} member when \texttt{use\_pilot\_carriers} is set to \texttt{true} during initialization.

Figure 12: Illustration of burst formatting with $Ns=5$, \texttt{known\_pos=2 \ reorder='''[0,1,4,6,7]'''},\texttt{ offset=20}, and \texttt{prefix\_length=10}.

Figure 13: Illustration of burst formatting with $Ns=5$, \texttt{known\_pos=2 \ reorder='''0:4'''}, \texttt{offset=20}, and \texttt{prefix\_length=10}.
5.2.1 Interleaving

In modern communication systems, erroneous frames are typically re-transmitted, and combined at the receiver. This can be emulated by transmitting the same frame several times and then determine how many transmission that were actually necessary, in post-processing. However, this implies that the re-transmitted symbols will experience the same channel and interference on each transmission. This is not realistic. To overcome this limitation, the functions `interleave_symbols` and `de_interleave_symbols` have been introduced. These functions re-order the subcarriers of the OFDM symbols in order to randomize channels. The re-ordering is determined by the parameter `index`, which is taken modulo 8. By varying this parameter between bursts, the sought randomization can be achieved.

5.3 Channel State information (CSI) pilots

Many MIMO schemes, in particular multi-user MIMO such as coordinated multipoint (CoMP) and interference alignment (IA), require channel knowledge at the transmitter. The OFDM class has some two methods that helps implement this. The first one, `insert_pilot`, can be used to insert an auxiliary pilot symbol (CSI pilot) into the transmitted waveform. The receiving node may then estimate the channel by calling the `estimate_channel` method.

6 The example code: SISO_AMC_OFDM

6.1 General

To exemplify how to use the components of four_multi the example SISO_AMC_OFDM has been created. This example resides in the examples directory. The example consists of the following files

1. SISO_AMC_OFDM_node.cpp
2. SISO_AMC_OFDM_node.hpp
3. Make
4. SISO_AMC_OFDM_simulator.cpp
5. SISO_AMC_OFDM_realtime.cpp

The last two files are executables which are compiled by issuing `make SISO_AMC_OFDM_simulator` and `make SISO_AMC_OFDM_realtime`, respectively. By compiling either of these two files `SISO_AMC_OFDM_node.cpp` will be compiled - but there is no use in compiling `SISO_AMC_OFDM_node.cpp` directly.

The example code uses a frame format with twenty payload OFDM symbols and one pilot symbol. The pilot symbols is located in the middle with ten payload symbols on each side.

6.2 SISO_AMC_OFDM_node

The file `SISO_AMC_OFDM_node.hpp` contains the declaration of the object `SISO_AMC_OFDM_node` which is the core of the example. The constructor of the object has the following declaration

```cpp
SISO_AMC_OFDM_node(float gain_rx,
    float gain_tx,uint32_t node_ix , double frequency,
    bool use_same_antenna,
    uint32_t modulation_order, bool simulate, uint32_t codec_ix, bool chase,
    double delta_time).
```

As the name indicates, the example uses only a single USRP at the transmitter and the receiver and it’s fixed at address 192.168.10.2. The first two parameters define the gain of the transmitter and receiver USRPs, which is used throughout the run. The `node_ix` defines the role of the node, `node_ix`=1 identifies the transmitter and `node_ix`=0, the receiver. The parameters `same_antenna` and `simulate` are passed on the `four_multi_node` base class, see Section 4.1. The parameters `modulation_order` and `codec_ix` are passed on to the AMC object, see Section 5.1. Let’s assume `chase` is set to `false` for the time being. The `SISO_AMC_OFDM_node` object has the following important members:
Some of the members are only used in the receiver (i.e. when node_ix=0). The vector symbols are used as the interface between AMC and OFDM1. The transmitter uses the following procedure

1. Randomize input.
2. Encode with AMC to fill symbols and transmitted (this is done in the private member function format_tx_symbols).
3. Interleave the symbols
4. OFDM multiplex symbols to obtain waveform_tx
5. Convert waveform_tx to d_tx_buffers[n], see Section 3.1.

while the receiver does an almost reversed sequence of operations

1. Convert the received shorts of d_rx_buffers[n] to waveform_rx, see Section 3.1.
2. OFDM demultiplex waveform_rx to fill symbols.
3. De-interleave the symbols.
4. Demodulate symbols using the AMC object to obtain message_hat and transmitted_hat (this is done in the private member function decode_rx_symbols).

In addition, the receiver also randomizes input to obtain transmitted. Why? In a real system this step would make no sense. However, here we are interested in evaluating performance. Since we know the seed used in the transmitter, we can actually obtain the same set of bits in the receiver. By doing so we are able to calculate the bit-error-rate. We do this and place the results in BERraw (comparison of transmitted and transmitted_hat) and BER (comparison of input and message_hat).

Four frames are transmitted. When QPSK modulation is used each frame holds one coding block of 20*38*2=1520 bits i.e. code_word_size=1520, see Section 5.1. Depending on the code rate, the message size of the coding block consists of 380, 760, 950 or 1140 for code_ix=0,1,2,3. When higher modulation orders are used, we transmit an increasing number of code blocks per frame (2,4 and 6 for 16QAM, 64QAM and 256QAM, respectively).

When chase is set to true so-called chase combining is used. This is done so that the same information is transmitted in each frame. Since we are using symbol interleaving (rather than bit interleaving) chase combining is achieved by maximum-ratio combining of the symbols just as in a diversity receiver. This procedure helps us emulate automatic repeat request, in the simplest manner possible.

6.3 Running

To run SISO_AMC_OFDM_simulator first check if the files code_data1.it, code_data2.it, code_data3.it and code_data4.it, reside in the directory. If not, or if you don’t know if they were generated with code_word_size=1520 you need to re-generate them. This done by changing the line

amc.init(word_length,true);

to

amc.init(word_length,false);
exit(1);

The second line stops the simulation. Recompile SISO_AMC_OFDM_simulator and run, as ./SISO_AMC_OFDM_simulator, it will take a few minutes. Once completed you can change the code back, recompile and finally run the simulations. The output of the program should be something like
Processing in the transmitter
codec_ix=1
=================================================================
time=0
Processing in the receiver
Processing in the transmitter
codec_ix=1
=================================================================
time=0.03
Processing in the receiver
Processing in the transmitter
codec_ix=1
=================================================================
time=0.06
Processing in the receiver
Processing in the transmitter
codec_ix=1
=================================================================
time=0.09
BERraw=[0.0802632 0.0677632 0.0934211 0.0907895 0.0756579 0.0769737 0.0690789 0.0644737]
BER=[0 0 0 0 0 0 0 0]
The printout == and time= comes from the simulate function to help debugging.
To run in realtime, the code_data-files need to be generated in the transmitter as well as receiver. The command in the transmitter would typically be something like

```
sudo ./SISO_AMC_OFDM_realtime --node_ix=1 --freq=2490e6 --start_time=36540
```
and in the receiver

```
sudo ./SISO_AMC_OFDM_realtime --node_ix=0 --freq=69999000 --start_time=36540
```
The frequencies used correspond to the testbed described in Section 2.2. Note that the frequency of the receiver is not exactly 70MHz. This is due frequency offsets. This frequency offset can be estimated using the est_freq_offset utility. The GPS time is formatted as

gps_time_combined=hour*3600+minute*60+second.
Note that the GPS time is defined as GMT without daylight savings. The receiver can be run without the transmitter but then the BER will be close to 50% since there is no signal. An additional important parameter is delta_time. This parameter specifies the time between bursts and therefore dictates the time allowed for processing. For higher modulation constellations you may need to increase it's value from the default 30 milliseconds (0.03). This can be done from the user command line using the parameter delta_time.

7 An implementation of interference alignment and Coordinated Multipoint: IA_CoMP

Here we present the implementation used to produce the results in [3]. The implementation is packaged separately from four_multi. A class-by-class documentation is found at http://iacomp.sourceforge.net/.
7.1 Installing

To download the software do

```
git clone git://git.code.sf.net/p/iacomp/code IA_CoMP
```

The software has a root directory and a subdirectory `IA1` where all the code resides. The name of the subdirectory reflects that interference alignment was the main focus of the work. In the future we expect additional implementations `IA2`, `IA3` and so on. Online documentation can be found at [http://iacomp.sourceforge.net/](http://iacomp.sourceforge.net/). To compile do

```
make IA1_simulate
make IA1_super_realtime
```

from the `IA1` directory. The code consists of the following important files

- IA1_node.cpp
- IA1_node.hpp
- calculate_beamformers.hpp
- IA1_simulate.cpp
- IA1_super_realtime.cpp
- run.m
- Illustrate_IA.m
- estimate_Hbig.m

The files `IA1_node.hpp` and `IA1_node.cpp` contain the declaration and definition of the object `IA1_node` (together with `calculate_beamformers.hpp`) which defines the behaviour of all the schemes and variants described in [3]. Again, the naming follows from the interest in interference alignment (IA) which spawned the work.

7.2 Running

To simulate the system, simply issue the command `.IA1_simulate`. This will run through all the schemes and variants described in [3], on the Rayleigh fading channel `block_fading` described in see Section 5 above. To run the system in real-time using the platform described in Section 2.2, the `IA1_super_realtime` need to be run in the transmitter and receiver computer. This can be achieved by calling `run.m` from matlab on transmitter and receiver computer. This function will then in turn run `IA1_super_realtime`. To do this super-user privilidges are needed. This is easiest to achieve by running matlab as root. The script `run` needs two variables to be set, `start_time_str`, and `super_node_ix`. The first one is a string which is changed by editing the file `run.m`. The string indicates the time when the system should run defined by GPS time (GMT). The format is “hhmmss”. The second one, `super_node_ix`, is given from the matlab command line as zero in the transmitter PC and one in the receiver PC. In the receiver computer three additional inputs are needed, namely, `df0`, `df1` and `df2`. These are the three frequency offsets associated with the three receiver nodes. These are obtained as described in Section 2.2 above.

When running the system, the transmitter-PC will beep (an 800Hz sinusoid) around 20 seconds before start in order to alert the user. When this beep goes off people should stand still (or move very slowly) if transmitter channel state information is to be useful. When the run is completed, another beep (400Hz) is emitter by the transmitter-PC, to indicate to the users that they can move freely. During the run, debug information is presented on both screens. The receiver displays raw bit error rates and a graphical representation of the performance of interference-alignment and coordinated multipoint. This is done by the function `illustrate_IA`. The signals received in the receivers are saved on harddisc as desribed in Section 4.6. The time of the measurement is used to calculate an integer `start_time` defined as `hour*3600+minute*60+sec`. The `start_time` is used to prefix the measurement files as
This parameter `start_time` is also used to spawn the seed of the random number generators used to generate the data bits, see also Section 4.6. At the end of the run, the receiver PC uses the measured files and the matlab script `estimate_Hbig`, to calculate statistics which are presented graphically by `Illustrate_IA`, as described in Section 7.2.1 below. The “phase evolution”, i.e. the phase-shift, between the first and second frame, from the viewpoint of one of the receiver nodes is also printed on screen. The user should check that this phase-shift is approximately the same for all transmitter antennas (within a few degrees), in order to verify that all transmitter antennas are sufficiently sample and frequency aligned.

### 7.2.1 Illustrate_IA

The matlab function `estimate_Hbig` uses the data received from all six antennas to calculate three two-by-six MIMO matrices, each one representing the channel between all base-station antennas and each of two antenna mobiles. This is done for all frames and subcarriers. The output is then used in `Illustrate_IA` to present viewgraphs as indicated in Figure 14. Each of the six subplots indicate the performance at a mobile. The mobiles has two antennas, thus from the viewpoint of the mobile, each of the three streams (one desired and two interfering) arrive over 2x1 SIMO channels. The mobile can only successfully detect the bits of the desired signal if the interfering signals are weak, or if they arrive in the same subspace (or almost in the same subspace). This is illustrated with three arrows representing the desired and the two interfering signals. The length of the arrows corresponds to the signal strengths in dB. The desired signal is black and is oriented in the direction of the x-axis. The interfering signals (in red and blue) are offset at an angle corresponding to the angle between its vector channel and that of the desired signal. The angle between the interfering signals, correspond to the angle between the vector channels of those two signals. Thus, good performance can be expected when either the two interfering arrows are aligned or if they are significantly weaker than the desired signal.

The illustration is animated where the motion is obtained by stepping through the 38 subcarriers (not by temporal changes). The upper row of subplots consider CoMP and the lower IA.

The output of `estimate_Hbig` is also used to calculate the phase evolution already mentioned.

### 7.3 IA1_node: sequence of processing

The implementation of `IA1_node` borrows heavily from `SISO_AMC_OFDM_node`. Just as `SISO_AMC_OFDM_node`, `IA1_node` has private methods called `format_tx_symbols` and `decode_rx_symbols` which are almost identical to those of `SISO_AMC_OFDM_node`. The following enumeration illustrates the operation of the transmitter nodes

1. Obtain the beamformers
2. Randomize input.
3. Encode with AMC to fill symbols and transmitted (this is done in the private member function `format_tx_symbols`).
4. Interleave the symbols
5. OFDM multiplex symbols to obtain waveform_tx, for all transmit antennas, using the calculated beamformers.
6. Convert waveform_tx to d_tx_buffers[n], see Section 3.1, for all transmit antennas.

The enumeration is almost the same as the one for `SISO_AMC_OFDM_node` presented in Section 6.2 above, except for the additional steps marked with italic font. The first step can be further broken down. This breakdown depends on whether the node is master or not. The master node is the node with d_node_ix=0. The master nodes collects the channel state information (CSI) from all three receivers and calculates the beamformers to be applied by the other transmitters. In CoMP there is only one logical node at the transmitter. For open-loop MIMO, no beamformers are needed but the feedback procedures are used anyway for programming simplicitly. For the master node, the task of obtaining the beamformers can be broken down as follows
Figure 14: Screenshot
1. Obtain CSI from the first mobile.
2. Obtain CSI from the second mobile.
3. Obtain CSI from the third mobile.
4. Calculate all beamformers (calculate_beamformers())
5. Send beamformers to the second base-station.
6. Send beamformers to the third base-station.

New beamformers can be relatively easily implemented by just modifying calculate_beamformers(), see Section 7.6 below.

For the base-stations which are not the master, all they need to do is to receive the beamformers from the master. The CSI feedback format we have used is simply to send the received raw samples (std::complex<int16_t>) to the transmitter. This works since we are using a high-capacity backbone for the transfer (i.e. not a wireless channel). In the receiver the work can be outlined as follows

1. *Feedback the CSI to the (master) transmitter.*
2. Convert the received shorts of d_rx_buffers[n] to waveform_rx, see Section 3.1, for all receive antennas.
3. OFDM demultiplex waveform_rx to fill symbols by using MMSE combining of the received signals as described in Section 5.2.
4. De-interleave the symbols.
5. Demodulate symbols using the AMC object to obtain message_hat and transmitted_hat (this is done in the private member function decode_rx_symbols).

### 7.4 IA1_node: important members

The IA1_node implementation has the following important matrices/vectors which hold the state of the node and determine it’s behaviour:

- **cvec waveform_rx**: This is a matrix with the received signal samples.
- **waveform_tx**: This is a matrix of samples to be transmitted.
- **BERraw**: The raw bit error rates of the bursts.
- **EVM**: The error vector magnitude of the bursts, inherently includes both noise, interference and impairments.
- **SINR**: This is a structured SINR estimate made by the receiver.
- **message_hat**: Storage for the hard detected bits.
- **transmitted_hat**: Storage for the detected code-word.
- **transmitted**: The encoded code word.
- **ai[]**: A vector of OFDM1 objects. One for each modulation stream in the node.
- **d_node_ix**: Node index. Starts with BSs and then MSs.
- **stream_ix_this_node**: The modulation streams are numbered. This vector holds the stream indices of the corresponding node (a global stream indices).
- **stream_ix_to_ms_ix**: Mapping from streams to MS index. The mobile indices starts with zero for the MS with the smallest node index.
• **bs_to_stream_ix**: Mapping from base-station index (=node index) to stream index. Each column contains the stream indexes of the corresponding base-station.

• **is(bs)**: Is true if the node is BS otherwise false.

• **num_ant_ms**: Number of antennas in all MS.

• **num_ant_bs**: Number of antennas in all BS.

• **d_no_ant**: Number of antennas in the node.

• **H[20]**: This entity holds the transmitter CSI in the system. Element $H[k][l]$ is the CSI between the $k$:th transmitter antenna and the $l$:th mobile. The transmitter antennas, are all the base-station antennas in the system (maximum 20, so far 6). Each elements is a num_ant_ms by length_ix_all matrix of complex values, where length_ix_all, is the number of subcarriers.

• **Vbig[]**: A vector of matrices. Each column contains a beamformer (precoder) for the corresponding subcarrier.

• **do_calculate_beamformers**: This boolean is set to true in scenarios where calculate_beamformers(), see Section 7.6, is used to globally optimize beamformers.

### 7.5 IA1_node: parameters

The IA1_node implementation can be used to implement all the schemes used in [3]. We have tried to realize all the schemes using using the same code but with different parameters. These parameters are set in the constructor. Some of the parameters are based on the input parameters to the constructor while some are hardcoded in the beginning of the constructor. Some combinations of the parameters may not create a meaningful result.

- **scenario**: 1=IA, 2=CoMP, 3=MIMO, 4=SIMO

- **node_ix**: Determines the role of the node. For all schemes except CoMP: node_ix=0,1,2 = base-station, 3,4,5 mobile-station. For CoMP: node_ix=0 base-station, 1,2,3=mobile-station.

- **no_ant**: Number of antennas in the node.

- **num_streams_per_ms**: Number of modulation streams in ms.

- **num_streams_per_bs**: Number of modulation streams in bs.

- **num_antennas_in_system**: Total number of transmit antennas of all base-stations combined.

- The following parameters are only used with single-user MIMO:
  - **MIMO_all_active**: Set to true if all three links are active all the time.
  - **MIMO_active_tx_node_tx**: Selects the active link (0,1,2) when MIMO_all_active is false.
  - **MIMO_closed_loop**: Set to true to employ channel dependent pre-coding. Not implemented.

### 7.6 IA1_node: calculate_beamformers()

This function calculates the beamformers (a.k.a linear precoders) for all the base-stations and all subcarriers according to the principles of the max-SINR algorithm described in [5]. These calculations are located in the function calculate Beamformers() which in turn calls the function iteration. The two functions have been written so that they should be possible to understand with only minimal knowledge of the rest of the code. The code is also located separately from the rest of the implementation in the file calculate_beamformers.cpp. The baseline schemes MIMO and SIMO do not use calculate_beamformers().

The first thing to observe is that we are considering OFDM in this paper which is not the case in [5]. What we do is therefore to apply the scheme of [5] on each subcarrier. Do we calculate the beamformers for each subcarrier independently? Well, in principle we do. However, since the max-SINR algorithm is an iterative one, we use the results from the previous subcarrier as a starting point for the iteration (except for the first subcarrier of course). The iteration function has the following definition.
void IA1_node::iteration(const cmat v_old_in, int subcarrier_ix_start, int subcarrier_ix_stop, int increment)

This function implements “Algorithm 2” of [5], on subcarrier subcarrier_ix_start to subcarrier_ix_stop. The matrix v_old_in represents the starting values of corresponding to \( V^k \) of [5] when operating on subcarrier subcarrier_ix_start. For the next subcarrier, the optimized values for the adjacent subcarrier is used to initialize the algorithm.

Our v_old_in matrix corresponds related to \( V^k \) more specifically as

\[
v_{\text{old in}} = [V^{[1]}, \ldots, V^{[K]}].
\]  

(3)

When setting the initial value \( v_{\text{old in}} \), we use the values from the previous frame. We call iteration four times in calculate_beamformers(). Referring to Figure 10, these four passes processes the subcarriers from \( m = 10 \) to \( m = 19 \), \( m = 10 \) to \( m = 1 \), \( m = 30 \) to \( m = 38 \) and \( m = 30 \) to \( m = 20 \).

Inside the iteration function, the reader may look for the loop over ix. Inside this loop is where the beamformer weights are calculated. This code have been written to follow Algorithm 2 as outlined in [5] as clearly as possible e.g. through the use of similar variable names.

Received combining vectors are calculated as well. However, these are never used in the receiver. New weights are instead calculated. These new weights compensate (to some degree) for the channel drift between the time of CSI feedback and use.

The results of calculate_beamformers() and iteration is ultimately stored in \( V_{\text{big}}[] \).

7.7 The physical layer

The physical layer of IA_CoMP is based on the modulation OFDM1, see Section 5.2, and the modulation and coding AMC, see Section 5.1. The MAC frame used is illustrated in Figure 15 below. The MAC frame contains 10+10 payload symbols, three demodulation pilots and six channel state information pilots. The overhead of nine pilots is way to large for any real implementation compared to the payload of 20 symbols. Our main interest here is however to investigate the principles of interference-alignment.

There are three modulated streams in the system, these streams are simultaneously transmitted during the 20+20 payload symbols. This is made possible by beamforming according to the principles of interference alignment (IA) and coordinated multipoint (CoMP). However, these schemes need global channel knowledge. This is achieved by the six pilots, one for each transmit antenna, in the system. The pilots are inserted into the transmitted signal using the insert_pilot method of OFDM1. The corresponding channel state information is extracted from the channel using the method estimate_channel method of OFDM1. Strangely enough this done by the transmitter, how can this be? The reason is that the mobile actually feed-backs the raw samples corresponding to the pilot transmission and then the estimation is done in the master transmitter (d_node_ix=0). The code corresponding to this processing is found under the comment line

```
// Receive data from mobiles
```

in IA1_node.cpp. This is of course unrealistic in a practical implementation. The feedback of the data is done in the receiver using the call

\[
x2_{\text{tx}}(0, \text{feedback_buffer}, \text{feedback_buffer_size});
\]

which is found around line 500. In the receiver MMSE combining is performed using the extract_multi_antenna method of OFDM1. This method needs knowledge of the channels of the desired and interfering signals. This is provided by the three demodulation pilots.

The nodes (i.e. IA1_node objects) have a member called a1 which is a vector of OFDM1 elements. In the base-station of the CoMP implementation, this vector has three elements, in all other cases it has one element. The elements are configured as described above. The general parameters are set using the init function and the multi-antenna receiver specific ones using init_multi_antenna. This done around line 232-299 in the code of IA1_node.
8 Compile Using Intel Compiler

The following sections describe how to use the INTEL compiler icpc instead of gcc when working with four_multi. The reason for doing so, is of course that the INTEL compiler can deliver faster code. We describe below how to have the two compilers readily available. To do this, we re-compile the IT++ library and four_multi. We do not install multiple versions of the libraries but it is easy to switch back and forth.

8.1 Install the INTEL compiler

Visit the webpage http://software.intel.com/en-us/c-compilers and download the compiler, in our case Intel C++ composer xe 2013 for linux. Install according to the instruction. In our case the installation is found in the directory /opt/intel/composer_xe_2013.3.163 and its subdirectories (if it is located elsewhere on your system change the files below accordingly).

Go to the directory /etc/ld.so.conf.d, create a new file, it may have any name, except it must have extension conf e.g. my.intel.conf (the file may not be called just .conf). In this file add the following two lines

```
/opt/intel/composer_xe_2013.3.163/mkl/lib/intel64/
/opt/intel/composer_xe_2013.3.163/compiler/lib/intel64/
```

These lines instructs the system to look at this locations when searching for libraries at runtime. The first two lines helps the loader find icpc related libraries the third line helps the compiler find our new boost libraries compiled with icpc, see below. Next the user should specify the location of icpc, and the location of the new boost libraries. This done by specifying this in the /etc/bash.bashrc file as

```
export PATH=$PATH:/opt/intel/composer_xe_2013.3.163/bin/intel64
export INTEL_LIBS=/opt/intel/composer_xe_2013.3.163/mkl/lib/intel64
export INTEL_LIBS2=/opt/intel/composer_xe_2013.3.163/compiler/lib/intel64
```

Start a new shell (terminal window) to make the above changes take effect.

8.2 Install high-speed message passing library MPICH

```
sudo apt-get install libmpich-mpd1.0gf
cd /usr/lib
sudo ln -s libmpich-p4mpd.so.1.0 libmpich.so
```

8.3 Recompile IT++ using Intel Compiler

First uninstall the gcc compiled version of IT++. This is simply achieved by changing directory to itpp-4.0.7 and do sudo make uninstall. A flexible installation is obtained by creating a new directory which we may call itpp_icpc (located in a good place). Unpack the IT++ source code as

```
1. tar jxvf itpp-4.0.7.tar.bz2
2. cd itpp-4.0.7
```
First go to the directory in the IT++ source where `elem_math.cpp` and `elem_math.h` are located. Copy `elem_math.cpp.patch` and `elem_math.h.patch` from `four_multi/patches` to this directory. Apply the patches by issuing the commands

1. `patch elem_math.cpp < elem_math.cpp.patch`
2. `patch elem_math.h < elem_math.h.patch`

Then change directory to `itpp-4.0.7`. Issue the following commands

1. `export CXX=icpc`
2. `export THE_LIB=$INTEL_LIBS/libmkl_rt.so`
3. `./configure --with-blas=$THE_LIB --with-lapack=$THE_LIB --with-fft=$THE_LIB`
4. `make`
5. `sudo make install`

### 8.4 Rebuild four_multi

Change directory to `four_multi/libfour_multi`. Issue the following command

`make --makefile=Makefile_icpc IA1_super_realtime`

### 8.5 Rebuild applications

The Makefile for the example code `SISO_AMC_OFDM_realtime` and `SISO_AMC_OFDM_simulator` as well as `IA_Comp` are all compatible with the INTEL compiler. The are compiled using the modified command

`make --makefile=Makefile_icpc target_name`

A close inspection of the modified makefiles shows that only targets that involve signal processing and no use of the boost library are compiled with `icpc`. Other files are still compiled with `gcc`. The applications are linked with `gcc` but the math libraries and some other related libraries are taken from the `intel` compiler implementation.

### 9 Install real-time kernel

The following webpage describes how to switch to realtime (i.e. low latency) kernel in ubuntu 12.04: http://askubuntu.com/questions/72964/how-can-i-install-a-realtime-kernel. At the time of writing this section, i.e. june 2013, these instructions needed to be updated. The following is an updated list of commands:

3) `tar xjvf linux-3.8.13.tar.bz2`
4) `cd linux-3.8.13`
5) `patch -p1 < (<(bunzip2 -c ../patch-3.8.13-rt10.patch.bz2`
6) `cp /boot/config-$(uname -r) .config && make oldconfig`
7) `sed -rie 's/echo \"\"/#echo \"\"/\' scripts/setlocalversion`
8) `make-kpkg clean`
9) `CONCURRENCY_LEVEL=$(getconf _NPROCESSORS_ONLN) fakeroot make-kpkg --initrd \
--revision=0 kernel_image kernel_headers`
10) `sudo dpkg -i ./linux-headers-image-3.8.13-rt10_0_*.deb`

This installs a realtime version of the 3.8.13 kernel. During configuration i.e. line 6 above, the user is asked make a number of choices. The first one, is the selection between “1. Simple tick based cputime accounting (TICK_CPU_ACCOUNTING) (NEW)” and “2. Fine granularity task level IRQ time accounting (IRQ_TIME_ACCOUNTING)” Here we choose the first alternative. Another selection is among the following five alternative
1. No Forced Preemption (Server) (PREEMPT_NONE).

2. Voluntary Kernel Preemption (Desktop) (PREEMPT_VOLUNTARY)

3. Preemptible Kernel (Low-Latency Desktop) (PREEMPT_LL) (NEW)

4. Preemptible Kernel (Basic RT) (PREEMPT_RT) (NEW)

5. Fully Preemptible Kernel (RT) (PREEMPT_RT_FULL) (NEW)

Here we select “5”. For the remaining choice (which are plenty) we simply choose the default alternative by just pressing return.

References


