IEEE802.11 SDR OFDM Receiver

Digital Communications Project

Miguel Amorim
up202208665@up.pt

Lucas Aparício
up202206594@up.pt

December 8, 2025

Contents

1 Introduction to OFDM

Orthogonal Frequency Division Multiplexing (OFDM) is the fundamental physical layer technology defining the IEEE 802.11a standard. It represents a paradigm shift from traditional single-carrier systems to a robust multi-carrier modulation technique designed specifically to handle the challenges of high-speed wireless data transmission.

1.1 Definition

OFDM is a multi-carrier modulation scheme that transmits a large amount of digital data over a radio wave. It works by splitting the radio signal into multiple smaller sub-signals that are then transmitted simultaneously at different frequencies to the receiver. In the context of 802.11a, the 20 MHz channel is divided into 52 subcarriers (48 for data and 4 for pilots).

1.2 The Problem: Multipath Propagation and ISI

In high-speed single-carrier wireless communications, the primary adversary is multipath propagation. When a signal is transmitted, it reflects off walls, furniture, and other obstacles, arriving at the receiver at slightly different times. This phenomenon causes the "echoes" of a previous symbol to interfere with the current symbol being received, a distortion known as Inter-Symbol Interference (ISI).

As the data rate increases, the symbol duration (T_{sym}) decreases. If T_{sym} becomes shorter than the delay spread of the channel (the time difference between the first and last arriving multipath components), ISI becomes severe, making data recovery effectively impossible without extremely complex equalization.

1.3 The Solution: Parallel Substreams

OFDM addresses the ISI problem by dividing the high-rate serial data stream into N parallel substreams. If the original data rate is R, each substream carries data at a rate of R/N. Consequently, the symbol duration on each subcarrier is expanded to $N \times T_{sym}$.

By making the symbol duration significantly longer than the channel's delay spread, the relative impact of multipath echoes is minimized. This transforms a frequency-selective fading channel (where different frequencies fade differently) into multiple flat-fading channels, which are much easier to equalize.

1.4 Spectral Efficiency

A key distinction between OFDM and traditional Frequency Division Multiplexing (FDM) lies in spectral efficiency.

- **Traditional FDM:** Requires "guard bands" (empty frequency spaces) between subcarriers to prevent adjacent carriers from interfering with each other. This results in wasted spectrum.
- **OFDM:** Allows subcarriers to overlap significantly without interference. This is achieved through the property of *orthogonality*. Conceptually, the peak of each subcarrier corresponds exactly to the zero-crossings of all other subcarriers, allowing 802.11a to pack more data into the available 20 MHz bandwidth compared to non-orthogonal systems.

1.5 Modulation Integration

In 802.11a, OFDM is not just a transmission method but a framework for carrying modulated data. Each of the parallel subcarriers is independently modulated using a digital modulation scheme. The system adapts the modulation density based on the channel quality (Signal-to-Noise Ratio):

- **BPSK (Binary Phase Shift Keying):** Used for basic rates (6 Mbps), carrying 1 bit per symbol per subcarrier.
- **QPSK (Quadrature Phase Shift Keying):** Used for 12/18 Mbps, carrying 2 bits per symbol per subcarrier.
- **16-QAM (Quadrature Amplitude Modulation):** Used for higher rates (24/36 Mbps), carrying 4 bits per symbol.
- **64-QAM:** Used for the maximum data rates (48/54 Mbps), carrying 6 bits per symbol.

This combination of OFDM for channel robustness and high-order QAM for throughput allows 802.11a to achieve data rates up to 54 Mbps.

2 Mathematical Framework

2.1 Orthogonality Condition

The core principle enabling the spectral efficiency described in Section 1 is the mathematical orthogonality of the subcarriers. In the continuous time domain, two signals $\phi_i(t)$ and $\phi_j(t)$ are considered orthogonal over a symbol period T if their inner product is zero. Mathematically, this is defined as:

$$\int_0^T \phi_i(t)\phi_j^*(t)\,dt = 0 \quad \text{for } i \neq j$$

where $(\cdot)^*$ denotes the complex conjugate. This property ensures that the demodulator for the k-th subcarrier is not affected by the signal energy in adjacent subcarriers, allowing for dense spectral packing without Inter-Carrier Interference (ICI).

2.2 IFFT/FFT Implementation

2.2.1 The Practicality Challenge

Theoretically, an OFDM signal is a sum of independent sinusoidal subcarriers. However, implementing this with analog banks of oscillators and demodulators for a large number of subcarriers (52 in 802.11a) is hardware-intensive, expensive, and prone to frequency drift.

2.2.2 Digital Implementation

Modern 802.11a transceivers solve this by performing modulation and demodulation in the digital domain:

- **Transmitter (IFFT):** The Inverse Fast Fourier Transform (IFFT) is used to map the frequency domain data (complex constellation points assigned to specific subcarriers) into the time domain waveform. It efficiently converts a block of N parallel data symbols into a sequence of time-domain samples.

- **Receiver (FFT):** The Fast Fourier Transform (FFT) performs the reverse operation. It takes the received time-domain samples and converts them back into the frequency domain, recovering the original phase and amplitude of each subcarrier.

2.3 The Cyclic Prefix (CP)

A crucial component of the 802.11a symbol structure is the Cyclic Prefix (CP). The CP is formed by copying the last $0.8\mu s$ of the time-domain symbol and prepending it to the front of the symbol.

2.3.1 Purpose 1: Guard Interval

The CP acts as a guard interval. By ensuring that the length of the CP is longer than the channel's maximum delay spread (τ_{max}), multipath echoes from the previous symbol decay completely before the "useful" portion of the current symbol is processed. This effectively eliminates Inter-Symbol Interference (ISI).

2.3.2 Purpose 2: Circular Convolution

Beyond simple guarding, the CP serves a profound mathematical purpose. The physical channel performs a *linear convolution* between the transmitted signal and the channel impulse response. However, the FFT assumes the input signal is periodic. By making the transmitted signal appear periodic within the FFT integration window (via the CP), the linear convolution of the channel is converted into a *circular convolution*.

$$y[n] = h[n] \circledast x[n] \iff Y[k] = H[k] \cdot X[k]$$

This simplifies the receiver design significantly. Instead of complex time-domain equalization, the receiver can perform simple single-tap equalization in the frequency domain, where each subcarrier is divided by the estimated channel frequency response.

2.3.3 Purpose 3: Power Continuity and AGC Stability

Finally, using a cyclic copy rather than a silent guard interval (zero padding) maintains the average power of the transmitted signal. A silent period would cause significant fluctuations in the signal envelope, potentially destabilizing the receiver's Automatic Gain Control (AGC) circuits. By preventing these power drops, the CP ensures the receiver maintains optimal resolution and quantization levels throughout the sampling window, avoiding the transient distortions or dynamic range issues that a silent interval might introduce.

3 The 802.11a Physical Layer (PHY) Specifications

While the general principles of OFDM apply to many standards (such as LTE and DVB), IEEE 802.11a defines a specific implementation tailored for Wireless LANs in the 5 GHz band.

3.1 Frequency Domain Structure

IEEE 802.11a operates within a 20 MHz bandwidth channel. To generate the signal, a 64-point IFFT/FFT is used. However, not all 64 available inputs to the IFFT correspond to data. The subcarriers are mapped as follows:

- **Total Subcarriers:** 64 (N_{FFT})

- **Active Subcarriers:** 52 (Indices -26 to $+26$, excluding 0)
    - **Data Subcarriers (N_{SD}):** 48 subcarriers are used to transmit the actual payload. They carry the BPSK, QPSK, 16-QAM, or 64-QAM constellation symbols.
    - **Pilot Subcarriers (N_{SP}):** 4 subcarriers are dedicated to pilot signals. Located at indices ± 7 and ± 21, these carry known BPSK sequences used by the receiver to track phase drift and frequency offset during reception.
- **Null Subcarriers:** 12 subcarriers are set to zero energy.
    - **DC Carrier:** Index 0 is null to prevent DC offset in the Digital-to-Analog conversion from corrupting the signal (carrier leakage).
    - **Guard Subcarriers:** Indices -32 to -27 and $+27$ to $+31$. These "empty" carriers at the edges of the spectrum act as a buffer to ensure the signal decays sufficiently before entering adjacent channels, meeting spectral mask requirements.

3.2 Time Domain Parameters

The timing parameters in 802.11a are derived directly from the channel bandwidth and the FFT size. The standard assumes a sampling rate of 20 MHz.

Parameter	**Value**
Channel Bandwidth (B)	20 MHz
FFT Size (N_{FFT})	64
Subcarrier Spacing ($\Delta f = B/N_{FFT}$)	312.5 kHz
Inverse FFT Period ($T_{FFT} = 1/\Delta f$)	3.2 μs
Guard Interval Duration ($T_{GI} = T_{FFT}/4$)	0.8 μs
Guard Interval Samples (N_{GI})	16 samples
Total Symbol Samples (N_{SYM})	80 samples (64 + 16)
Total Symbol Duration ($T_{SYM} = T_{GI} + T_{FFT}$)	**4.0** μs

Table 1: IEEE 802.11a OFDM Timing Parameters

The choice of a 0.8 μs guard interval (16 samples) allows the protocol to tolerate delay spreads up to roughly 200–250 ns without significant performance degradation, which is typical for indoor office and home environments.

4 Frame Structure (PPDU Format)

The full PHY frame, formally known as the PLCP Protocol Data Unit (PPDU), is composed of three distinct parts: the PLCP Preamble, the Signal Field, and the Data Field.

4.1 PLCP Preamble

The preamble is the first component transmitted. It enables the receiver to detect the signal, adjust its gain, and synchronize its frequency and timing before decoding the actual data. It lasts for a total of 320 samples and contains two parts:

4.1.1 Short Training Sequence (STS)

The first 160 samples of the preamble consists of 10 identical short OFDM symbols, each 16 samples long.

- **Composition:** Only 12 out of the 52 subcarriers are active (populated with non-zero values). This periodicity in the frequency domain (every 4th subcarrier) creates a shorter repeating pattern in the time domain.
- **Primary Functions:** Signal detection, Automatic Gain Control (AGC) convergence, diversity selection, and coarse frequency offset estimation.

4.1.2 Long Training Sequence (LTS)

Following the STS is the Long Training Sequence, also 160 samples long. It consists of a guard interval (32 samples, which is effectively a ”double” GI) followed by two identical long OFDM symbols (64 samples each).

- **Composition:** All 52 active subcarriers and the DC carrier are used.
- **Primary Functions:** Channel estimation (determining the amplitude and phase shift introduced by the channel for each subcarrier) and fine frequency offset tuning.

4.2 Signal Field

The Signal Field follows the preamble. It is a single OFDM symbol of 80 samples that conveys the parameters necessary to decode the subsequent payload. Crucially, the Signal Field is always encoded using the most robust modulation available (**BPSK** with a **1/2 coding rate**) to ensure it can be decoded even in poor channel conditions. It contains:

RATE (4 bits): Indicates the data rate and modulation scheme of the data payload. The encoding is not a simple binary count but a specific mapping defined by the standard to identify the 8 supported data rates (see Table 2).

LENGTH (12 bits): Indicates the number of octets (bytes) in the MAC payload (PSDU).

Parity (1 bit): An even parity check bit for error detection on the Signal Field itself.

Tail (6 bits): Set to zero to flush the convolutional encoder state, ensuring the decoder is reset before processing the Data Field.

Data Rate (Mbps)	**Bit Pattern (R_1-R_4)**
6	1101
9	1111
12	0101
18	0111
24	1001
36	1011
48	0001
54	0011

Table 2: Signal Field RATE Encoding

4.3 Data Field

The Data Field contains the actual user payload and operates at the rate specified in the Signal Field. It is composed of a variable number of OFDM symbols.

- **Service Field (16 bits):** The first 16 bits of the data field. These are used to initialize the scrambler.
- **PSDU (PLCP Service Data Unit):** The actual data passed down from the MAC layer.
- **Tail Bits (6 bits):** Appended after the payload to return the convolutional encoder to the "zero state" at the end of the packet.
- **Pad Bits:** Because OFDM transmits data in fixed-size blocks (symbols), the total number of bits (Service + PSDU + Tail) might not fit exactly into an integer number of symbols. Pad bits are added to fill the last OFDM symbol completely.

5 Modulation and Data Rates Summary

IEEE 802.11a offers eight distinct data rates. The system adapts to channel conditions by selecting the appropriate Modulation and Coding Scheme (MCS). A robust scheme (BPSK) is used for noisy channels, while high-density schemes (64-QAM) are used for clear channels to maximize throughput.

Data Rate (Mbps)	**Modulation**	**Coding Rate (R)**	**Coded Bits per Subcarrier (N_{BPSC})**	**Coded Bits per Symbol (N_{CBPS})**	**Data Bits per Symbol (N_{DBPS})**
6	BPSK	1/2	1	48	24
9	BPSK	3/4	1	48	36
12	QPSK	1/2	2	96	48
18	QPSK	3/4	2	96	72
24	16-QAM	1/2	4	192	96
36	16-QAM	3/4	4	192	144
48	64-QAM	2/3	6	288	192
54	64-QAM	3/4	6	288	216

Table 3: IEEE 802.11a Rate-Dependent Parameters

6 Short Sequence Synchronization

The `sync_short` block acts as the initial gatekeeper of the receiver pipeline. Its primary objectives are to detect the presence of an incoming IEEE 802.11a/g/p frame (Packet Detection) and to perform a Coarse Frequency Offset (CFO) correction to align the receiver's center frequency with the transmitter's.

6.1 Mathematical Principles

6.1.1 Frame Detection (Autocorrelation)

The detection algorithm exploits the periodic structure of the Short Training Sequence (STS), which consists of a 16 sample pattern that is repeated 10 time. The receiver computes the autocorrelation $C[n]$ between the received signal $r[n]$ and a version of itself delayed by $D = 16$

samples. This value is normalized by the signal power $P[n]$ to produce a metric $M[n]$ independent of absolute signal strength.

$$C[n] = \sum_{k=0}^{L-1} r[n+k] \cdot r^*[n+k-16] \tag{1}$$

$$M[n] = \frac{|C[n]|}{P[n]} \tag{2}$$

Detection is triggered when the normalized autocorrelation $M[n]$ exceeds a specific threshold γ for a sustained number of samples ($N_{plateau}$), forming a "plateau". $N_{plateau}$ and γ are inputs.

Code Implementation: In the `SEARCH` state, the block monitors this normalized correlation (`in_cor`):

```
// sync_short.cc
if (in_cor[i] > d_threshold) {
    if (d_plateau < MIN_PLATEAU) {
        d_plateau++; // Accumulate confidence
    } else {
        d_state = COPY; // Threshold met for N samples -> Frame Detected
        // ...
    }
}
```

6.1.2 Coarse Frequency Offset Estimation

A Carrier Frequency Offset (CFO), denoted as Δf, introduces a time-varying phase rotation to the received signal.

$$r[n] = x[n] \cdot e^{j2\pi\Delta f n T_s} \tag{3}$$

The autocorrelation over the delay $D = 16$ captures the phase rotation accumulated over exactly one period of the Short Training Sequence (STS). Since the underlying symbol $x[n]$ is identical to $x[n-16]$, their product is real, leaving only the phase term:

$$C[n] = r[n] \cdot r^*[n-16] \propto e^{j2\pi\Delta f \cdot 16 T_s} \tag{4}$$

The argument (angle) of this complex correlation value corresponds to the total phase shift over 16 samples. Therefore, the phase shift **per sample**, $\Delta\phi$, is derived as:

$$\Delta\phi = \frac{\text{arg}(C[n])}{16} \tag{5}$$

Code Implementation: The offset is calculated immediately upon entering the `COPY` state using the `in_abs` input (which holds the complex correlation product):

```
// sync_short.cc
// Calculate phase shift per sample (radians)
d_freq_offset = arg(in_abs[i]) / 16;
```

6.1.3 Frequency Correction

To compensate for the offset (which is calculated once at the beginning), the receiver must apply a counter-rotation to the input signal. If the received signal is rotating by $+\Delta\phi$ per sample, the corrected output $y[n]$ is obtained by multiplying by $e^{-j\Delta\phi n}$ (done iteratively for each n):

$$y[n] = r[n] \cdot e^{-j \cdot \Delta\phi \cdot n} \tag{6}$$

Code Implementation: This is applied iteratively in the `COPY` state, where `d_copied` tracks the sample index n:

```
// sync_short.cc
// Apply correction: Signal * exp(-j * offset * n)
out[o] = in[o] * exp(gr_complex(0, -d_freq_offset * d_copied));
```

6.2 State Machine Operation

The block functions as a Finite State Machine (FSM) with two modes:

- **SEARCH State:** The default ”listening” mode. The block consumes input samples but produces no output. It continuously scans the autocorrelation input for a valid plateau. When detected, it transitions to `COPY`.
- **COPY State:** The active data transfer mode. The block passes input samples to the output while applying the frequency correction formula derived above. It maintains this state until a maximum sample count is reached or a new frame is detected.

6.3 Parameter Analysis

The robustness of the detection depends on two key parameters:

1. **Threshold ($\gamma = 0.8$):**
   - *Effect:* Sets the similarity required to recognize a preamble.
   - *Implication:* A value too low (< 0.5) causes false alarms on noise. A value too high (> 0.9) causes missed detections of valid but weak frames.
2. **Min Plateau ($N_{plat} = 2$):**
   - *Effect:* Requires the threshold to be exceeded for consecutive samples.
   - *Implication:* Acts as a temporal filter. Setting this to 2 or 3 prevents instantaneous noise spikes from triggering the receiver.

6.4 Advanced Frame Handling: Valve vs. Tagging

A critical improvement in this implementation compared to early designs (such as the one described in the original 2013 paper) is the handling of back-to-back frames.

6.4.1 The Legacy ”Valve” Approach

The original design functioned as a simple valve that opened for a fixed number of samples upon detection.

- **Blindness:** Once the valve opened, the detector stopped searching.

- **RTS/CTS Failure:** If a second frame (e.g., CTS) arrived while the valve was open for the first frame (RTS), the second preamble would be buried in the data stream, ignored by the detector, and ultimately lost.

6.4.2 The Modern "Tagging" Solution

The current code implements a continuous scanning mechanism even while in the `COPY` state.

- **Mechanism:** If the autocorrelation rises again after a minimum gap (`MIN_GAP`), the block identifies a new frame *inside* the current stream.
- **Tagging:** Instead of closing and reopening, it inserts a `wifi_start` stream tag at the new preamble location and recalculates the frequency offset immediately.

Code Implementation:

```
// sync_short.cc inside COPY loop
} else if (d_copied > MIN_GAP) {
    // NEW FRAME DETECTED!
    d_copied = 0;       // Reset counter
    d_freq_offset = arg(in_abs[o]) / 16; // Recalculate offset for NEW frame
    insert_tag(nitems_written(0) + o, d_freq_offset, nitems_read(0) + o);
    break; // Restart loop
}
```

6.5 Practical Example Scenario

To visualize the operation, consider a noisy channel where an 802.11 frame arrives.

1. **Idle:** The autocorrelation output fluctuates around 0.1-0.3 (Noise). The state machine is in `SEARCH`.
2. **Arrival:** The Short Preamble arrives. The autocorrelation spikes to 0.85, then 0.9, then 0.88.
3. **Trigger:** The block sees values > 0.8 for 2 consecutive samples. It switches to `COPY`.
4. **Correction:** It calculates that the phase is rotating by 0.1 radians per sample. It begins multiplying the stream by $e^{-j0.1n}$.
5. **Output:** The downstream blocks receive a clean, frequency-corrected stream starting exactly at the first preamble symbol.

7 The OFDM Sync Long Block: Fine-Grained Synchronization

The `sync_long` block represents the second and final stage of the synchronization chain. While the preceding `sync_short` block is responsible for detecting the presence of a frame and performing a coarse frequency correction, the `sync_long` block is tasked with the high-precision operations required to prepare the signal for demodulation: **Fine Frequency Offset (FFO) Correction** and **Symbol Timing Alignment**.

These operations are performed on the *Long Training Sequence* (LTS) of the preamble. Crucially, both steps must be completed in the time domain before the signal reaches the FFT block. If the signal enters the FFT with residual frequency errors or misalignment, the orthogonality of the subcarriers is lost, resulting in Inter-Carrier Interference (ICI) and Inter-Symbol Interference (ISI) that no subsequent equalizer can correct.

7.1 Block Architecture and the Delay Buffer

A critical architectural feature of the receiver is the handling of processing latency. The `sync_short` block requires a finite amount of time (processing a window of samples) to calculate the auto-correlation plateau and confirm a detection. By the instant the "Frame Detected" decision is made, the actual start of the frame has already passed through the block.

To prevent the loss of the preamble and the subsequent data payload, the receiver employs a split-path architecture with a **Delay Block** (typically set to 240 samples).

7.1.1 The Split-Path Mechanism

The `sync_long` block receives two input streams:

1. **Input 0 (Control/Tag Path):** Receives the direct, real-time output from `sync_short`. This stream carries the `wifi_start` stream tag, which serves as the trigger to wake up the `sync_long` state machine.
2. **Input 1 (Data/Buffered Path):** Receives the signal passed through the Delay Block. This buffer effectively holds the "past" samples in memory.

7.1.2 The Function of the Delay

The delay is not intended to shift the timing of the STS or LTS for algorithmic reasons, but rather to function as a **pre-roll buffer**. When the `sync_short` block triggers the `wifi_start` flag (typically at the end of the Short Training Sequence), the `sync_long` block activates.

Without the delay, the block would begin reading the stream from the middle of the preamble, potentially missing the critical Long Training Sequence or the start of the Data field. However, by reading from the delayed Input 1, the `sync_long` block can "reach back in time." The 240-sample delay ensures that while the detection logic was running, the start of the frame (including the STS, LTS, Signal Field, and Payload) was held safely in the buffer. This guarantees that the `sync_long` block can process the frame from the very first sample, ensuring zero data loss.

7.2 Fine Frequency Offset Correction

7.2.1 The Need for a Two-Stage Correction

The receiver employs a cascaded approach to frequency correction. The `sync_short` block uses the Short Training Sequence (STS) to estimate and remove the bulk of the Carrier Frequency Offset (CFO). However, because the STS is short (16 samples), its estimation is prone to noise and cannot resolve small, residual errors.

The `sync_long` block addresses this residual error. Any remaining frequency offset Δf causes a time-varying phase rotation $\theta(n) = 2\pi \Delta f n T_s$. If uncorrected, this rotation would cause the constellation points to spin during the payload, causing bit errors.

7.2.2 Algorithm and Implementation

The block calculates the fine offset using the Long Training Sequence (LTS), which consists of two identical 64-sample symbols (T_1 and T_2). The algorithm compares these two symbols. Since they are identical, any phase difference between them is solely due to the residual frequency offset accumulated over the 64-sample duration.

Mathematically, the fine offset is estimated as:

$$\Delta f_{\text{fine}} = \frac{1}{64} \arg \left(\sum_{k=0}^{63} r[n+k] \cdot r^*[n+k+64] \right) \tag{7}$$

In the C++ implementation (`sync_long.cc`), this is computed inside the `search_frame_start` function. The code takes the two identified training symbols, `first` and `second`, and computes the angle of their product:

```
// sync_long.cc :: search_frame_start()
d_freq_offset = arg(first * conj(second)) / 64;
```

7.2.3 Application

This fine offset value is added to the coarse offset. During the `COPY` state, the block applies a counter-rotation to every sample in the frame to nullify this drift:

```
// sync_long.cc :: general_work()
out[o] = in_delayed[i] * exp(gr_complex(0, d_offset * d_freq_offset));
```

7.3 Symbol Timing Alignment

7.3.1 The Precision Challenge

The FFT requires a discrete block of samples (the "FFT Window") corresponding exactly to one OFDM symbol. If this window is shifted by even a few samples:

- **Too Early:** The window captures the end of the previous symbol, causing ISI.
- **Too Late:** The window captures the cyclic prefix of the next symbol, causing phase rotations.

While the `sync_short` block detects that a frame *exists*, it is not precise enough to define exactly *where* the data starts. The `sync_long` block solves this using **Matched Filtering**.

7.3.2 Matched Filter Implementation

Matched filtering is computationally expensive (≈ 64 complex multiplications per sample), so it is performed only on the 160-sample LTS buffer, not the continuous stream. The block convolves the received signal with a known time-domain template of the LTS (`LONG` vector in code).

```
// sync_long.cc :: general_work()
d_fir.filterN(d_correlation, in, ...);
```

This operation produces a correlation peak when the template perfectly aligns with the received symbol. Since the LTS contains two repeated symbols, the filter output produces two distinct peaks.

7.3.3 Peak Detection and The +64 Offset

To ensure robustness against noise, the block identifies the indices of the highest peaks (N_p). It selects the **latest** valid peak index to align with the second LTS symbol, which provides the most stable reference.

Crucially, the algorithm adds an offset of 64 samples to this peak index.

$$n_{\text{start}} = \max(N_p) + 64 \tag{8}$$

The necessity of the +64 offset is rooted in the operation of the **matched filter** (or the correlation calculation) used for synchronization:

1. $\max(N_p)$ **identifies the end of the matched sequence**. The correlation peak is naturally delayed until the entire 64-sample second LTS symbol has passed into the observation window. Therefore, $\max(N_p)$ points to the sample where the second LTS symbol finishes.
2. **The duration of the LTS symbol is exactly 64 samples.**
3. The resulting index n_{start} is calculated by adding the 64-sample duration to the end of the LTS. This operation lands precisely on the sample index where the **preamble ends** and the **Signal Field (the start of the data/payload)** begins. This is the desired starting point for the subsequent OFDM processing stages.

7.4 Data Preparation for the FFT

Once the frequency is corrected and the start index is defined, the block transitions to the `COPY` state. It acts as a gatekeeper for the FFT block:

- **Preamble Stripping:** It discards the initial 320 samples (Short + Long preambles) as they have served their purpose.
- **Cyclic Prefix Removal:** For every 80 incoming samples (Data Symbol + CP), it removes the first 16 samples (the Guard Interval).
- **Output Generation:** It outputs streams of 64-sample ”core” symbols. These are passed to the **Stream to Vector** block, which bundles them into vectors of length 64, satisfying the input requirement of the FFT block.

8 Analysis of Frame Equalization and Channel Estimation

This section details the signal processing mechanisms within the `frame_equalizer` block of the GNU Radio IEEE 802.11 receiver. The analysis focuses on phase tracking, channel estimation, and signal field decoding, correlating the theoretical background from the reference paper with the C++ implementation found in `frame_equalizer_impl.cc`.

8.1 Phase Offset Correction

Phase correction is critical in OFDM receivers to counteract two specific hardware imperfections caused by non-synchronized clocks between the transmitter and receiver: Sampling Frequency Offset (SFO) and Carrier Frequency Offset (CFO).

8.1.1 Theoretical Model

The receiver models the phase error Φ_k for a subcarrier k as a linear equation:

$$\Phi_k = \text{Slope} \cdot k + \text{Intercept} \tag{9}$$

- **Slope (SFO):** Caused by the sampling clock ticking faster or slower than the transmitter, manifesting as a delay that accumulates over time. In the frequency domain, this appears as a phase rotation proportional to the frequency index k.

- **Intercept (CFO):** Caused by a mismatch in center frequency, manifesting as a Common Phase Error (CPE) where the entire constellation rotates uniformly.

8.1.2 Algorithmic Implementation

The correction is applied in a three-step process within the `general_work` loop.

Step A: SFO Slope Correction Before pilot estimation, the symbol is corrected based on the current estimated clock drift ϵ. The code applies a frequency-dependent phase rotation:

$$S'_k = S_k \cdot e^{j2\pi \cdot N_{\text{sym}} \cdot 80 \cdot (\epsilon_0 + \epsilon_r) \cdot (k/64)} \tag{10}$$

where N_{sym} is the physical time (symbol index), 80 is the total samples per OFDM symbol (64 data + 16 guard), and $\epsilon_0 + \epsilon_r$ represents the coarse and fine drift estimates.

Step B: CPE Intercept Correction The code calculates the "intercept" using the four pilot subcarriers (indices -21, -7, 7, 21; mapped to FFT indices 11, 25, 39, 53). The pilots are descrambled by multiplying with the polarity p, and the resulting vectors are averaged to find the angle β:

$$\beta = \text{arg}\left(\sum_{p \in \text{pilots}} S_p \cdot \text{polarity}_p\right) \tag{11}$$

Note that the pilot at index 53 is natively negative in the IEEE 802.11 standard, so the code multiplies it by $-p$ to align it constructively. The entire symbol is then rotated by $e^{-j\beta}$ to correct the Common Phase Error.

Step C: Feedback Loop (SFO Refinement) A residual SFO may remain if the slope correction in Step A was imperfect. The receiver tracks this by measuring the phase drift of the pilots relative to the *previous* symbol:

$$\text{er}_{\text{rad}} = \text{arg}\left(\sum \text{conj}(P_{\text{prev}}) \cdot P_{\text{curr}}\right) \tag{12}$$

This angle is converted into a dimensionless Clock Error Ratio using the physical relationship $\Delta\Phi = 2\pi\Delta fT$. The update rule is derived as:

$$\epsilon_{new} = \text{er}_{\text{rad}} \cdot \frac{BW}{2\pi \cdot f_c \cdot 80} \tag{13}$$

This error is fed into a simple IIR (Exponential Moving Average) filter to smooth the estimate over time: $d_{er} = (1 - \alpha)d_{er} + \alpha \cdot \text{er}$.

8.2 Channel Estimation and Magnitude Correction

While the frame equalizer corrects phase rotation, the magnitude of the subcarriers must also be equalized to support Quadrature Amplitude Modulation (QAM).

8.2.1 Least Squares (LS) Estimation

Contrary to the 2013 paper, which described a static "sinc-shaped" magnitude correction limiting the receiver to BPSK/QPSK, the actual implementation in `ls.cc` utilizes a Least Squares (LS) estimator. This allows the receiver to support 16-QAM and 64-QAM by dynamically estimating the channel response $H[k]$.

The estimator uses the two Long Training Symbols (LTS) found in the preamble. It averages the received symbols Y_{LTS} and divides by the known training sequence X_{LTS}:

$$H[k] = \frac{Y_{LTS1}[k] + Y_{LTS2}[k]}{2 \cdot X_{LTS}[k]} \tag{14}$$

The data subcarriers for the remainder of the frame are equalized via Zero-Forcing:

$$\hat{S}[k] = \frac{Y_{\text{data}}[k]}{H[k]} \tag{15}$$

8.2.2 Limitations

While superior to the static method described in the paper, the LS implementation has two notable limitations:

1. **Noise Enhancement:** For subcarriers in deep fades (small $|H[k]|$), dividing by $H[k]$ amplifies the noise, potentially causing bit errors.
2. **Static Estimation:** The channel is estimated only once during the preamble. If the channel changes during the payload (e.g., due to Doppler shift), the initial $H[k]$ becomes outdated, as the LS block does not adaptively update the magnitude during the frame.

8.3 Signal Field Decoding

The `OFDM Decode Signal` mechanisms are integrated directly into the `general_work` function of the `frame_equalizer_impl` class. When the symbol index $N_{sym} = 2$ (the first symbol after the preamble), the block executes a specialized decoding pipeline.

8.3.1 Purpose and Mechanism

The primary objective of decoding the SIGNAL field at this early stage is to configure the receiver for the upcoming payload. The SIGNAL field contains critical information required to process the Data field:

- **Frame Duration:** Determining the length of the packet allows the equalizer to know exactly how many symbols to process before resetting for the next frame.
- **Payload Modulation:** While the SIGNAL field is always BPSK, the payload can be modulated with higher-order schemes (up to 64-QAM). The receiver must know the modulation scheme beforehand to configure the constellation de-mapper correctly.

To extract this information, the block isolates the 48 data subcarriers of the SIGNAL symbol and performs deinterleaving and Viterbi decoding. Since the IEEE 802.11 standard mandates that the header is always transmitted using the most robust modulation, the decoder is hardcoded to **BPSK with a coding rate of 1/2**. This yields 24 decoded bits containing the PHY header.

8.3.2 Configuration and Tagging

The 24 decoded bits are parsed to extract the `RATE` (bits 0–3) and `LENGTH` (bits 5–16) fields. These values are used to calculate the total number of OFDM symbols in the frame and to update the demodulation object (e.g., switching `d_frame_mod` to 16-QAM). Finally, these parameters are attached to the sample stream as GNU Radio tags, enabling downstream blocks to adapt their processing logic dynamically for the payload.

9 MAC Decoding and De-Interleaving

9.1 Overview of Actions

Actions taking place on this block: Here is where we recover the original user data. For that, we start by doing symbol slicing, where it converts complex symbols into raw bits. Then, de-interleaving is performed, where the bits are reordered to reverse the permutation applied at the transmitter. Then, Viterbi decoding is done, correcting bit errors using the convolutional code. Following this, de-scrambling is performed; this is where the data passes to an XOR operation. Since it is scrambled with a pseudo-random sequence, the initial state is recovered using the first few bits. Finally, a CRC check is done to verify the integrity of the frame.

9.2 Symbol Slicing

Symbol slicing is performed because the subsequent processing blocks (De-interleaver and Viterbi Decoder) operate on bits, while the input consists of demodulated symbols (represented as integer indices of constellation points) BPSK -> 1 symb. = 1 bit , QPSK -> 1 symb. = 2 bit, etc.. The code implementation iterates through all symbols in the frame and extracts the individual bits (k) from each symbol integer using bitwise operations. Basically what this means is ”Based on this position, what bits does this symbol represent?” and decides what each one represents. However, this selection is done in the frame decoder, but the `decode_mac` receives `d_rx_symbols` as an array of bytes, where each byte is an integer index representing the constellation point. The ”slicing” here is actually a bit extraction process. The code loops through each symbol and extracts the individual bits (b0,b1,...) from that integer, as we can see bellow:

```
// d_rx_symbols contains the demodulated symbols
// d_rx_bits will contain the raw binary stream
for (int i = 0; i < d_frame.n_sym * 48; i++) {
    for (int k = 0; k < d_ofdm.n_bpsc; k++) {
        // Bitwise AND (&) with shift (<<) extracts the k-th bit
        d_rx_bits[i * d_ofdm.n_bpsc + k] = !!(d_rx_symbols[i] & (1 << k));
    }
}
```

Example:

```
d_rx_symbols[i] holds the integer value 3 (binary 11) and we are in QPSK

k=0: 3 & 1 result is 1. Bit 0 is 1.
k=1: 3 & 2 result is 2. !!2 is 1. Bit 1 is 1.
The result is that the symbol 3 is sliced into two bits [1, 1].
```

9.3 De-Interleaving Implementation

Since wireless channels are frequency-selective, if a deep fade affects specific subcarriers, adjacent bits in the OFDM symbol might be corrupted. Interleaving spreads adjacent bits across non-adjacent subcarriers. Consequently, if any burst errors occur, they appear as scattered random errors to the Viterbi decoder. They appear separated to the Viterbi process since the adjacent bits will not suffer the same fate, being separated between non-adjacent carriers. This is crucial because the Viterbi decoder handles random errors much better than burst errors.

The IEEE 802.11 standard uses a two-step permutation. For the k-th bit of the block, and the

N_{cbps} coded bits per symbol (e.g., 48 for BPSK, where $N_{bpsc} = 1$), the code implements two mappings.

The first permutation is the frequency mapping, calculated as:

$$i = s \times \text{floor}(j/s) + (j + \text{floor}(16 \times j/N_{cbps})) \mod s \tag{16}$$

where $s = \max(N_{bpsc}/2, 1)$.

The second permutation is the bit mapping, which is more important for higher-order modulations like QAM to ensure coded bits are mapped alternately to less significant bits. It is calculated as:

$$j = 16 \times i - (N_{cbps} - 1) \times \text{floor}(16 \times i/N_{cbps}) \tag{17}$$

In the C++ implementation, the code pre-calculates these permutations into Lookup Tables (LUTs) named `first` and `second`, and then applies them:

```
// Preparing the lookup tables
int s = std::max(d_ofdm.n_bpsc / 2, 1);

// First Permutation
for (int j = 0; j < n_cbps; j++) {
    first[j] = s * (j / s) + ((j + int(floor(16.0 * j / n_cbps))) % s);
}

// Second Permutation
for (int i = 0; i < n_cbps; i++) {
    second[i] = 16 * i - (n_cbps - 1) * int(floor(16.0 * i / n_cbps));
}

// Applying the De-interleaving
for (int i = 0; i < d_frame.n_sym; i++) {
    for (int k = 0; k < n_cbps; k++) {
        // d_rx_bits is the input (interleaved)
        // d_deinterleaved_bits is the output (original order)
        d_deinterleaved_bits[i * n_cbps + second[first[k]]] =
            d_rx_bits[i * n_cbps + k];
    }
}
```

The code above iterates through every subcarrier. It uses the `first` array to identify the intermediate index (frequency mapping) and the `second` array to calculate the final destination of each bit (bit mapping). Then, it moves the bit from `d_rx_bits` (the received, interleaved buffer) to `d_deinterleaved_bits` (the output buffer), placing it in the correct order.

10 Viterbi Decoding

The Viterbi decoder acts as the engine for **Forward Error Correction (FEC)**. In the IEEE 802.11a standard, data is encoded using a Convolutional Encoder with a constraint length $K = 7$ and rate $R = 1/2$. This process adds redundancy to the data stream, enabling the receiver to correct bit errors caused by noise or interference.

The decoder employs the **Viterbi algorithm**, which uses Maximum Likelihood Sequence Estimation (MLSE) to search for the "most likely" sequence of states the transmitter passed through to generate the received bits.

10.1 Implementation in Code

In the provided `decode_mac.cc` file, the Viterbi decoder is abstracted into a helper class. The `decode_mac` block manages the data flow and invokes the decoder rather than implementing the trellis search directly.

Location and Execution: A decoder object is instantiated in the class header:

```
viterbi_decoder d_decoder;
```

The actual decoding occurs within the `decode()` method:

```
uint8_t* decoded = d_decoder.decode(&d_ofdm, &d_frame, d_deinterleaved_bits);
```

Arguments Explanation:

- `&d_ofdm`: Provides modulation parameters, specifically the **coding rate** (e.g., 1/2, 2/3, 3/4). High rates (like 3/4) involve "puncturing" (deleting bits) at the transmitter. The decoder uses this info to insert "dummy" bits ("depuncturing") at the correct locations before decoding, to later insert correct ones.
- `d_deinterleaved_bits`: These are the "hard decisions" (0s and 1s) produced by the symbol slicer in the previous stage.

10.2 How It Works (The Algorithm)

A. The Setup (Trellis) The encoder maintains a "memory" of the last $K-1$ bits. For 802.11a ($K=7$), this memory is 6 bits.

- **States:** There are $2^6 = 64$ possible states the encoder can occupy at any moment.
- **Transitions:** From any state, an input of '0' moves the encoder to a specific new state and outputs 2 bits. An input of '1' moves it to a different state with a different 2-bit output.

B. The Decision Process (Add-Compare-Select) As the receiver processes a pair of incoming bits (e.g., `10`), it compares them against the expected output for every possible state transition.

1. **Branch Metric (BM):** Measures the "distance" (error) between received bits and expected bits. For example, if `11` is received but `01` was expected, the Hamming distance is 1.
2. **Path Metric (PM):** The accumulated error score for a specific path up to the current point.
3. **Add-Compare-Select (ACS):**
    - **Add:** The Branch Metric is added to the previous Path Metric for every state.
    - **Compare:** Since multiple paths can converge on the same state, their total scores are compared.
    - **Select:** The path with the lowest error score (the "Survivor Path") is kept; others are discarded.

C. Traceback Once the entire frame is processed, the decoder identifies the final state with the lowest score and "traces back" through the survivor paths to reconstruct the original input sequence.

10.3 Real Example

Because the standard $K=7$ encoder (64 states) is complex to visualize, we use a simplified $K=3$ encoder (4 states: 00, 01, 10, 11).

Scenario:

- **Encoder Logic:** Input `1` $\rightarrow$ Output `11`; Input `0` $\rightarrow$ Output `00`.
- **Transmitted Data:** `1 0 1` $\rightarrow$ Encoded as `11 00 11`.
- **Received Stream:** `11 01 11` (The middle `00` was corrupted to `01`).

Step-by-Step Decoding:

1. **Step 1: Received** `11`
   - *Hypothesis A (Input 1):* Expected `11`. Distance = 0. (Good match).
   - *Hypothesis B (Input 0):* Expected `00`. Distance = 2. (Bad match).
   - **Winner:** Hypothesis A (Score 0).
2. **Step 2: Received** `01` **(The Error)**
   - *Path from A (Input 1 $\rightarrow$ 1):* Expected `11`. Received `01`. Distance = 1. Total Score = $0 + 1 = 1$.
   - *Path from A (Input 1 $\rightarrow$ 0):* Expected `00`. Received `01`. Distance = 1. Total Score = $0 + 1 = 1$.
   - *Path from B:* Previous score was 2. Adding distance makes it worse (> 2).
   - **Result:** Both paths from A are kept as candidates.
3. **Step 3: Received** `11`
   - *Path (1 $\rightarrow$ 1 $\rightarrow$ 1):* Expected `11`. Received `11`. Distance = 0. Total Score = $1 + 0 = 1$.
   - *Path (1 $\rightarrow$ 0 $\rightarrow$ ...):* Transition would expect `00`. Received `11`. Distance = 2. Total Score = $1 + 2 = 3$.
   -
   - **Result:** we only keep the value 1 for the third option since it has 0 distance.

Final Decision: The decoder compares the final scores. The path `1` $\rightarrow$ `1` $\rightarrow$ `1` has a total error score of 1, while the alternative `1` $\rightarrow$ `0` $\rightarrow$ `1` also has a total error score of 1. Since the decoder does not have a lower code, it traces back, and outputs the sequence **1 0 1**, ideally. The middle bits `01` hopefully were corrected back to `00` because the surrounding context made the "Input 0" path the most likely survivor despite the noise, ignoring the option of 11 on that spot instead.

11 De-Scrambling

The de-scrambling stage is responsible for reversing the randomization applied at the transmitter. Scrambling is used to "whiten" the data spectrum, preventing long sequences of identical bits (0s or 1s) which could cause synchronization issues or high peak-to-average power ratios. The scrambler is implemented using a Linear Feedback Shift Register (LFSR) with the generating polynomial $S(x) = x^7 + x^4 + 1$.

11.1 Seed Recovery

To synchronize the receiver's LFSR with the transmitter, the 802.11 standard mandates that the first 7 bits of the Service Field (the start of the payload) must be zeros before scrambling. Since

$0 \oplus x = x$, the first 7 bits output by the Viterbi decoder reveal the initial state (seed) of the scrambler directly. The code reads these first 7 bits and initializes the `state` variable.

11.2 The De-Scrambling Loop

Once the seed is known, the receiver enters a loop to process the rest of the payload. For each incoming bit, the scrambler generates a pseudo-random bit (the "feedback") using the polynomial $x^7 + x^4 + 1$. This feedback bit acts as the dynamic key to decipher the data.

Code Implementation: The core logic is found in the `descramble` function in `decode_mac.cc`:

```
// 0. Seed Recovery (from Section 11.1)
// Read the first 7 bits (Service Field) to reconstruct the initial state
int state = 0;
for (int i = 0; i < 7; i++) {
    if (decoded_bits[i]) {
        state |= 1 << (6 - i);
    }
}

// Loop through the payload bits
for (int i = 7; i < d_frame.psdu_size * 8 + 16; i++) {
    // 1. Calculate Feedback (The Key)
    // Polynomial x^7 + x^4 + 1: XOR bit 7 (mask 64) and bit 4 (mask 8)
    feedback = ((!!(state & 64))) ^ (!!(state & 8));

    // 2. Recover Data
    // XOR the calculated feedback with the received scrambled bit
    bit = feedback ^ (decoded_bits[i] & 0x1);

    // 3. Store Result
    out_bytes[i / 8] |= bit << (i % 8);

    // 4. Update State
    // Shift left, discard old MSB, and insert feedback at LSB
    state = ((state << 1) & 0x7e) | feedback;
}
```

11.3 Algorithm Summary

The de-scrambling process for each bit follows these specific steps:

1. **Check Polynomial:** Identify the 7th bit and the 4th bit of the current LFSR state.
2. **Calculate Feedback:** XOR these two bits to generate the feedback bit. This bit represents the scrambling sequence value for the current moment.
3. **Recover Data:** XOR the feedback bit with the received scrambled bit. The result is the original user data.
4. **Update State:** Shift the LFSR state left by one position and insert the feedback bit into the first position (LSB). This prepares the dynamic key for the next iteration.

11.4 Step-by-Step Example

Consider a scenario where the recovered seed is `1011001` and the receiver gets the scrambled sequence `1, 0, 1`.

Iteration 1 (Input: 1)

- **State:** `1011001` (Bit 7 is 1, Bit 4 is 1).
- **Feedback:** $1 \oplus 1 =$ **0**.
- **Data Recovery:** Input $1 \oplus$ Feedback $0 =$ **1**.
- **New State:** Shift left and insert 0 $\rightarrow$ `0110010`.

Iteration 2 (Input: 0)

- **State:** `0110010` (Bit 7 is 0, Bit 4 is 0).
- **Feedback:** $0 \oplus 0 =$ **0**.
- **Data Recovery:** Input $0 \oplus$ Feedback $0 =$ **0**.
- **New State:** Shift left and insert 0 $\rightarrow$ `1100100`.

Iteration 3 (Input: 1)

- **State:** `1100100` (Bit 7 is 1, Bit 4 is 0).
- **Feedback:** $1 \oplus 0 =$ **1**.
- **Data Recovery:** Input $1 \oplus$ Feedback $1 =$ **0**.
- **New State:** Shift left and insert 1 $\rightarrow$ `1001001`.

Result: The recovered data sequence is **1, 0, 0**.

12 Cyclic Redundancy Check (CRC) Validation

The final stage of the decoding process is the CRC, which verifies the integrity of the received frame. After the payload has been de-scrambled, the receiver computes a checksum over the data to ensure no errors remain after the Viterbi decoding step.

12.1 Implementation

The implementation uses the `boost::crc_32_type` library to calculate a 32-bit checksum. The process typically validates the payload (excluding the header or specific fields as defined by the frame format).

In the code, the validation is performed by processing the bytes of the payload (starting from the third byte to skip the Service Field) over the frame size. According to the IEEE 802.11 standard, processing the data including the transmitted FCS (Frame Check Sequence) should result in a specific constant residue value (558161692 in decimal or $0x2144DF1C$ in hex). If the calculated checksum generates this expected residue, the frame is considered valid and published to the output; otherwise, it is discarded as corrupt.

```
// Skip service field (first 2 bytes) and process the payload
boost::crc_32_type result;
result.process_bytes(out_bytes + 2, d_frame.psdu_size);
```

```
// Check against the standard residue for a valid CRC-32
if (result.checksum() != 558161692) {
    dout << "checksum wrong -- dropping" << std::endl;
    return;
}
```

13 MAC Parsing

13.1 Block Overview

The `Parse MAC` block acts as the protocol analyzer of the receiver pipeline. While the preceding blocks handle physical layer demodulation and bit-level integrity, this block interprets the payload as an IEEE 802.11 Data Link Layer frame. Its primary purpose is to extract semantic information (Frame Type, Addresses, SSIDs) and enrich the message metadata for downstream applications (such as Wireshark or user logging).

13.2 Parsing Logic

The block accepts Protocol Data Units (PDUs) and casts the raw byte blob into a `mac_header` structure. It performs bitwise operations on the **Frame Control** field (first 2 bytes) to classify the frame.

1. **Type Extraction:** The block isolates bits 2-3 of the Frame Control field to determine the main category: Management (00), Control (01), or Data (10).
2. **Subtype Decoding:** It isolates bits 4-7 to identify the specific function, such as Beacons (Management), RTS/CTS (Control), or QoS Data.
3. **Address Formatting:** It extracts the 48-bit MAC addresses (Address 1 (Receiver address), 2 (Transmitter address), and 3 (Filtering Address, BSSID or Gateway)) and formats them into standard hexadecimal strings (e.g., `aa:bb:cc:dd:ee:ff`).

13.3 Metadata Enrichment

A key function of the block is populating the PDU metadata dictionary. For example, when a Beacon frame is detected, the block parses the variable-length body to extract the **SSID** string and attaches it to the message.

```
// Code snippet: SSID Extraction logic
// The block locates the SSID tag in the beacon body
std::string s(buf + 24 + 14, *len);
// It adds the SSID to the metadata dictionary
d_meta = pmt::dict_add(d_meta, pmt::mp("ssid"), pmt::mp(s));
```

13.4 Link Quality Estimation

The block also computes an instantaneous Frame Error Rate (FER) by monitoring the 12-bit **Sequence Number** field. By detecting gaps between consecutive sequence numbers, it estimates how many intermediate frames were lost due to channel errors.

$$N_{lost} = Seq_{current} - Seq_{last} - 1 \tag{18}$$

$$FER = \frac{N_{lost}}{N_{lost} + 1} \tag{19}$$

14 GNU Radio & Wireshark Measurements

14.1 Parameter Determination and Experimental Validation

To ensure robust frame detection, extensive testing was conducted to determine the optimal values for the synchronization variables. Specifically, we focused on the Moving Average Window Size (N_{win}) used for smoothing the autocorrelation, and the Decision Threshold (γ) used to trigger the state machine.

14.2 Determination of Moving Average Window Size

The raw autocorrelation output is inherently noisy due to channel impairments. To create a stable detection signal (a ”plateau”), the output is passed through a moving average filter. We experimentally evaluated three distinct window sizes: 16, 96, and 48 samples.

Analysis of Results:

- **Window = 16 (Figure 1):** A window size of 16 corresponds to the length of a single Short Training Symbol. As seen in the figure, this resulted in a highly unstable signal. The ”plateau” is jagged and noisy, making it difficult to define a reliable threshold that avoids false positives.
- **Window = 96 (Figure 2):** A window size of 96 averages over 6 symbols. While this smoothed the noise effectively, it introduced significant distortion. The rise and fall times of the plateau became sloped rather than sharp, blurring the exact start and end of the preamble. This latency can lead to synchronization errors.
- **Window = 48 (Figure 3):** This value was selected as the optimal configuration. Mathematically, it corresponds to averaging over exactly 3 short symbols ($16 \times 3 = 48$). As shown in the figure, this produces a clean, flat plateau with sharp edges, offering the best compromise between noise immunity and temporal precision.

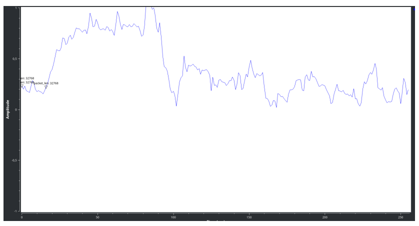


Figure 1: $N_{win} = 16$ Samples

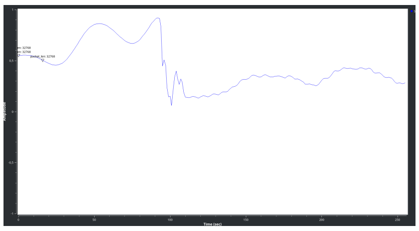


Figure 2: $N_{win} = 96$ Samples

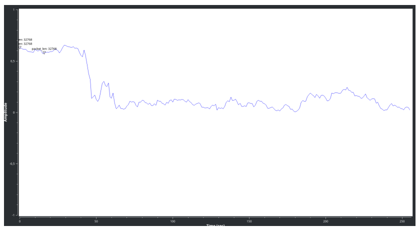


Figure 3: $N_{win} = 48$ Samples

Figure 4: Comparison of autocorrelation smoothing windows. $N = 16$ is too noisy, $N = 96$ is over-smoothed, while $N = 48$ provides an distinct, sharp plateau.

14.3 Threshold Configuration

With the window size fixed at 48, we analyzed the normalized autocorrelation amplitude to determine the detection threshold. Ideally, a perfect signal yields a correlation of 1.0.

As illustrated in Figure 5, the received signal consistently produced a plateau amplitude above 0.9 during the preamble. However, to account for potential fading or lower Signal-to-Noise Ratio

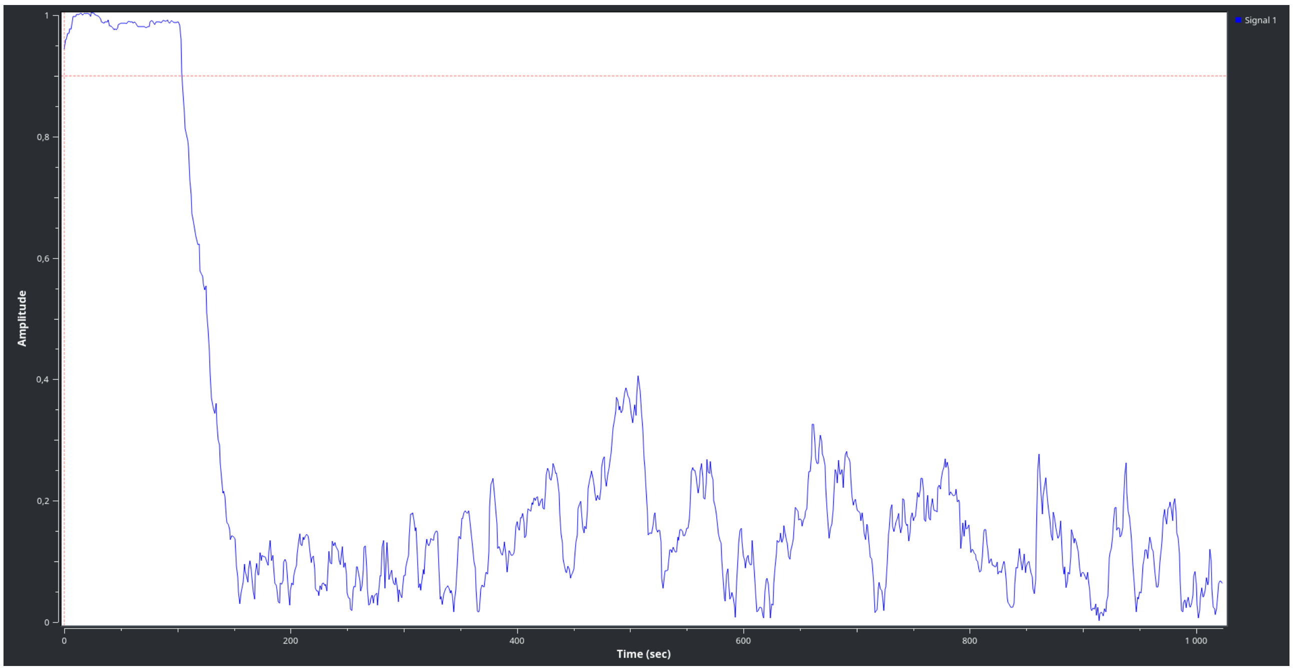
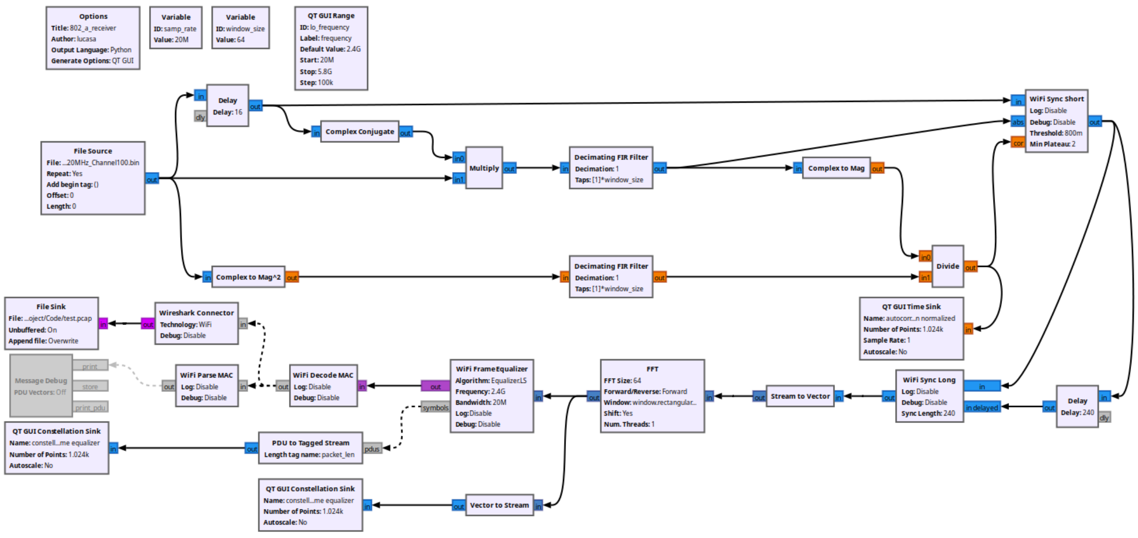


Figure 5: Normalized autocorrelation output showing a stable plateau exceeding 0.9. The red dotted line represents the chosen threshold of 0.8.

(SNR) conditions, we selected a conservative threshold of **0.8** (represented by the red dotted line). This provides a safe margin to prevent missed detections while remaining high enough to reject thermal noise.

14.4 System Validation and Results

After configuring the detection parameters ($N_{win} = 48$, Threshold $= 0.8$), the full receiver flowgraph (Figure 6) was tested against live IEEE 802.11a/g signals.



Options
Title: 802_a_receiver
Author: lucasa
Output Language: Python
Generate Options: QT GUI

Variable
ID: samp_rate
Value: 20M

Variable
ID: window_size
Value: 64

QT GUI Range
ID: lo_frequency
Label: frequency
Default Value: 2.4G
Start: 20M
Stop: 5.8G
Step: 100k

File Source
File: ...20MHz_Channel100.bin
Repeat: Yes
Add begin tag: ()
Offset: 0
Length: 0

Delay
Delay: 16

Complex Conjugate

Multiply

Decimating FIR Filter
Decimation: 1
Taps: [1]*window_size

Complex to Mag

WiFi Sync Short
Log: Disable
Debug: Disable
Threshold: 800m
Min Plateau: 2

Complex to Mag^2

Decimating FIR Filter
Decimation: 1
Taps: [1]*window_size

Divide

QT GUI Time Sink
Name: autocorr...n normalized
Number of Points: 1.024k
Sample Rate: 1
Autoscale: No

File Sink
File: ...oject/Code/test.pcap
Unbuffered: On
Append file: Overwrite

Wireshark Connector
Technology: WiFi
Debug: Disable

Message Debug
PDU Vectors: Off

WiFi Parse MAC
Log: Disable
Debug: Disable

WiFi Decode MAC
Log: Disable
Debug: Disable

WiFi Frame Equalizer
Algorithm: Equalizer.LS
Frequency: 2.4G
Bandwidth: 20M
Log: Disable
Debug: Disable

FFT
FFT Size: 64
Forward/Reverse: Forward
Window: window.rectangular...
Shift: Yes
Num. Threads: 1

Stream to Vector

WiFi Sync Long
Log: Disable
Debug: Disable
Sync Length: 240

Delay
Delay: 240

QT GUI Constellation Sink
Name: constell...me equalizer
Number of Points: 1.024k
Autoscale: No

PDU to Tagged Stream
Length tag name: packet_len

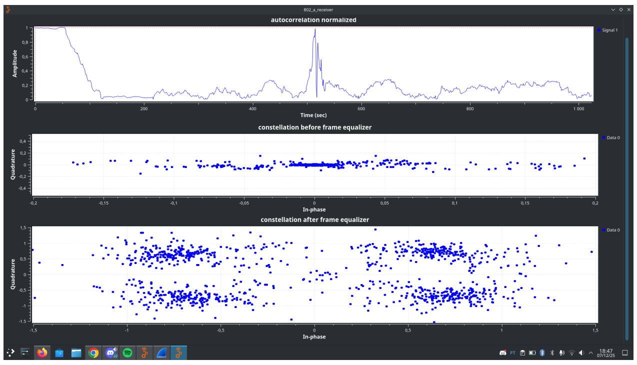
QT GUI Constellation Sink
Name: constell...me equalizer
Number of Points: 1.024k
Autoscale: No

Vector to Stream

Figure 6: The complete GNU Radio receiver flowgraph used for validation.

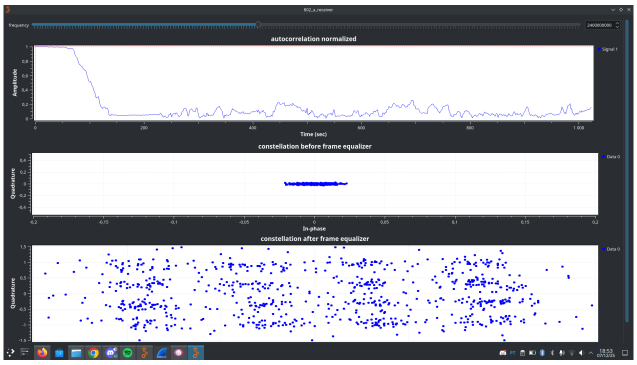
14.4.1 Constellation Analysis

The effectiveness of the synchronization and equalization was verified by observing the constellation plots of the decoded symbols.



autocorrelation normalized
Amplitude
Time (sec)
constellation before frame equalizer
Quadrature
In-phase
constellation after frame equalizer

Figure 7: BPSK/QPSK Constellation



autocorrelation normalized
Amplitude
Time (sec)
constellation before frame equalizer
Quadrature
In-phase
constellation after frame equalizer

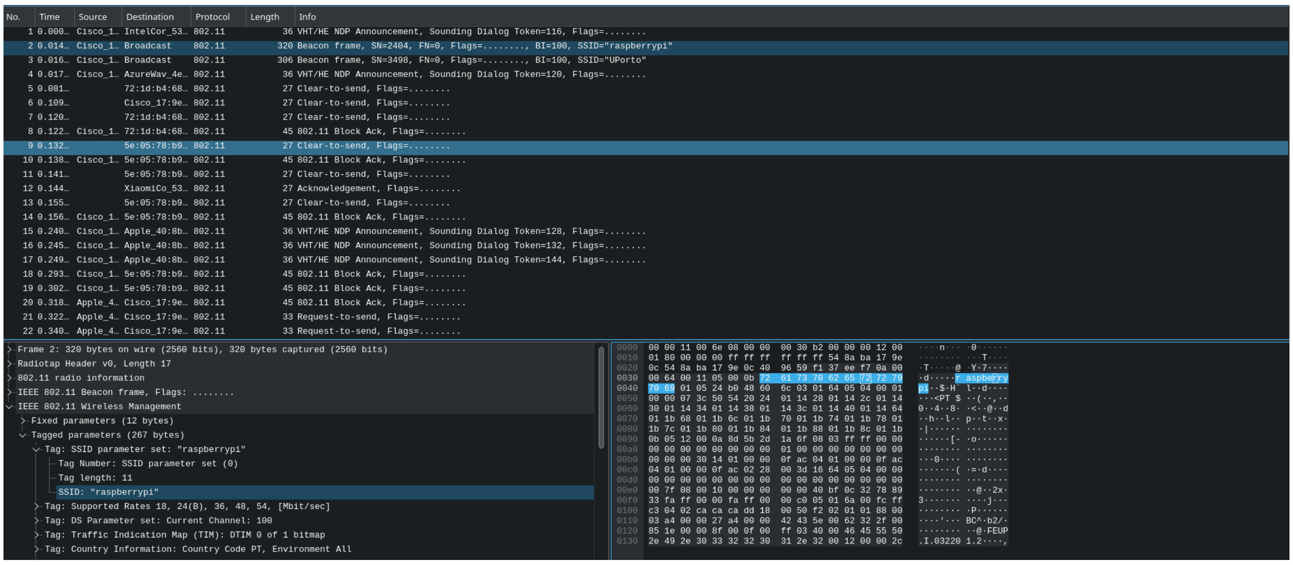
Figure 8: 16-QAM Constellation

Figure 9: Received constellations after frame equalization. The distinct separation of clusters in the 16-QAM plot (Right) indicates precise frequency offset correction and symbol alignment.

Figure 8 specifically demonstrates the receiver's ability to decode higher-order modulations like 16-QAM. The clear separation of the 16 constellation points confirms that the `Sync Short` and `Sync Long` blocks successfully corrected the carrier frequency offset and timing errors.

14.4.2 Packet Decoding Verification

Finally, the output of the receiver was piped to Wireshark to verify that the MAC layer data was being parsed correctly.



```
No. Time     Source    Destination   Protocol  Length Info
  1 0.000…  Cisco_1…  IntelCor_53…  802.11        36 VHT/HE NDP Announcement, Sounding Dialog Token=116, Flags=........
  2 0.014…  Cisco_1…  Broadcast     802.11       320 Beacon frame, SN=2404, FN=0, Flags=........, BI=100, SSID="raspberrypi"
  3 0.016…  Cisco_1…  Broadcast     802.11       306 Beacon frame, SN=3498, FN=0, Flags=........, BI=100, SSID="UPorto"
  4 0.017…  Cisco_1…  AzureWav_4e…  802.11        36 VHT/HE NDP Announcement, Sounding Dialog Token=120, Flags=........
  5 0.081…            72:1d:b4:68…  802.11        27 Clear-to-send, Flags=........
  6 0.109…            Cisco_17:9e…  802.11        27 Clear-to-send, Flags=........
  7 0.120…            72:1d:b4:68…  802.11        27 Clear-to-send, Flags=........
  8 0.122…  Cisco_1…  72:1d:b4:68…  802.11        45 802.11 Block Ack, Flags=........
  9 0.132…            5e:05:78:b9…  802.11        27 Clear-to-send, Flags=........
 10 0.138…  Cisco_1…  5e:05:78:b9…  802.11        45 802.11 Block Ack, Flags=........
 11 0.141…            5e:05:78:b9…  802.11        27 Clear-to-send, Flags=........
 12 0.144…            XiaomiCo_53…  802.11        27 Acknowledgement, Flags=........
 13 0.155…            5e:05:78:b9…  802.11        27 Clear-to-send, Flags=........
 14 0.156…  Cisco_1…  5e:05:78:b9…  802.11        45 802.11 Block Ack, Flags=........
 15 0.240…  Cisco_1…  Apple_40:8b…  802.11        36 VHT/HE NDP Announcement, Sounding Dialog Token=128, Flags=........
 16 0.245…  Cisco_1…  Apple_40:8b…  802.11        36 VHT/HE NDP Announcement, Sounding Dialog Token=132, Flags=........
 17 0.249…  Cisco_1…  Apple_40:8b…  802.11        36 VHT/HE NDP Announcement, Sounding Dialog Token=144, Flags=........
 18 0.293…  Cisco_1…  5e:05:78:b9…  802.11        45 802.11 Block Ack, Flags=........
 19 0.302…  Cisco_1…  5e:05:78:b9…  802.11        45 802.11 Block Ack, Flags=........
 20 0.318…  Apple_4…  Cisco_17:9e…  802.11        45 802.11 Block Ack, Flags=........
 21 0.322…  Apple_4…  Cisco_17:9e…  802.11        33 Request-to-send, Flags=........
 22 0.340…  Apple_4…  Cisco_17:9e…  802.11        33 Request-to-send, Flags=........
```

```
> Frame 2: 320 bytes on wire (2560 bits), 320 bytes captured (2560 bits)
> Radiotap Header v0, Length 17
> 802.11 radio information
> IEEE 802.11 Beacon frame, Flags: ........
v IEEE 802.11 Wireless Management
  > Fixed parameters (12 bytes)
  v Tagged parameters (267 bytes)
    v Tag: SSID parameter set: "raspberrypi"
        Tag Number: SSID parameter set (0)
        Tag length: 11
        SSID: "raspberrypi"
    > Tag: Supported Rates 18, 24(B), 36, 48, 54, [Mbit/sec]
    > Tag: DS Parameter set: Current Channel: 100
    > Tag: Traffic Indication Map (TIM): DTIM 0 of 1 bitmap
    > Tag: Country Information: Country Code PT, Environment All
```

```
0000  00 00 11 00 6e 08 00 00  00 30 b2 00 00 00 12 00   ····n···  ·0······
0010  01 80 00 00 00 ff ff ff  ff ff ff 54 8a ba 17 9e   ········  ···T····
0020  0c 54 8a ba 17 9e 0c 40  96 59 f1 37 ee f7 0a 00   ·T·····@  ·Y·7····
0030  00 64 00 11 05 00 0b 72  61 73 70 62 65 72 72 79   ·d·····r  aspberry
0040  70 69 01 05 24 b0 48 60  6c 03 01 64 05 04 00 01   pi··$·H`  l··d····
0050  00 00 07 3c 50 54 20 24  01 14 28 01 14 2c 01 14   ···<PT $  ··(··,··
0060  30 01 14 34 01 14 38 01  14 3c 01 14 40 01 14 64   0··4··8·  ·<··@··d
0070  01 1b 68 01 1b 6c 01 1b  70 01 1b 74 01 1b 78 01   ··h··l··  p··t··x·
0080  1b 7c 01 1b 80 01 1b 84  01 1b 88 01 1b 8c 01 1b   ·|······  ········
0090  0b 05 12 00 0a 8d 5b 2d  1a 6f 08 03 ff ff 00 00   ······[-  ·o······
00a0  00 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00   ········  ········
00b0  00 00 00 30 14 01 00 00  0f ac 04 01 00 00 0f ac   ···0····  ········
00c0  04 01 00 00 0f ac 02 28  00 3d 16 64 05 04 00 00   ·······(  ·=·d····
00d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········  ········
00e0  00 7f 08 00 10 00 00 00  00 00 40 bf 0c 32 78 89   ········  ··@··2x·
00f0  33 fa ff 00 00 fa ff 00  00 c0 05 01 6a 00 fc ff   3·······  ····j···
0100  c3 04 02 ca ca ca dd 18  00 50 f2 02 01 01 88 00   ········  ·P······
0110  03 a4 00 00 27 a4 00 00  42 43 5e 00 62 32 2f 00   ····'···  BC^·b2/·
0120  85 1e 00 00 8f 00 0f 00  ff 03 40 00 46 45 55 50   ········  ··@·FEUP
0130  2e 49 2e 30 33 32 32 30  31 2e 32 00 12 00 00 2c   .I.03220  1.2····,
```

Figure 10: Wireshark capture showing successfully decoded Beacon frames. The receiver correctly parsed the SSID ”raspberrypi” and ”UPorto”.

As shown in Figure 10, the system successfully identified Beacon frames and parsed variable-length fields such as the SSID (”raspberrypi” and ”UPorto”). The presence of valid checksums and coherent header information confirms the integrity of the entire receive pipeline. number of points in constellation sinks, the delay, fft size, window_size, etc.