Evaluation of Synchronization Algorithms with USRP

Martin Lülf

A Thesis submitted for the Degree of
Bachelor of Science

Institute for Communications and Navigation

Prof. Dr. Christoph Günther

Supervised by Dipl.-Ing. Ronald Böhnke

München, May 2012

Institute for Communications and Navigation

Technische Universität München
Theresienstrasse 90
80333 Munich

Abstract

An interface between the Universal Software Defined Radio Peripheral (USRP) and MATLAB has to be developed, as well as a scheme for frequency, delay and phase synchronization to detect received M-PSK symbols after transmission over the USRPs.

Different methods for the interface implementation are considered and the final implementation is done as a shared Mex C++ library that can be accessed from within MATLAB like usual functions. The synchronization is done by a feedforward estimator using a BPSK pilot sequence proposed by [1] followed by a delay tracking loop using Gardner's timing error detector [2, 3] and a phase tracking loop using the Viterbi&Viterbi algorithm [4, 3] for M-PSK signals. Frame synchronization and resolution of the phase ambiguity is done by a Barker code as start of frame sequence.

It is shown that the feedforward estimator achieves the Cramér-Rao bounds and the following tracking is able to keep the received samples synchronized. Using the MATLAB interface, a set of random data is transmitted over the USRPs, synchronized and detected at the receiver and compared to the original data sequence to validate the single components of this work.

Contents

Chapter 1

Introduction

When investigating new signal processing schemes the simulation of all relevant parameters is a useful tool to get a better understanding of the investigated system. Programmable numerical math software like MATLAB or Octave are well suited for these simulations as they already have functions for most of the mathematical problems as well as for graphical visualization of data, so the implementation of the simulation can focus on the signals and processing algorithms themselves.

After successful simulations it is often desirable to test the algorithms in a real world scenario in order to ensure that there have been no wrong assumptions in the simulations and the algorithms work in a complete communication system. Software defined radios (SDRs) allow the transmission of a wide variety of signals and can thus be easily adopted to such tests. In Chapter 2 the Universal Software Defined Radio Peripherals (USRP), a software defined radio product family by Ettus Research and National Instruments, are described to use them for testing.

If the SDRs can be accessed from the same software that was used for the simulations, another advantage over specialized RF hardware is that signal processing code from the simulations can be reused. In Chapter 3, an interface from MATLAB to the USRPs is developed so MATLAB code can directly access the USRP devices for testing.

While simulations can simulate special aspects of the communication system, when using SDRs, all aspects of a communication system have to be considered. The often made assumption of coherent detection is not directly possible for real measurement data. The received samples have to be synchronized first to continue with a coherent detection. In Chapter 4, a synchronization scheme is discussed which removes carrier frequency and phase offsets as well as timing errors after transmission over USRPs based on a maximum likelihood feedforward estimation for an acquisition, followed by a feedback tracking loop using a Gardner timing error detector and the Viterbi&Viterbi phase error detector to follow further changes of the timing and phase for arbitrary M-PSK modulated data. These synchronization schemes allow the test of the work from the previous chapters both in simulation and with real measured data from USRPs. Additionally they can be used for future works to transmit data symbols over the USRP and focus on the received coherent samples without worrying about synchronization.

Chapter 2

Universal Software Defined Radio Peripheral

The Universal Software Defined Radio Peripheral (USRP) family are software defined radios that allow transmission and reception of arbitrary baseband signals. Due to the modularization into a mother- and daughterboard they can be adopted to a wide range of operating frequencies. The USRP daughterboards are responsible for modulation and antialiasing filtering and the motherboards are dealing with amplification, up- and down converting, decimating and interpolating operations that require high processing power but little knowledge about the signal itself in a Field Programmable Gate Array (FPGA). All waveform dependent operations and signal processing focused part are done on a PC which enables a high variety of transmission systems with the same hardware.

The USRPs can either sample a signal mixed to an intermediate frequency with one sinusoidal signal which results in a one dimensional real valued discrete signal or use the orthonormal property of a sine and a cosine wave to sample the Inphase and Quadrature component individually on two channels to get a complex[1] two dimensional discrete signal.

2.1 Overview

Initially Ettus started with one SDR, the USRP1 device. After its success newer USRP devices with more features were developed. The next two subsections will give an overview about the different devices in the USRP family.

2.1.1 USRP1

The first generation, the USRP1 has four input and four output channels and is connected via USB[2]. The USRP1 can contain two daughterboards and all streams together can share a data rate of 8 complex mega samples per second [5]. With a few hardware modifications an external 10MHz oscillator can be

[1] Where the real part corresponds to the Inphase component and the imaginary part to the quadrature component

[2] Universal Serial Bus 2.0 with a maximum data rate of 32 MByte per second [5]

Figure 2.1: USRP1 device equipped with one daughterboard connected to two RF antennas and no external clock connected.

connected to the USRP to provide a more stable oscillator. Figure 2.1 shows a USRP1 device with no external clock and one daughterboard.

2.1.2 Next Generation USRPs

The second generation of USRPs consists of a few different devices with similar properties. These new USRP devices can hold only one daughterboard and have two input and two output channels. Their FPGAs can be reprogrammed for more control of the data processing before decimation and transmission to the PC. Beside the USRP2, each model comes with two versions (e.g. USRP N200 & N210) where a tailing 10 in the model number indicates a larger FPGA than in the tailing 00 models which gives more space for custom preprocessing code.

The next generation USRP devices can be connected with an external oscillator for a more stable frequency and an external pulse generator for a more precise timing reference. These USRP devices can switch between the different oscillation references and the different timing references (internal, external, MIMO[3]) via software. The new USRP devices can tag incoming and outgoing samples to know when exactly they have been received or have to be send. This leads to much higher accuracy in runtime measurements or Time Division Multiple Access (TDMA) systems as the delay introduced by USB or Ethernet does no longer affect the delay of transmission or measurement of reception time.

USRP2

This model was an improvement of the original USRP1. This USRP model is MIMO capable which means multiple USRPs can be connected and operate as a single USRP with more channels. It is no longer sold in favour of the newer N2x0 devices.

- connected by Gigabit Ethernet
- external frequency reference possible
- external one pulse per second (pps) timing reference possible
- reprogrammable FPGA
- MIMO capable

[3]Multiple Input Multiple Output using a special MIMO cable to synchronize the USRPs

USRP B100 & B110

The B1x0 is an improved version of the USRP1 targeted as low-cost USRP.

- connected by USB 2.0
- external frequency reference possible
- external pps timing reference possible
- reprogrammable FPGA

USRP E100 & E110

This USRP model has a small embedded Linux system onboard to allow signal processing independent of an additional PC.

- internally connected by general purpose input/output (GPIO)
- externally connected by USB 2.0 and Fast Ethernet
- external frequency reference possible if additional connector soldered to the board
- reprogrammable FPGA

USRP N200 & N210

Figure 2.2 shows a USRP N210, the successor of the USRP2 model.

- connected by Gigabit Ethernet
- external frequency reference possible
- external pps timing reference possible
- reprogrammable FPGA
- auxiliary analog and digital I/O signals
- MIMO capable

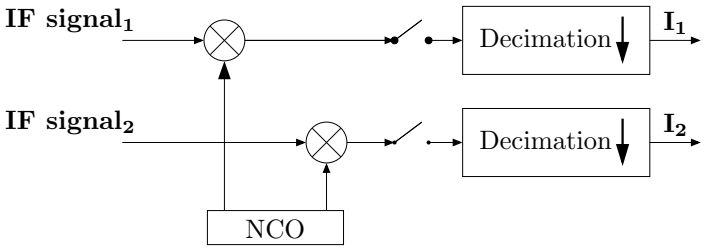
Figure 2.2: USRP N210 device without external references equipped with one daughterboard connected to two RF antennas.

2.1.3 Motherboards

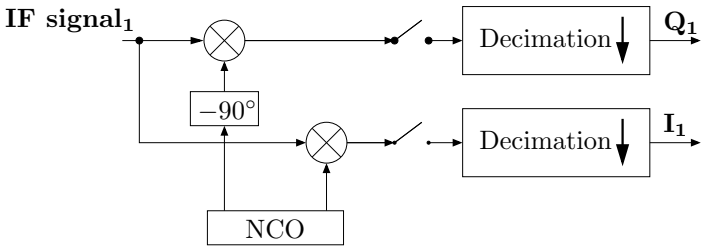
The heart of a USRP is it's motherboard which handles the communication with the PC and the signal processing at an intermediate frequency (IF). Daughterboards described in Section 2.1.4 handle the modulation from the intermediate to the operating frequency.

The input signal that was filtered and modulated to the IF by the daughterboard is first sampled and multiplied with a time discrete cosine wave or a sine and cosine wave for complex samples as shown in Figure 2.3. Afterwards the signal is again filtered and decimated to cancel out the double frequency parts and reduce the number of samples. The resulting baseband samples are stored in a buffer and transmitted to the PC over the USB or Ethernet interface.

Receive Part



(a) real samples



(b) complex samples

Figure 2.3: Reception path in the motherboard for real and complex samples

Transmit Part

At the transmission side the data is received over Ethernet or USB and stored in a buffer. Out of this buffer the samples are upsampled, interpolated and modulated to the intermediate frequency by a cosine for real samples or a sine and a cosine for complex samples. Then the samples are converted into analog signals and are send to the daughterboard.

2.1.4 Daughterboards

The operating frequency band of the USRP can be controlled by exchanging the daughter board. Each daughterboard has access to two RX and two TX channels

for either one complex or two real valued streams in RX and TX direction with full duplex capabilities. An oscillator with a fixed frequency of 10MHz on the motherboard is used to generate an oscillator on the daughterboard's target frequency. This oscillator modulates the signal from the operating RX frequency to the IF and the TX streams from the IF into the operating TX frequency. For real valued streams the two streams can be connected to different antennas on the same operating frequency. Most daughterboards are also capable of filtering the received signal to suppress aliasing effects.

2.2 Interfaces

Depending on the USRP device there are different ways to access it from several software distributions.

2.2.1 USRP1

The USRP1 device can be interfaced with a driver provided by Ettus [6]. It is a C++ Interface which can be used to switch between real and complex samples, set the operating frequencies, amplification and send or receive the samples from the device's buffer. There are several software that utilize this interface to allow the use of USRPs:

GnuRadio
: A block based open source radio software. A USRP can be used like any other signal source/sink from within this software. All necessary configurations of the USRP are accessible from the USRP block.

GnuRadio companion
: A graphical user interface to GnuRadio that utilizes the GnuRadio interface to the USRP.

MATLAB
: A programmable numerical math software by MathWorks. An interface between MATLAB and the USRP1 was developed by Institute for Communications Engineering (LNT) at Technische Universität München (TUM). It can set the basic configuration options of the USRP and send and receive samples to and from a vector.

SimuLink
: A MATLAB powered block based signal processing environment. An interface to access the USRP as Simulink block is available by the Communications Engineering Lab (CEL) at Karlsruhe Institute of Technology (KIT) [7].

2.2.2 Next Generation USRPs

To support the various improvements of the new USRP generation Ettus provided a new driver, the USRP Hardware Driver (UHD) [6]. The UHD is a C++ interface and offers the same functionality as the old USRP1 driver with the additional functionality to

- Switch between oscillation and timing reference sources
- Configure time tagged transmission
- Read out time tags from received samples
- Configure channel and antenna assignment in MIMO configuration
- Transmit and receive a specific number of samples by start and end of burst flags
- Detection and notification on stream interruptions[4]

The UHD can also access USRP1 devices where the features from above are emulated in the UHD driver on the PC, except for the start and end of burst capability which is ignored by the driver if a USRP1 device is accessed. This way all USRP devices can be accessed by the UHD with the maximum possible functionality for each device.

Similar to the USRP1 driver there are software distributions that use the UHD to access the USRP devices.

GnuRadio
: Very similar to the implementation of the old USRP1 driver. Most of the new functionality has been adopted

GnuRadio companion
: Uses the same interface as GnuRadio. There is no time tagging available in GnuRadio companion.

MATLAB /SimuLink
: MathWorks provides the *Support Package for USRP© Hardware* to access the USRP from within MATLAB/SimuLink [8]. Although this package uses the UHD it can only be used for the two network based USRP2 and USRP N2x0 devices with limited functionality

2.2.3 Conclusion

Although there are two existing Interfaces to access the USRP devices from within MATLAB both of them lack some key features. The USRP1 interface from the LNT can only access USRP1 devices and cannot benefit from the new functionality of the UHD, while the MathWorks driver is artificially limited to network based devices and is also lacking time tagging of samples. Due to the closed source nature of the support package it is impossible to change this. At the Institute we have both USRP1 and USRP N210 devices and it is very desirable to use both types of devices with the same codebase. Also the possibility of doing accurate range measurements by time tagging is very appealing. These considerations led to the development of a new MATLAB interface which is described in the following Chapter.

[4]Such as under- or overflow of buffers and impossible time tags at reception or transmission

Chapter 3

MATLAB Interface Implementation

As motivated in the previous Chapters, this work will implement an interface between MATLAB and the UHD, which should be able to

- access all types of USRP devices with the same MATLAB code
- receive and transmit simultaneously from the same MATLAB code
- tag the time of incoming and outgoing samples

in a way that the full potential of the UHD can be used.

3.1 General Considerations

3.1.1 Single Thread

MATLAB programs run in a single thread, which means that all operations are done in sequence and no parallel operations are possible. Therefore it is not possible to directly interact with the channel as this would lead to interruptions in the data stream as shown in Figure 3.1a on the next page. Instead the interaction with the channel needs to be buffered in a memory that is faster than the channel data rate. For two data streams the memory needs to be at least twice as fast as the channel. If the memory is even faster there is some additional time in the thread to do calculations with the received data and generate the transmission samples as shown in Figure 3.1b.

The UHD has an internal buffer for USB/Ethernet operations that is sufficient for operating two data streams into a single thread. The C++ code in Appendix A.1 on page 45 demonstrates the parallel reception and transmission from a single thread.

3.1.2 Approaches

It is not possible to access the UHD library directly in MATLAB as there is no way of using C++ libraries and data structures from within MATLAB. However there are a few different approaches that have been taken and are discussed in the following.

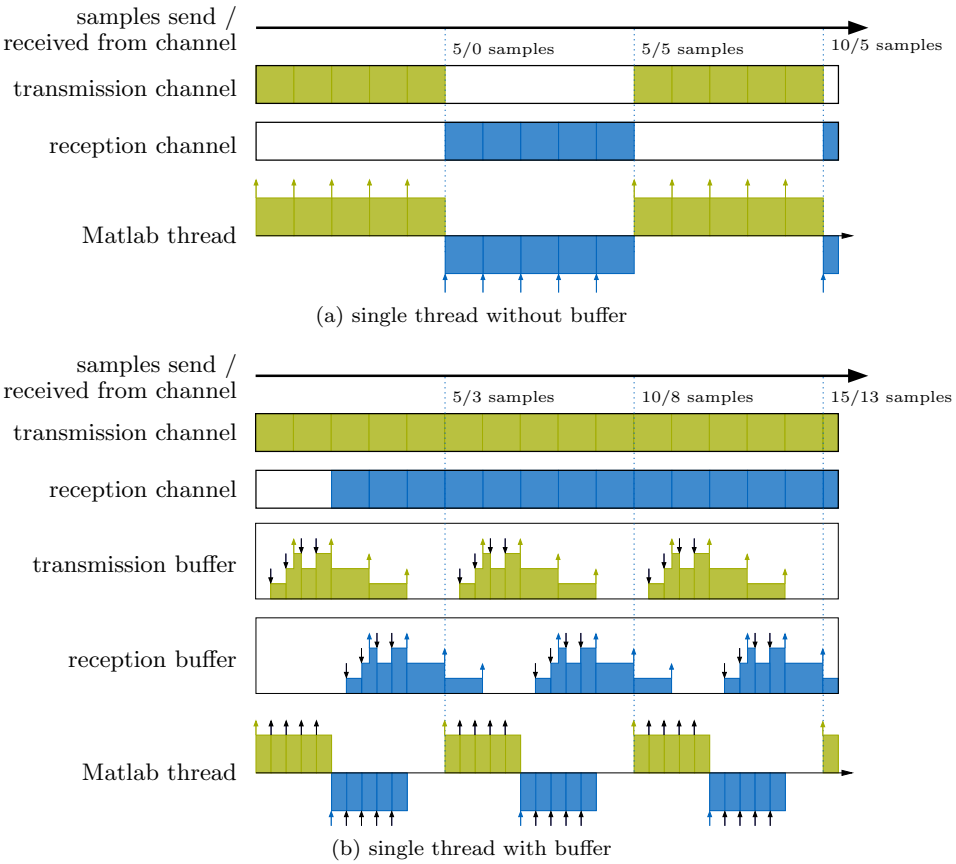


Figure 3.1: Channel access with and without buffering in between. Without buffering each channel is empty for half of the time for a single thread.

C/MEX Wrapper

MATLAB can call plain C libraries which can handle C++ code and it is possible to write special C++ code that can be executed from MATLAB[1]. Both the C library as well as the MEX code would then work with the C++ library internally and wrap the data into C/MATLAB data structures for further processing.

This approach is the most direct one compared to the two following approaches, as it directly interacts with the MATLAB data. In the beginning this close interaction led to errors with incompatibilities between the Boost library[2] used by the system and the ones used within MATLAB[3]. If these versions do not match, the attempt to call the UHD library from MATLAB either by a MEX file or by calling a wrapping C library loads the Boost version deployed with MATLAB instead of the system's one which results in a segmentation violation.

[1] MATLAB Executables (MEX) code that has to be compiled with special compiler options and only returns MATLAB compatible data structures

[2] A C++ library for standard operations in C++.

[3] MATLAB R2011b is linked to Boost version 1.44.0

Because of this problem the two approaches explained below have been considered. After MathWorks's technical support pointed us to the use of an incompatible boost version inside MATLAB[4] we were able to compile the UHD against the same Boost version as MATLAB which resolved the segmentation violations. After these issues are solved this approach is the most promising and thus is used for the implementation described in Section 3.2 on the next page.

Server/Client Separation

To have a clear separation between the MATLAB and the USRP side to work around segmentation violation problems, all UHD related tasks can be run as a stand alone program which exchanges configuration and samples with MATLAB over a local network[5].

As MATLAB has no own network access the fact that MATLAB can naively run Java code is exploited. Java has Remote Procedure Calls[6] (RPC) that enable the network separation with almost no implementation effort. Another advantage of Java code is that it is object oriented. This way multiple USRP devices can be easily addressed with multiple objects. Figure 3.2 shows the basic principle of this approach.

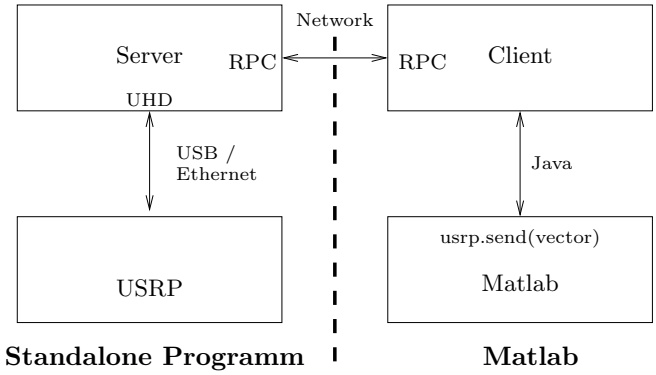


Figure 3.2: Separation of UHD and MATLAB address spaces by a server /client model.

The intended clear separation of MATLAB and UHD in this approach also has the drawback, that there is no common memory. All data between the two instances has to be copied before it can be used. This lead to very strong performance issues. Receiving data with a rate of up to one Megasample per second was possible without any transmission at the same time, but any higher data-rates or parallel transmission led to buffer overruns, because the data was not transferred fast enough over the interface.

[4]The use of Boost inside MATLAB is not documented at a prominent place

[5]Preferably over the loopback device that is a pure virtual network that has no data-rate limit through a physical connection

[6]That allow access from one Java program to data and Methods of another Java Program over the network without having to specify the interface first

Files

When it is not necessary to react to the received samples, e.g. no acknowledgements in the communication, the interface can be relaxed to sending from, and receiving into a file. Beside the much lower complexity this approach is reproducible by just reimporting the received samples. The sample C++ code in listing A.4 in Appendix A.2.2 on page 51 receives samples and writes them into a file as a sequence of double values while the sample code listing A.2 reads a file of subsequent complex double values and transmits them over the USRP. The MATLAB code in listing A.5 shows how to import this data into MATLAB and listing A.3 shows how to export it.

3.2 Interface Structure

The whole interface is implemented in a single mex function. This way all necessary data is available to all functions and only has to be kept persistent between subsequent calls of the same Mex function. To tell the Mex function what it should do during a specific call, a command has to be passed to the function. To access multiple USRPs in a non object orientated manner each USRP object is assigned with an unique integer index at initialization which is returned to MATLAB. By passing this index together with a command, the Mex function can search the right USRP object corresponding to the index and call the desired function of that object. Listing 3.1 shows the general structure of this function.

The entry point of the Mex function is at line 22. As MATLAB can clear Mex function out of memory at any time to ensure that the data is persistent between two calls a Mex function can lock itself. A locked Mex function is not cleared from memory unless it is unlocked again, or MATLAB exits. If the Mex function is not locked in line 25 it initializes itself by locking and registering a message handler, which is called by the UHD library to exchange messages. As the UHD uses multiple threads it has to be ensured that the message exchange between UHD and the single threaded Mex function is Thread safe. At each call to the message handler, in line four, a message is appended to a buffer. A mutex[7] ensures that either the Mex function or the UHD is accessing the buffer, but not both at the same time. At each call to the Mex function the message buffer is read and the messages are printed to the MATLAB console in line 32. Afterwards the input parameters are checked for existence and sanity. The interface expects at least one input string with the command to call. If there is at least one USRP object initialized, but no USRP index given the first USRP object (index 0) is used by default, otherwise the USRP index is expected before the command string. After the command string an optional third parameter can be given as parameter to the command, e.g. the frequency when setting the centre frequency of a USRP. In lines 43 to 67 all possible commands are checked in sequence and executed if all conditions for a command are fulfilled. After a command is executed the Mex function returns to MATLAB. When the program reaches line 69, no command has been executed and an according error message is printed out before returning to MATLAB.

[7] A programming concept which ensures mutual exclusion

Listing 3.1: Structure of the Mex file to achieve access to all functions on multiple USRPs from one Mex function.

```
static uhd::usrp::multi_usrp::sptr[] usrp;
static boost::mutex messages_mutex;

void message_handler(uhd::msg::type_t type, const std::
    string &msg){
  // ensure messages are not read out during this run
  boost::mutex::scoped_lock lock(messages_mutex);

  // ...
  // store message
  // ...
}

void process_messages(void) {
  // ensure messages are not stored during this run
  boost::mutex::scoped_lock lock(messages_mutex);

  // ...
  // print out stored messages
  // ...
}

extern "C" void mexFunction(int nlhs, mxArray *plhs[],  int
    nrhs, const mxArray *prhs[]) {

  // make sure we do not get cleared without our permission
  if(!mexIsLocked()) {
    // lock the file
    mexLock();
    // register message handler so errors can be printed out
    uhd::msg::register_handler(&message_handler);
  }
  // read out messages, that arrived during time in Matlab
  process_messages();

  // ...
  // check inputs
  // ...

  int uhd = getUInt(prhs[0]);
  std::string command = getString(prhs[1]);
  const mxArray *arg = prhs[2];

  // check individual commands
  if(command.compare("init") == 0) {
    std::string devarg = getString(arg);
    uhd::usrp::multi_usrp::sptr usrp_local = uhd::usrp::
        multi_usrp::make(devarg);
    // ...
    // store usrp_local at i-th position in usrp[]
    // ...
    plhs[0] = retInt(i);
```

```
50      return;
51    }
52    // only possible if the given USRP is initialized
53    if(command.compare("set_rx_frequency") == 0 and usrp[uhd])
          {
54      usrp[uhd]->set_rx_freq(getDouble(arg));
55      return;
56    }
57    // ...
58    if(command.compare("exit") == 0) {
59      // unlock Mex function so it can be cleared
60      mexUnlock();
61      return;
62    }
63    if(command.compare("flush") == 0) {
64      // do nothing, just print out new messages
65      // from the UHD, which has been done above
66      return;
67    }
68
69    // nothing to do yet :(
70    mexErrMsgIdAndTxt( "MATLAB:uhdInterface:noCommand", "
          Please specify a command");
71    return;
72  }
```

3.3 Usage

To use the interface, the compiled mex code needs to be accessible to MATLAB. On a new system it might be necessary to compile the code with the *make.m* MATLAB script. Afterwards the binary file *uhdinterface.mex****[8] needs to be added to MATLAB's path. The interface can then be used by a call to *uhdinterface* from within MATLAB. Section A.5 on page 57 lists every possible command of the interface.

3.3.1 Send and Receive Samples

The sample code in listing A.6 in Appendix A.3 on page 54 shows how to send and receive samples over the interface and demonstrates the usage of the most common commands of the interface. In lines two and three the two used USRPs are initialized and their index values are stored in the two variables *uhdsend* and *uhdrecv*. Instead of sending from one USRP to another one it is also possible to transmit and receive from the same USRP. In this case only one USRP object has to be initialized and both the transmission as well as the reception commands need the index of the same USRP object as argument. In lines six to eleven the parameters of the transmission are set. The frequencies are given in Hertz, the data rates in samples per second and the gains in the range from zero to 30 dB. Whenever parameters which are not possible with the USRPs are

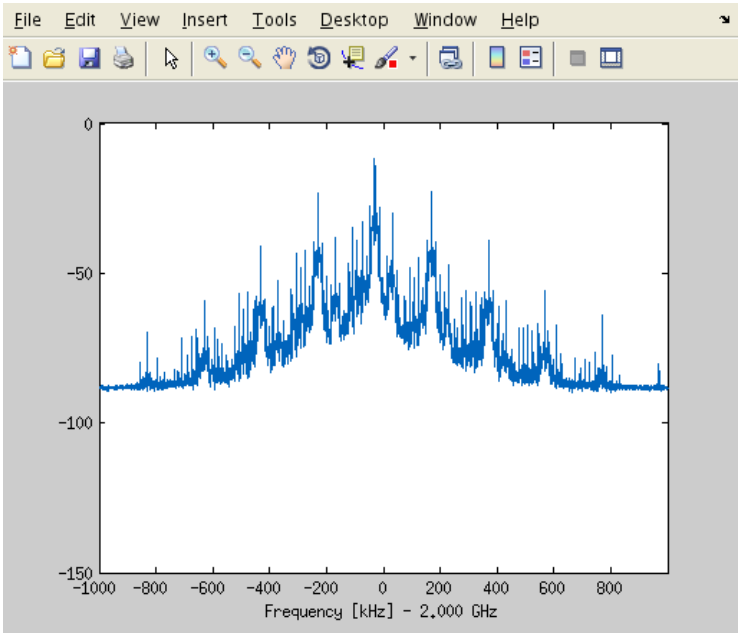
[8] The exact file suffix depends on the used operating system and the computer's architecture.

given the UHD notifies the user. To show the output of the last command in line eleven the output of the UHD is flushed to the MATLAB console in line 14. Afterwards the number of received and transmitted samples is set and the send and receive buffers are initialized. In line 27 the reception of samples is started and in line 28 the transmission of samples is started. Unless otherwise configured the transmission from the internal sample buffer starts 100ms after the call of this function. As the USRP needs some time during its transient state the first received samples are useless and are thrown away in line 32. If it is important to receive even the first transmitted bit it is possible to send zero symbols before the actual message to give the USRPs some time to tune in. The main reception loop starts at line 35. In this example a constant symbol is transmitted so it is not necessary to recalculate the send buffer. The previous send buffer is retransmitted in line 36. In line 37 the next set of samples is received and stored into the reception buffer. In line 45 and 46 the two initialized USRPs are closed again and if no other USRP device is initialized the interface unlocks itself. When using the same USRP device for transmission and reception this device has to be closed only once.

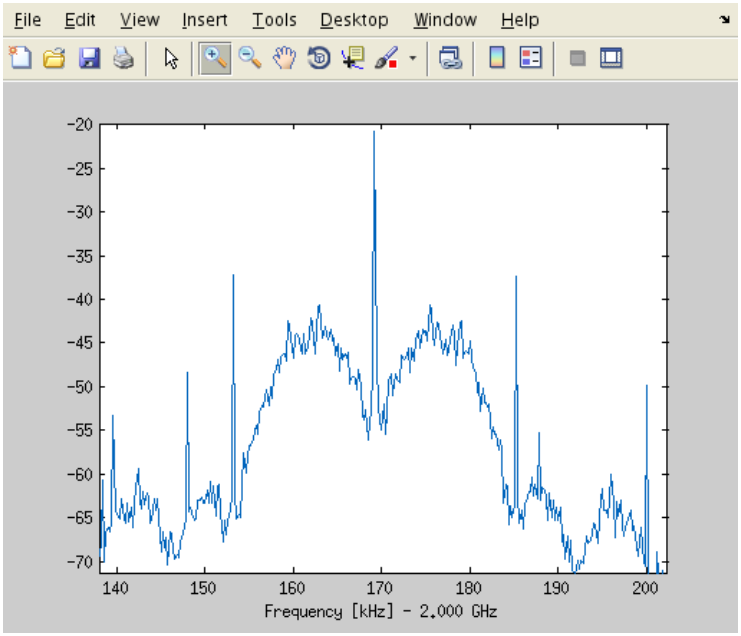
3.3.2 Spectrum Analyzer

The USRP interface can also be used to display the spectrum of the received samples. Figure 3.3 on the next page shows the spectrum of an ECSS[9] compatible ranging and telecomand signal generated by [9]. The according code is shown in listing A.7 on page 55 in the Appendix. While permanently receiving samples, their discrete Fourier transform is computed and averaged over a short time period. The resulting spectrum is plotted and updated in a configured interval. By using MATLAB for this task it is possible to view the spectrum through the same hardware as a sample code would do and it is possible to zoom and pan the spectrum like any other plot in MATLAB while it is still being updated.

[9] European Cooperation for Space Standardization



(a) Overall spectrum of an ECSS compatible ranging and telecommand signal.



(b) Zoomed in detail of a side peak of an ECSS compatible ranging and telecommand signal.

Figure 3.3: Screenshots of a spectrum analyzer implemented in MATLAB using the USRP interface

Chapter 4

Synchronization

When transmitting a signal it is in general modified by the channel before reception. This work will assume an additive white Gaussian noise (AWGN) channel. Beside the additive noise whose effect is shown in Figure 4.1a on the following page, the signal also needs its time to propagate through this channel which is seen as a delay at the receiver side and can lead to sampling at suboptimal time instances as shown in Figure 4.1b. Because of the Doppler shift due to relative movements between sender and receiver and because of imperfections in their local oscillators the signal is also rotated in the signal space by a constant frequency offset as shown in Figure 4.1c as well as by a constant rotation because of a phase offset in the local oscillators and the signal propagation as shown in Figure 4.1d.

All these effects lead to a modification of the send signal at the receiver side. Synchronization aims at estimating the frequency offset and channel delay, as well as the phase offset in a coherent receiver to remove or at least reduce the modifications before the detection of the sent data symbols.

4.1 System Model

Starting with the N data symbols $d[i] \in \mathcal{D} \subset \mathbb{C}$ the baseband transmission signal s is formed by the root-raised cosine transmit filter g_T

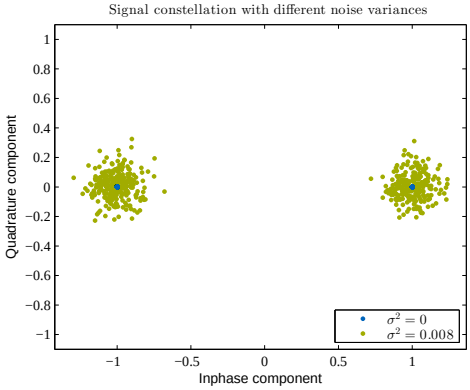
$$s(t) = \sum_{i=1}^{N} d[i] \cdot g_T\left(t - iT\right). \tag{4.1}$$

The baseband signal s is then modulated to the carrier frequency f_s with an arbitrary phase of the local oscillator ϕ_s resulting in the analytic representation of the real valued send passband signal s_{RF} at radio frequency

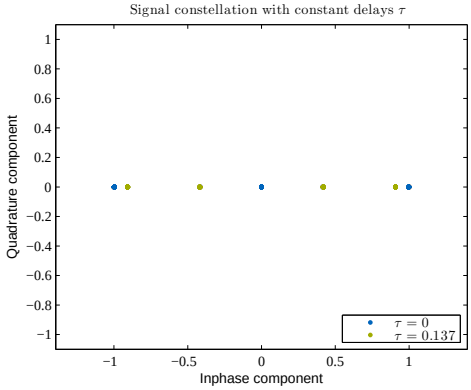
$$s_{RF}(t) = s(t) \cdot e^{j(2\pi f_s t + \phi_s)}. \tag{4.2}$$

After transmission over the AWGN channel the signal at the receiver is modelled with a transmission delay t' and a noise term w_{RF} with one-sided power spectral density N_0

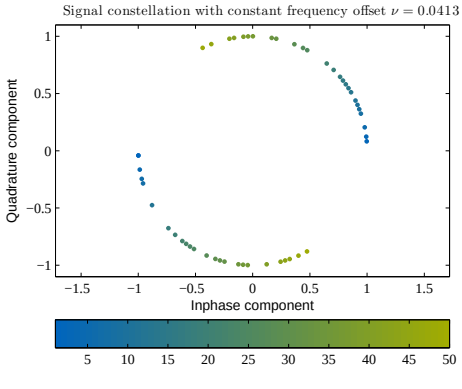
$$\begin{aligned} r_{RF}(t) &= s_{RF}(t - t') + w_{RF}(t) \\ &= s(t - t') \cdot e^{j\left(2\pi f_s \left(t - t'\right) + \phi_s\right)} + w_{RF}(t). \end{aligned} \tag{4.3}$$



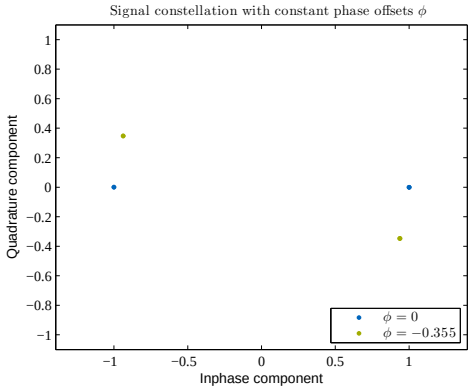
(a) Additive white Gaussian noise offsets the samples around the send data symbol



(b) Propagation delay leads to sampling at suboptimal time intervals



(c) Frequency offset continuously rotates the received samples



(d) Phase offset rotates the samples by a constant angle

Figure 4.1: Influences of the delayed AWGN channel on a BPSK signal

This received signal is then downmodulated with a frequency f_r similar to f_s. Because of differences in the local oscillators at sender and receiver side and because of the Doppler shift due to relative movement between sender and receiver f_s and f_r are in general not identical. ϕ_r is an arbitrary phase of the local oscillator at the receiver

$$r(t) = r_{RF}(t) \cdot e^{-j(2\pi f_r t + \phi_r)}. \tag{4.4}$$

With

$$w(t) := w_{RF}(t) \cdot e^{-j(2\pi f_r t + \phi_r)} \tag{4.5}$$

equation (4.4) becomes

$$r(t) = s(t - t') \cdot e^{j2\pi(f_s - f_r)t} \cdot e^{j\left(\phi_s + \phi_r - 2\pi f_s t'\right)} + w(t). \tag{4.6}$$

The received signal is filtered with a root raised cosine low pass filter with a one-sided bandwidth of $\frac{1}{2T_s}$ that limits the noise, but keeps the signal undis-

torted. Sampling the filtered signal with a frequency of $\frac{1}{T_s}$ leads to

$$r[k] = r(kT_s) = s(kT_s - t') \cdot e^{j2\pi(f_s - f_r)kT_s} \cdot e^{j\left(\phi_s + \phi_r - 2\pi f_s t'\right)} + \tilde{w}(kT_s) \quad (4.7)$$

with

$$\tilde{w}(kT_s) \propto \mathcal{N}_{\mathbb{C}}\left(0, \sigma^2 = \frac{N_0}{T_s}\right). \quad (4.8)$$

To come to a simpler notation and eliminate the units the following normalization is applied. The oversampling factor N_{os} is defined as the ratio between the symbol interval T and the sampling interval T_s

$$N_{os} := \frac{T}{T_s}. \quad (4.9)$$

The normalized delay $\mathcal{D}$ is defined as

$$\mathcal{D} := \frac{t'}{T} \quad (4.10)$$

which is divided into the integral delay η and the fractional delay τ. In this work, only the fractional delay is considered and the integral delay is assumed to be zero, as the integral delay has no impact on synchronization and only introduces a delay in the received symbols

$$\mathcal{D} = \eta + \tau \quad , \tau \in [-0.5, 0.5) \wedge \eta \in \mathbb{N}. \quad (4.11)$$

The frequency offset is normalized by the symbol interval and is considered to be limited

$$\nu = (f_s - f_r)\, T \quad , \nu \in [-0.5, 0.5). \quad (4.12)$$

The phase offset is collected in a single variable ϕ_0

$$\phi_0 = \phi_s + \phi_r - 2\pi f_s \mathcal{D} T \, , \phi_0 \in [-\pi, \pi). \quad (4.13)$$

Applying the normalizations from (4.9) to (4.13), equation (4.7) becomes

$$r[k] = s\Big(\,(k - N_{os}\tau)\, T_s\Big) \cdot e^{j2\pi \frac{\nu k}{N_{os}}} \cdot e^{j\phi} + \tilde{w}(kT_s). \quad (4.14)$$

4.2 Feedforward Estimator

In [1] a feedforward synchronization scheme is presented that will be used in this work. It is designed for burst mode transmission where the time between the reception of the first sample and the time of correct detection of the data plays an important role. Therefore a synchronization sequence of known data symbols is transmitted first, followed by a short, known start of frame (SoF) sequence and the message's data symbols as shown in Figure 4.2 on the next page.

The synchronization sequence consists of alternating ± 1 BPSK symbols. This sequence optimizes the Cramér-Rao lower bound for timing estimation, because the corresponding spectrum has only two peaks at the maximum frequencies. This sequence also decomposes the estimation of the three parameters timing τ, carrier frequency offset ν and the carrier phase ϕ into a one dimensional

Figure 4.2: Organization of synchronization sequence, start of frame sequence and the message's data symbols.

frequency search as shown in the next Section. While following the derivations from [1] instead of estimating the phase at the beginning of the synchronization sequence, this work will estimate the phase at the middle of the sequence which leads to a lower overall variance of the phase estimation. The algorithm operates on the received samples with an oversampling factor of two.

4.2.1 Joint Maximum Likelihood Estimation

The probability for a received sample $r[k]$ from equation (4.14) conditioned on fixed synchronization parameters $\tilde{\nu}, \tilde{\tau}$ and the phase at the middle of the synchronization sequence $\tilde{\phi} = \phi_0 + \pi\tilde{\nu}L$ can be expressed as

$$p(r[k] \mid s, \tilde{\nu}, \tilde{\tau}, \tilde{\phi}) = \frac{1}{\pi\sigma^2} \exp\left(-\frac{1}{\pi\sigma^2}\left|r[k] - s\left(\left(k - N_{os}\tau\right)T_s\right) e^{j\left(2\pi\frac{\nu k}{N_{os}}+\phi\right)}\right|^2\right). \tag{4.15}$$

When transmitting the synchronization sequence of alternating ± 1 the resulting filtered spectrum consists of two peaks at $\pm\frac{1}{2T}$ which corresponds to a cosine. Thus, the bandlimited continuous form of the send signal $s(t)$ can also be written as

$$s(t) = \sqrt{N_{os}} \cos\left(\frac{\pi t}{T}\right) \tag{4.16}$$

Because of the independence of the individual samples, the likelihood function for the synchronization sequence of length L given the synchronization parameters $\tilde{\nu}$, $\tilde{\tau}$ and $\tilde{\phi}$ can be expressed as the product of all $2L$ samples

$$p(r[k] \mid s, \tilde{\nu}, \tilde{\tau}, \tilde{\phi}) = \prod_{k=0}^{2L-1} \frac{1}{\pi\sigma^2} \exp\left(-\frac{1}{\pi\sigma^2}\left|r[k] - \sqrt{N_{os}} \cos\left(\frac{\pi k}{N_{os}} - \pi\tilde{\tau}\right) e^{j\left(2\pi\frac{\nu k}{N_{os}}+\phi\right)}\right|^2\right). \tag{4.17}$$

To fulfil the maximum likelihood principle, the parameters $\tilde{\nu}, \tilde{\tau}$ and $\tilde{\phi}$ are chosen such that the likelihood function (4.17) is maximized

$$(\hat{\nu}, \hat{\tau}, \hat{\phi}) = \arg\max_{\tilde{\nu}, \tilde{\tau}, \tilde{\phi}} \left\{p(r[k] \mid s, \tilde{\nu}, \tilde{\tau}, \tilde{\phi})\right\}. \tag{4.18}$$

Instead of maximizing p directly the logarithmic likelihood function $\Lambda = \ln(p)$ is maximized

$$\Lambda(r[k] \mid \tilde{\nu}, \tilde{\tau}, \tilde{\phi}) = -2L \ln\left(\pi\sigma^2\right) - \frac{1}{\pi\sigma^2} \sum_{k=0}^{2L-1} \left|r[k] - e^{j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)} \sqrt{N_{os}} \cos\left(\frac{\pi k}{N_{os}} - \pi\tilde{\tau}\right)\right|^2. \tag{4.19}$$

When leaving out constant parts, equation (4.19) transforms to

$$\begin{aligned}\psi(\tilde{\nu},\tilde{\tau},\tilde{\phi}) =& -\sum_{k=0}^{2L-1}\left|r[k]-e^{j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)}\cdot\sqrt{N_{os}}\cos\left(\frac{\pi k}{N_{os}}-\pi\tilde{\tau}\right)\right|^2\\ =& -\sum_{k=0}^{2L-1}\left(r[k]-e^{j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)}\cdot\sqrt{N_{os}}\cos\left(\frac{\pi k}{N_{os}}-\pi\tilde{\tau}\right)\right)\\ &\cdot\left(r^*[k]-e^{-j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)}\cdot\sqrt{N_{os}}\cos\left(\frac{\pi k}{N_{os}}-\pi\tilde{\tau}\right)\right)\\ =& -\sum_{k=0}^{2L-1}|r[k]|^2+N_{os}\cos^2\left(\frac{\pi k}{N_{os}}-\pi\tilde{\tau}\right)\\ &-2\sqrt{N_{os}}\Re\left\{r[k]\cdot e^{-j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)}\cdot\cos\left(\frac{\pi k}{N_{os}}-\pi\tilde{\tau}\right)\right\}. \qquad (4.20)\end{aligned}$$

Using the two times oversampled signal ($N_{os}=2$), equation (A.6.1) on page 60 shows, that the sum over the squared cosine term is independent of $\tilde{\nu}$, $\tilde{\tau}$ and $\tilde{\phi}$. Therefore, again omitting constant terms, the maximization of (4.20) is simplified as

$$\Psi(\tilde{\nu},\tilde{\tau},\tilde{\phi}) =\Re\left\{e^{-j(\tilde{\phi}-\pi L\tilde{\nu})}\sum_{k=0}^{2L-1}r[k]\cdot e^{-j\pi k\tilde{\nu}}\cdot\cos\left(\frac{\pi k}{2}-\pi\tilde{\tau}\right)\right\}.$$

Regrouping the samples into odd and even ones leads to

$$\begin{aligned}\Psi(\tilde{\nu},\tilde{\tau},\tilde{\phi}) =&\Re\left\{e^{-j(\tilde{\phi}-\pi L\tilde{\nu})}\cdot\left(\sum_{k=0}^{L-1}r[2k]\cdot e^{-j\pi 2k\tilde{\nu}}\cdot\cos\left(\frac{\pi 2k}{2}-\pi\tilde{\tau}\right)\right.\right.\\ &\left.\left.+\sum_{k=0}^{L-1}r[2k+1]\cdot e^{-j\pi(2k+1)\tilde{\nu}}\cdot\cos\left(\frac{\pi\,(2k+1)}{2}-\pi\tilde{\tau}\right)\right)\right\}\\ =&\Re\left\{e^{-j(\tilde{\phi}-\pi L\tilde{\nu})}\cdot\left(\sum_{k=0}^{L-1}r[2k]\cdot e^{-j\pi 2k\tilde{\nu}}\cdot\cos\left(\pi k-\pi\tilde{\tau}\right)\right.\right.\\ &\left.\left.+\sum_{k=0}^{L-1}r[2k+1]\cdot e^{-j\pi 2k\tilde{\nu}}\cdot e^{-j\pi\tilde{\nu}}\cdot\cos\left(\pi k+\frac{\pi}{2}-\pi\tilde{\tau}\right)\right)\right\}\\ =&\Re\left\{e^{-j(\tilde{\phi}-\pi L\tilde{\nu})}\cdot\left(\sum_{k=0}^{L-1}r[2k]\cdot e^{-j\pi 2k\tilde{\nu}}\cdot(-1)^k\cdot\cos\left(\pi\tilde{\tau}\right)\right.\right.\\ &\left.\left.+\sum_{k=0}^{L-1}r[2k+1]\cdot e^{-j\pi 2k\tilde{\nu}}\cdot e^{-j\pi\tilde{\nu}}\cdot(-1)^k\cdot\sin\left(\pi\tilde{\tau}\right)\right)\right\}. \qquad (4.21)\end{aligned}$$

Naming the splitted received samples in (4.21) as

$$Y_e(\tilde{\nu}) := \sum_{k=0}^{L-1}(-1)^k\cdot e^{-j2\pi k\tilde{\nu}}\cdot r[2k] \qquad (4.22)$$

$$Y_o(\tilde{\nu}) := \sum_{k=0}^{L-1}(-1)^k\cdot e^{-j2\pi k\tilde{\nu}}\cdot r[2k+1] \qquad (4.23)$$

reveals that these quantities can be computed very efficiently using the fast Fourier transformation (FFT). Also this leads to a simpler notation of Ψ

$$\Psi(\tilde{\nu},\tilde{\tau},\tilde{\phi}) = \Re\left\{e^{-j(\tilde{\phi}-\pi L\tilde{\nu})}\left[Y_e(\tilde{\nu})\cdot\cos(\pi\tilde{\tau}) + e^{-j\pi\tilde{\nu}}\cdot Y_o(\tilde{\nu})\cdot\sin(\pi\tilde{\tau})\right]\right\}. \quad (4.24)$$

Defining the inner part as $\mathcal{Z}(\tilde{\nu},\tilde{\tau})$

$$\mathcal{Z}(\tilde{\nu},\tilde{\tau}) := Y_e(\tilde{\nu})\cdot\cos(\pi\tilde{\tau}) + e^{-j\pi\tilde{\nu}}\cdot Y_o(\tilde{\nu})\cdot\sin(\pi\tilde{\tau}) \quad (4.25)$$

equation (4.24) can be rewritten as

$$\Psi(\tilde{\nu},\tilde{\tau},\tilde{\phi}) = |\mathcal{Z}(\tilde{\nu},\tilde{\tau})|\cdot\cos\left(\measuredangle\mathcal{Z}(\tilde{\nu},\tilde{\tau}) - \tilde{\phi} + \pi L\tilde{\nu}\right). \quad (4.26)$$

From equation (4.26) the optimum $\tilde{\phi}$ for a fixed $\tilde{\nu}$ and $\tilde{\tau}$ can be observed when the cosine term becomes one, which leads to

$$\hat{\phi} = \measuredangle\mathcal{Z}(\tilde{\nu},\tilde{\tau}) + \pi L\tilde{\nu}. \quad (4.27)$$

Inserting the optimal phase estimate from above, equation (4.26) leads to

$$\Psi(\tilde{\nu},\tilde{\tau},\hat{\phi}) = |\mathcal{Z}(\tilde{\nu},\tilde{\tau})| \quad (4.28)$$

which has the same maximum as

$$\begin{aligned}\Gamma(\tilde{\nu},\tilde{\tau}) =& 2\cdot\Psi^2(\tilde{\nu},\tilde{\tau},\hat{\phi})\\ =& 2\left|\mathcal{Z}(\tilde{\nu},\tilde{\tau})\right|^2.\end{aligned}$$

Equation (A.6.2) on page 61 shows the equality to

$$\Gamma(\tilde{\nu},\tilde{\tau}) = |Y_e(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^2 + \Re\left\{e^{-j2\pi\tilde{\tau}}\mathcal{A}(\tilde{\nu})\right\} \quad (4.29)$$

with

$$\mathcal{A}(\tilde{\nu}) := |Y_e(\tilde{\nu})|^2 - |Y_o(\tilde{\nu})|^2 + j2\Re\left\{e^{j\pi\tilde{\nu}}Y_e(\tilde{\nu})Y_o^*(\tilde{\nu})\right\}. \quad (4.30)$$

Equation (A.6.3) on page 62 shows the argument phase notation of $\mathcal{A}$ as

$$\mathcal{A}(\tilde{\nu}) = \left|Y_e^2(\tilde{\nu}) + e^{-j2\pi\tilde{\nu}}Y_o^2(\tilde{\nu})\right| e^{\measuredangle\mathcal{A}(\tilde{\nu})}. \quad (4.31)$$

Inserting (4.31) into (4.29) leads to

$$\begin{aligned}\Gamma(\tilde{\nu},\tilde{\tau}) &= |Y_e(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^2 + |\mathcal{A}(\tilde{\nu})|\cos\left(\measuredangle\mathcal{A}(\tilde{\nu}) - 2\pi\tilde{\tau}\right)\\ &= |Y_e(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^2 + \left|Y_e^2(\tilde{\nu}) + e^{-j2\pi\tilde{\nu}}Y_o^2(\tilde{\nu})\right|\cos\left(\measuredangle\mathcal{A}(\tilde{\nu}) - 2\pi\tilde{\tau}\right)\end{aligned} \quad (4.32)$$

which is maximized for fixed $\tilde{\nu}$ when the cos term equals to one, which leads to

$$\hat{\tau} = \frac{1}{2\pi}\measuredangle\mathcal{A}(\tilde{\nu}). \quad (4.33)$$

When inserting (4.33), equation (4.32) becomes

$$\begin{aligned}P(\tilde{\nu}) =& \Gamma(\tilde{\nu},\hat{\tau})\\ =& |Y_e(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^2 + \left|Y_e^2(\tilde{\nu}) + e^{-j2\pi\tilde{\nu}}Y_o^2(\tilde{\nu})\right|.\end{aligned} \quad (4.34)$$

Finally the maximum of the likelihood function can be obtained by a one dimensional search over the frequency offset, which is described in the next Section, with

$$\boxed{\hat{\nu} = \arg\max_{\tilde{\nu}} \{P(\tilde{\nu})\}.} \tag{4.35}$$

With the result of the frequency offset search the other two synchronization parameters can be calculated from (4.33) with $\mathcal{A}(\hat{\nu})$ from (4.30)

$$\boxed{\hat{\tau} = \frac{1}{2\pi} \measuredangle \mathcal{A}(\hat{\nu})} \tag{4.36}$$

and from (4.27) with $\mathcal{Z}(\hat{\nu}, \hat{\tau})$ from (4.25) with

$$\boxed{\hat{\phi} = \measuredangle \mathcal{Z}(\hat{\nu}, \hat{\tau}) + \pi L \hat{\nu}.} \tag{4.37}$$

4.2.2 Frequency Search

As showed in the previous Section the frequency estimation is achieved by a maximum search over $P(\tilde{\nu})$ which is a combination of the two spectra $Y_e(\tilde{\nu})$ and $Y_o(\tilde{\nu})$, which can be computed using the FFT algorithm. The spectral resolution of P is given by the spectral resolution of Y_e and Y_o, which is given by the length of the FFT input y_e and y_o. To increase the spectral resolution these input sequences can be enlarged by adding additional zeros at the end. This technique is called zero padding, where a pruning factor K is defined as

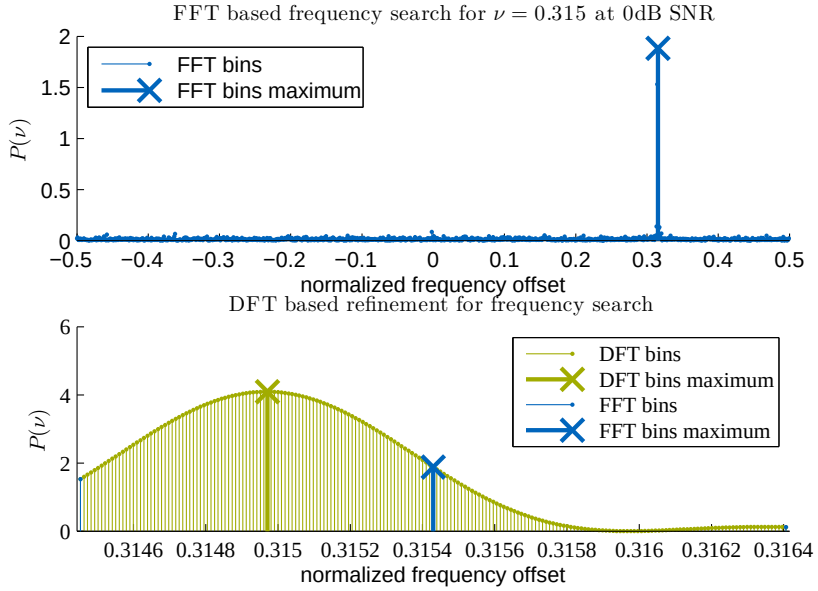


Figure 4.3: Frequency Search principle with a coarse search using the FFT algorithm and a refinement using the DFT in a second step.

the ratio between the length of the zero padded and the original sequence. By

doubleing the amount of input samples the resolution is doubled, however the computational complexity is also more than doubled. The drawback of zero padding in this work is that the frequency resolution over the whole spectrum is increased while only a high spectral resolution around the maximum is beneficial for the maximum frequency search. Thus a lot of computational power to compute large FFTs is needed as well as the memory to store the refined FFT for the small benefit of having the frequency refinement also around the maximum.

To overcome this drawback the frequency search is done in more than one step. First the spectrum of P is computed using the FFT without zero padding. From this spectrum a coarse frequency estimation $\hat{\nu}_c$ is determined. With knowledge of the rough position of the maximum and the assumption that the spectrum is locally concave around the maximum, the spectrum can be refined around the maximum using the discrete Fourier transform (DFT). The frequency range around the maximum is divided into 1000 equidistant frequency bins and the DFT of y_e and y_o for these frequency bins is computed using a DFT matrix. The fine frequency estimate $\hat{\nu}_f$ can be found by recomputing P with the new spectra and taking the maximum as shown in Figure 4.3 on the preceding page. To further increase the spectral resolution the computed $\hat{\nu}_f$ can be considered as another coarse estimation and the previous step can be repeated.

For a productive implementation of the frequency search, instead of computing a DFT matrix and performing the matrix multiplication on every repetition, more performant methods like the golden section search with the Goertzel algorithm to compute the single spectral points should be used.

4.2.3 Simulation Results

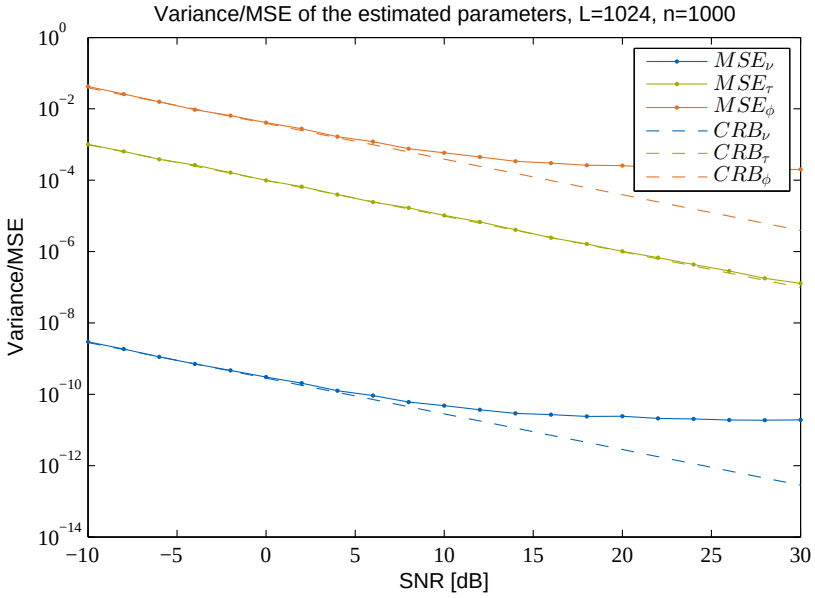
To evaluate the performance of the synchronization scheme from above, the mean squared error (MSE) between uniformly distributed random synchronization parameters and their estimates is computed for different signal to noise ratios (SNRs). These MSEs are compared to the Cramér-Rao bounds (CRBs) for frequency, phase and timing estimation. The CRB is the theoretical lower bound of the variance of an unbiased estimator and serves as a reference for a perfect synchronization scheme. The CRBs are computed as [1]:

$$\mathrm{CRB}_\nu = \frac{1}{\mathrm{SNR}} \cdot \frac{12}{\pi^2 L \left[4L^2 - 4 + 3\sin^2(2\pi\tau)\right]} \tag{4.38}$$

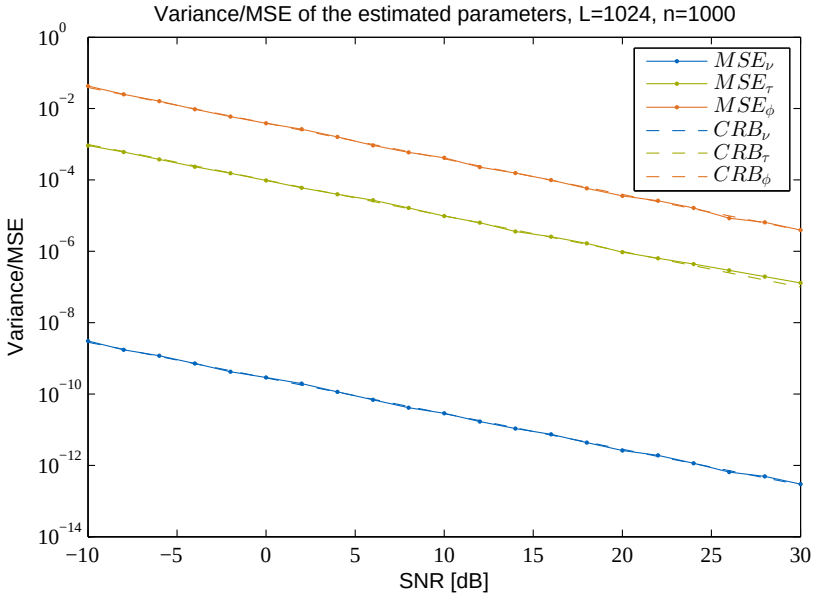
$$\mathrm{CRB}_\tau = \frac{1}{\mathrm{SNR}} \cdot \frac{1}{\pi^2 L} \tag{4.39}$$

$$\mathrm{CRB}_\phi = \frac{1}{\mathrm{SNR}} \cdot \frac{2\,(2L-1)\,[4L - 1 - 3\cos(\tau)]}{L\left[4L^2 - 4 + 3\sin^2(2\pi\tau)\right]} \tag{4.40}$$

The simulated results of the joint MLE are plotted in Figure 4.4 on the next page. For each SNR value the MSE is computed for 1000 frequency, timing and phase offsets with an observation length of 1024. The frequency search is done in two different ways. In Figure 4.4a a FFT with a zero padding pruning factor of 64 is used for coarse frequency estimation. In Figure 4.4b the DFT refinement described in the previous Section is used with one repetition. In Figure 4.4a the MSEs hit the CRBs for SNRs up to 8dB. For higher SNRs the frequency



(a) zero padded FFT with pruning factor of 64



(b) DFT refinement repeated once

Figure 4.4: Mean squared errors of the joint maximum likelihood estimator compared to the CRBs for different frequency search methods

estimation hits an error floor because of the coarse frequency search. As the phase estimation is strongly dependent on the correct frequency estimation it also exhibits an error floor, while even the very inaccurate frequency estimations for high SNRs are good enough to compute an accurate timing estimation. Thus the MSE for the timing estimation hits the corresponding CRB over the whole simulated SNR range. In Figure 4.4b the frequency estimation is computed with a higher accuracy. In this case the MSEs of all three parameters hit the CRBs in the complete simulated SNR range, which means that with the same parameters any other estimator can only be as good as the discussed joint MLE estimator in terms of estimation accuracy.

4.2.4 Synchronization Sequence Detection

Before starting the maximum likelihood estimation described above on measured data, the receiver first has to detect that the synchronization sequence is received. Therefore the receiver iterates over small blocks of the received samples (e.g. 64 samples) and checks this sequence for the presence of the synchronization data. Equation (4.34) seems suitable for the detection as a synchronization sequence creates one distinct peak in $P(\tilde{\nu})$ while noise just raises the average value of $P(\tilde{\nu})$. A signal to noise ratio can be calculated by computing the quotient of $\max_{\tilde{\nu}} \{P(\tilde{\nu})\}$ as a measure of signal and noise energy and $\mathcal{E}[P(\tilde{\nu})]$ as a measure for the noise energy. Once the SNR raises over a threshold value the presence of the synchronization sequence is assumed and the joint ML estimation is started on the following samples.

Unfortunately the USRP devices transmit a strong carrier signal even when transmitting zero symbols. This carrier signal also creates a distinct peak in $P(\tilde{\nu})$ and thus mimics the presence of a synchronization sequence even when the transmission of this sequence has not yet been started. Although from a subsequent view of the SNR estimates an increased SNR is observed when the synchronization sequence is transmitted due to the higher energy of the synchronization data over the carrier, there is no absolute value at which the presence of a synchronization sequence can be assumed, when not knowing the attenuation of the channel. To reliably decide between the presence or absence of a synchronization sequence without knowing the channel attenuation, hypothesis testing instead of SNR estimation is used. The following three hypotheses are made:

H1 A synchronization sequence is present

H2 A carrier signal, but no synchronization sequence is present

H3 The received samples do not contain a signal, but only noise

The probability of each of these hypotheses under the condition of the received samples is evaluated and compared to the other ones. Instead of directly calculating the probability p of each hypothesis a monotonic function $\tilde{\Psi}$ is used

instead, with

$$\begin{aligned}\tilde{\Psi}(p) =& \frac{1}{2}\left(\psi(p) + L + \sum_{k=0}^{2L-1} |r[k]|^2\right) \\ =& \pi\sigma^2 \ln(p) + 2\pi\sigma^2 L \ln\left(2\pi\sigma^2\right) + \frac{1}{2}\left(L + \sum_{k=0}^{2L-1} |r[k]|^2\right)\end{aligned} \tag{4.41}$$

This definition of $\tilde{\Psi}$ comes from the derivation of the maximum likelihood estimator with $\tilde{\Psi} = \Psi - \frac{L}{2}$ from equations (4.17) to (4.21) with ψ being the negative sum of the Euclidean distances between the received samples and the expected data symbols in the signal space. The likelihood of hypothesis one can be taken from (4.28) as

$$\Psi_{\mathrm{H1}}(\hat{\nu}, \hat{\tau}) = |\mathcal{Z}(\hat{\nu}, \hat{\tau})| - \frac{L}{2} \tag{4.42}$$

Ψ_{H2} is computed in (A.7.1) on page 62

$$\Psi_{\mathrm{H2}}(\hat{\nu}) = \left| \sum_{k=0}^{2L-1} r[k] \cdot e^{-j\pi k\hat{\nu}} \right| - \frac{L}{2} \tag{4.43}$$

For hypothesis three no data symbols are expected, so $-\psi$ is just the norm of the received samples and thus

$$\Psi_{\mathrm{H3}} = \frac{1}{2}\left(-\sum_{k=0}^{2L-1} |r[k]|^2 + \sum_{k=0}^{2L-1} |r[k]|^2\right) = 0 \tag{4.44}$$

To avoid false detection during transitions between two hypotheses the presence of a synchronization sequence is only assumed when the likelihood of hypothesis one is larger than the likelihood of the two other hypotheses plus certain thresholds $\kappa^- = 0.05$ and $\kappa^+ = 1.05$, which lead to the following decision criteria

$$\Psi_{\mathrm{H1}} > \kappa^- > 0 = \Psi_{\mathrm{H3}} \quad \wedge \quad \Psi_{\mathrm{H1}} > \kappa^+ \cdot \Psi_{\mathrm{H2}}. \tag{4.45}$$

Figure 4.5 on the following page shows the calculated likelihoods for two measured USRP signals which start transmitting the synchronization sequence at block number 25. It can be observed that the detection criteria of equation (4.45) will always lead to a clear and correct detection in the plotted cases.

4.2.5 Measured Results

To test the performance of the synchronization scheme over a real channel the synchronization sequence is transmitted from one USRP device to another. For high SNR cases the data is transmitted over a cable, for lower SNR it is transmitted over antennas with low transmit power. As the estimation errors of the synchronization parameters are hard to measure, the received and corrected signal constellation is inspected instead, to see if a clear detection is possible. Figure 4.6 on page 33 shows the signal constellation for two data sequences, once after reception and once after synchronization and decimation. It can be

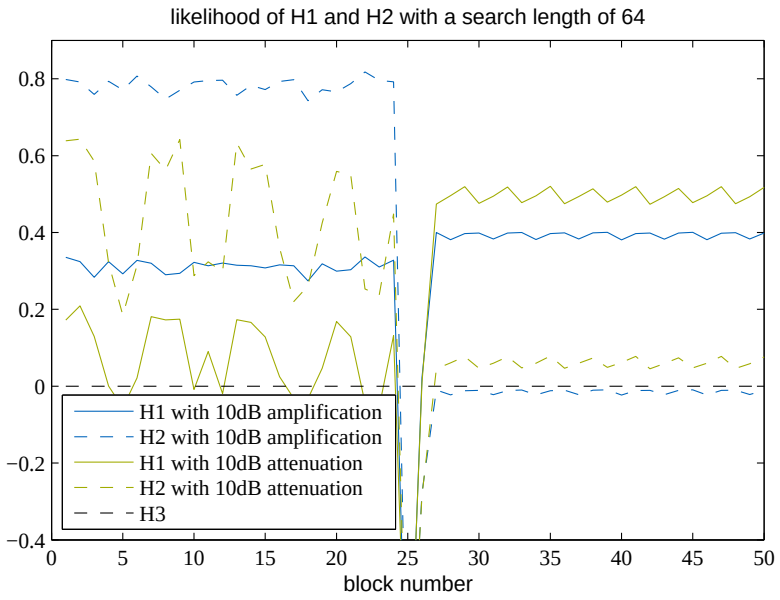
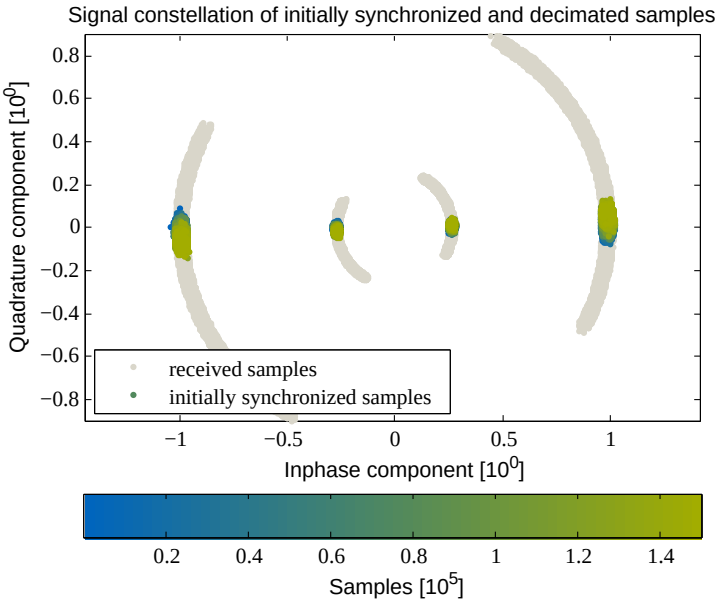


Figure 4.5: Hypothesis testing over subsequent blocks of samples, while switching from transmitting only a carrier to the synchronization symbols at block number 25.

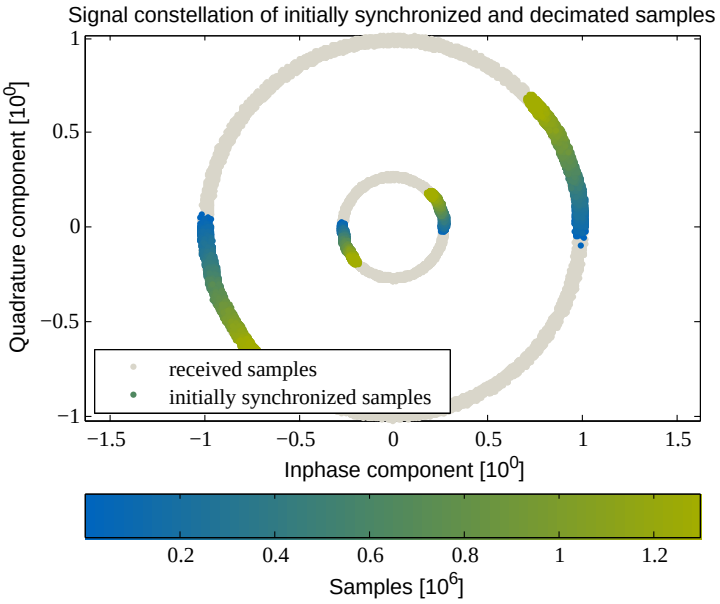
observed that the received samples in Figures 4.6a and 4.6b are not suited for detection of the transmitted BPSK symbols, while the synchronized samples in Figure 4.6a allow a clear separation of the transmitted symbols. In Figure 4.6a and especially in Figure 4.6b it can be observed that there is still an error in the frequency estimation which leads to slightly rotating symbols in the signal space. While this rotation can be neglected for the small number of samples in Figure 4.6a, the large amount of samples in Figure 4.6b leads to an overall rotation that will make it impossible to detect the correct symbols. In the next Section, tracking of the received samples is discussed which will correct these remaining rotations and keep track of the timing estimation.

4.3 Tracking

As discussed in the previous Section, even small errors in the initial synchronization can lead to detection problems when receiving over longer time instances. But also the parameters themselves can vary over time due to movements as well as clock drifts between sender and receiver. To compensate for this changes the receiver does not only estimates the synchronization parameters once, but keeps track of them and continuously corrects them during further reception. In this work the delay and the phase is tracked with two closed loops. Both loops work with the samples after the matched filter y, while the initial estimation of the phase and frequency offset is corrected before the matched filter, the timing error is preserved and initializes the timing error loop.



(a) Short sequence of samples



(b) Large sequence of samples

Figure 4.6: Measured synchronization results compared to the received samples

4.3.1 Delay Tracking

Timing Error Detector

For tracking the delay the Gardner timing error detector (TED) [3, 2] is used. In the Gardner TED the difference between the two interpolated sample values $\hat{y}$ at the estimated maximum is multiplied with the value at the estimated zero crossing using the samples y as input for the interpolation:

$$e_\tau(k) = \Big[\hat{y}\big([k-1]T + \hat{\tau}_k\big) - \hat{y}\big(kT + \hat{\tau}_k\big)\Big] \cdot \hat{y}\big([k-0.5]T + \hat{\tau}_{k-1}\big) \tag{4.46}$$

Figure 4.7 on the facing page shows the position of the interpolated samples for various situations. In the upper plot of Figure 4.7 at $k = 3$ the estimated $\hat{\tau}$ is too small. The difference of the two outer samples is a large positive number, multiplied by the positive value of the middle sample. This results in a positive error signal which will increase the delay estimate. If the delay is already correctly estimated the middle sample will be located at the zero crossing and so the error signal will be zero as at $k = 5$. If a raising instead of a falling zero crossing is considered the difference of the two outer samples will be negative and thus inverse the sign of the middle sample, as shown at $k = 6$. When there are no symbol transitions it is not possible to do timing estimations, but as shown in the lower plot of Figure 4.7 the difference of the two outer samples is very small in this case and thus the resulting error signal is very small or even zero for an arbitrary delay.

Filter

The generated error signal is filtered by a first order filter to generate the timing estimate $\hat{\tau}$

$$\hat{\tau}_k = \hat{\tau}_{k-1} + \gamma \cdot e_\tau(k) \tag{4.47}$$

The choice of the proportional constant γ affects both the settling time as well as the bandwidth of the loop. While a higher γ results to a lower settling time, the higher loop bandwidth collects more noise and thus the accuracy of the loop decreases. The relation between the loop bandwidth B_L and γ is [3, p. 214]

$$B_L T = \frac{\gamma A}{2(2 - \gamma A)} \approx \frac{\gamma A}{4} \tag{4.48}$$

The constant A is the slope of the S-curve[1] at the origin and is two for the Gardner timing error detector. The loop bandwidth can be related to an equal observation length of a feedforward estimator with [3, p. 216]

$$L_{eq} = \frac{1}{2B_L T} \tag{4.49}$$

To achieve a similar tracking performance as the feedforward estimator of section 4.2.1 on page 24 it's observation length L is used to compute γ.

$$\gamma = \frac{2}{LA} = \frac{1}{L} \tag{4.50}$$

[1] The S-curve describes the expectation of the error signal given a certain error.

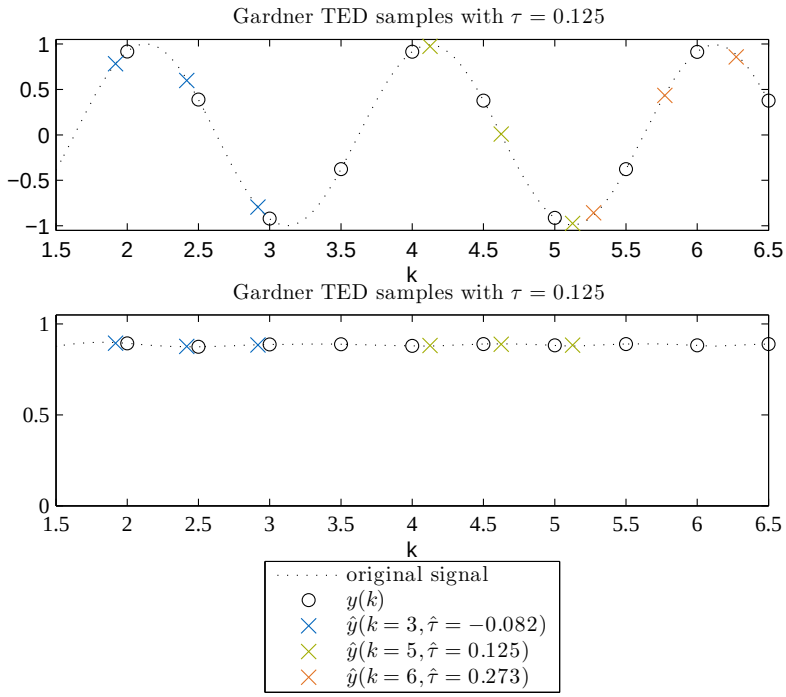


Figure 4.7: Interpolated samples for the Gardner timing error detector for alternating (top) and constant (bottom) symbols.

4.3.2 Phase Tracking

Phase Error Detector

The phase of an M-PSK modulated signal is dependant on the send symbols which are unknown during synchronization. Therefore, the modulation has to be removed from the received signal to get an estimate of the unmodulated carrier phase. In this work the modulation is removed by a modified M-power algorithm, the Viterbi & Viterbi phase error detector [3, 4].

The idea behind the M-power algorithm is that having a modulation alphabet of M-PSK symbols, the M^{th} powers of these symbols will all lie at the same place in the signal space. The phase of the carrier can then be estimated from the interpolated received sample $\hat{y}_k$ by

$$\hat{\phi}_k = \frac{\measuredangle \left\{ \hat{y}_k^M \right\}}{M} \tag{4.51}$$

The received sample with the removed modulation $\hat{y}_k^-$ can then be calculated

$$\hat{y}_k^- = |\hat{y}_k| \cdot e^{-j\hat{\phi}} \tag{4.52}$$

Figure 4.8 shows this effect for a QPSK (M=4) modulation. The estimated phase is affected by noise. To generate an error signal for a tracking loop the estimated phase is usually weighted with a factor depending on the quality of

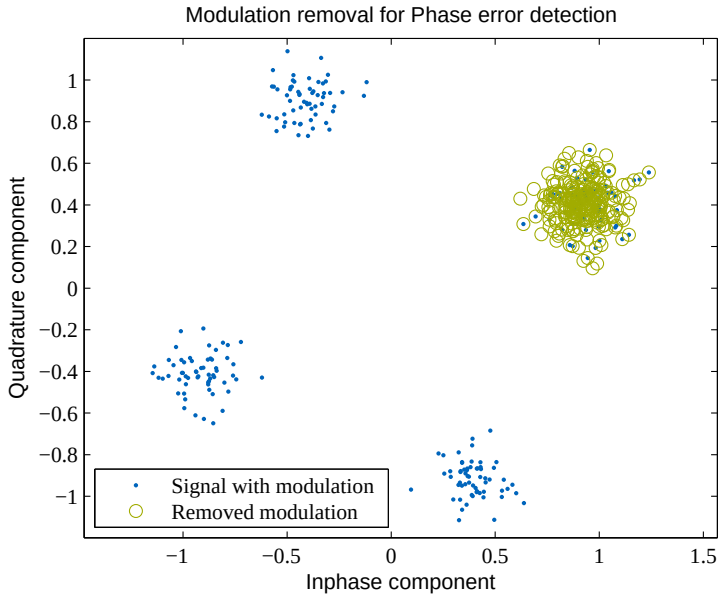


Figure 4.8: Basic principle of modulation removal using the M-power algorithm.

the sample. In the M-power algorithm the phase is weighted by the M^{th} power of the samples absolute value, thus samples with a higher absolute value, which have a higher energy and are thus less affected by noise, are weighted stronger. Taking the M^{th} power as weighting means that stronger signals are much more weighted than weaker signals. The Viterbi & Viterbi algorithm uses a smaller power as weighting to preserve the higher weighting of stronger samples, but does not degrade the weaker samples that strong.

Filter

Again a loop filter is used to smooth out the noise impact on the phase estimate. As the phase can change in a linear manner due to a frequency offset, a proportional regulator will produce a residual error when following such a linear input. To also compensate the frequency offset an additional integral part is used to form a second order loop. The filtered phase estimate is controlled by the two constants ρ and γ.

$$\hat{\phi}_k = \hat{\phi}_{k-1} + \xi(k) \tag{4.53}$$

$$\xi(k) = \xi(k-1) + \gamma \cdot ((1+\rho) \cdot e_\phi(k) - e_\phi(k-1)) \tag{4.54}$$

The relation between the two parameters and the loop bandwidth is [3, p. 223]

$$B_L T = \frac{2\rho + \gamma A(2+\rho)}{2(4 - \gamma A[2+\rho])} \tag{4.55}$$

and a damping factor can be defined as

$$\zeta = \frac{(1+\rho)\sqrt{\gamma A}}{2\sqrt{\rho}} \tag{4.56}$$

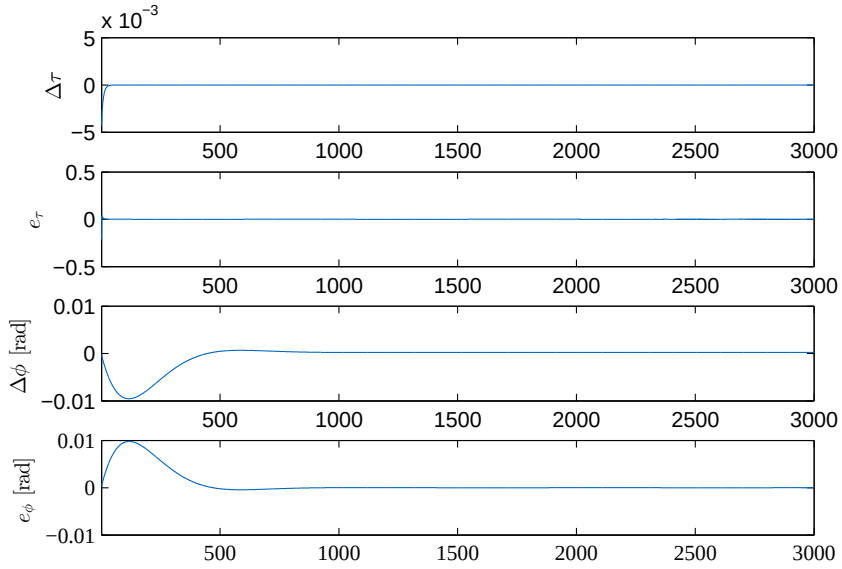
The slope of the S-curve at the origin is one and a damping factor $\zeta = \frac{1}{\sqrt{2}}$ is chosen. Choosing $B_L T$ from the observation length L of the feedforward estimator and using equation (4.49) the two equations can be solved for the two unknown parameters γ and ρ.

4.3.3 Simulation Results

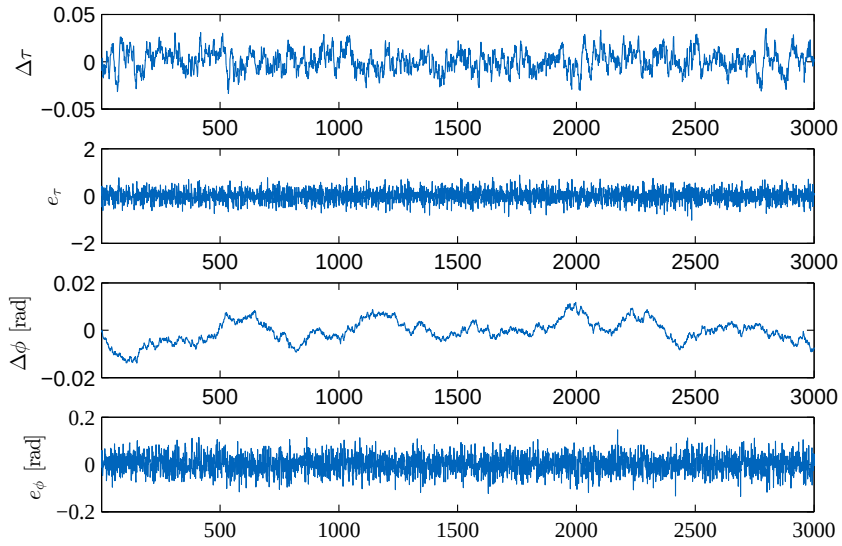
To ensure the proper function of the tracking a generated sample sequence with known phase offset and delay is generated and tracked. The resulting error between estimated and real parameter as well as the corresponding error signals from Sections 4.3.1 and 4.3.2 for a noise free case are plotted in Figure 4.9a on the following page and with noise in Figure 4.9b.

4.3.4 Measured Results

The results from tracking the initially synchronized samples from Section 4.2.5 on page 31 are shown in Figure 4.10 on page 39. It can be seen that the already well synchronized samples from Figure 4.10a are rotated even a bit more to the original BPSK symbols, but are almost untouched. In comparison the initially synchronized samples from Figure 4.10b are as well concentrated around the two BPSK symbols by the tracking loop and a detection is now possible. As the tracking operates on single samples the number of samples does not affect the accuracy of the estimation and even for much longer sequences the separation between the M-PSK samples will be possible.

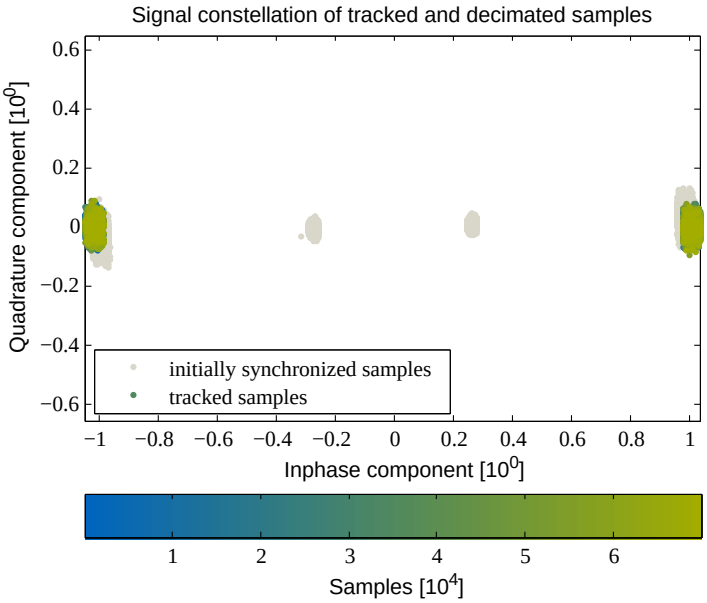


(a) without noise

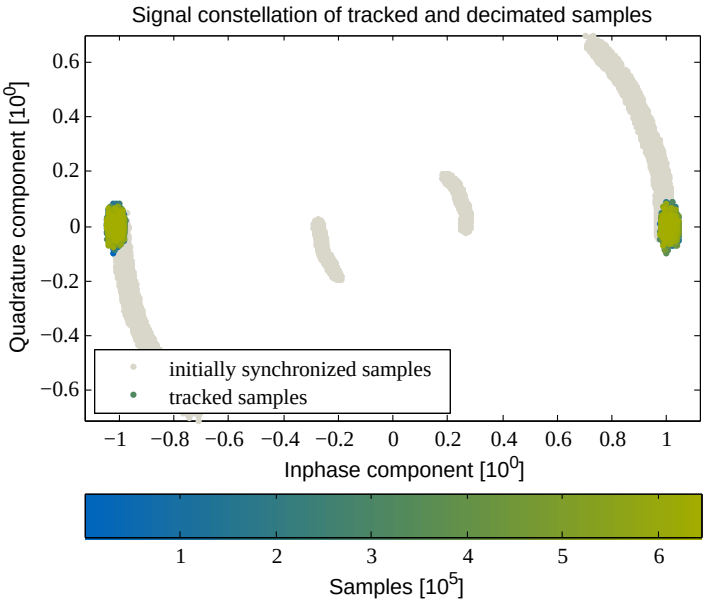


(b) with noise ($\sigma^2 = 0.0316 \Leftrightarrow$ SNR $= 15$dB)

Figure 4.9: Simulated tracking results



(a) Short sequence of samples



(b) Large sequence of Samples

Figure 4.10: Measured tracking results compared to the initially synchronized samples

4.4 Start of Frame Detection

After the successful bit synchronization of the previous sections there are still two open points. Because of the π-ambiguity of the phase of the BPSK synchronization sequence it is unclear whether a received $+1$ corresponds to a one or a zero and it has to be detected which bit is the first bit that belongs to the send data. These two issues can be resolved by sending a known start of frame sequence and correlate the received bits with this sequence. A Barker sequence of length 13 is used as start of frame sequence either directly or the Kronecker product of two barker sequences to get a sequence of length 13^2 for higher robustness. Barker sequences have an autocorrelation function with a off-peak autocorrelation of at most one. The maximum crosscorrelation of the synchronization sequence with the 13 bit Barker code is five, while the autocorrelation of two aligned Barker sequences is 13. The received samples are BPSK

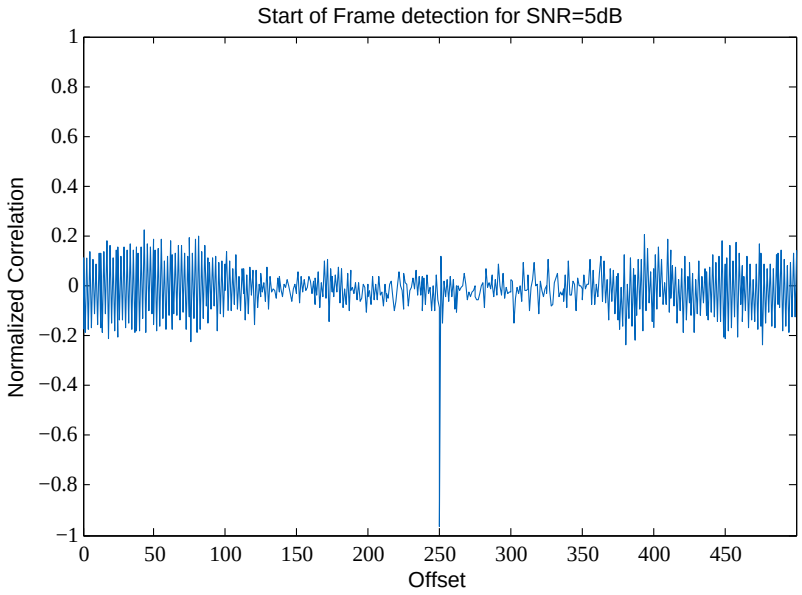


Figure 4.11: Simulated correlation between the demodulated samples and the start of frame sequence

demodulated by taking the real value of them and are then correlated with a zero padded start of frame sequence. Figure 4.11 shows the normalized correlation of such a sequence. A clear and distinctive peak can be observed at an offset of 250. In this case the correlation at the detected maximum is negative, so the received samples have to be rotated by π before detection, otherwise the samples are already aligned.

4.5 Data Transmission over the USRPs

Finally all the thoughts from this Chapter should be used to transmit data over the USRPs. First the vector of transmission symbols is generated starting with

the synchronization sequence from Section 4.2, followed by the start of frame sequence of Section 4.4 and a M-PSK modulated random data sequence. These symbols are filtered with a root-raised cosine filter and are transmitted over an USRP.

The received samples are checked for the presence of a synchronization sequence as described in Section 4.2.4 and the frequency and phase offsets as well as the delay of the received samples from the synchronization sequence are estimated by the feedforward estimator of Section 4.2.1 at the middle of the synchronization sequence.

The samples starting at the middle of the synchronization sequence are rotated by the estimated phase and frequency offset and are filtered with a matched root raised cosine filter. These filtered samples are then tracked and decimated.

A fixed number of the corrected and decimated samples of the tracking loop are then send to the BPSK detector to search for the start of frame and resolve the phase ambiguity, as described in Section 4.4. All symbols that belong to the data sequence are then M-PSK demodulated taking the detected phase ambiguity into account. The demodulated data sequence is then compared to the generated data sequence to see if any bit errors have occurred.

Chapter 5

Summary

In the first part of this work a fully functional interface between MATLAB and the USRPs has been developed. With this interface it is now possible to access the USRP devices, set all necessary parameters and transmit and receive samples with or without time tagging even from multiple USRPs as long as the MATLAB Thread is fast enough to catch up with the data rates.

In the second part a synchronization scheme was investigated to be used for frequency, timing and phase synchronization of USRP transmissions. It has been shown in simulations that the used feedforward estimator achieves the Cramér-Rao bounds. It has also been shown that the used tracking algorithms are capable of keeping the M-PSK symbols separable over long durations of random transmitted data. Afterwards the start of frame sequence has been used to align the bit synchronized data stream and resolve the phase ambiguity of the received samples. Finally the received samples have been demodulated and detected both in simulations as well as with measured data from the USRPs, using the interface of the first part of this work.

Based on these two parts it is now possible to use a coherent USRP channel in MATLAB for further testing of communication schemes where existing simulation MATLAB code can be reused and without having to worry about synchronization.

Appendix A

A.1 Send and Receive Multiple Data Streams from a Single Thread

Listing A.1: Sample code to send and transmit from one single Thread

```
#include <uhd/usrp/multi_usrp.hpp>
#include <stdio.h>
#include <string>
#include <complex>
#include <unistd.h>

int main(void){
  // desired data rate in samples per seconds
  double datarate = 2000000.0; // 2 Msamples per second
  // desired center frequency
  double frequency = 1490000000.0; // 1.49 GHz
  // maximum time for read our buffer, before return
  double timeout = 0.008; // 8 ms

  // how many samples should be read out at once
  const size_t max_samps_per_packet = 25000;
  // number of read/write cycles that should be done
  const int num_packets = 10000;

  // counting variables
  size_t i;
  size_t rx_num, tx_num;

  // default device
  std::string args("");
  uhd::usrp::multi_usrp::sptr usrp = uhd::usrp::multi_usrp::
      make(args);

  // set datarates
  usrp->set_rx_rate(datarate);
  usrp->set_tx_rate(datarate);

  // set center frequencys
  usrp->set_rx_freq(frequency);
```

```
34  usrp->set_tx_freq(frequency);
35
36  // no amplification needed
37  usrp->set_rx_gain(0.0);
38  usrp->set_tx_gain(0.0);
39
40  // space to store metadata
41  uhd::rx_metadata_t rx_md;
42  uhd::tx_metadata_t tx_md;
43  uhd::async_metadata_t async_md;
44
45  // send as soon as possible (no time tagging)
46  tx_md.has_time_spec = false;
47
48  // buffers for sending and receiving
49  std::vector<std::complex<double> > rx_buff(
        max_samps_per_packet);
50  std::vector<std::complex<double> > tx_buff(
        max_samps_per_packet);
51
52  // create send signal inside the buffer
53  for(i=0;i<max_samps_per_packet;i++) {
54  // create complex cosine (alternating +/-1+0i)
55  tx_buff[i] = std::complex<double>(((i%2 == 0) ? -1 : 1),
        0);
56  }
57
58  // start streaming
59  printf("start streaming\n");
60  usrp->issue_stream_cmd(uhd::stream_cmd_t::
        STREAM_MODE_START_CONTINUOUS);
61
62  for(i=num_packets ;i>0 ;i--) {
63  // send out the whole buffer
64  tx_num = usrp->get_device()->send(&tx_buff.front(),
        tx_buff.size(), tx_md, uhd::io_type_t::COMPLEX_FLOAT64,
         uhd::device::SEND_MODE_FULL_BUFF);
65
66  // receive until the buffer is full, or timeout seconds
        went by
67  rx_num = usrp->get_device()->recv(&rx_buff.front(),
        rx_buff.size(), rx_md, uhd::io_type_t::COMPLEX_FLOAT64,
         uhd::device::RECV_MODE_FULL_BUFF, timeout);
68
69  // receive asynchronous messages (errors during
        transmission) with 1ms timeout
70  if (usrp->get_device()->recv_async_msg(async_md, 0.001) &&
         async_md.event_code != uhd::async_metadata_t::
        EVENT_CODE_BURST_ACK) {
71    // metadata was not an acknowledgement, every other
          metadata packet signals transmission problems
72    printf("tx error\n");
73  }
74
```

```
75   // if received less samples than transmitted, print out
         the numbers
76   if(rx_num < max_samps_per_packet) {
77     printf("send: %d, recv: %d, max:%d\n",(int)tx_num, (int)
           rx_num, (int)max_samps_per_packet);
78   }
79
80   // when there are receive errors, print them out
81   if(rx_md.error_code != uhd::rx_metadata_t::ERROR_CODE_NONE
         ) {
82     printf("rx error\n");
83   }
84
85   // sleep some time to simulate data processing in MATLAB
86   usleep(5000);
87   }
88
89   // signal the USRP to stop transmitting
90   tx_md.end_of_burst = true;
91   usrp->get_device()->send("", 0, tx_md, uhd::io_type_t::
         COMPLEX_FLOAT64, uhd::device::SEND_MODE_FULL_BUFF);
92
93   // stop receiving
94   usrp->issue_stream_cmd(uhd::stream_cmd_t::
         STREAM_MODE_STOP_CONTINUOUS);
95
96   printf("finished\n");
97   return 0;
98 }
```

A.2 Send and Receive Samples to a File

A.2.1 Sending Samples

Listing A.2: Sample code to read out samples from a file and transmit them over the USRP

```
1  // The following code is based on the rx_samples_to_file
       example from the UHD library code
2  #include <uhd/utils/thread_priority.hpp>
3  #include <uhd/utils/safe_main.hpp>
4  #include <uhd/usrp/multi_usrp.hpp>
5  #include <iostream>
6  #include <fstream>
7  #include <csignal>
8  #include <complex>
9
10 // prepare the function to abort reception later on
11 static bool stop_signal_called = false;
12 void sig_int_handler(int){stop_signal_called = true;}
13
14 // programm entry point
```

```
15 int UHD_SAFE_MAIN(int argc, char *argv[]){
16   // print help message when lacking parameters
17   if(not (argc == 7 || argc == 6)) {
18     printf("usage: %s usrpargs filename frequency rate gain
         [numsamps]\n", argv[0]);
19     return 1;
20   }
21
22   // give the uhd threads their desired priorities
23   uhd::set_thread_priority_safe();
24
25   // parse input parameters
26   const std::string   args(argv[1]);
27   const std::string   file(argv[2]);
28   const double freq = atof(argv[3]);
29   const double rate = atof(argv[4]);
30   const double gain = atof(argv[5]);
31   const size_t samps_total = (argc == 7) ? strtoul(argv[6],
       NULL, 10) : 0;
32
33   // number of samples to receive before writing them into a
        file
34   const size_t samps_per_buff = samps_total; // first
       receive all samples and then write the file
35   // number of samples to read from the UHD in each loop
       cycle
36   const size_t samps_per_loop = 10000; // 10 thousand
37
38   // give the uhd threads their desired priorities
39   uhd::set_thread_priority_safe();
40
41   // create the USRP device
42   uhd::usrp::multi_usrp::sptr usrp = uhd::usrp::multi_usrp::
       make(args);
43
44   // set transmission rate, frequency and gain
45   usrp->set_rx_rate(rate);
46   usrp->set_rx_freq(freq);
47   usrp->set_rx_gain(gain);
48
49   // if no fixed number of samples is given, let the user
       abbort the reception
50   if (samps_total == 0){
51     std::signal(SIGINT, &sig_int_handler);
52     printf("Press Ctrl + C to stop streaming...\n");
53   }
54
55   //create a receive streamer
56   uhd::stream_args_t stream_args(std::string("fc64"));
57   uhd::rx_streamer::sptr rx_stream = usrp->get_rx_stream(
       stream_args);
58
59   // store metadata
60   uhd::rx_metadata_t md;
```

```
61
62  // allocate the buffer
63  std::vector<std::complex<double> > buff(samps_per_buff);
64
65  // create the file stream
66  std::ofstream outfile((file.c_str()), std::ofstream::
        binary);
67
68  // counter of the overall received samples
69  size_t rx_total=0;
70
71  printf("Start recording:\n");
72
73  //setup streaming
74  uhd::stream_cmd_t stream_cmd(
75      (samps_total == 0) ?
76          uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS :
77  uhd::stream_cmd_t::STREAM_MODE_NUM_SAMPS_AND_DONE
78  );
79  stream_cmd.num_samps = samps_total;
80  stream_cmd.stream_now = true;
81  usrp->issue_stream_cmd(stream_cmd);
82
83  // mail loop
84  while(not stop_signal_called && rx_total<samps_total){
85    // counter of received samples in this loop
86    size_t i = 0;
87    while(i<samps_per_buff) {
88      size_t num_rx_samps = rx_stream->recv(&buff.at(i),
          samps_per_loop, md);
89      i += num_rx_samps;
90
91      // show progress
92      std::cout << ".";
93      flush(std::cout);
94
95
96      if (md.error_code == uhd::rx_metadata_t::
          ERROR_CODE_TIMEOUT) {
97        printf("rx timeout\n");
98      }
99      if (md.error_code == uhd::rx_metadata_t::
          ERROR_CODE_OVERFLOW){
100       printf("rx_error\n");
101     }
102     if (md.error_code != uhd::rx_metadata_t::
          ERROR_CODE_NONE){
103       printf("Unexpected error code 0x%x", md.error_code);
104     }
105   }
106   // write buffer to file
107   outfile.write((const char*)&buff.front(), i*sizeof(std::
        complex<double>));
108   // increase the overall counter
```

```
109     rx_total += i;
110   }
111   // close file
112   outfile.close();
113 
114   printf("\n\nDone!\n");
115   return 0;
116 }
```

Listing A.3: Sample code to write sample data into a file

```
1  function writesamples(data, filename, varargin)
2  % writesamples - Write samples into a file and transmit them
       over the USRP
3  %   Syntax:   writesamples(data, filename, [usrp_arg, freq,
       rate, gain])
4  %   Input:    data     - Vector of 1xN samples
5  %             filename - The name and path of the file to
       store the samples
6  %             usrp_arg - The device argument for the USRP
7  %             freq     - Center frequency of the USRP
8  %             rate     - Sampling rate of the USRP
9  %             gain     - Receive gain of the USRP
10 
11 if(~(isempty(varargin) || length(varargin) == 4))
12     % wrong number of input variables
13     error('usage: writesamples(data, filename, [usrp_arg,
           freq, rate, gain])');
14 end
15 % write sequential double values as alternating real and
       imaginary part
16 tmp = zeros(2*length(data(:)), 1);
17 tmp(1:2:end) = real(data(:));
18 tmp(2:2:end) = imag(data(:));
19 % clear original data sequence
20 clear data;
21 % open the samples file
22 fid = fopen(filename, 'w');
23 %write data
24 fwrite(fid, tmp, 'double');
25 %close file and clean up temporary variables
26 fclose(fid);
27 clear tmp fid;
28 
29 if(length(varargin) == 4)
30     % all parameters are given, receive befor reading the
           file
31     usrp_arg = varargin{1};
32     freq     = varargin{2};
33     rate     = varargin{3};
34     gain     = varargin{4};
35     % execute the transmit script from within matlab
36     command = sprintf('./tx_samples_from_file %s %s %f %f %f
           ',usrp_arg,filename, freq, rate, gain);
```

```
37     system(command);
38 end
39 end
```

A.2.2 Receiving Samples

Listing A.4: Sample code to receive samples from the USRP and store them in a file

```
1  // The following code is based on the rx_samples_to_file
       example from the UHD library code
2  #include <uhd/utils/thread_priority.hpp>
3  #include <uhd/utils/safe_main.hpp>
4  #include <uhd/usrp/multi_usrp.hpp>
5  #include <iostream>
6  #include <fstream>
7  #include <csignal>
8  #include <complex>
9
10 // prepare the function to abort reception later on
11 static bool stop_signal_called = false;
12 void sig_int_handler(int){stop_signal_called = true;}
13
14 // programm entry point
15 int UHD_SAFE_MAIN(int argc, char *argv[]){
16   // print help message when lacking parameters
17   if(not (argc == 7 || argc == 6)) {
18     printf("usage: %s usrpargs filename frequency rate gain
         [numsamps]\n", argv[0]);
19     return 1;
20   }
21
22   // give the uhd threads their desired priorities
23   uhd::set_thread_priority_safe();
24
25   // parse input parameters
26   const std::string   args(argv[1]);
27   const std::string   file(argv[2]);
28   const double freq = atof(argv[3]);
29   const double rate = atof(argv[4]);
30   const double gain = atof(argv[5]);
31   const size_t samps_total = (argc == 7) ? strtoul(argv[6],
       NULL, 10) : 0;
32
33   // number of samples to receive before writing them into a
       file
34   const size_t samps_per_buff = samps_total; // first
       receive all samples and then write the file
35   // number of samples to read from the UHD in each loop
       cycle
36   const size_t samps_per_loop = 10000; // 10 thousand
37
38   // give the uhd threads their desired priorities
```

```
39  uhd::set_thread_priority_safe();
40
41  // create the USRP device
42  uhd::usrp::multi_usrp::sptr usrp = uhd::usrp::multi_usrp::
        make(args);
43
44  // set transmission rate, frequency and gain
45  usrp->set_rx_rate(rate);
46  usrp->set_rx_freq(freq);
47  usrp->set_rx_gain(gain);
48
49  // if no fixed number of samples is given, let the user
        abbort the reception
50  if (samps_total == 0){
51    std::signal(SIGINT, &sig_int_handler);
52    printf("Press Ctrl + C to stop streaming...\n");
53  }
54
55  //create a receive streamer
56  uhd::stream_args_t stream_args(std::string("fc64"));
57  uhd::rx_streamer::sptr rx_stream = usrp->get_rx_stream(
        stream_args);
58
59  // store metadata
60  uhd::rx_metadata_t md;
61
62  // allocate the buffer
63  std::vector<std::complex<double> > buff(samps_per_buff);
64
65  // create the file stream
66  std::ofstream outfile((file.c_str()), std::ofstream::
        binary);
67
68  // counter of the overall received samples
69  size_t rx_total=0;
70
71  printf("Start recording:\n");
72
73  //setup streaming
74  uhd::stream_cmd_t stream_cmd(
75      (samps_total == 0) ?
76          uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS :
77  uhd::stream_cmd_t::STREAM_MODE_NUM_SAMPS_AND_DONE
78  );
79  stream_cmd.num_samps = samps_total;
80  stream_cmd.stream_now = true;
81  usrp->issue_stream_cmd(stream_cmd);
82
83  // mail loop
84  while(not stop_signal_called && rx_total<samps_total){
85    // counter of received samples in this loop
86    size_t i = 0;
87    while(i<samps_per_buff) {
```

```
88        size_t num_rx_samps = rx_stream->recv(&buff.at(i),
              samps_per_loop, md);
89        i += num_rx_samps;
90
91        // show progress
92        std::cout << ".";
93        flush(std::cout);
94
95
96        if (md.error_code == uhd::rx_metadata_t::
              ERROR_CODE_TIMEOUT) {
97          printf("rx timeout\n");
98        }
99        if (md.error_code == uhd::rx_metadata_t::
              ERROR_CODE_OVERFLOW){
100         printf("rx_error\n");
101       }
102       if (md.error_code != uhd::rx_metadata_t::
              ERROR_CODE_NONE){
103         printf("Unexpected error code 0x%x", md.error_code);
104       }
105     }
106     // write buffer to file
107     outfile.write((const char*)&buff.front(), i*sizeof(std::
            complex<double>));
108     // increase the overall counter
109     rx_total += i;
110   }
111   // close file
112   outfile.close();
113
114   printf("\n\nDone!\n");
115   return 0;
116 }
```

Listing A.5: Sample code to read out a file with raw sample data

```
1  function data = readsamples(filename, offset, varargin)
2  % readsamples - Receive samples from the USRP into a file
       and read them out
3  %   Syntax:   data = readsamples(filename, offset, [usrp_arg
       , freq, rate, gain, numsamps])
4  %   Input:    filename - The name and path of the file to
       store the samples
5  %             offset   - Number of samples at the beginning
       to ignore
6  %             usrp_arg - The device argument for the USRP
7  %             freq     - Center frequency of the USRP
8  %             rate     - Sampling rate of the USRP
9  %             gain     - Receive gain of the USRP
10 %             numsamps - Number of samples to receive
11 %   Output:   data     - Vector or 1xN samples with
12 %                        N = numsamps if provided
13
```

```
14 if(length(varargin) == 5)
15     % all parameters are given, receive befor reading the
           file
16     usrp_arg = varargin{1};
17     freq     = varargin{2};
18     rate     = varargin{3};
19     gain     = varargin{4};
20     numsamps = varargin{5};
21     % execute the receive script from within matlab
22     command = sprintf('./rx_samples_to_file %s %s %f %f %f %
           d',usrp_arg,filename, freq, rate, gain, numsamps+
           offset);
23     system(command);
24 elseif(~isempty(varargin))
25     % wrong number of input variables
26     error('usage: data = readsamples(filename, [usrp_arg,
           freq, rate, gain, numsamps])');
27 end
28 % open the samples file
29 fid = fopen(filename);
30 % read sequential double values and store them as doubles
31 tmp=fread(fid, '*double');
32 % odd data are the real values and even data the imaginary
       values of the samples
33 data=complex(tmp((2*offset+1):2:length(tmp)),tmp((2*offset
       +2):2:length(tmp)));
34 %close file
35 fclose(fid);
36 end
```

A.3 Send and Receive Samples from MATLAB

Listing A.6: Sample code to send and receive samples over the Mex interface from within MATLAB.

```
1  % choose USRP devices
2  uhdsend = uhdinterface('init','serial=4e2610f4');
3  uhdrecv = uhdinterface('init','addr=192.168.10.2');
4
5  % set parameters
6  uhdinterface(uhdsend,'set_tx_freq',1.5e9)
7  uhdinterface(uhdsend,'set_tx_rate',1e6)
8  uhdinterface(uhdsend,'set_tx_gain',10)
9  uhdinterface(uhdrecv,'set_rx_freq',1.5e9)
10 uhdinterface(uhdrecv,'set_rx_rate',1e6)
11 uhdinterface(uhdrecv,'set_rx_gain',30)
12
13 % get messages from the UHD lib
14 uhdinterface('flush');
15
16 % number of samples to receive/transmitt per loop
17 buf_len = 50000;
```

```
18
19 % number of loop cycles
20 n = 50;
21
22 % generate send signal (constant carrier) and preassign
       receive buffer
23 send_buf = ones(buf_len,1)*(1+0j);
24 recvvec = zeros(buf_len*n,1);
25
26 % start receiving and streaming
27 uhdinterface(uhdrecv,'rx_stream_start')
28 uhdinterface(uhdsend,'tx_stream_start')
29 uhdinterface(uhdsend,'send',send_buf);
30
31 % throw away the first samples (which already include the
       transmit signal)
32 recvvec(:,1) = uhdinterface(uhdrecv,'receiven',buf_len);
33
34 % send and rceive loop
35 for i=1:n
36     uhdinterface(uhdsend,'send',send_buf);
37     recvvec((i-1)*buf_len+1:i*buf_len) = uhdinterface(
           uhdrecv,'receiven',buf_len);
38 end
39
40 % stop streaming
41 uhdinterface(uhdsend,'tx_stream_stop');
42 uhdinterface(uhdrecv,'rx_stream_stop');
43
44 % close USRP and unlock mexfile
45 uhdinterface(uhdsend,'close')
46 uhdinterface(uhdrecv,'close')
```

A.4 Spectrum Analyzer

Listing A.7: Sample code to use the USRP interface as a spectrum analyzer in MATLAB.

```
1 clear all;
2 close all;
3
4 % choose USRP device
5 % uhd = uhdinterface('init','serial=4e2610f4');   %USRP1
6 uhd = uhdinterface('init','serial=4e2611ca');   %USRP1
7 % uhd = uhdinterface('init','addr=192.168.10.2'); %USRP N210
8
9 % data rate in samples per second
10 rate = 0.5e6;
11 % freq = 1541.6e6; % Inmarsat 4F2
12 % freq = 1878.5e6; % Meteosat 9
13 freq = 2e9;
14 gain = 20;
```

```
15 update_time = 0.1;
16 avrg_len = 10;
17
18 % setparameters
19 uhdinterface(uhd,'set_rx_freq',freq)
20 uhdinterface(uhd,'set_rx_rate',rate)
21 uhdinterface(uhd,'set_rx_gain',gain)
22
23 % get messages from the UHD lib
24 uhdinterface('flush');
25
26 %length of buffer
27 N = 10e3;
28
29 % number of loop cycles to skip before recomputing the fft
30 n = ceil(update_time/N*rate);
31
32 % frequency vector
33 f = (((0:N-1) - ceil(N/2)) / N * rate) + freq;
34 % generate axis label
35 figurehandle = figure(1);
36 % set(figurehandle, 'Windowstyle', 'modal');
37 plothandle = plot(f,zeros(size(f)));
38 xlabel('Frequency [Hz]');
39 y_max = 0;
40 y_min = -150;
41 x_max = max(f);
42 x_min = min(f);
43 axis([x_min x_max y_min y_max]);
44
45 % start receiving
46 uhdinterface(uhd,'rx_stream_start');
47
48 % throw away the first N samples
49 vec = uhdinterface(uhd,'receiven',N);
50
51 i = 1;
52 reset_avrg = true;
53 % receive wave form
54 while true
55     vec = uhdinterface(uhd,'receiven',N);
56     if not(ishandle(plothandle))
57         break;
58     end
59     if(i >= n)
60         i = 1;
61         newfftvec = 20*log10(abs(fftshift(fft(vec))/N));
62         if reset_avrg
63             fftvec = newfftvec;
64             reset_avrg = false;
65         else
66             fftvec = (avrg_len*fftvec + newfftvec)/(avrg_len
                   +1);
67         end
```

```
68          set(plothandle,'YData',fftvec);  % Update the plot
                line
69          drawnow;
70      end
71
72  %     pressedKey = get(figurehandle, 'CurrentCharacter');
73  %     if pressedKey == 'q'
74  %         break;
75  %     end
76      i = i+1;
77  end
78
79  % stop streaming
80  uhdinterface(uhd, 'rx_stream_stop');
81
82  % close USRP and unlock mexfile
83  uhdinterface(uhd, 'close');
84
85  % set(figurehandle, 'Windowstyle', 'normal');
```

A.5 Available Interface Commands

The interface calls have the following form

```
[return] = uhdinterface([index], command, [parameter]);
```

where a value enclosed in brackets [] is an optional value. If an index is possible but none is supplied the default USRP with index 0 is used. The possible commands are:

init *returns USRP index, no index, optional string parameter*
Initializes a USRP object with the given device argument, or an empty string if no argument is supplied. The USRP object is created and stored. The index to this USRP object is returned as positive (or zero) integer value. If the returned value is negative there has been an error and no USRP object has been created.

flush *returns nothing, no index, no parameter*
Prints out new messages from the UHD on the MATLAB console, does nothing if there are no new messages.

set_rx_freq_offset *returns nothing, optional index, scalar double parameter*
Changes the receiving centre frequency offset of the selected USRP object to the provided frequency offset in Hz. This option offsets the operational frequency of the A/D converters and is afterwards compensated by the USRP's DSP. This offset is useful when transmitting and receiving at the same frequency on the same daughterboard to prevent crosstalk of the RX and TX path.

set_rx_freq *returns nothing, optional index, scalar double parameter*
Changes the receiving centre frequency of the selected USRP object to the provided frequency in Hz.

set_rx_rate *returns nothing, optional index, scalar double parameter*
Changes the receiving sampling rate of the selected USRP object to the provided sampling rate in samples per second.

set_rx_bw *returns nothing, optional index, scalar double parameter*
Changes the receiving channel filter of the selected USRP object to the provided bandwidth in Hz if this operation is supported by the daughterboard.

set_rx_gain *returns nothing, optional index, scalar double parameter*
Changes the receiving amplification gain of the selected USRP object to the provided gain in the range from 0 to 30.

set_rx_subdev *returns nothing, optional index, string parameter*
Changes the RX subdevice to the specified one.

set_rx_antenna *returns nothing, optional index, string parameter*
Changes the RX antenna to the specified one.

receiven *returns vector of complex samples, optional index, scalar integer parameter*
Tries to receive the provided amount of samples from the selected USRP. All received samples are returned in an Nx1 vector where N is either the number of requested samples or less, if a timeout occurred during reception.

set_tx_freq *returns nothing, optional index, scalar double parameter*
Changes the sending centre frequency of the selected USRP object to the provided frequency in Hz.

set_tx_rate *returns nothing, optional index, scalar double parameter*
Changes the sending sampling rate of the selected USRP object to the provided sampling rate in samples per second.

set_tx_bw *returns nothing, optional index, scalar double parameter*
Changes the sending channel filter of the selected USRP object to the provided bandwidth in Hz if this operation is supported by the daughterboard.

set_tx_gain *returns nothing, optional index, scalar double parameter*
Changes the sending amplification gain of the selected USRP object to the provided gain in the range from 0 to 30.

set_tx_subdev *returns nothing, optional index, string parameter*
Changes the TX subdevice to the specified one.

set_tx_antenna *returns nothing, optional index, string parameter*
Changes the TX antenna to the specified one.

set_tx_timespec *returns nothing, optional index, scalar double parameter*
Sets the absolute time of transmission of the next data package (initialized by the next send command) in seconds.

send *returns integer number, optional index, vector of complex doubles as parameter*
Tries to send the provided samples from the selected USRP. Returns the number of send samples which is either the number of provided samples or less, if a timeout occurred during transmission.

get_rx_freq *returns scalar double, optional index, no parameter*
Returns the receiving centre frequency of the selected USRP in Hz.

get_rx_rate *returns scalar double, optional index, no parameter*
Returns the receiving sampling rate of the selected USRP in samples per second.

get_rx_bw *returns scalar double, optional index, no parameter*
Returns the receiving channel filter bandwidth of the selected USRP in Hz if this operation is supported by the daughterboard..

get_rx_gain *returns scalar double, optional index, no parameter*
Returns the receiving amplification gain of the selected USRP in the range from 0 to 30.

get_rx_subdev *returns string, optional index, no parameter*
Returns the pretty print string of the RX subdevice.

get_rx_antenna *returns string, optional index, no parameter*
Returns the pretty print string of the RX antenna.

get_rx_timespec *returns scalar double, optional index, no parameter*
Returns the absolute timespec of the last received packet (initialized by the previous receiveN command) in seconds.

rx_stream_start *returns nothing, optional index, no parameter*
Starts the reception of samples into the internal buffer of the selected USRP.

rx_stream_stop *returns nothing, optional index, no parameter*
Stops the reception of samples into the internal buffer of the selected USRP.

get_tx_freq *returns scalar double, optional index, no parameter*
Returns the sending centre frequency of the selected USRP in Hz.

get_tx_rate *returns scalar double, optional index, no parameter*
Returns the sending sampling rate of the selected USRP in samples per second.

get_tx_bw *returns scalar double, optional index, no parameter*
Returns the sending channel filter bandwidth of the selected USRP in Hz if this operation is supported by the daughterboard..

get_tx_gain *returns scalar double, optional index, no parameter*
Returns the sending amplification gain of the selected USRP in the range from 0 to 30.

get_tx_subdev *returns string, optional index, no parameter*
Returns the pretty print string of the TX subdevice.

get_tx_antenna *returns string, optional index, no parameter*
Returns the pretty print string of the TX antenna.

tx_stream_start *returns nothing, optional index, no parameter*
Starts the transmission of samples from the internal buffer of the selected USRP with a delay of 1ms.

tx_stream_stop *returns nothing, optional index, no parameter*
Stops the transmission of samples from the internal buffer of the selected USRP.

close *returns nothing, optional index, no parameter*
Closes the selected USRP object and if no other USRP object is left, the Mex function is unlocked.

closeAll *returns nothing, no index, no parameter*
Closes all USRP objects and the Mex function is unlocked.

test *returns optional scalar double, no index, optional scalar double parameter*
If a parameter is given an internal test variable is assigned with the value of the parameter. If a return value is requested the current value of the test variable is returned.

A.6 Derivation of the Maximum Likelihood Estimator

The equality of

$$\sum_{k=0}^{2L-1} \cos^2\left(\pi\left[\frac{k}{2}-\tilde{\tau}\right]\right) = L \tag{A.6.1}$$

should be shown, as used in Section 4.2.1 on page 25

$$\begin{aligned}
&\sum_{k=0}^{2L-1} \cos^2\left(\pi\left[\frac{k}{2}-\tilde{\tau}\right]\right)\\
=&\sum_{k=0}^{2L-1} \frac{1}{2}\left(1+\cos\left(2\pi\left[\frac{k}{2}-\tilde{\tau}\right]\right)\right)\\
=&\sum_{k=0}^{2L-1} \frac{1}{2}\left(1+(-1)^k\cos\left(2\pi\tilde{\tau}\right)\right)\\
=&\sum_{k=0}^{2L-1} \frac{1}{2}+\cos\left(2\pi\tilde{\tau}\right)\cdot\underbrace{\sum_{k=0}^{2L-1}(-1)^k}_{=0}\\
=&L.
\end{aligned}$$

□

The equality of

$$\Gamma(\tilde{\nu},\tilde{\tau}) = |Y_e(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^2 + \Re\left\{e^{-j2\pi\tilde{\tau}}\mathcal{A}(\tilde{\nu})\right\} \tag{A.6.2}$$

should be shown, as used in Section 4.2.1 on page 26

$$\begin{aligned}\Gamma(\tilde{\nu},\tilde{\tau}) =& 2\left|\mathcal{Z}(\tilde{\nu},\tilde{\tau})\right|^2\\ =& 2\left|Y_e(\tilde{\nu})\cdot\cos(\pi\tilde{\tau}) + e^{-j\pi\tilde{\nu}}\cdot Y_o(\tilde{\nu})\cdot\sin(\pi\tilde{\tau})\right|^2\\ =& 2\left|Y_e(\tilde{\nu})\right|^2\cdot\left|\cos(\pi\tilde{\tau})\right|^2 + 2\left|Y_o(\tilde{\nu})\right|^2\cdot\left|\sin(\pi\tilde{\tau})\right|^2\\ &+ 4\Re\left\{Y_e(\tilde{\nu})\cdot Y_o^*(\tilde{\nu})\cdot e^{j\pi\tilde{\nu}}\cdot\cos(\pi\tilde{\tau})\cdot\sin(\pi\tilde{\tau})\right\}\end{aligned}$$

applying addition theorems leads to:

$$\begin{aligned}=& 2\left|Y_e(\tilde{\nu})\right|^2\cdot\frac{1}{2}\left(1+\cos\left(2\pi\tilde{\tau}\right)\right) + 2\left|Y_o(\tilde{\nu})\right|^2\cdot\frac{1}{2}\left(1-\cos\left(2\pi\tilde{\tau}\right)\right)\\ &+ 4\Re\left\{Y_e(\tilde{\nu})\cdot Y_o^*(\tilde{\nu})\cdot e^{j\pi\tilde{\nu}}\cdot\frac{1}{2}\sin\left(2\pi\tilde{\tau}\right)\right\}\\ =& \left|Y_e(\tilde{\nu})\right|^2 + \left|Y_o(\tilde{\nu})\right|^2\\ &+ \left(\left|Y_e(\tilde{\nu})\right|^2 - \left|Y_o(\tilde{\nu})\right|^2\right)\cos\left(2\pi\tilde{\tau}\right) + 2\Re\left\{Y_e(\tilde{\nu})Y_o^*(\tilde{\nu})e^{j\pi\tilde{\nu}}\sin\left(2\pi\tilde{\tau}\right)\right\}\end{aligned}$$

The already real valued second line can be enclosed with the $\Re$ operator without changing the result.

$$\begin{aligned}=& \left|Y_e(\tilde{\nu})\right|^2 + \left|Y_o(\tilde{\nu})\right|^2 +\\ &\Re\left\{\left(\left|Y_e(\tilde{\nu})\right|^2 - \left|Y_o(\tilde{\nu})\right|^2\right)\cos\left(2\pi\tilde{\tau}\right) + 2\Re\left\{Y_e(\tilde{\nu})Y_o^*(\tilde{\nu})e^{j\pi\tilde{\nu}}\sin\left(2\pi\tilde{\tau}\right)\right\}\right\}\end{aligned}$$

to replace the cos and sin terms by a single complex rotation

$$\begin{aligned}=& \left|Y_e(\tilde{\nu})\right|^2 + \left|Y_o(\tilde{\nu})\right|^2\\ &+ \Re\left\{e^{2\pi\tilde{\tau}}\left(\left|Y_e(\tilde{\nu})\right|^2 - \left|Y_o(\tilde{\nu})\right|^2 + j2\Re\left\{e^{j\pi\tilde{\nu}}Y_e(\tilde{\nu})Y_o^*(\tilde{\nu})\right\}\right)\right\}\end{aligned}$$

The inner part is defined as $\mathcal{A}(\tilde{\nu})$

$$\mathcal{A}(\tilde{\nu}) := \left|Y_e(\tilde{\nu})\right|^2 - \left|Y_o(\tilde{\nu})\right|^2 + j2\Re\left\{e^{j\pi\tilde{\nu}}Y_e(\tilde{\nu})Y_o^*(\tilde{\nu})\right\}$$

which leads to

$$\Gamma(\tilde{\nu},\tilde{\tau}) = \left|Y_e(\tilde{\nu})\right|^2 + \left|Y_o(\tilde{\nu})\right|^2 + \Re\left\{e^{-j2\pi\tilde{\tau}}\mathcal{A}(\tilde{\nu})\right\}. \qquad \square$$

$\mathcal{A}$ should be described in argument/phase notation as used in Section 4.2.1 on page 26 as

$$|\mathcal{A}(\tilde{\nu})| = \left|Y_e^2(\tilde{\nu}) + e^{-j2\pi\tilde{\nu}}Y_o^2(\tilde{\nu})\right|. \tag{A.6.3}$$

The squared argument of $\mathcal{A}$ is calculated with

$$|\mathcal{A}(\tilde{\nu})|^2 = |Y_e(\tilde{\nu})|^4 - 2\,|Y_e(\tilde{\nu})|^2\,|Y_o(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^4 + 4\Re\left\{e^{j\pi\tilde{\nu}}Y_e(\tilde{\nu})Y_o^*(\tilde{\nu})\right\}^2 \tag{A.6.4}$$

with $\Re\{z\} = \frac{1}{2}(z + z^*)$ equation (A.6.4) becomes

$$\begin{aligned}|\mathcal{A}(\tilde{\nu})|^2 &= |Y_e(\tilde{\nu})|^4 - 2\,|Y_e(\tilde{\nu})|^2\,|Y_o(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^4 \\ &\quad + 4\left(\frac{1}{2}\left(e^{j\pi\tilde{\nu}}Y_e(\tilde{\nu})Y_o^*(\tilde{\nu}) + e^{-j\pi\tilde{\nu}}Y_e^*(\tilde{\nu})Y_o(\tilde{\nu})\right)\right)^2 \\ &= |Y_e(\tilde{\nu})|^4 - 2\,|Y_e(\tilde{\nu})|^2\,|Y_o(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^4 + e^{j2\pi\tilde{\nu}}Y_e^2(\tilde{\nu})Y_o^{*^2}(\tilde{\nu}) \\ &\quad + 2\,|Y_e(\tilde{\nu})|^2\,|Y_o^*(\tilde{\nu})|^2 + e^{-j2\pi\tilde{\nu}}Y_e^{*^2}(\tilde{\nu})Y_o^2(\tilde{\nu})\end{aligned} \tag{A.6.5}$$

with $z + z^* = 2\Re\{z\}$ equation (A.6.5) becomes

$$\begin{aligned}|\mathcal{A}(\tilde{\nu})|^2 &= |Y_e(\tilde{\nu})|^4 - 2\,|Y_e(\tilde{\nu})|^2\,|Y_o(\tilde{\nu})|^2 + |Y_o(\tilde{\nu})|^4 \\ &\quad + 2\,|Y_e(\tilde{\nu})|^2\,|Y_o(\tilde{\nu})^*|^2 + 2\Re\left\{e^{j2\pi\tilde{\nu}}Y_e^2(\tilde{\nu})Y_o^{*^2}(\tilde{\nu})\right\} \\ &= |Y_e(\tilde{\nu})|^4 + |Y_o(\tilde{\nu})|^4 + 2\Re\left\{e^{j2\pi\tilde{\nu}}Y_e^2(\tilde{\nu})Y_o^{*^2}(\tilde{\nu})\right\} \\ &= \left|Y_e^2(\tilde{\nu}) + e^{-j2\pi\tilde{\nu}}Y_o^2(\tilde{\nu})\right|^2\end{aligned}$$

which leads to

$$|\mathcal{A}(\tilde{\nu})| = \left|Y_e^2(\tilde{\nu}) + e^{-j2\pi\tilde{\nu}}Y_o^2(\tilde{\nu})\right| \qquad \square$$

A.7 Derivation of the Likelihood of Hypothesis Two

The likelihood of hypothesis two, as used in Section 4.2.4 on page 31 is

$$\Psi_{\mathrm{H2}} = \max_{\tilde{\nu}}\left\{\left|\sum_{k=0}^{2L-1} r[k]\cdot e^{-j\pi\tilde{\nu}k}\right| - \frac{L}{2}\right\} \tag{A.7.1}$$

which can be calculated with

$$\begin{aligned}
\psi_{\mathrm{H2}} =& -\sum_{k=0}^{2L-1} \left| r[k] - e^{j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)} \cdot 1 \right|^2 \\
=& -\sum_{k=0}^{2L-1} \left(|r[k]|^2 - 2\Re\left\{ r[k] \cdot e^{-j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)} \right\} \right) - 2L \\
\Psi_{\mathrm{H2}} =& \sum_{k=0}^{2L-1} \Re\left\{ r[k] \cdot e^{-j\left(\pi\tilde{\nu}(k-L)+\tilde{\phi}\right)} \right\} - \frac{L}{2} \\
=& \Re\left\{ e^{j\left(\pi\tilde{\nu}L-\tilde{\phi}\right)} \cdot \sum_{k=0}^{2L-1} r[k] \cdot e^{-j\pi\tilde{\nu}k} \right\} - \frac{L}{2} \qquad \text{(A.7.2)}
\end{aligned}$$

The optimal value for $\pi\tilde{\nu}L - \tilde{\phi}$ will rotate the sum onto the real-axis.

$$\begin{aligned}
\hat{\phi} - \pi\tilde{\nu}L =& -\measuredangle\left\{ \sum_{k=0}^{2L-1} r[k] \cdot e^{-j\pi\tilde{\nu}k} \right\} \\
\hat{\phi} =& -\measuredangle\left\{ \sum_{k=0}^{2L-1} r[k] \cdot e^{-j\pi\tilde{\nu}k} \right\} + \pi\tilde{\nu}L \qquad \text{(A.7.3)}
\end{aligned}$$

Inserting this optimal phase into (A.7.2)

$$\Psi_{\mathrm{H2}} = \left| \sum_{k=0}^{2L-1} r[k] \cdot e^{-j\pi\tilde{\nu}k} \right| - \frac{L}{2}$$

The maximum likelihood of this hypothesis can then be achieved by taking the maximum over all frequencies $\tilde{\nu}$

$$\Psi_{\mathrm{H2}} = \max_{\tilde{\nu}} \left\{ \left| \sum_{k=0}^{2L-1} r[k] \cdot e^{-j\pi\tilde{\nu}k} \right| - \frac{L}{2} \right\} \qquad \square$$

Bibliography

[1] M. Morelli and A. D'Amico, "Maximum Likelihood Timing and Carrier Synchronization in Burst-Mode Satellite Transmissions," *EURASIP Journal on Wireless Communications and Networking*, 2007.

[2] F. Gardner, "A BPSK/QPSK Timing-Error Detector for Sampled Receivers," *IEEE Transactions on Communications*, May 1986.

[3] U. Mengali and A. D'Andrea, *Synchronization Techniques for Digital Receivers*. Plenum Press, 1997.

[4] A. J. Viterbi and A. Viterbi, "Nonlinear Estimation of PSK-Modulated Carrier Phase with Application to Burst Digital Transmission," *IEEE Transactions on Information Theory*, July 1983.

[5] F. A. Hamza. (2011) The USRP under 1.5X Magnifying Lens! [Online]. Available: http://gnuradio.org/redmine/attachments/download/129

[6] Ettus Research. (2012) UHD Wiki. [Online]. Available: http://code.ettus.com/redmine/ettus/projects/uhd/wiki

[7] Karlsruhe Institute of Technology Communications Engineering Lab. (2011) Simulink-USRP: Universal Software Radio Peripheral (USRP) Blockset. [Online]. Available: http://www.cel.kit.edu/english/downloads.php

[8] MathWorks. (2011) MATLAB and Simulink Support Package for USRP© Hardware. [Online]. Available: http://www.mathworks.de/discovery/sdr/usrp.html

[9] Q. Funke, "Implementation of an ECSS Compliant Ground Station Using Software Defined Radios," Diploma Thesis, Technische Universität München, July 2012.