Implementation and Analysis of Multi-Carrier Synchronization Techniques

Fachbereich 1 - Physics and Electrical Engineering
Institute for Telecommunications and High-Frequency Techniques (ITH)
Department of Communications Engineering
P.O. Box 33 04 40
D-28334 Bremen

Supervisor:	M.Sc. Johannes Demel
First examiner:	Prof. Dr.-Ing. Armin Dekorsy
Second examiner:	Dr.-Ing. Carsten Bockelmann
Date:	December 14, 2017

I ensure the fact that this thesis has been independently written and no other sources or aids, other than mentioned, have been used.

Bremen, December 14, 2017 _____

Contents

1 Abstract

This thesis will present an implementation and evaluation of a multi-carrier synchronization algorithm implemented for use in the GNU Radio framework.

To point out the special synchronization requirements imposed by using multi-carrier systems in contrast to single carrier systems the basic concepts of both methods will be presented with a special focus on synchronization.

Based on these observations a synchonization alogrithm used in multi-carrier systems will be presented, the Schmidl and Cox (S&C) algorithm.

Finally a fast implementation of the S&C algorithm implemented in optimized C++ for the GNU Radio framework will be presented and its performance will be analyzed in terms of synchronization accuracy and processing speed, in comparison to more naive synchronization schemes.

The implementation will be tested for its ability for being used in realtime high datarate applications like a pure software implementation of a WiFi stack using software defined radios.

2 Introduction

This thesis demonstrates an implementation of the S&C algorithm optimized for processing speed, contained in the XFDMSync[1] library of GNU Radio modules that was written as part of this thesis.

To demonstrate the special requirements imposed by multi-carrier transmission methods, namely accurate synchronization in the time and frequency domain and resilience against frequency-selective channcels, a very basic introduction to multi-carrier systems will be given. This introduction will cover the concepts of multipath propagation in radio systems, OFDM and cyclic prefixes.

To give an intuition on why advanced synchronization techniques are needed in multi-carrier systems a simple input power based synchronization method will be presented that does not fulfil the requirements in frequency-domain synchronization and multipath-propagation resilience. This synchronization method will then be compared to the Schmidl and Cox algorithm, that can, in addtion to providing synchronization in the time domain, perform frequency-domain synchronizations and and has superior multipath resilience.

Following the demonstration of the algorithms used there will be an introduction of GNU Radio, a free and open source software-defined radio (SDR) processing framework. This introduction will cover the different ways of using GNU Radio, namely the GNU Radio companion, the Python interface and the C++ interface, and how GNU Radio can be used to create signal processing graphs and nodes. An appendix on this chapter will go into more detail on how GNU Radio schedules its processing and how the buffer management is optimized for processing speed.

Next the Schmidl and Cox implementation written alongside this thesis will be presented, demonstrating some considerations made while translating the S&C algorithm into a fast software implementation.

Finally the performance of the S&C implementation will be evaluated using a set of experiments. The experiments will test different aspects of the implementation. Firstly the accuracy of the synchronization in time, in the presence of different disturbances, namely white gaussian noise, a frequency offset between transmitter and receiver and a frequency-selective channel. Secondly the performance of the synchronization in the frequency-domain will be tested in a hardware test over an actual channel. And finally the processing performance of the implementation in terms of maximum achievable sample rate on common computing hardware.

3 Theoretical background

In order to give a better understanding of the special requirements multi-carrier systems impose on the synchronization between a transmitter and a receiver this chapter presents the basic operating principles of both methods in general and compares them with a focus on synchronization.

This chapter also introduces the Schmidl and Cox synchronization algorithm that is later implemented in software and some aspects of the preamble design.

3.1 Single carrier transmission

A classical single-carrier digital transmission standard usually maps digital bits to complex symbols from an alphabet A and modulate these symbols at a fixed rate onto a carrier frequency.

Figure 1 presents an example of a signal where six QPSK encoded symbols are transmitted in equal time intervals. In QPSK the alphabet consists of the complex symbols

$$A = \frac{1}{\sqrt{2}} \cdot \{1 + j, 1 - j, -1 - j, -1 + j\} \quad .$$

The symbols are shown in the complex baseband domain and a hypothetical filter is constructed such that the signal phase is interpolated linearily between symbols.

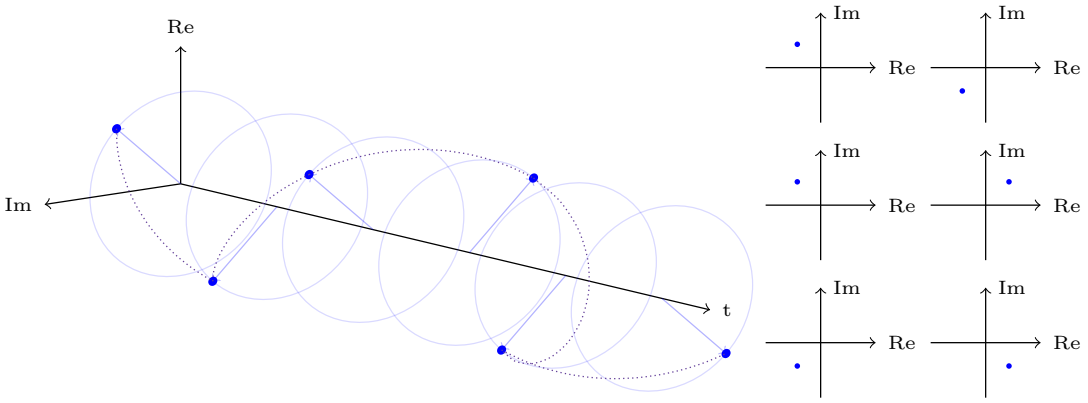


Figure 1: QPSK modulated symbols over time in the complex domain

When only one of the complex components (real- or imaginary part) is observed and the continous signal is displayed as an eye-diagram, meaning it is cut and overlayed at symbol-length intervals, as shown in Figure 2, one can immediately guess the optimal sampling time, indicated by the vertical line in the middle of the plot. The optimal sampling time can be found by observing the zero-crossings of the signal which have to

happen half a symbol period before or after the sampling [1]. In low noise situations, like in Figure 2, the modulated digital signal can be perfectly reconstructed when sampled at the correct points in time.

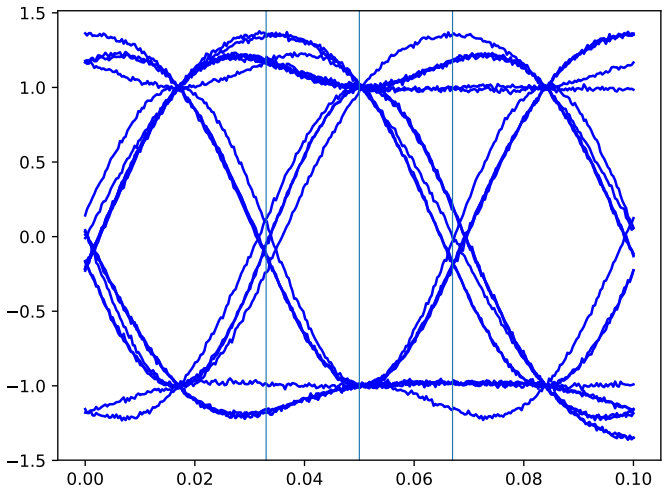


Figure 2: Eye diagram of a low noise QPSK signal

3.2 Multipath propagation

A phenomenon that is quite common in terrestrial communications is multipath propagation. What this means is that a signal that was transmitted once reaches the receiver multiple times, delayed and attenuated by differing amounts.

Figure 3 shows a simplified multipath propagation example. In this example the transmitter does not have a direct line of sight to the receiver, due to an object in the middle of the scene, so signal rays may not propagate directly between transmitter and receiver. Instead signals that reach the receiver do so by reflecting off of other objects in the scene.

In the depicted scenario there are two macro paths by which signals from the transmitter may reach the receiver, these macro paths consist of many micro paths. The effects of the macro paths are modeled as a combination of the micro paths it consists of. The macro path resulting from the reflection with the object on the top is longer than the other macro path, resulting in this signal being delayed and attenuated by a larger amount.

Figure 4 shows the effect of the two delayed signals superimposing on one another, on the same signal as shown in Figure 2. The signal arriving via the longer transmission

[1] This is a simplified formulation of the second Nyquist criterion

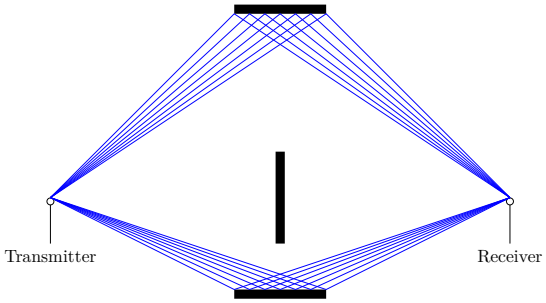


Figure 3: A Multipath scenario without a direct line of sight

path is delayed by 0.9 symbol durations and has a magnitude of 10 %, 30 %, 50 % and 70 % of the original signal amplitude, respectively.

With increasing amplitude of the superimposed signal the optimal sampling points and zero crossings used for clock recovery become less and less obvious. A simple method to reduce the negative effects of multipath propagation is to increase the symbol durations, reducing the relative shift between the differently delayed signals, leading to a less distorted output.

But doing so does obviously reduce the data rate, limiting the overall data throughput.

3.3 Multi-carrier transmission

As was already hinted at in the previous sub-section a reduction in symbol rate helps mitigating the negative effects of multipath propagation but also leads to reduced datarate.

A second positive effect of reducing the symbol rate, in addition to increasing multipath resilience is a reduction of the occupied bandwith. Halving the symbol rate leads to half the frequency range being occupied by the modulated signal.

If one were to fill this newly freed bandwith with a second signal of equal bandwith without degrading the original signal one could transfer the same datarate while utilizing the same bandwith as in the single-carrier case modulated at twice the symbol rate, while still increasing the resilience to multipath propagation.

In order to not degrade one another the two parallel sub-streams have to be modulated onto orthogonal carrier frequencies. One way of performing this modulation onto orthogonal frequencies is by using an inverse fast fourier transformation (iFFT), the inverse operation of the fast fourier transformation. A chunk of N symbols s_i to be modulated onto respective carriers are used as an input vector to the iFFT. The iFFT then synthesizes the symbols onto N orthogonal carrier frequencies as shown in Equation 1 [2].

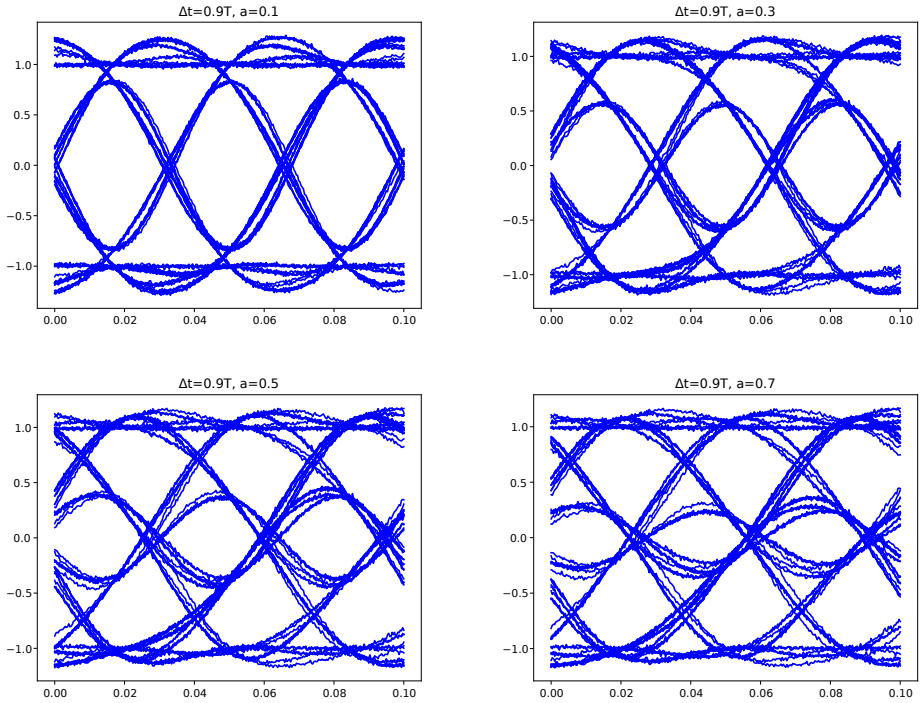


Figure 4: Eye diagrams

$$x(k) = \sum_{i=0}^{N-1} s_i \cdot e^{2\pi j \cdot \frac{ik}{N}} \tag{1}$$

The resulting time-domain signal can then be modulated onto a high frequency carrier to be sent by a transmitter.

This scheme of synthesising symbols onto orthogonal frequencies, usually using an iFFT, and sequentially transferring the output chunks is commonly called orthogonal frequency-division multiplexing (OFDM).

3.4 Cyclic prefix

So far it was demonstrated that OFDM is a modulation scheme that is useful to transfer high datarates over channels where multipath propagation would degrade the integrity of a single-carrier scheme.
There are however still some issues to consider when using OFDM over a multipath channel.

If one were to just sequentially output the OFDM-symbols generated in the previous chapter back-to-back one would encounter multipath-effects as shown in figure 5. In

the diagram the OFDM symbols -7 to 0 are received over two paths with different delays and equal attenuation. At the receiver the two signals will superimpose, leading to interference between adjacent symbols.

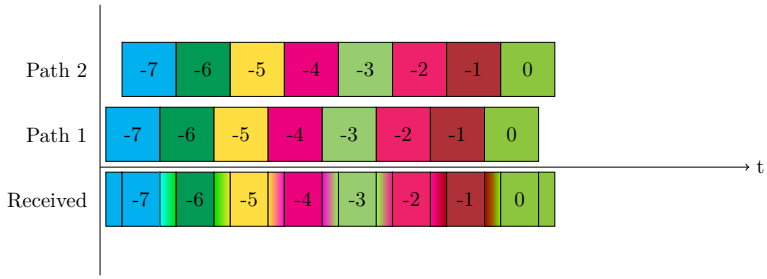


Figure 5: OFDM multipath propagation without guard interval

To prevent interference between subsequent symbols a guard interval has to be introduced to separate them, as shown in figure 5. The length of the guard interval has to be designed according to the targeted scenario, it should usually be at least as long as the impulse response of the targeted channel, which in terms of multipath propagation is the time difference between the signal being received over the shortest path and over the longest path[2].

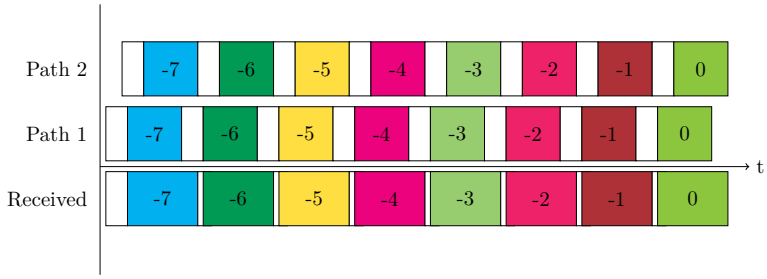


Figure 6: OFDM multipath with guard interval

Due to the cyclic nature of the fast fourier transformation (FFT), used to separate the carriers at the receiver, the guard interval is often filled with the last samples of the following OFDM symbol. Making the effects of the channel effectively cyclic as well. Guard intervals generated in this manner are called cyclic prefix (CP).

3.5 Synchronization using null-symbols

As presented earlier synchronization to a datastream in a single-carrier scheme may be performed as easily as observing zero-crossings in the timedomain to detect symbol transitions. In a multi-carrier system the actual sub-carriers are not available at the receiver prior to fourier-transformation, thus synchronization has to be performed differently.

One method to synchronize a receiver to a continuous stream of OFDM-symbols is the introduction a null-symbol, an OFDM-symbols where the signal power is zero for the entire symbol duration.

An example of such a symbol stream is shown in Figure 7. To synchronize onto the symbol stream the receiver has to monitor the received input power for dips with a length of one symbol duration.

This synchronization scheme is for example used in the DAB+ audio broadcasting standard [3].

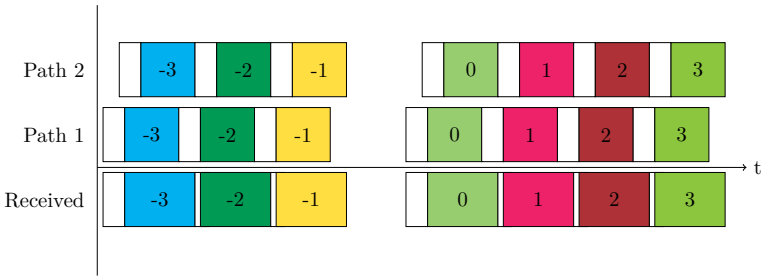


Figure 7: OFDM Synchronization using a null symbol

The start of a frame can then be detected using a method like the one shown in Equation 2, where the signal energy in a window before the detection point is subtracted from the signal energy in a window following the detection point. The output $d(k)$ has a maxmimum when the window before the detection point does not contain any input energy (null-symbol) and the window following the the detection points contains a non-null symbol.

$$d(k) = \sum_{n=0}^{N} \left|x(k+n)\right|^2 - \sum_{n=-N}^{-1} \left|x(k+n)\right|^2 \tag{2}$$

While being easy to implement this method does however have some drawbacks. For one it is not well suited for burst transfers, where the signal power of a received burst is not known a priori, complicating the selection of threshold values. Secondly it does not allow the receiver to estimate the frequency offset between its carrier and the carrier of the transmitter, a previously undiscussed topic, that is for example important when the receiver and transmitter are moving relative to one another, due to the resulting doppler shift in frequency.

3.6 Schmidl and Cox

A more advanced synchronization method is the Schmidl and Cox algorithm that is presented in this sub-chapter. This algorithm was first presented in 1997 by Timothy M. Schmidl and Donald C. Cox [4].

The gist of how a Schmidl & Cox synchronization sequence is introduced into a stream of OFDM-symbols is shown in Figure 8. In this example the Schmidl & Cox sequence consists of a complex sequence of half a symbol length that is sent twice, back-to-back, as shown in Figure 8.

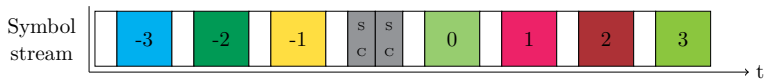


Figure 8: Introduction of a Schmidl and Cox preamble

The detection of the repeated sequence at the receiver works by delaying the input stream by half a symbol length, performing a complex conjuate multiplication with the original stream and averaging over a window with a length of half a symbol. A block diagram of this process is shown in Figure 9.

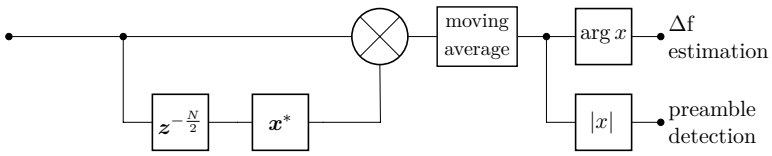


Figure 9: Schmidl & Cox detector block diagram

Figure 10 gives an intuition of what the output stream of this processing chain looks like over time when a synchronization sequence is received.

The simulation consists of a S&C sequence that is constructed from two halves of 512 random samples and a 128 samples long cyclic prefix. The signal is distorted by additive white gaussian noise (AWGN). The signal to noise ratio (SNR) is 0 dB.

The left plot shows the magnitude of the output of the moving average, while the right plot shows its phase.

The absolute value follows a trapezoidal shape. Before and after the synchronization sequence is received the value is nearly zero, due to the result of the complex conjugate multiplication of the input and its delayed version being randomly distributed, leading to it being canceled out in the averaging step.

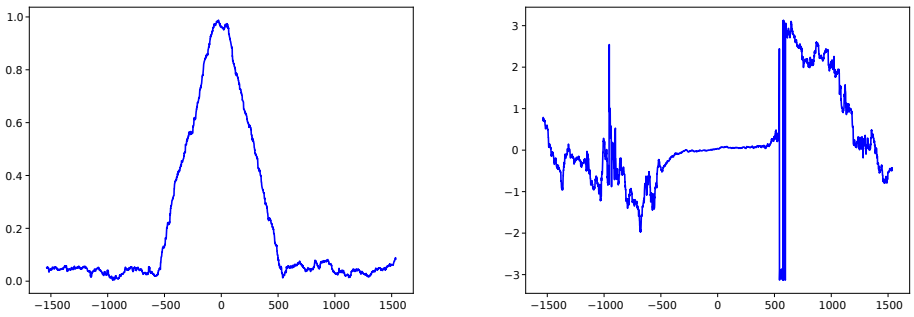


Figure 10: S&C output $|x|$ and $\arg x$ ($\Delta f = 0$)

The conditions leading to the peak in the plot are best described in Equation 3. Where x is the received signal, consisting of the random noise component n and the synchronization sequence s, and the sum represents the moving average in the block diagram, which is performed over half a symbol length N.

As the synchronization sequence is repeated after $N/2$ samples $s_{i-N/2}$ is equal to s_i and as n is randomly distributed its effect will mostly be canceled out by the averaging step.

$$\begin{aligned}\frac{1}{N/2}\sum_{i=0}^{N/2} x_i \cdot x_{i-N/2}^* &= \frac{1}{N/2}\sum_{i=0}^{N/2} (s_i + n_i) \cdot \left(s_{i-N/2} + n_{i-N/2}\right)^* \\ &= \frac{1}{N/2}\sum_{i=0}^{N/2} (s_i + n_i) \cdot \left(s_i + n_{i-N/2}\right)^* \\ &\approx \frac{1}{N/2}\sum_{i=0}^{N/2} s_i \cdot s_i^* = \frac{1}{N/2}\sum_{i=0}^{N/2} |s_i|^2 \end{aligned} \tag{3}$$

The linear slopes leading to the peak are a result of the correlated symbols sliding into and out of the averaging window.

If, due to doppler shift or a difference in the reference clocks, there is a offset frequency Δf superimposed onto the synchronization sequence s' where the sampling frequency is f_s

$$s_i' = s_i \cdot e^{j \cdot 2\pi \frac{\Delta f}{f_s} i}$$

and if the component of the input signal x is ignored the block diagram in Figure 9 can now be described by the new Equation 4.

$$\begin{aligned}\frac{1}{N/2}\sum_{i=0}^{N/2} s'_i \cdot s'^{*}_{i-N/2} &= \frac{1}{N/2}\sum_{i=0}^{N/2}\left(s_i \cdot e^{j\cdot 2\pi\frac{\Delta f}{f_s}i}\right)\left(s^{*}_{i-N/s}\cdot e^{-j\cdot 2\pi\frac{\Delta f}{f_s}\left(i-\frac{N}{2}\right)}\right)\\ &= \frac{1}{N/2}\sum_{i=0}^{N/2} s_i \cdot s_i^{*}\cdot e^{j\cdot 2\pi\frac{\Delta f}{f_s}\frac{N}{2}}\\ &= \frac{1}{N/2}\sum_{i=0}^{N/2}|s_i|^2\cdot e^{j\cdot 2\pi\frac{\Delta f}{f_s}\frac{N}{2}} \end{aligned} \tag{4}$$

Equation 4 shows that the frequency offset between transmitter and receiver can be estimated using the argument $\arg x$ of the complex output of the moving average block in Figure 9.

$$\arg x = 2\pi\frac{\Delta f}{f_s}\frac{N}{2} \implies \Delta f = \frac{f_s}{N\pi}\arg x$$

The magnitude and phase of the output values for a simulation with frequency offset can be seen in figure 11. For the given frequency offset the magnitude is largely unaffected but the phase is now non-zero.

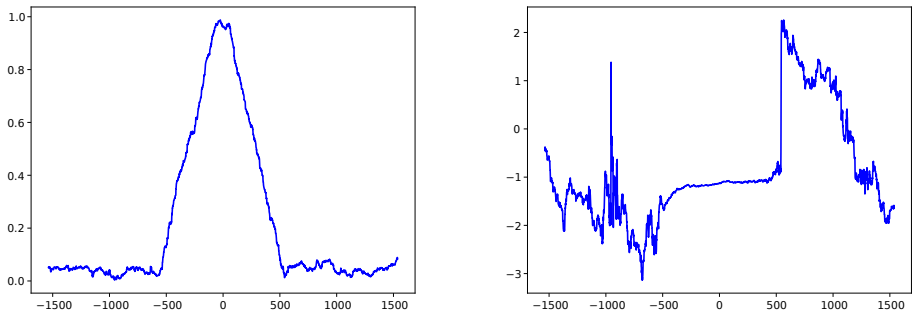


Figure 11: S&C output $|x|$ and $\arg x$ ($\Delta f \neq 0$)

Its ability to estimate frequency offsets in addtion to producing exact timing synchronization between transmitters and receivers makes the S&C algorithm especially useful in multi-carrier scenarios where carriers are spaced tightly and frequency offsets lead to interference between carriers.

3.7 Cross-correlation based preamble detection

To further improve the accurcacy of the synchronization in time a cross-correlation based approach can be used, where the cross-correlation between the input samples and a stored reference-preamble is calculated. A flow diagram and an exemplary output of

a cross-correlation applied to an input stream and a stored preamble are shown in the dotted segment in Figure 12.

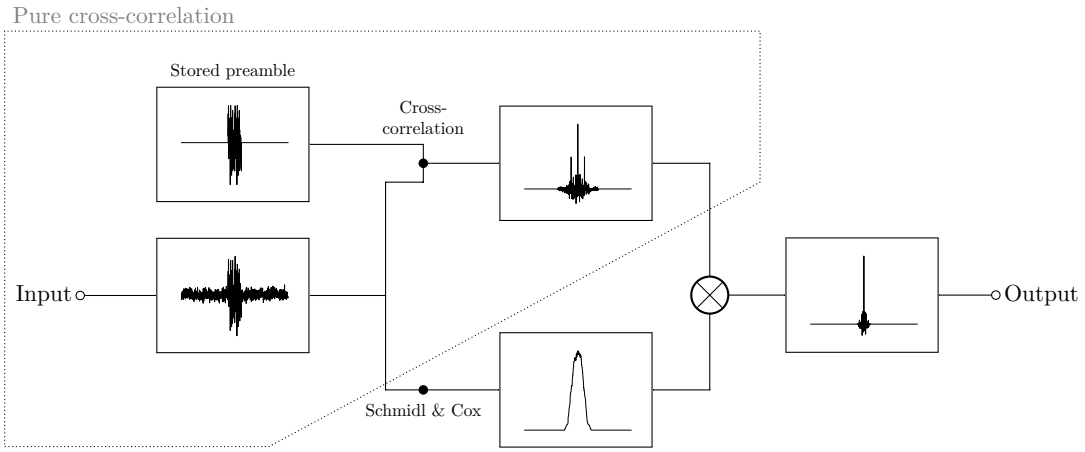


Figure 12: Improved synchronization in time using cross- and auto-correlation

In addition to the central peak the cross-correlation output in Figure 12 also shows two additional side-peaks. These side-peaks occur half a preamble length before and after the main peak and are a result of the repeating nature of the preamble.

A method to suppress these side-peaks is to use the auto-correlation output of the S&C detector and to multiply it with the output of the cross-correlation[5].

This effectively masks out the side-peaks and leads to an output as shown in the rightmost sub-diagram in Figure 12 containing a single sharp peak that can be used to accurately synchronize onto the preamble.

3.8 CAZAC sequences

In order for the cross-correlation based synchronization technique, introduced in the previous subsection, to work optimally the autocorrelation of the preamble should be minimal for time-offsets Δt where $|\Delta t| \neq \frac{N}{2}$.

A class of sequences where this requirement is fulfilled are Constant amplitude zero autocorrelation waveform (CAZAC) waveforms. One class of sequences in this CAZAC class are Zadoff-Chu sequences[6], constructed using Equation 5, where $n \in [0, N_{\mathrm{ZC}})$, $u \in [0, N_{\mathrm{ZC}})$, N_{ZC} and u are relatively prime, $q \in \mathbb{Z}$ and N_{ZC} is the length of the sequence[7].

$$x_{\mathrm{u}}(n) = e^{-\frac{j\pi un(n+1+2q)}{N_{\mathrm{ZC}}}} \tag{5}$$

A property of CAZAC sequences in general is that their autocorrelation is zero for

$\Delta t \neq 0$. So if one constructs a S&C preamble by repeating a CAZAC sequence the autocorrelation is zero for $|\Delta t| \neq \frac{N}{2}$.

Another useful property of CAZAC sequences is the their constant amplitude over time which is useful for example in the design of Automatic gain control (AGC) hardware.

4 GNU Radio

GNU Radio[8] is a free[9] and open source SDR signal processing software-framework. This chapter presents GNU Radio in general, the different interfaces the GNU Radio framework provides and the anatomy of a GNU Radio module.

4.1 GNU Radio Companion

The first point of contact for most new GNU Radio users is usually the GNU Radio companion (GRC), a program that lets users design signal flowgraphs in a graphical user interface. GRC's main interface is shown in Figure 13, the flowgraph presented in this figure is an implementation of the Schmidl and Cox detector block-diagram shown in Figure 9 in the previous chapter.

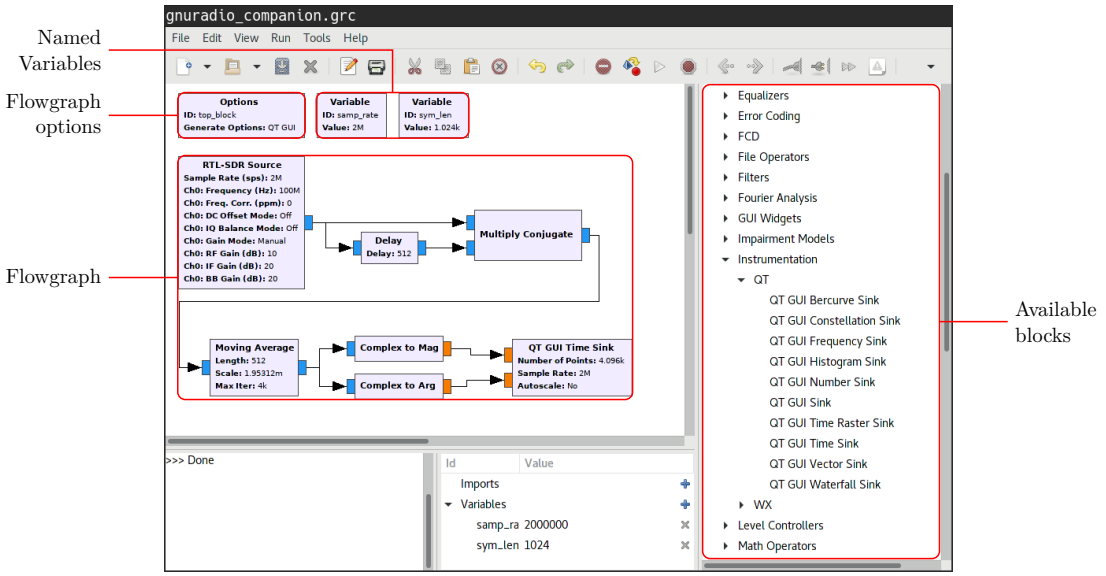


Figure 13: The GNU Radio companion

When a flowgraph containing graphical instrumentation like the `QT GUI Time Sink` in Figure 13 is executed, a window opens containing the live output of these instruments. Figure 14 shows such a window, containing a `QT GUI Time Sink`, a GNU Radio oscilloscope.

GRC works by generating a Python script[2] from the graphical flowgraph representation, based on predefined patterns.

A minimal flowgraph like the one shown in Figure 15, is translated to a Python script much like the one shown in Listing 1, the code in the listing is edited for brevity, as the autogenerated code is usually more verbose.

[2]Python is an interpreted progamming language

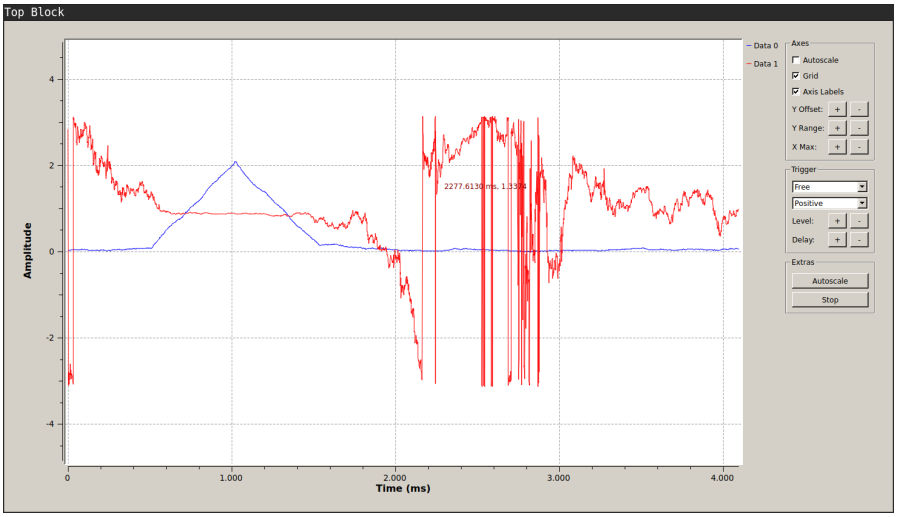


Figure 14: The graphical output of a running flowgraph

When an user selects the `run` option in GRC, a script is automatically generated and executed, so that the user does not have to deal with any Python code while experimenting.

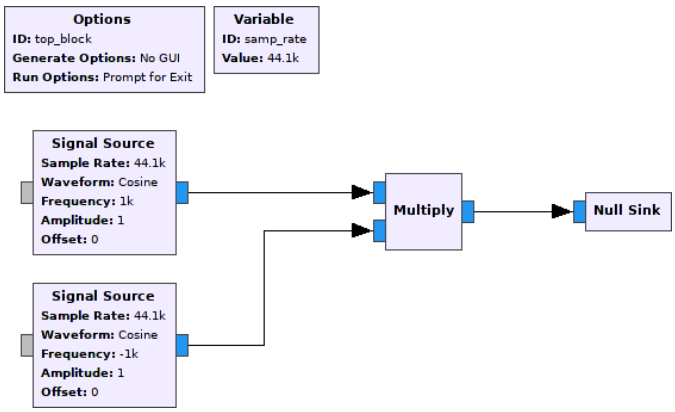


Figure 15: A minimal GRC flowgraph

4.2 GNU Radio native programming interfaces

While GRC is a great tool to experiment with signals, filters and prebuilt decoders, more advanced processing often requires writing custom processing blocks in, more flexible, textual programming languages.

GNU Radio provides two native interfaces to write custom blocks in, C++ and Python

2[3]. The Python interface has the benefit of being easier to program for, while the C++ interface usually provides better processing performance, as the code using it is compiled to native processor instructions. These tradeoffs have to be considered when writing a new GNU Radio processing block.

4.2.1 Python-interface

Listing 1 shows a block written in Python that does not perform its own signal processing but instead connects other blocks to produce a processing chain. Much like a flowgraph designed using the GRC.

The block is derived from the `gr.top_block` base class, as can be seen in line `5`. Blocks derived from this base class do not expose any input or output ports and do not perform their own signal processing, instead they are used to hierarchically connect other blocks to form a processing graph. A typical GNU Radio program will contain one top block that is used to setup the signal processing chain.

There are also some other base classes from which GNU Radio blocks can be derived[10]:

`gr.sync_block` A block that, for every input item, will produce exactly one output item. This can, for example, be a block that, for a stream of complex input symbols, outputs the real magnitude of each symbol.

Or the multiply block, as seen in figure 15, which produces one output symbol for each input symbol present at both inputs.

`gr.sync_decimator` Decimator blocks are quite similar to sync blocks, but will produce output items at lower rate than input items are consumed.

The relative input and output rate has to be fixed at runtime.

One example would be a block that takes two consecutive input items, adds them and outputs them as a single output item.

Decimator blocks are especially useful for sample rate reducing filters.

`gr.sync_interpolator` Interpolator blocks are the rate-increasing pendant to decimator blocks.

`gr.block` The base class for every other class in this list, the most generic block type.

Thise base class does not impose any rate restrictions on the block, allowing for greater flexibility when programming a block derived from this base class.

There is however some more housekeeping work to be done that can no longer be handled automatically when choosing this block instead of the more restricted base classes.

[3]There is some work done in supporting Python 3, but as of the time of writing there is no mainline support for it.

gr.hier_block2 Blocks derived from this base class are pureley hierarchical, they do not perform any signal processing themselve but instead combine other blocks to perform signal processing on the hierarchical block's inputs and output the results on the hierarchical block's outputs.

gr.top_block The top block, as discussed earlier, is a special kind of hierarchical block that does not provide any inputs or outputs.

Instead it contains some additional functionality to execute the contained flowgraph.

The behavior of this block is discussed in in more detail in the apendix, where the interna of GNU Radio's scheduling and buffer management are presented.

The rest of listing 1 performs the equivalent of the flowgraph in figure 15.

Lines 10 to 19 create instances of the blocks used in the flowgraph and provide them with the correct parameters, like sample rate and frequency. Lines 22 to 24 create the connections between the blocks that were previously instantiated. Lines 27 to 31 create an instance of the top block and execute it until the enter key is pressed on the terminal.

```
1  #!/usr/bin/env python2
2
3  from gnuradio import analog, blocks, gr
4
5  class top_block(gr.top_block):
6      def __init__(self):
7          gr.top_block.__init__(self, "Top Block")
8
9          # Instanciate blocks
10         self.sink = blocks.null_sink(gr.sizeof_gr_complex)
11         self.multiply = blocks.multiply_vcc(1)
12
13         self.source_a = analog.sig_source_c(
14             44100, analog.GR_COS_WAVE, -1000, 1, 0
15         )
16
17         self.source_b = analog.sig_source_c(
18             44100, analog.GR_COS_WAVE, 1000, 1, 0
19         )
20
21         # Connect blocks
22         self.connect((self.source_a, 0), (self.multiply, 0))
23         self.connect((self.source_b, 0), (self.multiply, 1))
24         self.connect((self.multiply, 0), (self.sink, 0))
25
26 if __name__ == '__main__':
27     tb = top_block()
28     tb.start()
29     raw_input('Press Enter to quit: ')
30     tb.stop()
31     tb.wait()
```

Listing 1: Using GNU Radio from Python

4.2.2 C++-interface

Most of GNU Radio's backend code is not written in Python but in C++, as it provides lower level access to the computer memory and better computing performance.

A lot of this backend code is acessable from Python thanks to C++-to-Python wrappers that are automatically generated using the SWIG [11] framework.
One example of C++ code for which wrappers are available are the, previously discussed, `gr.*_block` base classes which are actually implemented in C++ but are also useable to derive blocks from in Python.

Accordingly custom blocks can also be implemented in C++, as demonstrated in Listing 2. The block shown is derived from the `gr:sync_block` base class, as can be seen in line `14`, as such it is bound to produce one output item per input item consumed at its inputs.

The input (line `15`) and output (line `16`) signature specifiers specify the number of input and output ports of the block and their type. The simple input signature

```
gr::io_signature::make(1, 1, sizeof(float))
```

breaks down to: "This block has at least one input port '`1,`' and at most one input port '`1,`' an input item consists of a single real number '`sizeof(float)`' ".
Thus the block has exactly one real-valued input port and exactly one real-valued output port.

The actual signal processing for the demonstrated block is performed in the `work` method, starting in line `29`. The work method is passed the following arguments by the GNU Radio scheduler:

`noutput_items` The number of output items the block should write to the output buffers in this iteration, as this is a sync block this is also the minimum number of input items that are available in the input buffers.

`input_items` A list of input buffers. In this case the list only contains a single reference to an input buffer as there is only a single input port.

`output_items` A list of output buffers. Again, in the case at hand, this will only contain a single reference.

In the `for`-loop starting in line `32` every element in the input buffer is squared and written to the corresponding position in the output buffer.
In line `36` the number of items that were actually consumed is returned, in this case it is equal to the number of items the method was requested to produce.

```
1 #include <gnuradio/io_signature.h>
2 #include "square2_ff_impl.h"
3
4 namespace gr {
5   namespace hnez {
6     square2_ff::sptr
7     square2_ff::make()
8     {
9       return gnuradio::get_initial_sptr
10        (new square2_ff_impl());
11    }
12
13    square2_ff_impl::square2_ff_impl()
14      : gr::sync_block("square2_ff",
15                       gr::io_signature::make(1, 1, sizeof(float)),
16                       gr::io_signature::make(1, 1, sizeof(float)))
17    {
18    }
19
20    square2_ff_impl::~square2_ff_impl()
21    {
22    }
23
24    int
25    square2_ff_impl::work(int noutput_items,
26                          gr_vector_const_void_star &input_items,
27                          gr_vector_void_star &output_items)
28    {
29      const float *in = (const float *) input_items[0];
30      float *out = (float *) output_items[0];
31
32      for (int i=0; i<noutput_items; i++) {
33        out[i]= in[i] * in[i];
34      }
35
36      return noutput_items;
37    }
38  }
39 }
```

Listing 2: A simple custom block in C++

5 Schmidl and Cox detector implementation

The GNU Radio blocks implemented as part of this thesis are shown in Figure 16, circled in red.

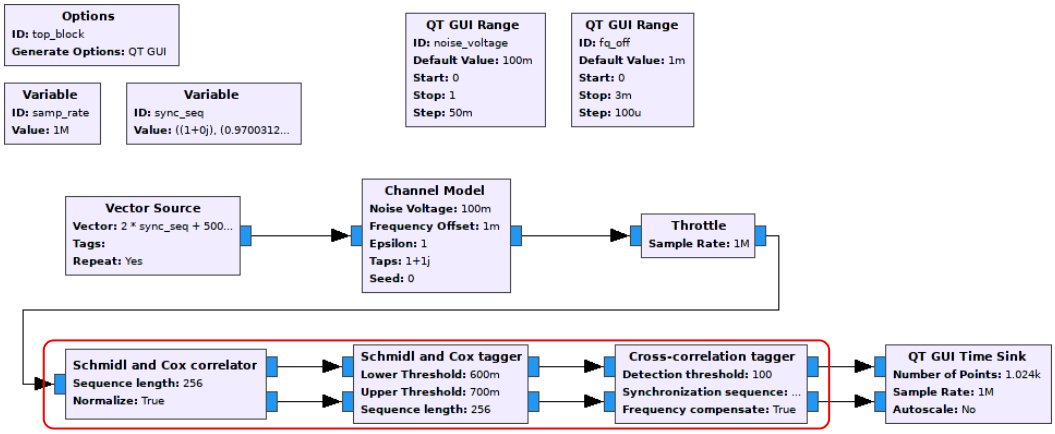
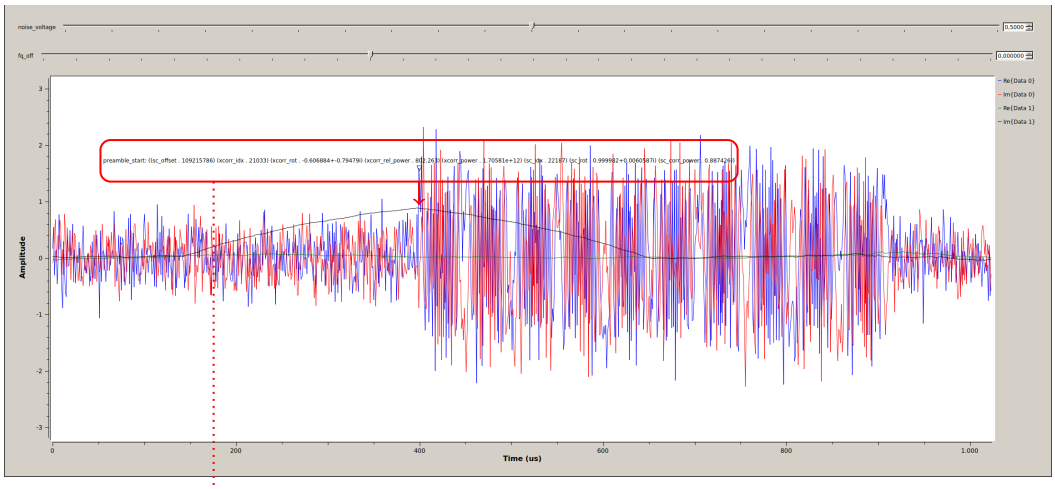


Figure 16: The interface of the S&C detectors

The purpose of these blocks is to take a stream of input signals and to add annotations to this stream whenever a synchronization-preamble is detected. Figure 17 shows an example output of the detector blocks, where an an annotation is placed right at the start of a synchronization sequence.



`preamble_start` tag

Figure 17: The output of the S&C detectors

As performance is a major design goal in the implementation of these blocks, all of them are written in C++ and some optimizations are used to make them perform well

on a modern CPU, like utilizing SIMD instructions that allow performing the same operation on multiple values using a single machine-instruction.

5.1 Schmidl and Cox correlator

The functionality of the Schmidl and Cox correlator block is shown in Figure 18. The block takes one input signal, calculates the correlation as described in the Schmidl and Cox chapter and normalizes it with the average input power in the analyzed duration.

It also outputs a delayed version of the input sequence so that a peak at the correlation output corresponds to the start of a preamble and not its end.

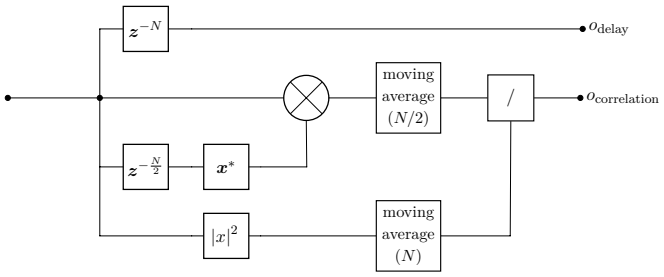


Figure 18: Block diagram of the Schmidl and Cox correlator

5.2 Schmidl and Cox tagger

The next step in the detection pipeline is to find peaks in the output of the Schmidl and Cox correlator and to put a tag[4] on them.

Depending on the input noise-level the correlation output may be quite noisy. To prevent local fluctuations from being detected as multiple start-of-frame preambles and thus producing bursts of start-of-frame tags the S&C tagger applies a user-defined amount of hysteresis to the detection process.

An illustration is shown in Figure 19. The upper and lower detection thresholds are provided by the user as a parameter to the block.

Whenever the upper threshold is exceeded a detection window starts. The detection window ends when the input falls below the lower threshold. Exactly one tag will be placed in this detection window at the position of the highest magnitude. The sequence length parameter places a constaint onto the maximum length of this detection window, which is necessary as the block needs to keep a copy of the samples inside the window in order to be able to output a tag at the correct location.

[4] A tag in GNU Radio is a an annotation to a datastream that attaches metadata to a specific point in time

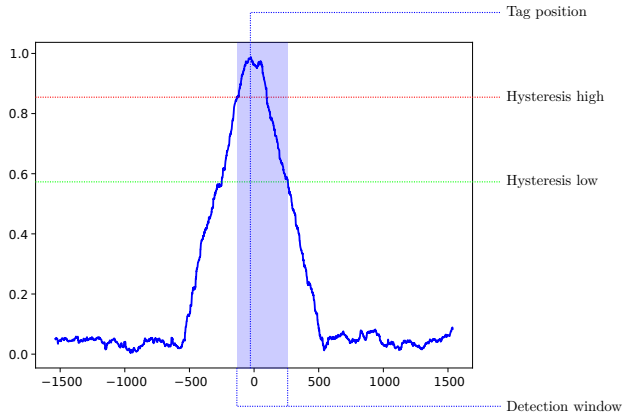


Figure 19: Detection levels of the S&C tagger

5.3 Cross-correlation tagger

The last detection step may be used to increase the synchronization-in-time accuracy of the preamble detection and to prevent triggering on unknown preambles.

It takes as inputs the sample stream that was tagged with start-of-frame tags by the S&C tagger and a reference preamble.

Whenever a start-of-frame tag is encountered in the input stream, a chunk of samples is extracted around the tag and the cross-correlation of this chunk and a preamble stored for reference is computed.

If the maximum cross-correlation is below a preset threshold the tag that triggered the check is not forwarded to the next block.

If the threshold is exceeded the tag is moved to the sample $\hat{k}_{\text{new}}$ where the cross-correlation $c(k)$ is at maximum.

$$\hat{k}_{\text{new}} = arg\,max\left(\left|c(k)\right|\right)$$

A more detailed view of the inner workings of the cross-correlation tagger is shown in Figure 20.

The computation of the cross-correlation is, as shown in the block-diagram, implemented using the fast fourier transformation and its inverse. This method provides superior computation speed when dealing with large chunks of samples, when compared to the naive implementation of the cross-correlation [2].

The extracted chunk of samples is zero-padded to a power-of-two size. This has two effects: for one the FFT algorithm performs best when operating on chunks that are a

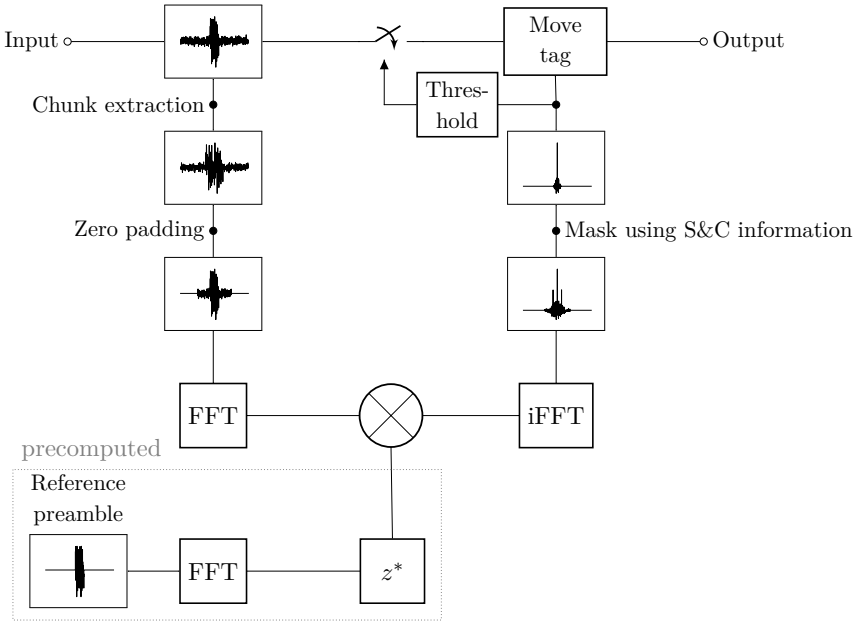


Figure 20: In-depth operation of the cross-correlation tagger

power-of-two in length. Secondly the zero padding effectively turns the circular cross-correlation calculation performed by the FFT into a non-circular cross-correlation [2].

After the calculation of the iFFT the resulting cross-correlation contains two side lobes in addition the desired maximum. These side lobes are a result of the repeating nature of the preamble, as discussed in subsection 3.7.

To mask out these side lobes the cross-correlation is multiplied onto the the correlation output of the Schmidl and Cox correlator, which has a triangular pattern with a peak around the desired maximum, as can be seen in Figure 19.

6 Results

In order to analyze the performance of the GNU Radio blocks discussed earlier a set of testcases were devised.

The metrics that the testcases were designed to test were, firstly, the precision of the tag-placement in the time domain in the presence of different disturbances, secondly, the precision of the carrier frequency offset (CFO) estimation and finally the effective computational complexity.

6.1 Start-of-frame detection

The first testcase was designed to measures the quality of synchronisation when the signal is disturbed by, noise, frequency offsets or frequency selective channels. The respective results are shown in Figures 21 to 24.

All of these figures compare three different synchronization techniques:

Schmidl & Cox: The Schmidl and Cox correlator and tagger, as discussed in the previous chapters. The cross-correlation tagger component was not used in these tests.

The synchronization sequence used was a Zadoff-Chu sequence with the parameters $N_{\mathrm{ZC}} = 256$, $u = 47$, $q = 13$.

Burst & Silence: A custom GNU Radio block that detects a burst in input power followed by an equally long period of silence by calculating the average signal power in two consecutive windows and subtracting the energy in the later window from the energy of the earlier window.

The noise floor affects both windows in a roughly equal manner and is thus canceled out.

Frequency Sweep: Another custom GNU Radio block that mixes the input signal with a repeating frequency sweep.

The synchronization sequence consists of a complementary frequency sweep, such that the mixing of both results in a constant frequency output.

The constant frequency results in a constant phase difference between consecutive samples that is utilized for detection.

The algorithms were tested under these common conditions:

- The preambles were followed by random OFDM symbols
- Each algortihm used a preamble of 512 symbols
- The energy of the whole preamble was equal for each algorithm and equal to the average energy of the OFDM symbols

6.1.1 AWGN Channel

Figure 21 shows the effect of different SNRs on the output of the synchronization methods. The histograms show the distribution of tag positions relative to the actual position of the synchronization sequence. For a perfect synchronization method the histogram would show a single peak in the "0" bin, indicating that every synchronization sequence was detected without even a single sample of offset.

A particulary bad method would show a single peak in the "missed" bin, indicating that the sequence was missed by more than ten samples or not detected at all.

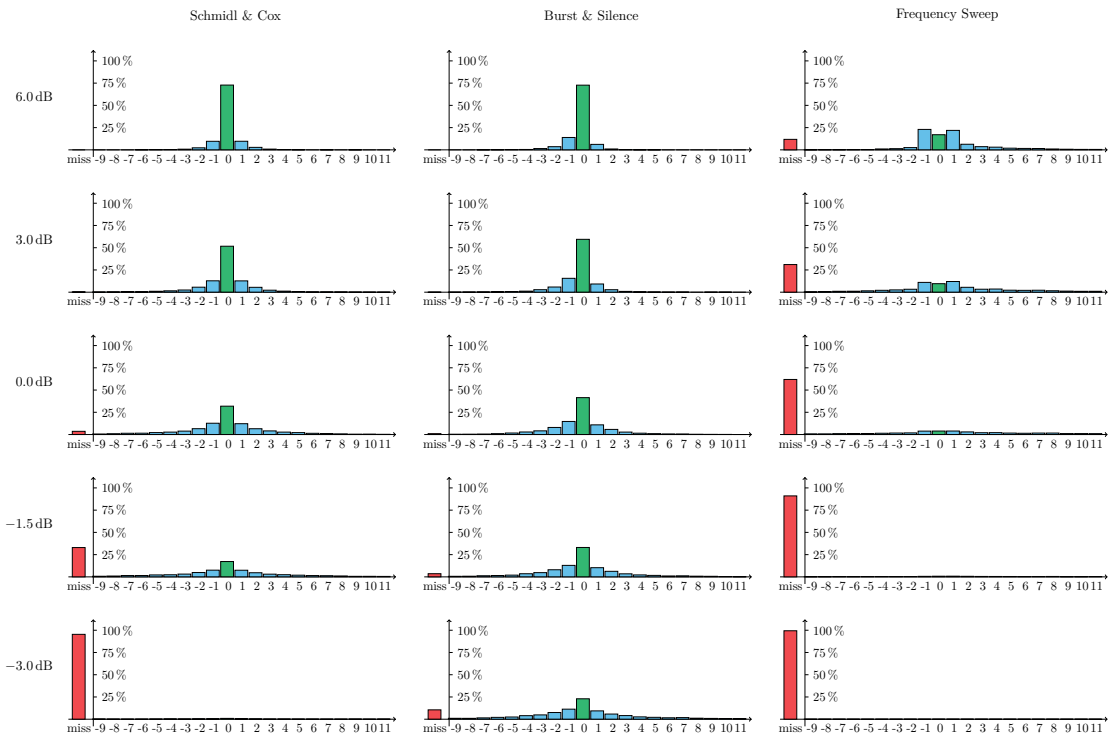


Figure 21: Detection time distributions for different SNRs

For high SNRs all of the tested methods produced usable results - tagging most preambles to within a few samples.

For decreasing SNRs the results obtained by the frequency sweep based method degraded rather quickly, already missing most preambles at an SNR of just 0 dB while the other methods still provided reliable output.

The best method in this test was the rather simple power-level based Burst & Silence detector, dominating over the more complex Schmidl & Cox detector. It should be noted however that the detection hysteresis levels for all methods were tweaked to produce a minimal amount of false positives at low noise levels.

When tweaking the parameters for different design criteria e.g. a minimal amount of false negatives the Schmidl & Cox and Burst & Silence show similar performance while

the frequency sweep based approach always provides inferior results.

6.1.2 Frequency offset

The second test was designed to test the effect a frequency offset between transmitters and receivers has on the detection accuracy of the respective algorithms.

In this test the baseband signals containing the synchronization sequences were mixed up using a signal of $n\%$ the baseband nyquist frequency resulting in a cyclic frequency shift. The results are shown in Figure 22.

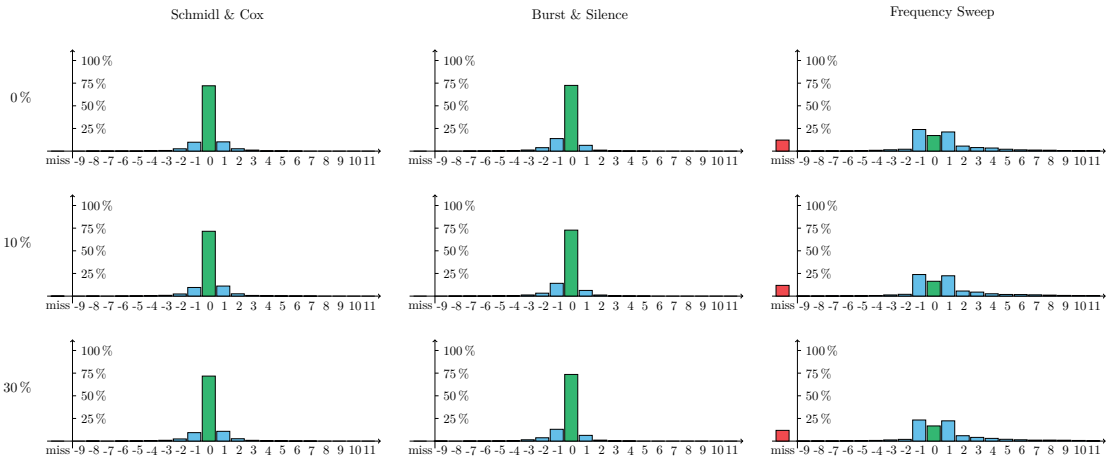


Figure 22: Detection time distributions for different frequency ofssets

Figure 22 shows that none of the methods express any dependency on cyclic frequency shifts. This is to be expected as none of the methods actually use the absolute frequency information contained in the preambles for timing synchronization.

6.1.3 Frequency-selective channel

A major application for multi-carrier based techniques are transmissions over frequency-selective channels like reflection-heavy urban areas.

Good performance in these use-cases is thus also a requirement for multi-carrier synchronization systems.

To characterize the performance of the synchronization methods at hand a selection of increasingly hostile channels were constructed by designing different impulse responses.

The frequency responses of these channels are shown in the linear plots in Figure 23.

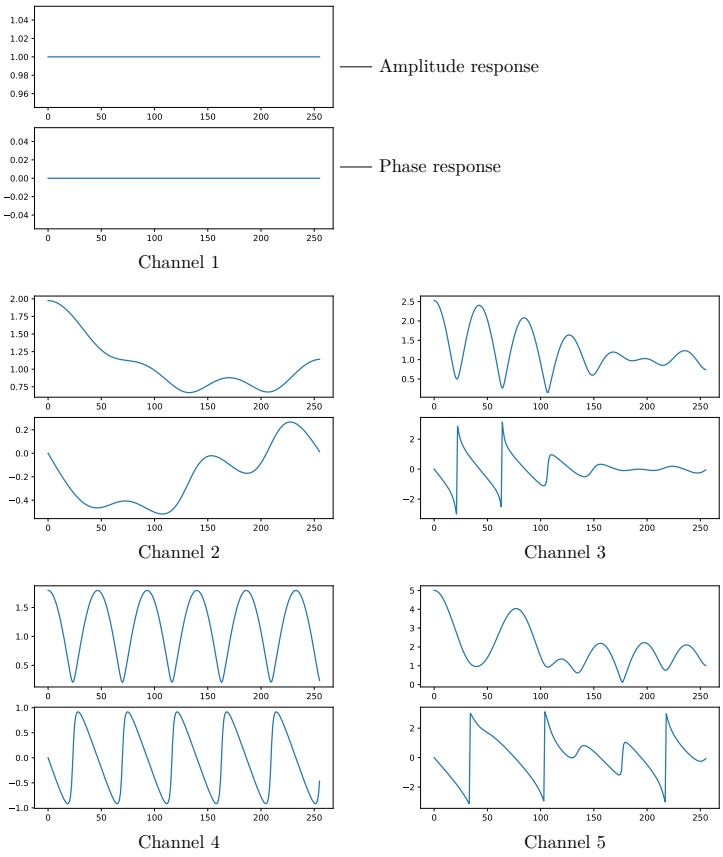


Figure 23: Channel models used in the simulation

The first channel was completely flat, the impulse response consisted of only a single pulse.

The second channel roughly resembled the behavior of an indoor communication with a direct line of sight. The impulse response consisted of a large peak, representing the direct line of sight, followed by an exponentional falloff and a highly attenuated reflection pulse.

The remaining channels were intentionally designed to be hostile by placing large peaks in the impulse response over a wide span of time.

The results for the different algorithms, when used over these channels, are shown in Figure 24.

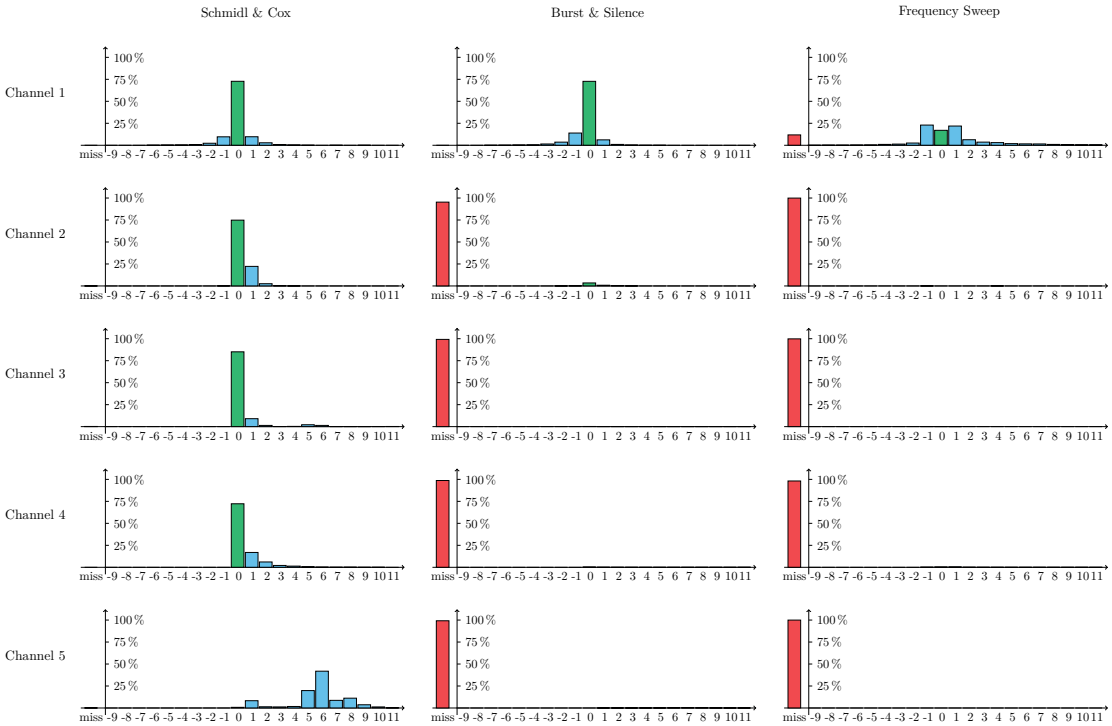


Figure 24: Detection time distributions for different channel models

Both custom synchronization techniques failed dramatically when used over frequency-selective channels.

In the case of the Burst & Silence detector verifying the plausibility of these results is quite simple as any additional non-zero value in the impulse response in addition to the main pulse leads to parts of the burst-half of the preamble "bleeding" into the silent-half, reducing the difference in energy between the windows.

6.2 Carrier frequency offset estimation

In addition to the previously analyzed synchronization in time the S&C algorithm also provides an estimation of the difference in carrier frequency between receiver and transmitter, the carrier frequency offset (CFO).

As S&C was the only method presented in this thesis providing a means of estimating the CFO its performance was characterized separately.

The test was designed as an actual-hardware test and used common audio hardware. A speaker and a microphone[5] were placed on opposite sides of a small room. An image showing the components used is shown in Figure 25. The speaker periodically

[5] Due to the lack of an actual microphone a headphone, connect to the microphone jack of a sound card, was used as a crude replacement

broadcasted S&C preambles consisting of two 256 symbols long sequences, occupying a bandwith of 4410 Hz around a carrier of 4000 Hz. These preambles were received by the microphone and mixed down by an LO-frequency between 3985 Hz and 4015 Hz, the S&C algorithm was then used to estimate this carrier-frequency offset of ±15 Hz.

Figure 25: Main hardware components used in the CFO test

The playback volume was adjusted to the lowest level that still provided reliable preamble detection at the receiver.

The results are shown in Figure 26, which depicts the estimated LO-frequencies over the actual LO-frequencies used to mix down the signal. The measurements are depicted as blue dots while a green line shows the identity mapping between estimated and actual frequencies.

An obvious first observation is that for frequency offsets of more than about ±8 Hz the estimated frequencies were widely off from the expected values.

This is a direct result of using the output phase of the S&C algorithm for frequency estimation, as the phase is periodic in $(-\pi, \pi]$ leading to ambiguities in the phase-to-frequency mapping.

The maxium detectable frequency offset is thus a criterium in the design of a S&C based system.

To analyze the CFO performance, assuming unambiguous frequency offsets, the same measurements as in Figure 26 where plotted again, this time showing the error in frequency estimation plotted over the actual carrier frequency. The resulting diagram is shown in Figure 27. The values were clipped to a range of $[-2, 2]$.

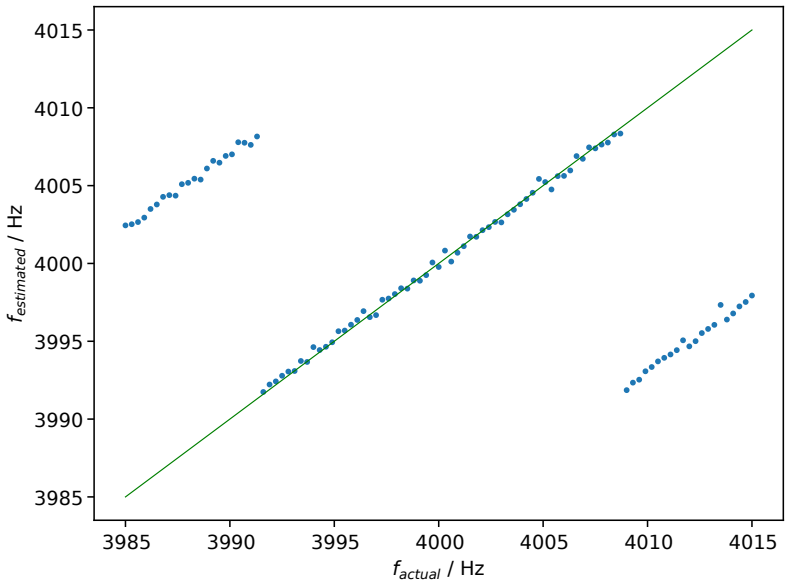


Figure 26: Estimated f_c versus actual f_c

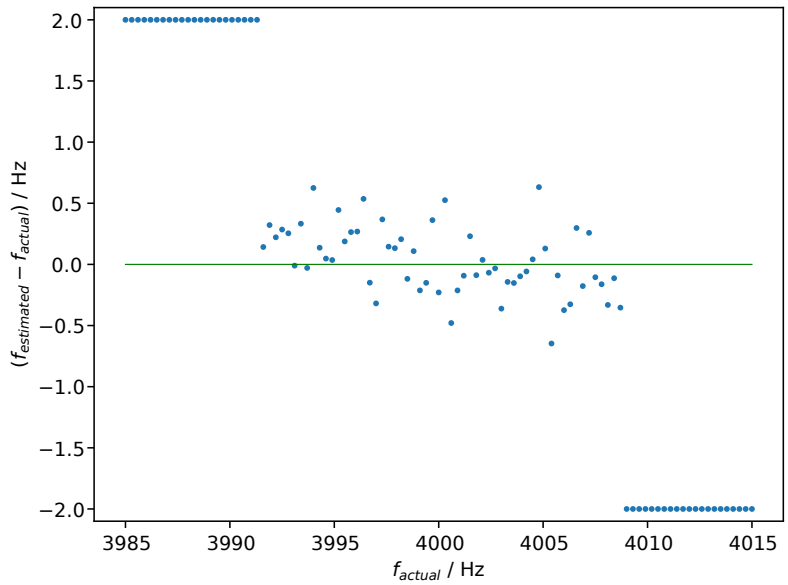


Figure 27: Estimation error Δf_c versus f_c

The figure shows that the carrier frequencies were estimated to within a range of $\pm 0.7\,\mathrm{Hz}$. The carrier spacings in a multi-carrier system would have to be designed to tollerate these residual frequency offsets.

6.3 CPU load

A major design goal of the S&C implementation presentated in this thesis was processing speed on general-purpose computing hardware. To facilitate this goal various optimizations were made in the software implementation.

To test the processing speed of the S&C correlator and -tagger blocks a custom GNU Radio block was implemented that repeatedly outputs a preset sequence of symbols over and over, while keeping track of the number of samples produced per unit of time.

The overall processing speed of a GNU Radio program is determined by the slowest block in the flowgraph, this is usually a hardware device that is limited in the number of samples it produces by its sample rate.

In the absence of hardware devices, meaning a pure simulation scenario, the flowgraph will run as fast as possible, only limited by the processing speed of the computing hardware.

For this test the custom speed measuring block introduced earlier was tasked to output spurious S&C sequences disturbed by gaussian noise.

This measuring source was then connected to the S&C correlator block in series with the S&C tagger block. The outputs of the tagger were dumped using null sink blocks which do not perform any processing on their inputs and do not produce any output. A block diagram of this setup is shown in Figure 28.

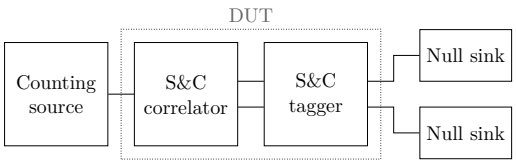


Figure 28: Benchmark setup

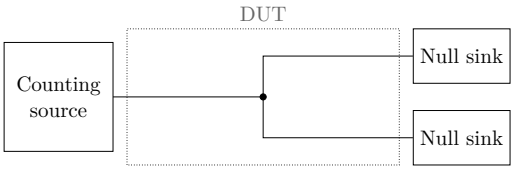


Figure 29: Baseline setup

This program was then executed on two reasonably recent computers, containing central processing units (CPUs) designed with different use-cases in mind.

The first CPU was an AMD Ryzen 5 model intended for the mid-to-high range Desktop PC market where power consumption is a minor concern. It provides, among others, the AVX2 group of instructions that allow processing multiple values in a single processor instruction (SIMD).

The second CPU was an Intel Atom model intended for very low power laptop and

tablet computers. To reduce the power consumption and die-area of this CPU only the SSE to SSE4.2 line of SIMD instructions were implemented instead of the more powerful AVX2 SIMD instruction present in the Desktop CPU.

The samplerates achieved in this test are shown in Table 1 in the "Benchmark samplerate" column.

Processor	Benchmark samplerate	Baseline samplerate
AMD Ryzen 5 - 1600 6 CPU cores 3.4 GHz Linux 4.13.12-1-ARCH	$114 - 133\,\mathrm{MS\,s^{-1}}$	$190 - 198\,\mathrm{MS\,s^{-1}}$
Intel Atom x5-Z8350 4 CPU cores 1.6 GHz Linux 4.13.12-1-ARCH	$19 - 20\,\mathrm{MS\,s^{-1}}$	$33 - 34\,\mathrm{MS\,s^{-1}}$

Table 1: achievable samplerates for different processors

The table also contains a "Baseline samplerate" column. This column contains the samplerates obtained when the flowgraph is executed without the S&C blocks, leaving only the measuring source and the null sinks to dump the output samples, as shown in Figure 29.

This baseline figure was used to verify that the processing speed is actually limited by the S&C blocks, as the baseline samplerates were substantially larger than the bechmark rates it is relatively certain that the processing speed was indeed limited by the S&C blocks and the measured performance was a characteristic of these blocks.

Another observation the benchmarking allowed can be seen in Figure 30: only a small number of the available processor cores were actually utilized by the benchmarking program (the processor utilization is shown in the topmost 12 bargraphs).

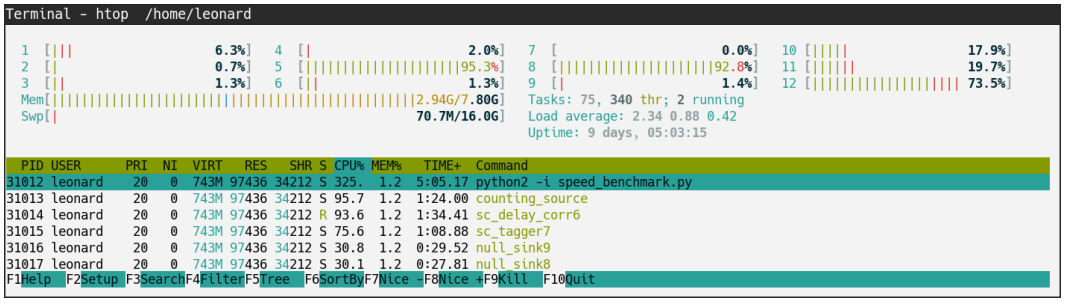


Figure 30: Processor load while executing the benchmarking flowgraph

This is due to the correlator block and the tagger block running in a single thread each, allowing them to utilize at most a single processor core at once, respectively.

While utilizing more CPU cores could possibly result in a higher samplerate, the limitation to at most two cores used by S&C components can also be seen as a feature, whereby the remaining cores may still be used for other processing blocks without negatively impacting the overall archievable samplerate.

7 Conclusion

The main goal of the project accompanying this thesis was to take the abstract formulation of the Schmidl and Cox synchronization algorithm and to transfer it into an actually useable software product.

The result of this work is XFDMSync[1], a set of GNU Radio processing blocks to be used for multi-carrier synchronization tasks.
Writing this piece of software imposed some interesting challenges not commonly seen in academic software development, like concentrating on interoperability with an existing software framework and spending a large proportion of development time on optimizing for maximum data throughput.

As a result the blocks should be usable by the general GNU Radio userbase to synchronize onto reasonably high-datarate data-streams.

Another aspect of this project was to quantify the performance of the synchronization blocks at hand. To perform these test a collection of testcases were implemented to characterize the time and frequency aspects of synchronization as well as the computational throughput.
The implemented components fared well in all of these tests when compared to more naive synchronization methods and provided sufficient throughput even for demanding applications like realtime WiFi frame detection.

Some aspects that were not explored in depth and could be targets of future research work were the optimization of detection parameters like threshold levels for specific applications or hardware tests on actual software-defined radio devices.

8 Appendix - GNU Radio internals

As the code samples in Listing 1 and Listing 2 show, GNU Radio blocks do not have to perform any buffer management themselves to handle the streaming input and output ports.

The buffer management and scheduling of when to execute the processing block is instead handled internally by GNU Radio. As the processing blocks are only allowed to communicate with other blocks using the previously discussed input and output ports and a, not yet discussed, message passing interface they can also be executed concurrently on multiple CPU cores by the GNU Radio scheduler without having to fear race conditions on shared data.

8.1 Buffer management and scheduling

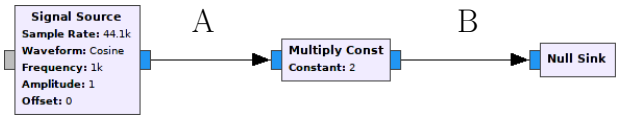


Figure 31: A minimal flowgraph to demonstrate buffer management

For a flowgraph as shown in figure 31 GNU Radio allocates two internal buffers, one for the "Signal Source" block to write into and the "Multiply Const" block to read from and one for the "Multiply Const" block to write into and the "Null Sink" block to read from [12].

In the following diagrams the first buffer is called "Buffer A" and the second buffer is be called "Buffer B". The buffers in GNU Radio are organized as circular buffers, this means that size of the occupied memory region remains constant during execution and that there is one pointer into the memory region where the next values are to be written (`write_ptr`) and one or more pointers pointing to the next value to be read (`read_ptr`).

If both pointers point to the same position in memory the buffer is empty. If a pointer is incremented past the end of the memory region it wraps around to the beginning.

Figure 32 shows the initial state of the two buffers in the example demonstrating the buffer handling of a flowgraph like the one in Figure 31, right when the execution of the flow graph starts. In both buffers `write_ptr` and `read_ptr` point to the same location, the beginning of the buffers, so both buffers are considered empty.

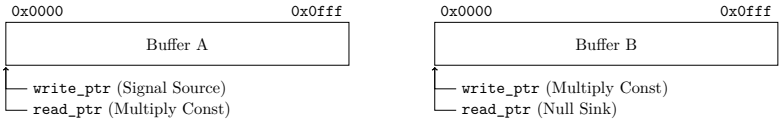


Figure 32: Initial state of the two buffers

The GNU Radio scheduler tries to keep all the buffers filled and will execute blocks until every buffer is filled. Blocks that depend on input to process like the "Multiply Const" or "Null Sink" block in the example can not be executed unless there is data in their input buffers.

To determine which block to execute next the scheduler asks every block, which does not have a completely filled output buffer, to estimate how many input items it needs to fill its output buffers.

For synchronous blocks, like the "Multiply Const" block, this corresponds to a simple 1:1 mapping, as the block needs n input values to produce n output values.

Blocks without output ports, like the "Null Sink" block, can be scheduled whenever there is data in their input buffers. Blocks without input ports, like "Signal Source" can always be scheduled, but might not actually produce the desired number of output values, for example when a hardware device did not produce enough samples.

In the example the scheduler asks the "Multiply Const" block how many input items it needs to fill its output buffer of length n_0, the block answers with n_0 items. The scheduler can not provide that many input items, as the input buffer is empty. The scheduler will then sucessively halve the number of output items it requests $n_{i+1} = n_i/2$ until it determines that it can not fulfill the blocks input requirements ($n_{i+1} = 0$).

The scheduler then determines that the "Signal Source" block can be executed, as it does not depend on any inputs. The "Signal Source" is executed and produces some output values that it puts into "Buffer A", it is assumed that the block is not able to fill the complete buffer. The states of the buffers after the "Signal Source" block is executed is shown in figure 33.

The diagram shows some valid data in "Buffer A" that was not yet consumed, "Buffer B" remains empty.

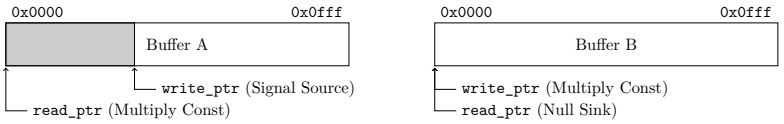


Figure 33: Buffer states after "Signal Source" was executed

In the next scheduling round “Signal Source” has data available in its input buffer to be processed. Figure 34 shows the buffer states after “Signal Source” was executed. “Buffer A” is completely drained and “Signal Source” has written some output items to “Buffer B”.

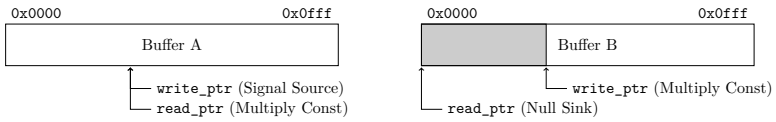


Figure 34: Buffer states after “Multiply Const” was executed

Once there is data in “Buffer B” the “Null Sink” block can be scheduled and it consumes all the available input samples. The state of the buffers is shown in figure 35.

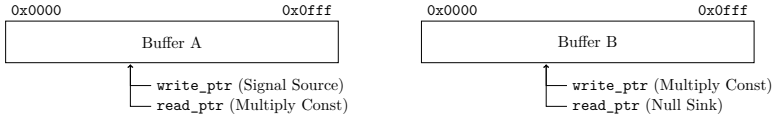


Figure 35: Buffer states after “Null Sink” was executed

The scheduling decisions in this chapter are choosen to be as illustrative as possible. In an actual flowgraph there are further considerations to be made, like the possibility of scheduling multiple blocks concurrently on machines with multiple CPU cores or buffers with multiple readers or multiple block producing data in the same scheduling cycle.

8.2 Efficient circular buffers

As hinted at in listing 2, when a block is asked to fill or read a buffer it is passed a region in that buffer indicated by its starting address and size.

An illustration is shown in figure 36. The block is asked to write `size` elements into the buffer, starting at `write_ptr`.

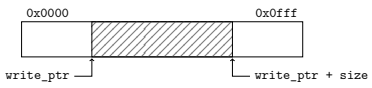


Figure 36: Filling a memory region

A problem arrises when the block is asked to write or read at the end of the buffer. As the buffer is circular reads and writes beyond the end of the buffer have to be wrapped to addresses at the start of the buffer.

Figure 37 shows a situation where a write to the passed region would lead to an overflow to memory not belonging to the buffer.

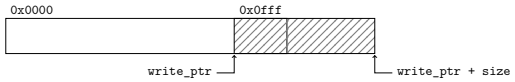


Figure 37: Overflowing the memory region

A possible workaround is to execute the block twice whenever the buffer boundary is crossed, as shown in 38.

This leads to worse performance, as the block might have to perform some initialization on every execution. Further problems arrise when the size of the buffer is not a multiple of the size of an item.

Figure 38: Scheduling twice to prevent overflowing

GNU Radio uses another approach, it instructs the operating system to map the buffer twice into the address space of the program, right after one another [13].

This leads to a memory layout like in figure 39. Reads and writes to addresses `0x1000 - 0x1fff` in the program are redirect to the same physical memory as reads/writes to addresses `0x0000 - 0x0fff` by the computers memory management unit (MMU).

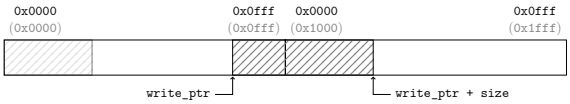


Figure 39: Mapping the same memory region twice to prevent overflowing

The scheduler can thus just pass the start address and the length of the data to write/read to the block without having to explicitly handle the wrapping.

Glossary

AGC Automatic gain control. 14

AVX2 Advanced Vector Extensions. 32, 33

AWGN additive white gaussian noise. 10

CAZAC Constant amplitude zero autocorrelation waveform. 13, 14

CFO carrier frequency offset. 25, 29, 30

CP cyclic prefix. 8, 10

CPU central processing unit. 22, 32–34, 38

DAB+ digital audio broadcasting. 9

FFT fast fourier transformation. 6, 8, 23, 24

GRC GNU Radio companion. 3, 15–17

iFFT inverse fast fourier transformation. 6, 7, 24

MMU memory management unit. 39

OFDM orthogonal frequency-division multiplexing. 3, 7–10, 25

QPSK Quadarture phase shift keying. 4, 5

S&C Schmidl and Cox. 2–4, 9–15, 21–25, 29, 30, 32–35

SDR software-defined radio. 3, 15

SIMD single instruction multiple data. 22, 32, 33

SNR signal to noise ratio. 10, 26

SSE4.2 Streaming SIMD Extensions. 33

References

[1] URL: https://github.com/hnez/XFDMSync (visited on 11/28/2017).

[2] Kristian Kroschel Karl-Dirk Kammeyer. *Digitale Signalverarbeitung*. Springer Vieweg, 2012. ISBN: 978-3-8348-1644-3.

[3] URL: http://www.etsi.org/deliver/etsi_en/300400_300499/300401/02.01.01_60/en_300401v020101p.pdf (visited on 11/30/2017).

[4] Donald C. Cox Timothy M. Schmidl. "Robust Frequency and Timing Synchronization for OFDM". In: (1997).

[5] A. B. Awoseyila, C. Kasparis, and B. G. Evans. "Improved preamble-aided timing estimation for OFDM systems". In: *IEEE Communications Letters* 12.11 (2008), pp. 825–827. ISSN: 1089-7798. DOI: 10.1109/LCOMM.2008.081054.

[6] D. Chu. "Polyphase codes with good periodic correlation properties (Corresp.)" In: *IEEE Transactions on Information Theory* 18.4 (1972), pp. 531–532. ISSN: 0018-9448. DOI: 10.1109/TIT.1972.1054840.

[7] URL: https://en.wikipedia.org/wiki/Zadoff-Chu_sequence (visited on 12/12/2017).

[8] URL: https://www.gnuradio.org/ (visited on 10/10/2017).

[9] URL: https://www.gnu.org/philosophy/free-sw.en.html (visited on 10/10/2017).

[10] URL: https://wiki.gnuradio.org/index.php/BlocksCodingGuide (visited on 10/16/2017).

[11] URL: http://www.swig.org/ (visited on 10/16/2017).

[12] URL: https://www.gnuradio.org/blog/buffers/ (visited on 10/16/2017).

[13] URL: https://github.com/gnuradio/gnuradio/blob/master/gnuradio-runtime/lib/vmcircbuf_mmap_shm_open.cc#L131 (visited on 10/18/2017).

Urheberrechtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Stellen, die ich wörtlich oder sinngemäß aus anderen Werken entnommen habe, habe ich unter Angabe der Quellen als solche kenntlich gemacht.

Datum Unterschrift

Erklärung zur Veröffentlichung von Abschlussarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten.
Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils der ersten und letzten Bachelorabschlusses pro Studienfach und Jahr.

- ○ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ○ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.
- ○ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Datum Unterschrift