* Notice: this is an **automatic translation** for convenience only*

ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA

CESENA CAMPUS

DEPARTMENT OF COMPUTER SCIENCE – SCIENCE AND ENGINEERING

DEGREE COURSE IN ENGINEERING AND IT SCIENCE

Python implementation of the

Olivia digital modulation

Elaborated in

NUMERICAL METHODS

Speaker

Damiana Lazzaro

Presented by

Federico Santandrea

Single session

Academic Year 2019/20

Table of Contents

Introduction.....5
Theoretical Background.....7
1.1 Discrete Fourier Analysis.....10
1.2 Fast Transform Algorithms.....12
1.3 Walsh-Hadamard Transform.....13
Olivia Modulation.....16
2.1 Overview.....16
2.2 Modulation Stages.....19
2.2.1 Buffering and Housekeeping.....19
2.2.2 Character to Vector Mapping.....20
2.2.3 Inverse Fast Walsh-Hadamard Transform.....20
2.2.4 Scrambler.....21
2.2.5 Interleaving.....22
2.2.6 Gray Coding.....22
2.2.7 Generation of tones.....24
2.2.8 Audio frequency transmission.....25
2.3 Demodulation stages.....26
2.3.1 Fast Fourier Transform.....26
2.3.2 Gray decoding.....27
2.3.3 Buffering.....27
2.3.4 De-interleaving.....28
2.3.5 Descrambler.....28
2.3.6 Fast Walsh-Hadamard Transform.....28
2.3.7 Vector-to-character mapping.....29
Implementation details.....30

3.1 Command line interface.....31
3.2 Asynchronous transmission using a send queue.....32
3.3 Tone generation.....37
3.4 Error detection.....39
3.5 Walsh-Hadamard transform sorting.....44
Experiments and measurements.....46
4.1 Interoperability with reference implementations.....46
4.2 No-noise performance.....47
4.3 White noise performance.....48
4.4 Structured noise performance.....53
4.4.1 Music.....54
4.4.2 Speech.....55
4.5 Radio frequency transmission experiment.....56
Conclusions.....60
Bibliography and webography.....62

Introduction

This thesis aims to reconstruct and illustrate in a complete way the structure of the Olivia transmission system starting from the available documents, arriving at the development of a practical and clear implementation in Python language whose performance will be measured in simulated scenarios and in the real world.

Olivia is a digital modulation for the communication of textual data in high noise conditions, created for use in the amateur radio field where high resistance to disturbances is necessary and high transmission speeds are not required. It is a non-trivial transmission system that combines various numerical processing and error correction techniques.

At present, the formal specification of Olivia consists substantially of the reference implementation in C++ and a non-complete summary document. There is currently a limited choice of free or commercial computer programs that support it. A streamlined and easily understandable implementation in a modern programming language is not yet available.

Since its first incarnations, Python has experienced and is experiencing an intense evolution from a simple scripting language to a complete ecosystem. This makes it an increasingly attractive platform for scientific and numerical research and calculation. The wide availability of open source libraries of all types allows the verification of the implementations of the algorithms by the scientific community. The effectiveness of the syntax and the high degree of abstraction of the underlying hardware and interfaces give the developer the freedom to focus on the solution of the problem in question without spending too much energy on low-level management, making it a valid rapid prototyping system. At the same time, the solidity and efficiency achieved after years of improvements make it suitable for the final generation of stable and optimized production code. These characteristics make it a natural choice to pursue the objectives set out above.

The thesis is organized as follows:

• in Chapter 1, “Theoretical Background”, the concepts underlying digital modulations and digital signal processing, necessary for understanding the subject, will be illustrated.

• in Chapter 2, “Olivia modulation”, the structure of the Olivia transmission system will be shown, identifying and describing the functional blocks that make up the modulator and the demodulator;

• in Chapter 3, “Implementation Details”, the technical solutions chosen for the implementation will be highlighted, reporting and commenting on the most significant code fragments;

• in Chapter 4, “Experiments and Measurements”, the results of practical transmission and reception tests carried out using the implemented programs will be reported, collecting quantitative data on the performance in various scenarios;

• in Chapter 5, “Conclusions”, the conclusions drawn during the previous phases will be summarized.

Chapter 1

Theoretical background

This chapter illustrates the concepts underlying digital modulations and digital signal processing, which are necessary for understanding the subject.

A modulation, in the classical sense of the term, is a transmission technique used to impress the information contained in a signal that is continuous in time and values (called modulating) on a waveform (called carrier, typically sinusoidal) having a frequency orders of magnitude higher than the maximum frequency of the modulating signal and whichtherefore better adapts to the characteristics of the transmission channel. This impression is obtained by modifying some property of the carrier signal as a function of the behavior of the analog modulating signal. [LIT-1]

A digital modulation is a technique used to transmit data (therefore sequences of symbols) rather than analog signals. Conceptually, one could consider any sequence of digital symbols as a modulating signal characterized by the succession of different values of amplitude or frequency, or any combination of them, and then apply an analog modulation to this hypothetical signal. In practice, however, the fact that the modulating signal is discrete and can assume only a finite number of values allows a notable simplification of the theoretical treatment and also of the practical implementation.

In fact, in this case it is not usually necessary to resort to complicated non-linear analog circuits or complex digital algorithms to obtain the modulated signal, but it is possible to generate it directly, even using software synthesis techniques that require little computing power. For example, two tones of different frequencies and equal duration can be used to represent the bits 0 and 1; this is the simplest FSK (Frequency Shift Keying) modulation.

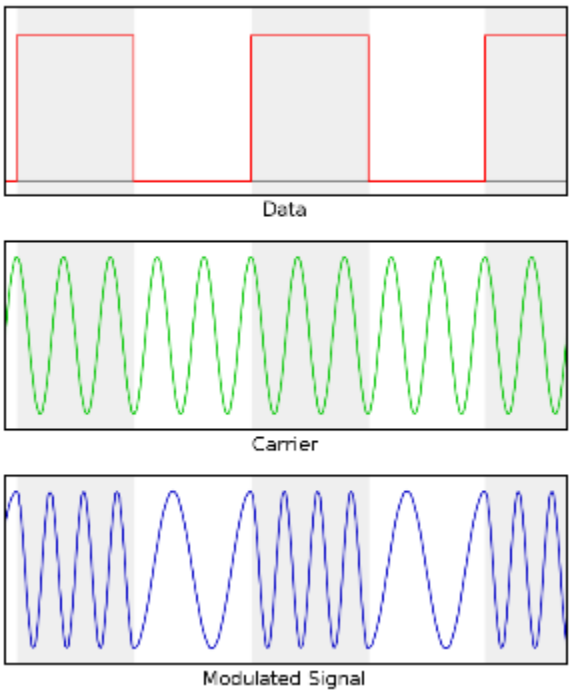


Figure 1: A simple FSK modulation. [WIKI-1]

Of course, it is possible to use more than two tones: in this case, the detection of a specific tone will contain more than one bit of information. With n tones, the information obtained from the detection of a tone will be equal to log2(n) bits.

There is a wide variety of digital modulations that, starting from a data flow, generate a baseband waveform with a very limited width spectrum, typically included within a range that goes from 500 to 2500 Hz. These modulations are also called sound card modes, because they lend themselves to being generated and interpreted by computers, using normal sound cards for the emission and input of the signal. Typical consumer-grade sound cards integrated into every modern PC work at a sampling frequency of 44100 Hz, which is much higher than the Nyquist frequency of a baseband signal with a maximum frequency of 2500 Hz. Using these sound card modes, it is possible to exchange digital data via audio signals, taking the signal generated by the speaker output of the card and vice versa by inserting the received signal into the microphone input. It is also possible to achieve radio frequency communication, by interposing and between the transmitting and receiving computers

radio transceiver equipped with SSB (single-sideband) modulation. SSB modulation, technically an amplitude modulation without the carrier and one of the two sidebands, essentially consists in the translation to the right of the spectrum of the modulating signal, by an amount equal to the frequency of the carrier signal. The audio signal resulting from the digital modulation, produced by the computer, can then be supplied as input to an SSB transmitting apparatus, which will take care of translating the spectrum of the signal upwards to a frequency appropriate to the desired propagation. In reception, after having performed the reverse translation using an SSB receiver, the audio signal obtained as input will be supplied to the computer, which will take care of the digital demodulation. With this technique, computers and wiring are freed from the complications of radio frequency electronics, and one simply works in digital processing on a low frequency signal that can be acquired and interpreted in software without particular problems.

Generating the modulated signal in the case of a digital modulator is therefore simple: it is sufficient to generate and emit the individual samples of the output waveform theoretically expected for the modulation in use. Demodulation is also theoretically simpler and in the most basic cases it can be carried out with direct methods. For example, the frequency of a pure sinusoidal signal can be determined simply by counting how many times the waveform changes sign in a unit of time (zero crossing); in the case of FSK, it is possible in this way to distinguish sinusoidal tones of various frequencies associated with different symbols in reception.

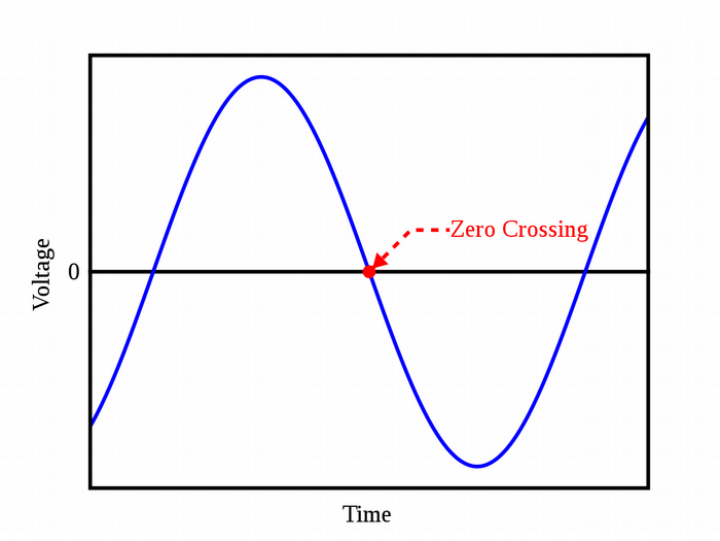


Figure 2: Frequency detection by counting zeros. [WIKI-2]

1.1 Discrete Fourier Analysis

The main tool for waveform analysis remains the discrete Fourier transform (DFT). Although it is a relatively computationally expensive method, it provides the receiver with all the information on the phase and amplitude of the sinusoidal tones that make up the signal. This opens up many possibilities for imprinting the information content of the symbols in the waveform generated during transmission.

The transform converts a finite sequence of equally spaced samples of an analog signal into a finite sequence of equally spaced samples of its frequency spectrum. It therefore represents a discretization of the Fourier transform. This discretization allows the calculation to be approached numerically, without having to find a way to express the input signal in symbolic form and then derive the spectrum. For this reason, it is very well suited to implementation with electronic calculators.

Assuming that the waveform of a signal to be transformed is periodic, the corresponding spectrum will be discrete and will correspond to the sequence of coefficients of the Fourier series. If we had to transform any portion limited in time of a waveform that is generally non-periodic, we would instead obtain a continuous spectrum; however, it is possible to consider the periodic repetition of the portion considered as a waveform, thus obtaining in this case too a discrete sequence of coefficients.

Furthermore, if the input waveform is sampled, i.e. expressed as a sequence of coefficients that indicate its values assumed at discrete intervals of time, the properties of the transform will result in the sequence of coefficients of the Fourier series being periodic and, therefore, it will be possible to consider a single period without any loss of information. The combination of the limitation in time to a portion of the input signal and its discretization therefore allows us to express its harmonic content in the form of a discrete and finite sequence of coefficients.

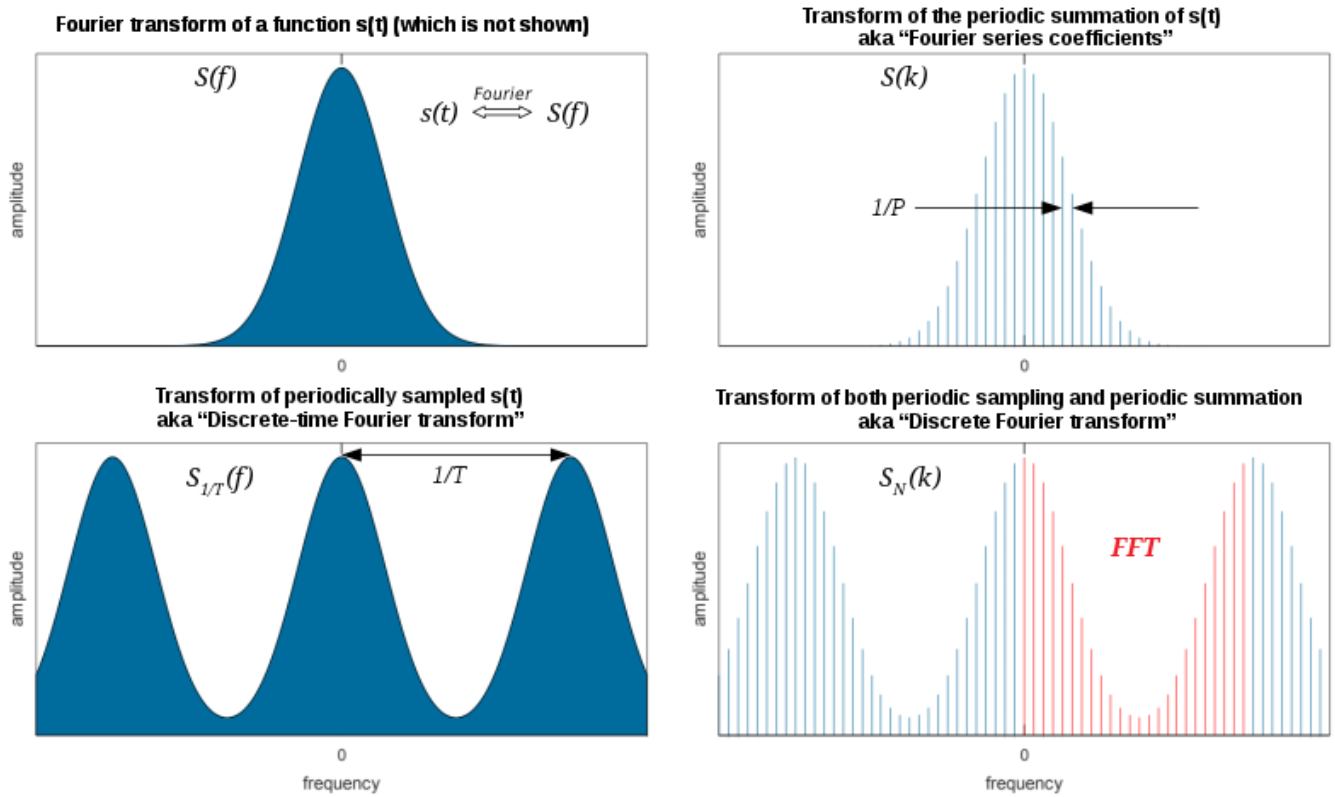


Figure 3: Visualization of the discretization of the Fourier transform. [WIKI-3]

The sequence of waveform samples is related to that of the spectrum samples by the following formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-ik\frac{2\pi}{N}n} \qquad k = 0, \ldots, N-1$$

where Xk is the sequence of the spectrum samples, xn the sequence of the waveform samples and N the length of the sequences. It can be noted that each sample of the spectrum depends on each sample of the waveform. The complex values of the sequence obtained each represent the phase and amplitude of a sinusoidal component of the waveform of the original signal. [LIT-2] The amount of information made available by the transform allows the implementation of modulations in amplitude, frequency and even quadrature. By combining the power of digital processing, it can be implemented time division multiplexing, frequency division multiplexing, spread spectrum and much more. This flexibility

makes an almost obligatory step after the acquisition of the original analog signal in the reception phase, except for the simplest modulation cases.

In theory, the Fourier transform, and thus also its discretization, would operate on complex and non-real values. However, from the sampling of a real analog signal, totally real samples result. The output result from a discrete Fourier transform made on real samples will be a sequence of complex values (spectrum samples) symmetrical with respect to the center. The values are complex because each of them has an absolute amplitude and a phase and this allows to maintain the information on the phase relationship that exists between the various frequency components.

In many cases, such as the one examined in this document, the information on the phase relationship can be ignored, for example when one wants to identify the presence or absence of a certain component it will be sufficient to check that the amplitude exceeds a certain threshold. In these situations, the output values can be considered in their absolute value. Under these simplifying conditions, we can therefore consider an algorithm for calculating the discrete Fourier transform that has as input a sequence of real values and as output another sequence of values that are also real.

1.2 Fast transformation algorithms

Given the usefulness of the discrete Fourier transform, recognized even before the advent of electronic calculators, much research effort has been invested in trying to make its calculation easier. Manually performing all the required steps was in fact a heavy job. In the literature, it is possible to find various fast algorithms developed to calculate the DFT.

One in particular, the Cooley-Tukey algorithm, has become extremely popular for software implementations because it can easily work with a number of samples that is a power of 2. It is a recursive algorithm that is based on the expression of the DFT in two DFTs of half the size.

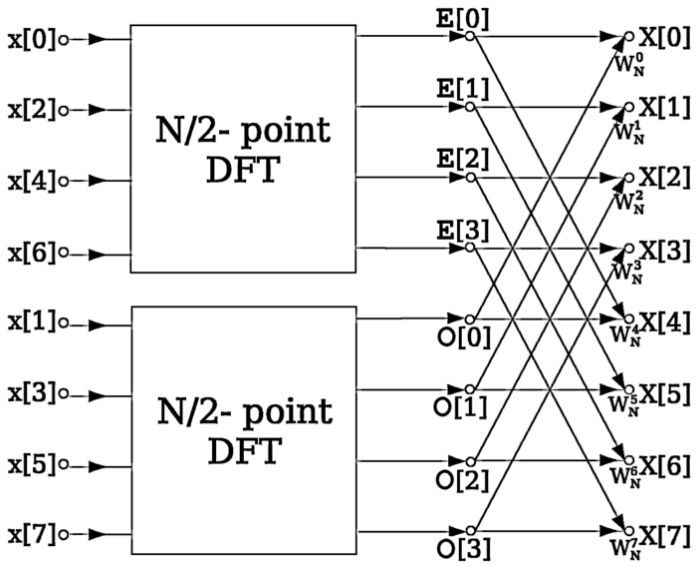


Figure 4: Conceptual scheme of the Cooley-Tukey algorithm. [WIKI-4]

Since the number of samples is even, it is possible to compute separately the discrete Fourier transforms of the sequence of the even terms and that of the odd terms and express the initial transform as a combination of them:

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{\frac{-2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{\frac{-2\pi i}{N}(2m+1)k}$$

Furthermore, since N is a power of 2, the length of the resulting transforms will also be even and therefore it is possible to recursively apply the decomposition until reaching elementary calculations.

The application of this algorithm allows to significantly reduce the complexity of calculating the DFT, from O(n2) to O(n log n). At the computing powers currently available, the application to signals acquired in real time becomes possible.

1.3 Walsh-Hadamard transform

The Walsh-Hadamard transform, belonging to a generalized class of Fourier transforms, is a linear operation that acts on a set of real numbers whose cardinality is a power of 2 (2m). It can be considered as a combination of several DFTs of dimension 2, and is effectively equivalent to a multidimensional DFT with dimension 2x2x2...x2, with many dimensions how many numbers are considered.

It is used to decompose the input vector into a superposition of Walsh functions (in the same way that the DFT decomposes the input vector into a superposition of sinusoidal functions). These Walsh functions are all orthogonal and therefore it can be said that every discrete function in time has a Walsh spectrum. Similarly, the transform can be inverted, reconstructing the original function from the spectrum. In practice, the transform is calculated by means of a matrix (Hadamard matrix) defined recursively as follows:

$$H_m = \frac{1}{\sqrt{2}} \begin{pmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{pmatrix}$$

where H_0 is assumed to be the identity matrix. The multiplicative constant often, as in the case of the implementation in analysis, can be omitted. [LIT-3]

Each of the functions that form the basis for the transform has a different number of zero crossings. These functions can be ordered in different ways, which correspond to a different ordering of the rows of the Hadamard matrix: it is evident that to successfully invert the function it is necessary to use the same ordering of the basis in both directions.

This transform also has its own fast numerical implementation:

the FWHT (Fast Walsh-Hadamard Transform). Conceptually similar to the DFT algorithm, it is based on the recursive calculation of transforms on a half number of elements, until the desired result is obtained, and naturally descends from the recursive definition of the WHT reported above.

The FWHT algorithm, formally expressed in the implementation code illustrated in chapter 3, uses only additions and subtractions between real numbers and is therefore much less expensive to execute than the DFT algorithm, even using the Cooley-Tukey method. Furthermore, since the resulting spectrum is composed only of real numbers, the memory requirements for representing it in the computer are also lower.

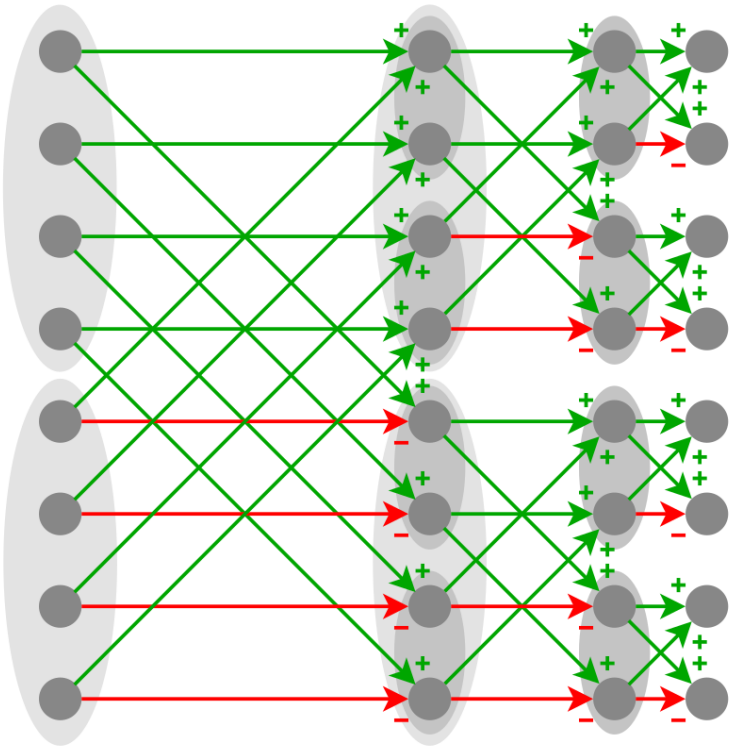


Figure 5: Schematic showing the recursive combination of values in smaller WHT.[WIKI-5]

The interesting property of this transform is that it can be used to implement error correction schemes. In fact, if you alter (introduce error) a not too large amount of elements of the output vector, applying the inverse transform to the altered vector you will obtain a function similar to the original one. By carefully selecting a subset of possible combinations that can be applied to the input, it is possible to determine the one with which the reconstructed signal has the greatest similarity, and thus conclude that with a high probability it must have been the original input.

A simple way to exploit this property is to limit the possible input vectors to those with a single non-zero element. By altering the output values of the transform, and then applying the inverse to the altered vector, you will typically obtain a vector with more than one non-zero element. However, one of these elements will be larger than the others and will be the one originally valued; the lower values of the other elements are interpretable as noise/error.

By encoding the information in the index of the single valued element, we obtain an effective way of detecting and correcting the error applied to the output values.

Chapter 2

The Olivia modulation

This chapter will show the structure of the Olivia transmission system, identifying and describing the functional blocks that make up the modulator and the demodulator.

2.1 Overview

The Olivia modulation is a sound card mode developed in 2003 by Pawel Jalocha for amateur radio use. The main goal was to create a text communication channel that is very resistant to noise, particularly on channels subject to variable or intermittent attenuations (fading). To achieve this, it is necessary to introduce error correction techniques and therefore to compromise on transmission speed and spectral efficiency; consequently, in standard configurations, the Olivia signal has a relatively wide bandwidth for a sound card mode and a much lower transmission speed than modulations with similar bandwidth requirements.

There is no definitive official specification, however a preliminary draft is available on the ARRL website at the web address http://www.arrl.org/olivia [ARRL-1]. The author's reference implementation, published on the web by the author and subsequently archived on GitHub, available in the repository at the address https://github.com/mrbostjo/Olivia_MFSK [GITH-1], can be considered the formal specification.

The fundamental principle is to use low-rate sequences of tones, chosen from an overwhelming number of possibilities and distributed over a wide bandwidth. In addition to this, there are scrambling, interleaving and encoding stages that statistically help to reduce the incidence of errors.

Olivia can operate in different configurations, which differ in the number of tones used and the bandwidth occupied. Some of these configurations allow slightly higher, while others increase the level of robustness to noise.

In this document, an Olivia configuration will be identified with the syntax n/b@f , where n is the number of equally spaced tones to be used, b the width of the band within which they are equally distributed, and f the central frequency in the baseband of the signal spectrum.

The configuration that best demonstrates a valid balance between performance and resistance is the one with 32 tones distributed over a 1 KHz band. It is customary to center the spectrum of the modulated signal in the available baseband; since this band, in sound card modes, is typically 3 KHz, a central frequency for the modulated spectrum of 1500 Hz is obtained. The resulting configuration is therefore 32/1000@1500, and this is the one that will be used for the experimental tests included in this document. The implementation presented is however able to operate also on different configurations.

The actual transmission speed (symbol rate) depends on the configuration according to the formula:

$$f_s = \frac{b}{n}$$

The symbol rate therefore coincides with the frequency spacing between one tone used and the next. By widening the band with the same number of tones used, the symbol rate increases; similarly, by decreasing the number of tones used, the symbol rate increases with the same bandwidth, thus maintaining a certain degree of noise resistance.

The various stages of modulation, detailed below, form a theoretically two dimensional error correction code. The author's desire was to create an iterative algorithm to exploit this two-dimensionality in order to obtain a more accurate demodulation, but this result has not yet been achieved.

Olivia is designed for the transmission of ASCII text, i.e. 7-bit characters. If you want to use it to transfer byte streams, such as files or UTF-8 text strings, you need to perform a preliminary transformation of the byte stream into a stream of ASCII characters. A standard way to to achieve this result is the Base64 encoding, RFC 4648: (https://tools.ietf.org/html/rfc4648)

Mainly used for file transfer via e-mail, Base64 produces an output stream composed exclusively of numbers and uppercase and lowercase letters plus two special characters, i.e. a subset of the 7-bit ASCII characters. The encoding causes an expansion of the content, in fact 24 input bits are mapped onto 4 7-bit characters, for a total of 28 bits. This introduces an overhead of about 17%,in comparison with the possibility of modulating arbitrary bytes. After demodulation, Base64 decoding will be applied to the ASCII character stream obtained, thus reconstructing the original byte stream.

2.2 Modulation stages

The block diagram shows the conceptual stages through which information passes during modulation, from the digital source to becoming an analog audio signal.

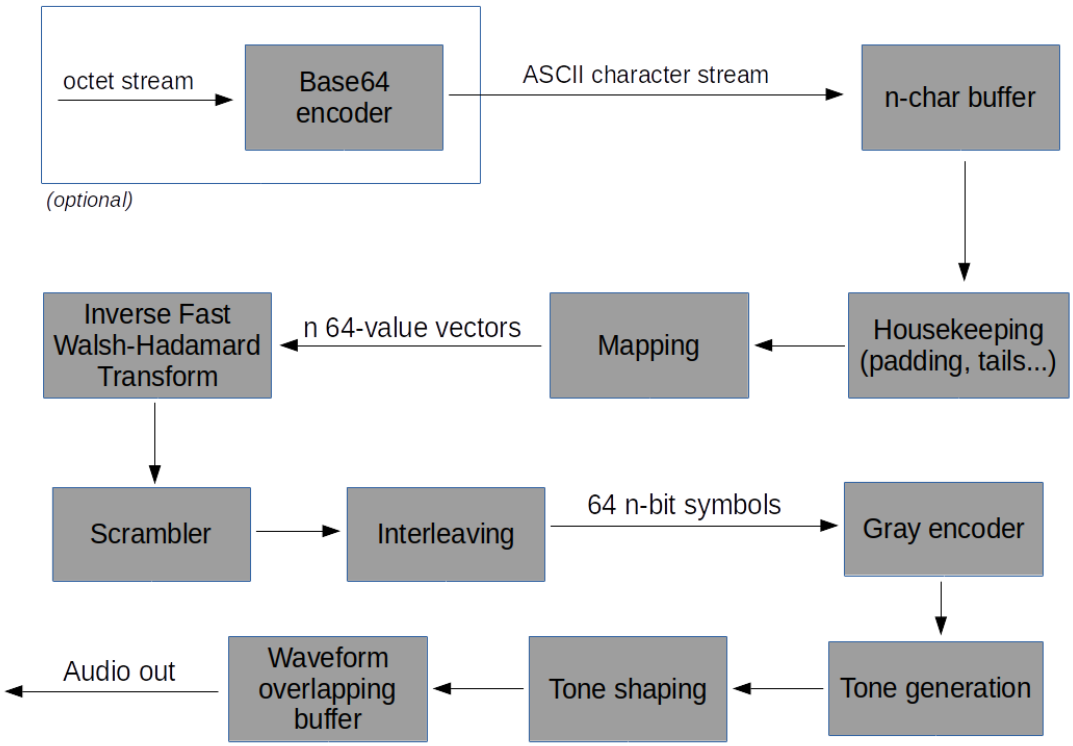


Figure 6: Block diagram of the Olivia modulator.

The individual stages of modulation are illustrated and discussed below.

2.2.1 Buffering and housekeeping

The characters are transmitted in blocks of log2(n), where n is the number of tones available in the chosen configuration. For example, using a 32-tone configuration, the characters will be transmitted in groups of 5. Since these are 7-bit characters, there will be an information content equal to 7n bits per block; in the example with n=32, there will be 35 bits of useful information transmitted per block.

It is therefore necessary to provide a stage to accumulate the flow of incoming characters and deliver only blocks to the next stage complete with log2(n)

characters, possibly completed with padding of ASCII 0 (NUL) characters in the case that the transmission ends with a number of characters that is not a multiple of the length of the block. To simplify the manual tuning of radio frequency receivers, two “tails” are also transmitted; one before the start of the transmission of the blocks, called preamble; and a final one, after the last block. The tails are composed of two alternating repetitions of the lowest and highest tone in the chosen configuration, and do not contain data.

2.2.2 Mapping from characters to vectors

The transformation function used in the next stage, for reasons explained later, requires as input vectors of 64 values, all null except one. A simple encoding is adopted to convert the 7-bit ASCII characters into vectors of this type:

$$w_i = \begin{cases} 1 & q < 64, i = q \\ -1 & q \geq 64, i = q - 64 \quad \forall \; i \in [0;63] \\ 0 & altrimenti \end{cases}$$

(otherwise)

where w indicates the vector being constructed, q the character to which it corresponds. Note that since q is a 7-bit ASCII character, its value will always be between 0 and 127; for each of these values of q, one and only one element of w will be different from zero.

At the output of this stage, therefore, n vectors of 64 values will be obtained, each of which corresponds to one of the n characters of the block output from the previous buffer.

2.2.3 Inverse Fast Walsh-Hadamard Transform

The inverse Walsh-Hadamard transform is applied to the individual vectors obtained. This introduces a first dimension of error control.

The reason why the inverse transform is used in the transmission phase is that the vector generated in the previous state, having a single valued element, can be considered as the simplest Walsh spectrum of a function: a single line with a

certain unit amplitude. Inverting this spectrum will therefore obtain a Walsh basis function.

Each input vector therefore corresponds to a specific transformed output vector. As will be seen later in the demodulation phase, this step introduces a level of redundancy for which it will be possible to trace the original input vector even in the presence of alterations on a sufficiently low number of values of the generated vector.

2.2.4 Scrambler

A possible type of noise on the channel is the narrow-band one, for example a sinusoidal tone with a fixed frequency, always present on the channel. The presence of this type of disturbance can make it significantly difficult to discriminate between adjacent tones in its immediate vicinity, introducing a greater probability of errors whenever these tones are used. To counteract this possibility it is appropriate to ensure that the tones are statistically used with uniform probability.

A stage is then introduced that combines the output values from the WH transform with a pseudo-random vector. This vector has been chosen and standardized once and for all, and is:

$$k = (E\,257\mathrm{E}6\,D\,0291574\,EC)_{16}$$

This key is applied directly in XOR bit by bit to the first vector of a block; for the subsequent vectors in the same block, it is rotated each time by 13 bits to the right before application, so as to generate different sequences even in the presence of equal characters in the same block.

The output values from this stage will have an almost uniform distribution. However, since a known and established a priori pseudo-random key value is used, it will be possible to deterministically invert this step during reception.

2.2.5 Interleaving

Another type of noise that can be encountered is the one with a broad spectrum but limited in time. A typical source of this type of noise is the electrostatic discharge, frequently spontaneous in the atmosphere or generated by human activities. This situation can introduce a decision error on the decoding of one or a few symbols around a certain instant of time. We want to avoid that the decision error on a symbol causes the incorrect decoding of characters.

A symbol contains log2(n) bits of information, a number by construction coinciding with the length of the block in characters. A vector reshuffling scheme is introduced to ensure that each bit of a symbol carries information relating to a different vector, so that each character of the block has a contribution in determining each symbol to be emitted. Consequently, in the reception phase, any error on a symbol will be distributed over all the vectors that compose a block, thus lending itself very well to being corrected by the Walsh-Hadamard transform.

The arbitrarily decided and standardized interleaving scheme for Olivia is the following:

- for each vector W constituting the block and defined w as its index,
  - for each symbol S constituting a block and defined s as its index,
    - let q = (w - s) mod log2(n),
    - let the w-th bit of the symbol S be equal to the s-th bit of the q-th vector.

2.2.6 Gray coding

The frequencies of the audio tones that will be generated by the subsequent stage, corresponding to the various possible symbols, have very small distances on the spectrum. Consequently, the most probable error that can be made during the reception phase is to confuse a tone with the previous or following one. To limit the error made in this scenario, Gray coding is used on the symbols to be sent.

Gray coding, invented in 1953 by Frank Gray, was introduced to represent digital values output from real devices (such as series of switches or rotary encoders). In these devices, switches or microswitches are mechanically actuated by a structure, usually rotating. Using the normal binary coding of integers, a problem is encountered in the transition between two adjacent states whose binary representations differ by more than one bit: in fact, it is physically not possible to create switches that change state at the same exact instant. This will result in fleeting, unwanted intermediate transitions to one or more irrelevant states (alea).

Gray coding associates integers with binary strings constructed in such a way as to minimize the number of bits that vary during a transition between two adjacent states. An n-bit Gray coding can be constructed with a recursive algorithm:

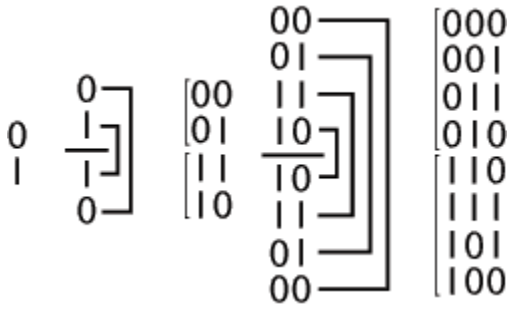


Figure 7: Construction of a Gray coding. [WIKI-6]

1. Write the numbers 0 and 1 in vertical columns.
2. Reflect the strings downward along an imaginary line under the last one.
3. Add a 0 to the beginning of the strings above the line, and a 1 to the beginning of those below.
4. Repeat from point 2 until you obtain an encoding of the desired length.

As you can see, the modification distance between a string and an adjacent one is constant and always equal to 1 bit.

In the demodulation phase, the error committed by exchanging a symbol for an adjacent one varies from 1 to n bits depending on the case. Let's assume a 32-tone scheme, i.e. with 5-bit symbols.

The error committed by exchanging tone 16 (10000) with tone 17 (10001) would be 1 bit, while the error committed by exchanging tone 15 (01111) with tone 16 (10000) would be 5 bits.

By applying Gray coding to the indices of the symbols obtained at the output of the previous stage, the error committed by exchanging a symbol with an adjacent one is minimized. In fact, the number of bits that differ between the Gray encoding of a symbol and that of the previous (or next) is, as a consequence of what has been seen above, constant and equal to the minimum possible of 1.

A simple algorithm to implement Gray encoding on a computer, also used in this implementation, is to perform the bit-by-bit XOR of the value with the value shifted to the right by one position.

2.2.7 Tone generation

The tones represented by the symbols obtained are synthesized directly, calculating the samples to be emitted for each instant of time. Knowing the modulation parameters (number of tones, center frequency and bandwidth) it is possible to calculate the values of the samples as follows:

$$f_t = (f_c - \frac{B}{2}) + \frac{f_{sep}}{2} + f_{sep} N$$
$$f(k) = A \sin(2 \pi f_t k + \phi_0)$$

where:
• k is the time instant corresponding to the sample;
• fc the central frequency of the modulation;
• B the bandwidth of the modulation;
• fsep the separation between one tone and the next (i.e. B/n);
• N the number of the tone to be produced;

• A multiplicative factor used to adjust the output signal power;
• ϕ0 the phase shift.

The phase shift is chosen randomly and consists of a delay or advance of π/2. It is introduced to avoid the generation of continuous sinusoids in the (unlikely) case of continuous production of the same tone, thus limiting the receiver synchronization problems.

The samples of the produced tone are modeled by a pre-established and shared filter function. The filter function is the following:

$$1+1.1913785723\cos(f(k))-0.0793018558\cos(2f(k))$$
$$-0.2171442026\cos(3f(k))-0.0014526076\cos(4f(k))$$

The coefficients of this function have been determined experimentally to limit intersymbol modulation as much as possible and are part of the specification.

2.2.8 Audio frequency transmission

The tones thus modeled are formed by a first phase in which their amplitude harmonically increases, and a second in which it decreases. An output audio buffer is set up to combine the waveforms so that the second half of one tone overlaps the first half of the next. In this way the output signal is free of sudden variations and therefore has a low harmonic content that does not dirty the transmitter output signal and does not interfere with any other adjacent signals.

In each sample window it is always possible in the reception phase to determine the prevalence of one tone over the other, measuring its amplitude and considering the strongest one. In fact, the phase of descending amplitude of a tone overlaps with that of ascending amplitude of the following tone, obtaining a gradual transition from the predominance of one tone to that of the other.

2.3 Demodulation stages

The block diagram shows the conceptual stages through which information passes during demodulation, from the analog audio signal at the input to the reconstructed digital stream at the output.

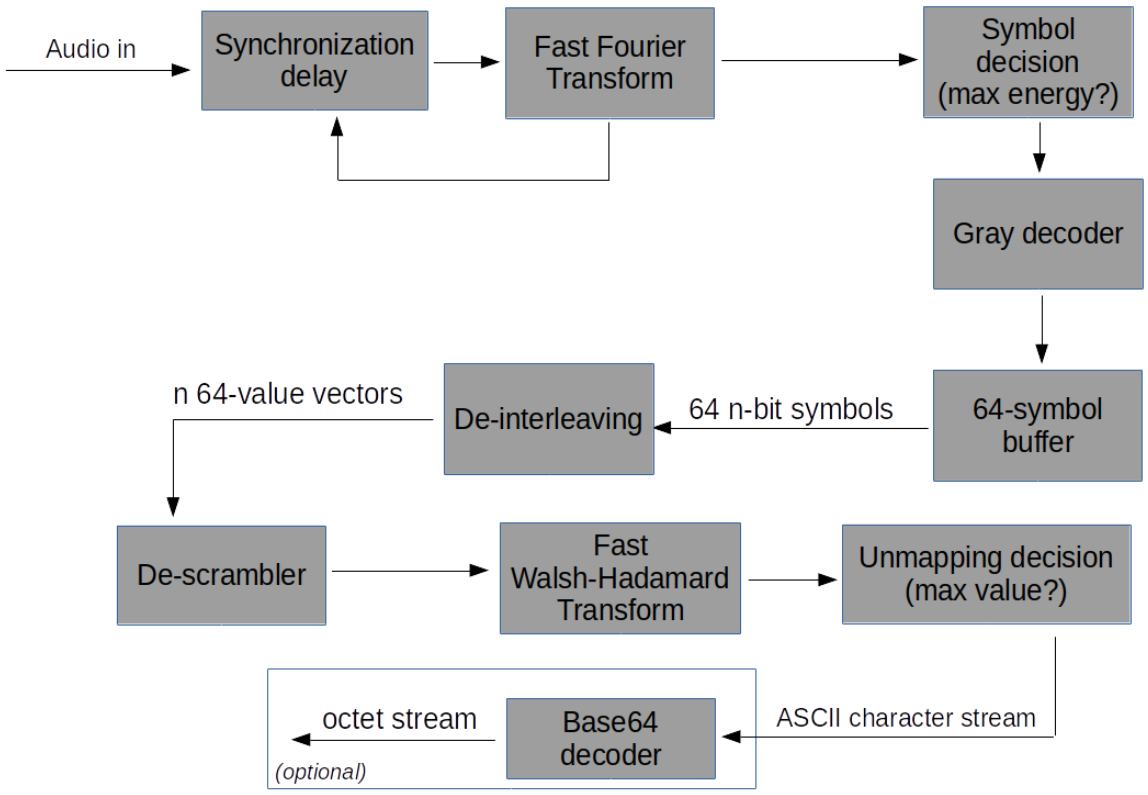


Figure 8: Block diagram of the Olivia demodulator.

The individual stages of demodulation are illustrated and discussed below.

2.3.1 Fast Fourier Transform

The Olivia modulation is designed to make the most of this fast transform. The sampling frequency typically used for Olivia (8000 Hz) is chosen so that the frequencies of the various tones used are, in standard configurations, exactly 1 apart on the discrete spectrum. This can simplify the implementation of demodulators.

The amplitudes of all frequencies corresponding to the possible tones are measured and it is determined which tone is most likely to be correct, discarding the possibilities that are most likely determined by noise.

To synchronize the transform window with the signal produced by the receiver, that is, to ensure that the modulation of a symbol is as much as possible contained in a buffer that is transformed, one can, for example, adopt an energy-based approach: one lets the window slide, discarding the input samples until the energy of the signal in the window considered does not exceed a certain threshold. In this way, one avoids transforming a window containing silence in the first part and only a small amount of signal in the second, a scenario that would certainly give a greater error since the ratio between the energy of the actual signal and that of the noise would be lower.

2.3.2 Gray decoding

Gray coding applied to the symbols output by the modulator has the role of protecting the information from excessive error during the frequency detection phase. At this point it is possible to invert it to obtain the values corresponding to the transmitted symbols, barring the errors previously introduced in the decision phase. A simple algorithm to implement Gray decoding, also used in this implementation, is to consider all the values deriving from the possible shifts to the right of the input value and apply the bitwise XOR between all of them.

2.3.3 Buffering

Before being able to proceed with the decoding of even a single character, it is necessary to receive a quantity of symbols corresponding to an entire transmitted block. A smaller quantity of symbols does not contain enough information to decode any character. The transmitted block, whatever n, is made up of 64 symbols, each of which will carry a variable quantity of information as a function of n. It is therefore necessary to prepare a temporary buffer that accumulates 64 symbols before passing them to the next stage. In the implementation considered this block is not a simple accumulator of values, but is implemented by means of a rotating array and this is particularly important to obtain reliable error correction with low code complexity, as will be better illustrated in the dedicated section below.

2.3.4 De-interleaving

The inverse operation of the interleaving performed in modulation is performed, in order to obtain n 64-bit vectors.

Due to the uncertainty on the decision of the received symbols, these vectors will typically not be identical to those that were input to the interleaving stage of the modulator, especially in noisy conditions.

The de-interleaving scheme, arbitrarily decided and standardized for Olivia and complementary to the interleaving scheme previously illustrated, is the following:

- for each vector W constituting the block and defined w as its index,
  - for each symbol S constituting a block and defined s as its index,
    - let q = (w + s) mod log2(n),
    - let the s-th bit of the vector W be equal to the q-th bit of the symbol S.

At this point we are back to the situation in which each vector represents a single character of the received block.

2.3.5 Descrambler

This stage is identical to the scrambler present in the modulator. In fact, by applying again the bit-by-bit XOR of the same shared key to the vectors, rotated each time by 13 bits for each successive vector in the block, the operation is exactly cancelled. This stage does not introduce error since the output values are exactly determined by the input ones.

2.3.6 Fast Walsh-Hadamard Transform

The Walsh-Hadamard transform is applied to the obtained vector. At this point the vector is an altered version of a Walsh basis function. Ideally, in the absence of error, constructed as the modulated signal the vector should be exactly a basis

function and consequently its transform should generate a spectrum containing a single line of unit amplitude (except for a multiplicative normalization constant that depends on the length of the vector). The vector obtained by applying the transform will consist of a spectrum with a line of predominant amplitude compared to the others. The difference between the amplitude of the predominant line and the average of the amplitudes of the other lines will depend on the amount of error introduced by the channel and the previous stages of the demodulator. The error correction is therefore carried out by the simple procedure of setting the element of the vector containing the largest value identically to 1 (if positive) or -1 (if negative); and setting all the other values identically to zero.

2.3.7 Mapping from vectors to characters

At this point we have log2(n) vectors 64 elements long, of which only one is valued with one of two possible values. The mapping operation performed during modulation is reversed to determine which of the 128 possible ASCII characters the vectors correspond to. The characters thus reconstructed form the received text and demodulation is completed.

Chapter 3

Implementation details

This chapter highlights the technical solutions chosen for the implementation, reporting and commenting on the most significant code fragments.

The choice of the language to be used for the implementation fell on Python. This language has a simple and easy-to-learn syntax, and is increasingly widely used for scientific applications due to the vast choice of third-party libraries available, in addition to compatibility on a very wide range of different platforms.

The strong point of Python is the ecosystem: in a short time, ready-made and widely used libraries have been found that have allowed the development effort to be concentrated on the problems of the topic in question, interfacing with the hardware in a practical way and avoiding having to reimplement already known methods and algorithms.

1. numpy.

This is a library that makes Python suitable for numerical calculation, an application for which it was not originally designed. It includes a large number of scientific andcalculation functions, many with an interface inspired by MATLAB, of which it achieves an equivalent basic functionality although with a very different syntax; but above all it introduces as a data type the multidimensional array that can be used generically in functions, opening up possibilities for vector calculation and representation of algorithms in a much more compact way. The introduction of this library has allowed researchers to approach Python as a plausible ecosystem for the implementations and rapid modeling of problems, also thanks to the introduction of integrated environments such as Spyder, taking the language out of its traditional application of system scripting and the creation of application programs.

2. scipy.

This library includes, among other things, implementations of many algorithms useful for numerical computation, for example functions for the computation of direct and inverse Fourier and Walsh-Hadamard transforms, and many other tools for digital signal processing, linear algebra, interpolation, and machine learning.

3. sounddevice.

This library allows you to use the computer's audio input and output devices through a sample-buffer interface, completely abstracting from the complexities of the underlying drivers, which are complicated to use and platform-dependent. Once an input or output object is created, the library takes care of outputting sample buffers delivered via the API to the selected output device in the background, as well as providing the application program with a constant stream of input buffers obtained from the selected input device. The emission and acquisition of sounds is therefore reduced to a simple preparation or analysis of arrays of values that represent the samples in time of the corresponding analog signal, already quantized and converted into an easily manipulable data type.

3.1 Command line interface

For the practical implementation of the software that implements modulation, a command line interface was chosen for the following reasons:

1. practicality. The text interface is more flexible, effective and quick to implement.

2. compatibility. The absence of graphic elements allows to reduce to a minimum the amount of third-party libraries on which one relies, furthermore it is more universally compatible between various systems.

3. scriptability. It lends itself to being used in combination, or integrated, with other programs or automatically. The programs obtained can be seen as Unix filters and inserted into scripts with little effort.

The program is invoked with an optional parameter that indicates the desired modulation configuration:

n/b@f

The default value in the absence of the parameter will be: 32/1000@1500, a widely used mode that represents a compromise between transmission speed and robustness to noise.

If the standard input is a terminal, a welcome message is displayed (or help, if the parameters entered are not correct) and for transmission, the prompt olivia> is displayed.

If the program is not being invoked from a terminal but from a script or via pipes/file redirection, no message or prompt is printed, to make integration easier.

3.2 Asynchronous transmission using a send queue

We chose to implement an asynchronous logic for the transmitter: the modulator runs continuously in the background and is not blocked waiting for keyboard input. When there is no data to transmit, the modulator continuously transmits empty padding sequences. A separate thread in parallel reads the keyboard input and, dividing it into blocks, inserts it into a transmission queue. The transmission thread fetches the data from this queue and takes care of modulating it. This implements the buffering logic described in section 2.2.1.

The transmitter initSound function prepares the library to use a callback on a dedicated thread to get samples when it needs them:

```
def initSound():
'''
Prepares global OutputStream for sample playback.
'''
global sout

sout = sd.OutputStream(samplerate=SAMPLE_RATE,   blocksize=64*wlen,
channels=1, callback=callback, dtype=np.float32)
sout.start()
```

The callback checks for any blocks to be transmitted on the queue. If the transmission queue is empty, it arranges for a null block to be sent, to prevent the sound generation from being interrupted due to a lack of a constant flow of samples; otherwise, the next block in the queue is selected.

```
def callback(outdata, frames, time, status):
	'''
	Handles sample generation and playback asynchronously.
	'''

	# If first call, generate the preamble.
	if callback.firstCall and ENABLE_PREAMBLE:
	callback.firstCall = False
	outdata[:,0] = generatePreamble()/ATTENUATION
	return None

	# If last call, just empty the samples buffer.
	if callback.lastCall:
	outdata.fill(0)
	return None

	# If transmission queue is empty, transmit null blocks.
	try:
		piece = q.get_nowait()
	except:
		piece = "\0" * spb

	outdata[:,0] = generateBlock(piece)/ATTENUATION

	# If transmission is over, transmit ending tail.
	if piece == None:
		callback.lastCall = True
	return None
```

The advantage of this technique is that the entire life cycle of the modulator program results in a single continuous transmission, on which actual data

may or may not pass depending on availability, minimizing the information lost during the synchronization phase of the receiver. In fact, once the receiver has synchronized on the beginning of the signal, it can continue to follow it for as long as the transmitter remains in operation without having to realign itself except in the case of excessive error. This increases the reliability of the channel and reduces the transmission latency of the character blocks, since the modulator is already in operation when the data becomes available for transmission.

To obtain samples starting from a block of characters, the callback makes use of the generateBlock function, which provides for the partial overlap of the waveforms obtained from the tone generation code (see paragraph 3.3) and possibly the insertion of preambles and tails:

```
def generateBlock(piece):
        '''
        Transmits samples corresponding to a full block.
        '''
        global trail

        wf = np.zeros(64*wlen+wlen)

        # Overlaps trail of last symbol, if any
        wf[0:wlen] += trail

        # If transmission is being stopped, add trailing tail
        if piece == None:
        if not ENABLE_PREAMBLE:
                return wf[0:64*wlen]
        trail = np.zeros(wlen)
        tail = generateTail()
        if len(tail) < 64*wlen:
                wf[wlen:wlen+len(tail)] = tail
        return wf[0:64*wlen]

        syms = prepareSymbols(piece)

        for i in range(0, 64):
```

```
            # Tone number is symbol number after Gray encoding
            # This minimized error made by mistaking one tone for
            # another one right next to it (1 wrong bit only).
            tone = oliviaTone(gray(syms[i]))
            wf[wlen*i:wlen*i+len(tone)] += tone

    trail = wf[64*wlen:64*wlen+wlen]
    return wf[0:64*wlen]
```

The heart of the modulator implementation is the prepareSymbols function. This is where the stages that transform the input data stream into a stream of symbols are implemented.

```
    def prepareSymbols(chars):
        '''
        Transform a block of characters into a block of symbols
        ready for transmission.
        '''
        w = np.zeros((spb, 64))
```

The first part of the function prepares the key that will be used for scrambling, expressing the standardized hexadecimal key as a Numpy array of bits, in order to simplify the subsequent manipulations:

```
    # Key is a 64-bit fixed value and can be found in specification.
    # key = 0xE257E6D0291574EC
    # It is a pseudorandom value and its role is to make the
    # output stream appear random.
    # Here it is decomposed into a bit array for easier use.
    key = np.flip(np.array(
        [1, 1, 1, 0, 0, 0, 1, 0,
        0, 1, 0, 1, 0, 1, 1, 1,
        1, 1, 1, 0, 0, 1, 1, 0,
        1, 1, 0, 1, 0, 0, 0, 0,
        0, 0, 1, 0, 1, 0, 0, 1,
        0, 0, 0, 1, 0, 1, 0, 1,
        0, 1, 1, 1, 0, 1, 0, 0,
        1, 1, 1, 0, 1, 1, 0, 0]))
```

Next, the character to vector mapping illustrated in section 2.2.2 is realized.

```
# Character to 64-value vector mapping to provide redundancy.
# Characters are 7-bit (value from 0 to 127)
# Values from 0 to 63 are mapped by setting nth value to 1
# Values from 64 to 127 are mapped by setting nth value to -1
#Every other value in vector is 0.
for i in range(0, spb):
        q = ord(chars[i])
        if q > 127:
                q = 0

        if q < 64:
                w[i, q] = 1
        else:
                w[i, q-64] = -1
```

The inverse Walsh-Hadamard transform is applied to introduce the redundancy (paragraph 2.2.3):

```
# Inverse Walsh-Hadamard Transform to encode redundancy
w[i,:] = ifwht(w[i,:])
```

At this point the scrambling (paragraph 2.2.4) is carried out vectors obtained, using the previously prepared key, rotated by 13 bits for each subsequent vector.

```
# XOR with key to ensure randomness
# (XOR is made by multiplying with -1 or 1)
w[i,:] = w[i,:] * (-2*np.roll(key, -13*i)+1)
```

The resulting bits are interleaved (section 2.2.5) to obtain the bits representing the symbols to be emitted:

```
syms = np.zeros((64, spb))
# Bit interleaving to spread errors over symbols
for bis in range(0, spb):
        for sym in range(0, 64):
```

```
                    q = 100*spb + bis - sym
                    if w[q % spb,sym] < 0:
                            syms[sym,bis] = 1

        # Convert to integer to find symbol numbers
        symn = np.zeros(64)
        or i in range(0,64):
                symn[i] = bits2int(np.flip(syms[i]))

        return symn
```

The generateBlock function above, before transmission, will apply Gray encoding (section 2.2.6) using the utility function:

```
def gray(n):
        '''
        Utility function to calculate Gray encoding of an integer value
        '''
        n = int(n)
        return n ^ (n >> 1)
```

before passing the symbols to the generation code.

3.3 Tone Generation

To generate the sample windows corresponding to the tone transmission, direct synthesis is used:

```
def oliviaTone(toneNumber):
        '''
        Tone generator. Creates output waveform for specified tone number.
        '''
        toneFreq = (CENTER_FREQUENCY - BANDWIDTH/2) + fsep/2 + fsep
        * toneNumber
        t = np.arange(0, 2/fsep, 1/SAMPLE_RATE)
        ph = np.random.choice([-pi/2, pi/2]);
        ret = np.sin(2*pi*toneFreq*t + ph)
        return toneShaper(ret)
```

The oliviaTone function produces samples of the sine wave thatrepresents the requested tone in the current configuration, introducing the expected random phase shift. The output tone is shaped by thetoneShaper function to reduce the effects of intersymbol modulation, as seen in section 2.2.7:

```
def toneShaper(toneData):
    '''
    Tone shaping to avoid intersymbol modulation.
    Cosine coefficients are fixed and can be found in specification.
    '''
    x = np.linspace(-pi, pi, len(toneData));
    shape = (1. + 1.1913785723*np.cos(x)
        - 0.0793018558*np.cos(2*x)
        - 0.2171442026*np.cos(3*x)
        - 0.0014526076*np.cos(4*x))
    return toneData * shape
```

An Olivia transmission typically includes a preamble and a coda, consisting of the highest and lowest tones in the current configuration, alternately, for a total duration of one second. These tone sequences are inserted to help a human operator fine-tune the receiver, as well as to recognize that Olivia modulation is being used and possibly determine the bandwidth for selecting the appropriate demodulator.
Since they are an aid to the operator, these sequences are not essential for data transmission and in fact do not contain any information. In the implemented modulator script they are therefore generated if it is possible to do so within a single playback buffer of the audio device, i.e. if the total duration in samples of the preamble or coda is less than the sample rate. Otherwise the preamble or tail

is omitted without affecting the transmission functionality, to avoid overcomplicating the code.

The generateTail function builds the preamble and tail tones by preparing a buffer of samples one whole second long, which contains the pure sine waves, alternately at the maximum and minimum frequencies.

```
def generateTail():
    '''
    A tail is made in this way:
    first tone, last tone, first tone, last tone each one lasting 1/4 seconds.
    '''
    pl = int(SAMPLE_RATE/4)
    t = np.arange(0, 1/4, 1/SAMPLE_RATE)
    wf = np.zeros(SAMPLE_RATE)
    wf[0:pl] = toneShaper(np.sin(2*pi*(CENTER_FREQUENCYBANDWIDTH/
    2+fsep/2)*t)/2)
    wf[pl:2*pl] =
    toneShaper(np.sin(2*pi*(CENTER_FREQUENCY+BANDWIDTH/2-fsep/2)*t)
    /2)
    wf[2*pl:3*pl] = wf[0:pl]
    wf[3*pl:4*pl] = wf[pl:2*pl]
    return wf
```

As for the receiver, it was not necessary to implement asynchronous logic, since the flow of incoming audio samples from the driver is constant and there are no requests for data from the keyboard that can be blocking for an indefinite time, so the sounddevice library was used in synchronous mode.

3.4 Error detection

Olivia modulation itself provides two possible dimensions of error control, the first on the recognition of the modulated symbols through the Fourier transform and the second on the determination of the vector element to be valorized after the Walsh-Hadamard transform. For the implementation in analysis, it was chosen to avoid the code and computational complexity due to the synchronization of the buffer for the

Fourier transform and it has been found that this still leaves room for effective error correction. The Fourier transform (section 2.3.1) is performed by the detectSymbol function, directly on the sample windows acquired by sounddevice:

```
def detectSymbol():
    '''
    Applies Fourier transform to audio buffer to detect
    symbol corresponding to sampled tone.
    Returns
    -------
    int
        Most likely symbol number.
    '''
    spectrum = np.abs(fft(buf))
    ix = CENTER_FREQUENCY - BANDWIDTH/2 + fsep/2
    measures = np.zeros(SYMBOLS)
    for i in range(0, SYMBOLS):
        ix += fsep
        measures[i] = spectrum[int(ix * wlen / SAMPLE_RATE)]
    mix = np.argmax(measures)

    return degray(mix)
```

Before returning the symbol, the function also removes the Gray encoding (section 2.3.2) using the utility function:

```
def degray(n):
    mask = n
    while mask != 0:
    mask >>= 1
    n ^= mask
    return n
```

Omitting the synchronization of the sample window on which the Fourier transform is performed, one can find oneself in a situation in which only a small part of the transformed samples contains a real audio emission related to the transmission

of a symbol, while the first part may consist of silence. This possibility is substantially random and depends on the instant in time in which the transmitter started transmitting and the instant in which the receiver started receiving.

Since the modulated output tones are superimposed, it will always be the case in each window considered that the amplitude of a certain tone will be predominant, thus eliminating ambiguity on which one should be considered (obviously considering the fact that the noise could have altered the input signal). However, it is possible that an additional tone is artificially introduced at the beginning of the transmission, due to the capture of a small signal tail at the end of the first reception window. From the next window onwards the problem does not exist since there will be one or two modulated signals present right from the start, with one preponderant in each case.

It can therefore be verified with a certain probability that the block of symbols detected by the Fourier transform and delivered to the next stage of the modulator contains a first spurious symbol. In addition to this, it must be considered that if a minimum energy threshold is not set (squelch) to consider the result of the transform valid, in the absence of modulation it will still extract the most probable symbol from the background noise.

Both these problems have been solved by implementing a shift buffer before the deinterleaving stage (section 2.3.3): once a complete block of symbols is received by the transform, it is demodulated and the noise energy is measured on the Walsh spectrum downstream of the Walsh-Hadamard transform. If this error exceeds a certain threshold, the complete block is not discarded, but simply the first symbol. Upon receipt of the next symbol, an attempt will be made again to decode the block consisting of the remaining symbols plus the new received symbol.

```
while True:
    # Fetch new samples
    updateBuffer()
    sym = detectSymbol()
    syms.append(sym)
```

```
        if len(syms) == 64:
            # Enough symbols to decode a block
            if decodeAndPrintBlock(syms):
                # Block decoded successfully, waiting for a new one
                syms = []
            else:
                # Probably not a complete block, try rolling
                syms = syms[1:]
```

When the considered block will be composed only of symbols actually transmitted by the modulator – and therefore not by symbols extracted from silence – the error rate will be lower than the threshold and the block will be valid. At this point the demodulator is synchronized and the next entire blocks of symbols can be acquired without further problems.

This is effectively the implementation of a sliding windowin the Walsh domain rather than in the time domain. Using this sliding logic, the problems highlighted above are addressed by the same code used for noise error correction without the need to implement expensive additional methods of synchronization of the received audio signal.

The symbols belonging to the block are de-interleaved (section 2.3.4):

```
    for i in range(0, spb):
        for j in range(0, 64):
            bit = (syms[j] >> ((i+j) % spb)) & 1
            if bit == 1:
                w[i,j] = -1
            else:
                w[i,j] = 1
```

The standardized key is applied in bit-by-bit XOR to the vectors obtained (section 2.3.5), so as to cancel the scrambling that was performed during transmission:

```
w[i,:] = w[i,:] * (-2*np.roll(key, -13*i)+1)
```

The Walsh-Hadamard transform is applied (section 2.3.6) to perform error correction. The maximum line in the Walsh spectrum is measured to determine which is most likely the originating vector. If the uncertainty is greater than a preset threshold, the reconstructed vector is considered doubtful.

```
w[i,:] = fwht(w[i,:])

c = np.argmax(np.abs(w[i,:]))

if abs(w[i,c]) < BLOCK_THRESHOLD:
    doubt += 1
```

Finally, the characters are obtained by inverting the character-vector mapping (paragraph 2.3.7):

```
if w[i,c] < 0:
    c = c + 64
if c != 0:
    output += chr(c)
```

However, the received block of characters is output only if there is no doubt about its decoding. The presence of at least one doubt in a block makes it invalid and causes a symbol to be scrolled in the buffer illustrated above, in addition to causing the block not to be printed on the screen:

```
if doubt == 0:
    print(output, end="", flush=True)
    return True
else:
    return False
```

3.5 Walsh-Hadamard Transform Sorting

The very popular SymPy library already contains high-quality code for the Walsh-Hadamard transform and its inverse, using the fwht and ifwht functions. At first, we tried, successfully, to use these ready-made functions; the system was working but was not compatible with existing Olivia implementations. After a thorough analysis, it emerged that the implementation provided by SymPy uses a matrix sorting that is different from the one used by the reference implementation of Olivia. This does not detract from the quality of the SymPy implementation; in fact, both methods are correct and simply differ in the presentation of the input and output information.

Rather than reordering the vectors to exploit the SymPy implementation for Olivia, we preferred to re-implement the transform and inverse code from scratch, taking as an example the source code of the reference implementation in C++ and translating it exactly into Python. We found that the conversion from C++ to Python was very simple because it was not necessary on the Python side to take care of the low-level management of the memory dedicated to the vectors.

```
def fwht(data):
        '''
        Fast Walsh-Hadamard transform.
        '''
        step = 1
        while step < len(data):
                for ptr in range(0, len(data), 2*step):
                        for ptr2 in range(ptr, step+ptr):
                                bit1 = data[ptr2]
                                bit2 = data[ptr2+step]

                                newbit1 = bit2
                                newbit1 = newbit1 + bit1

                                newbit2 = bit2
                                newbit2 = newbit2 - bit1
```

```
                        data[ptr2] = newbit1
                        data[ptr2+step] = newbit2
            step *= 2
    return data

def ifwht(data):
    '''
    Inverse Fast Walsh-Hadamard transform.
    There is a similar ready-made transform in sympy, but its output ordering
    (Hadamard order) is different from the Olivia specified one, and it's more
    efficient to reimplement it directly rather than converting the output.
    This is a 1:1 translation from the Olivia C++ reference implementation.
    '''
    step = int(len(data)/2)
    while step >= 1:
        for ptr in range(0, 64, 2*step):
            for ptr2 in range(ptr, step+ptr):
                bit1 = data[ptr2]
                bit2 = data[ptr2+step]

                newbit1 = bit1
                newbit1 = newbit1 - bit2
                newbit2 = bit1
                newbit2 = newbit2 + bit2

                data[ptr2] = newbit1
                data[ptr2+step] = newbit2
        step = int(step/2)
    return data
```

The portions of code related to initialization, interpretation of parameters and user interface have not been reported. The complete and directly executable source code of the modulator and the demodulator has been published on GitHub and can be found at:

https://github.com/sntfrc/olivia-python .

Chapter 4
Experiments and measurements

This chapter reports the results of practical transmission and reception tests carried out using the implemented programs, collecting quantitative data on performance in various scenarios.

4.1 Interoperability with reference implementations

Once the code implementation was completed, it was necessary to test it. For this purpose, transmission and reception tests were conducted using, at the other end of the communication, the Fldigi software, developed by Dave Freese starting from 2007 and widely used by radio amateurs to manage a large number of sound card modes, including Olivia. This software package can be considered an industry standard.

It was found that the olivia-tx.py script produces an audio signal perfectly decodable by Fldigi; vice versa, Fldigi produces an audio signal perfectly decodable by the olivia-rx.py script. This means that the implemented scripts can be used to communicate with other Olivia implementations without any special advice.

There are some differences in performance during the reception phase: the olivia-rx.py script has a lower reception latency than Fldigi, but the latter can tolerate higher amounts of noisebefore presenting errors in the received text. The reason for thisdifference lies in the error correction during the reception phase: the correction algorithm implemented here is simpler than the one used in Fldigi, since it simply relies on the redundancy that the signal presents in a single block. Fldigi is able to consider the values of multiple consecutive blocks to determine with more confidence whether the received characters are correct or not. As shown in more detail in the following sections, even the simplest implementation of error correction is able to provide excellent performance up to very high noise levels.

4.2 Noiseless performance

First, a noiseless performance test was performed, which will serve as a benchmark for subsequent tests. The test methodology, which will be the same for subsequent tests, is described below.

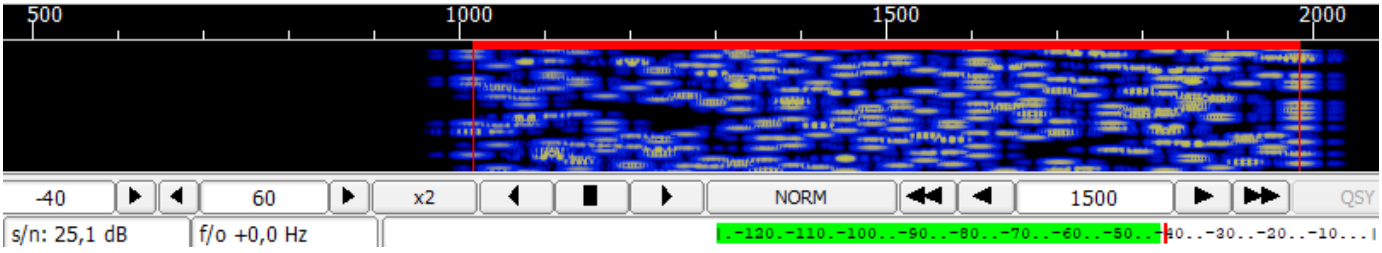
1) Transmission of the following short text:

Lorem ipsum dolor sit amet, consectetur adipiscing
elit. Nulla vehicula purus purus, iaculis pharetra
sapien sagittis in. Nulla quis dui nisi.

using the olivia-tx.py script, after it has stabilized with the transmission of several null blocks.

2) Reception of the modulation with the olivia-rx.py script, using the Windows “Stereo Mix” driver as input source, which accurately reports all the sounds generated in output from the PC without therefore introducing any environmental noise that would be present in the case of emission and reproduction by external devices.

3) Simultaneous reception of the same signal via “Stereo Mix” by Fldigi, in order to display the spectrum, obtain a measure ofthe signal/noise ratio and make a comparison on the quality of the demodulation guaranteed by the two applications. The message in question requires a time of about 60 seconds for transmission in Olivia with 32/1000 configuration. The test in the absence of noise presents a spectrum of this type:



The signal/noise ratio measured with the emission volume used is about 25 dB, while the absolute signal level is -40 dBFS. The emission volume will be kept unchanged in the subsequent tests to make the comparison meaningful.

As can be seen in the figure, the emitted tones have a soft shape on the spectrum waterfall and have sidebands that fade smoothly into the next and previous tone. This is a consequence of the tone shaping and overlapping done to decrease the incidence of intersymbol modulation.

The total absence of noise in this test can be verified by observing how the portion of the spectrum that does not include the modulated signal, below 1000 Hz and above 2000, appears totally black (absence of signal).

The text was received perfectly, with exactly zero transmission errors, both by olivia-rx.py and by Fldigi.

4.3 Performance in the presence of white noise

A test is performed in the presence of white noise. A white noise generator is started on the same machine that runs the scripts; in this way the audio output of the modulator will be combined with the noise by means of “Stereo Mix” before being presented to the input of the demodulator.

The white noise generator is adjusted to generate the same audio level measured in the pure modulation of the previous test (-40 dBFS). The spectrum relating to the noise signal alone is the following:

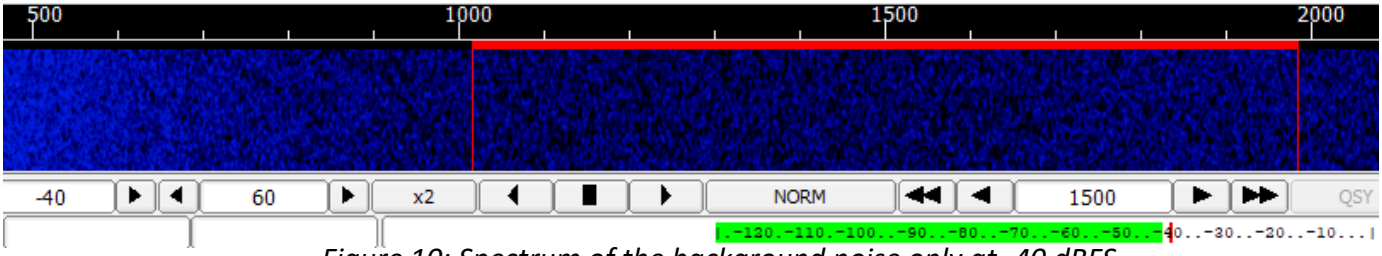


Figure 10: Spectrum of the background noise only at -40 dBFS.

As expected, the spectrum is uniform. Starting the demodulator at this point, in the absence of a modulated signal, it extracts from the noise the most coherent information that can be detected. The symbols generated for which there is sufficient “certainty” are printed on the screen. The result, in the test in question, was the following:

S$←?HP↑↕rZ:Gc]0}diMa,9♣¶8I4_♂S¶Q6{$‼&_DZ♥ 94uA♦:↓w♀↑(+hVq☺&

These meaningless character sequences are constructed from random oscillations of pure noise that are compatible with the emission of a modulator. It is possible to eliminate them by requiring that the input signal to the demodulator has an amplitude greater than a certain pre-established threshold (squelch) before being considered significant. However, this does not in any way affect the results obtained in the presence of a modulated signal.

The modulated signal was applied above the noise signal, obtaining the following spectrum:

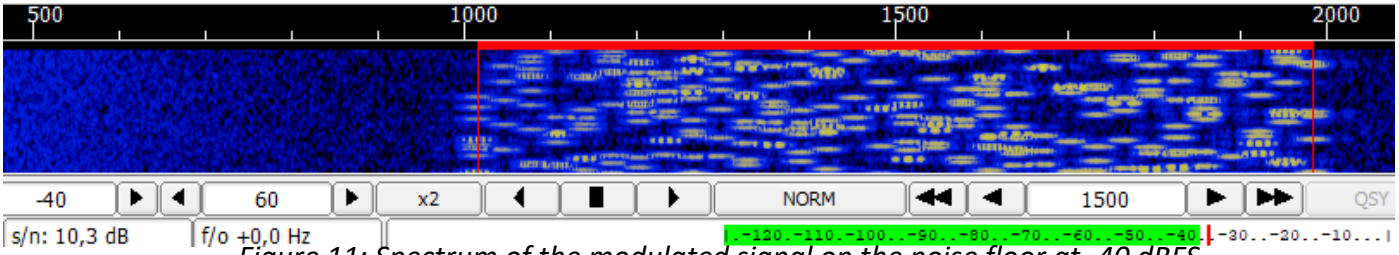


Figure 11: Spectrum of the modulated signal on the noise floor at -40 dBFS.

It can be seen how the sidebands of the various tones merge with the underlying noise. The signal-to-noise ratio has dropped to about 10 dB, a change of -15 dB compared to the case without noise. Despite this, the images of the tones on the spectrum are still perfectly visible and discernible. In fact, the text received from both olivia-rx.py and Fldigi is still completely error-free.

The next test was performed with a much higher noise floor, with amplitude equal to -30 dBFS. The noise spectrum considered is:

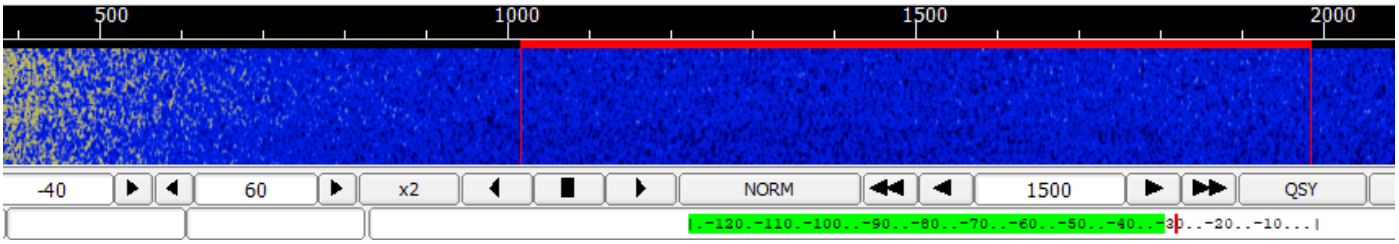


Figure 12: Spectrum of background noise only at -30 dBFS.

The noise amplitude peak that can be seen at frequencies close to zero is due to a limitation of the audio hardware. The phenomenon (which can also be seen in a milder form in the previous test) does not create anyway problems for the test since it is totally contained in frequencies below 1000 Hz and therefore never interacts

with the modulated signal.

The noise generated is therefore an order of magnitude greater in amplitude than that of the previous test. We proceed to emit the modulated signal above this new noise.

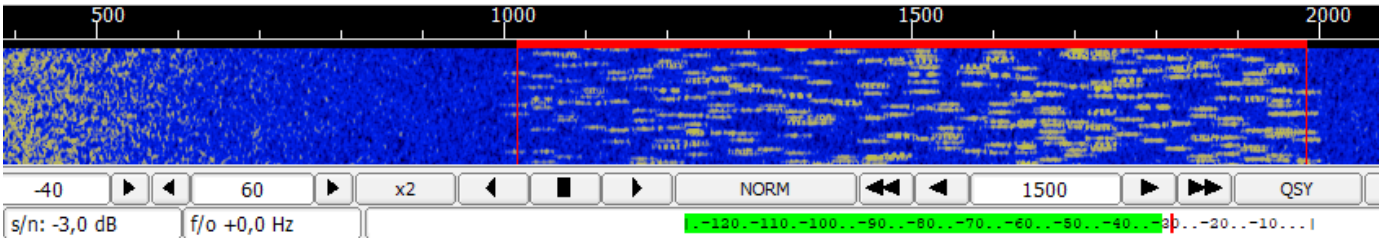


Figure 13: Spectrum of the modulated signal on the background noise at -30 dBFS.

You can see how the confusion of the sidebands of the tones corresponding to the symbols increases further, which at this point have practically disappeared in the noise. The signal to noise ratio has become negative: -3 dB. This means that at this point the power of the noise is double that of the useful signal! Although the signal has a total power even lower than that of the noise, the demodulation is once again perfect, and the text received by both olivia-rx.py and Fldigi is completely free of errors. This denotes a great robustness to noise and fundamentally that the main goal of the Olivia modulation can be considered well achieved.

At this point the maximum power of the noise generator is being used. To simulate even higher noise conditions, the modulated signal is progressively attenuated while keeping the amplitude of the underlying noise constant. This test is performed with the modulated signal attenuated by 3 dB compared to the previous tests, and the underlying noise level maintained at the previous -30 dBFS:

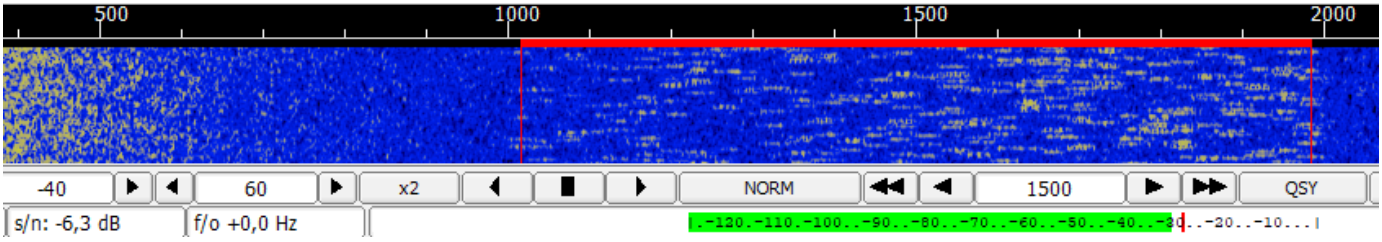


Figure 14: Spectrum of the signal attenuated by 3 dB, on the noise floor of -30 dBFS.

Visually, you can see how the shape of the tones on the spectrum waterfall starts to be seriously affected by the noise. The signal-to-noise ratio, as expected, is about -6 dB, or 3 dB lower than before due to the attenuation introduced. At this point, the modulated signal power is a quarter of that of the noise. Despite this, the demodulation is once again perfect, and the text received from both olivia-rx.py and Fldigi is completely error-free. We proceed to increase the attenuation of the modulated signal to 6 dB, obtaining the following spectrum:

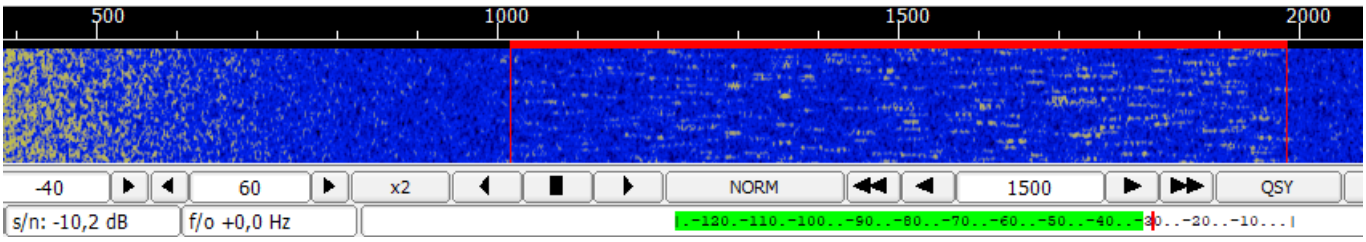


Figure 15: Spectrum of the signal attenuated by 6 dB, on the background noise of -30 dBFS.

The measured signal-to-noise ratio is about -10 dB, close to the theoretical -9 dB that was expected. At this point the power of the modulated signal is an order of magnitude lower than that of the noise. Despite this, the demodulation is once again perfect, and the text received from both olivia-rx.py and Fldigi is completely error-free. We proceed to increase the attenuation of the modulated signal, bringing it to 10 dB. The spectrum obtained is the following:

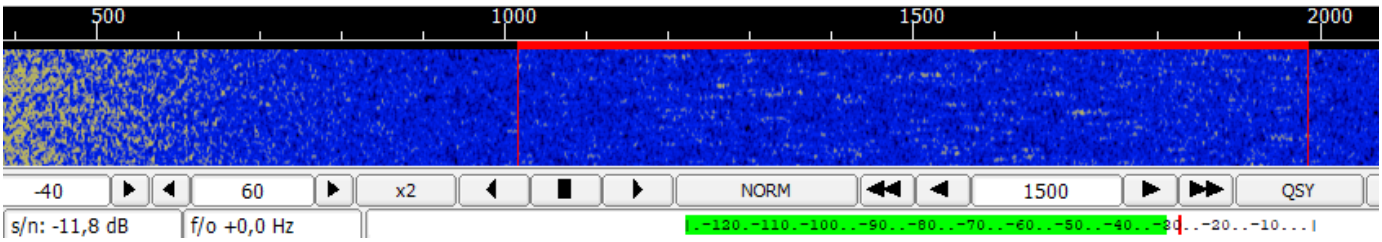


Figure 16: Spectrum of the signal attenuated by 10 dB, on the background noise of -30 dBFS.

Visually it can be seen how the modulated signal has almost disappeared in the noise and it has become very difficult to distinguish the images in the spectrum of the individual tones. The signal to noise ratio is about -12 dB, corresponding to what was expected.

At this point we see that errors are starting to be introduced in the text received by the demodulator. In the test in question, the text received by the olivia-rx.py demodulator is:

Lor!m ipsuor sit amensectetur adipiscing elit. Nulla
vea purus purus,sis pharetra sapien sla qu

The following errors were introduced:

• “Lorem” became “Lor!m”. The block of 5 characters received therefore contains an error on a single character. The high doubt on one character was considered acceptable (below threshold) by the receiver code, so the block was printed.

• “ipsum dolor” became “ipsuor”: in this case an entire block of 5 characters (“m dol”) was discarded by the receiving code since there was not a sufficient degree of certainty on its content, that is, there was considerable doubt on more than one character of the block.

• “vehicula” became “vea”: the 5-character block “hicul” was discarded for the same reason as the previous point.

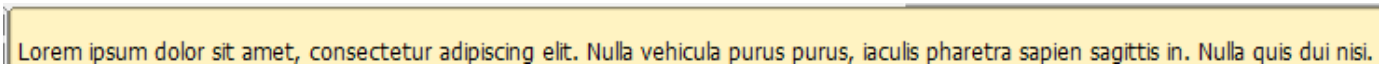
• “, iaculis” became “,sis”: the block “ iacu” was discarded for doubt, and “lis” became “sis” due to the introduction of a single error.

• The blocks “agitt”, “is in”, “. Nul”, “is du”, “i nis”, “i.” were discarded for doubt.

To see the appearance of the first errors, it was therefore necessary to reduce the power of the modulated signal until it was more than ten times lower than that of the noise. Up to a power equal to one tenth of that of the noise, the information was copied perfectly. It is still possible to partially discern the meaning of the text. Since background noise is random by nature, repeating the test with the same levels will probably result in errors of a different type and located in different points of the text. So eventually, in case of two-way communication, it is possible to ask to repeat the

transmission to copy the missing parts and it is therefore possible to have still a functioning communication channel.

It should be noted, however, that Fldigi has also perfectly copied this signal:



This is explained by the fact that Fldigi uses more complex integration algorithms, optimized to maximize the number of characters copied perfectly, combining the values obtained by decoding multiple blocks in a row, in exchange for a greater latency in reception before the appearance of the demodulated text. Olivia modulation itself therefore continues to be effective even at these (and probably even higher) noise levels; by increasing the sophistication of the receiving code it is still possible to extract information.

4.4 Performance in the presence of structured noise Tests with uniform white noise are useful for measuring the characteristics of the modulation. However, in real applications it may happen that the signal is transmitted over more structured background noise. Let's think for example of the case in which the transmission actually occurs via pressure waves (for example, output via the speaker of a smartphone and input via the microphone of a laptop). Typically there will be non-white environmental noises with powers significantly higher than those of the background noise.

4.4.1 Music
A piece of music is played on the same machine that runs the scripts, so that the audio output of the modulator will be combined with the music using “Stereo Mix” before being presented at the input of the demodulator.

The volume of the music is calibrated so that the energy of the overall music signal is -30 dBFS and any attenuation applied to the modulating signal is removed, thus resulting in the situation of the second test with white noise. The difference is that in this case the energy of the noise signal is not uniformly distributed in the

spectrum, but it condenses in some points creating images on the spectrum that closely resemble those of the tones produced by the modulator.

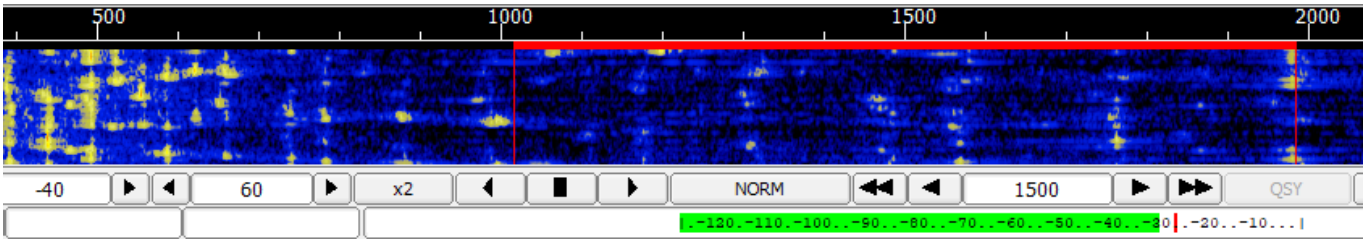


Figure 17: Spectrum of musical noise alone at -30 dBFS.

The spectrum resulting from the modulation of the signal above the musical noise is as follows:

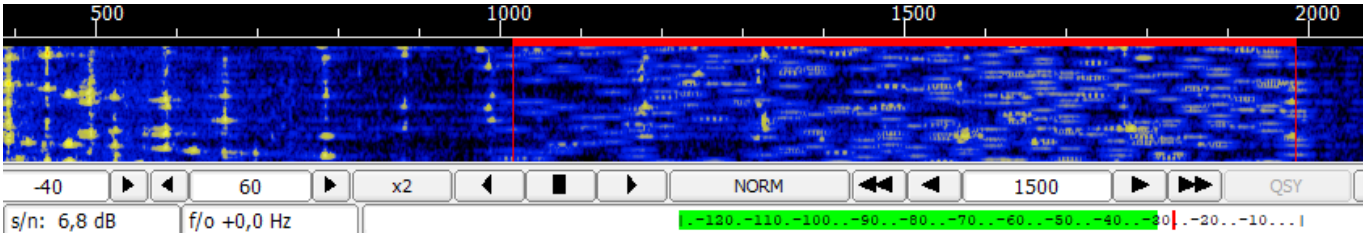


Figure 18: Spectrum of the modulated signal above the musical noise at -30 dBFS.

It can be seen how the tones generated by the modulator actually get mixed up with the high notes of the song. One would expect a large amount of errors in reception, but the phenomenon is contained by the error correction property of the demodulator. The final error in the text demodulated by olivia-rx.py is low:

Lorem ipsum dolor sit amet, consectetur adipiscing
elithicula purus purus, iaculis pharetra sapien s☺
gittis in. Nulla quis dui nisi.

This is a single incorrect character and the omission of two blocks. Once again, Fldigi has instead copied the transmitted text perfectly, thanks to its integration properties. We see then that the behavior of the demodulators is similar (although better) to that of the test carried out with white noise in the presence of an attenuation of the modulated signal equal to 10 dB.

We deduce that for the same energy, a noise more concentrated in frequency is more detrimental to the modulation than a noise of uniform density. This is

compatible with the fact that Olivia was designed for amateur radio use, where typically the channel noise is uniform. Direct use on pressure waves will therefore not be the most brilliant application for this type of modulation.

4.4.2 Speech

The last test was carried out using as noise signal a TED Talk given by a female voice (the choice of a high-pitched voice was necessary to fit most of the noise into the passband of the modulated signal).

The video with the speech was played on the same machine as the code execution, so the audio output of the modulator was combined with the speech using “Stereo Mix” before being presented to the demodulator input.

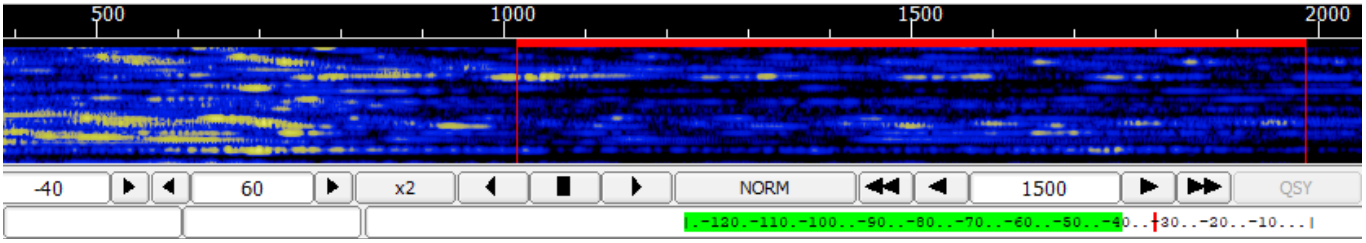


Figure 19: Spectrum of the speech noise signal only.

It can be seen that in this case the noise is of an intermediate type between white noise and musical noise: the energy remains concentrated on specific frequency intervals, but wider than those of musical notes. Observing the waterfall, we see how the images on the spectrum produced by speech are typically wider than one symbol.

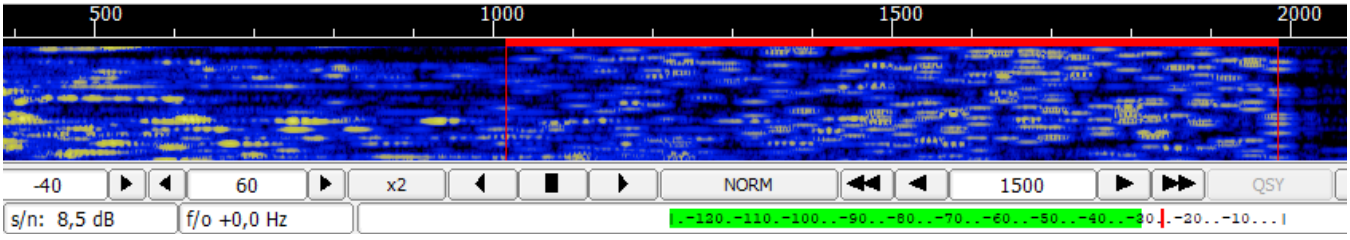


Figure 20: Spectrum of the modulated signal above the vocal noise.

The text received by the demodulator is the following:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nothing vehicula purus purus, iaculis pharetra sapien Qis in. Nothing heres dui nisi.

We therefore note only one omitted block and one incorrect character. The noise at the same intensity was actually more harmful than the white noise, and less than the musical one. This confirms the interpretation that the concentration of the noise energy on narrow frequency intervals is more penalizing, all other conditions being equal.

Finally, we note that Fldigi, as in all the other tests, perfectly copied the transmitted signal without any errors. We deduce that the implementation of the modulator made in olivia-rx.py is to be considered fast and computationally not very intense, but at the expense of performance.

The measurements made on Fldigi convince us that it is possible to take the Olivia modulation even further to the limit. However, it should be remembered that in these tests, overall, we are considering noise levels that are much higher than those that can occur in real applications. The difference between the two implementations will probably not be relevant in practical use.

4.5 Radio frequency transmission experiment

Finally, a practical test of radio communication was carried out,using off-the-shelf PMR446 devices (available in a common electronics store). These devices allow transmission with a power of 500 mW, without a license, in analog FM modulation in a frequency range around 446 MHz (UHF). The PMR446 channel number 7 was used, corresponding to a frequency of 446.08125 MHz, using a maximum frequency deviation of 2.5 kHz.

The fixed receiving station (identifier 1HFAF) was equipped with a PMR446 device model Radioddity PR-T1, connected via a cable to the USB sound card of the computer that runs the reception script. At the same time, Fldigi was kept running on the computer in order to graph the spectrum and measure the signal/noise ratio.

Figure 21:
Fixed station 1HFAF.

Figure 22:
Mobile station 1HFAG

The mobile transmitting station (identifier 1HFAG), equipped with a Midland G9 Pro PMR446 device, was placed at a distance of 200 meters as the crow flies from the fixed station, interposing several reinforced concrete walls in order to reduce the quality of the signal and make the test channel noisier. A high-fidelity audio recorder was used with which the modulated signal from the transmission script corresponding to this message was previously acquired:

1HFAF DE 1HFAG Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Nulla vehicula purus purus, iaculis
pharetra sapien sagittis in. Nulla quis dui nisi. K

The recorder speaker was acoustically coupled to the microphone of the radio device without the use of cables, potentially introducing additional environmental noise.

It should be noted that the use of the FM modulation mode instead of the more typical SSB meets the requirements of simplicity, cost containment and wider diffusion of unlicensed radio, but typically worsens the signal/noise ratio of digital modulations at the same power used for emission. The results obtained are to be considered pessimistic compared to the real performances obtainable in a situation without obstacles between the two stations and possibly using more suitable analog modulations as a substrate.

The resulting spectrum is the following:

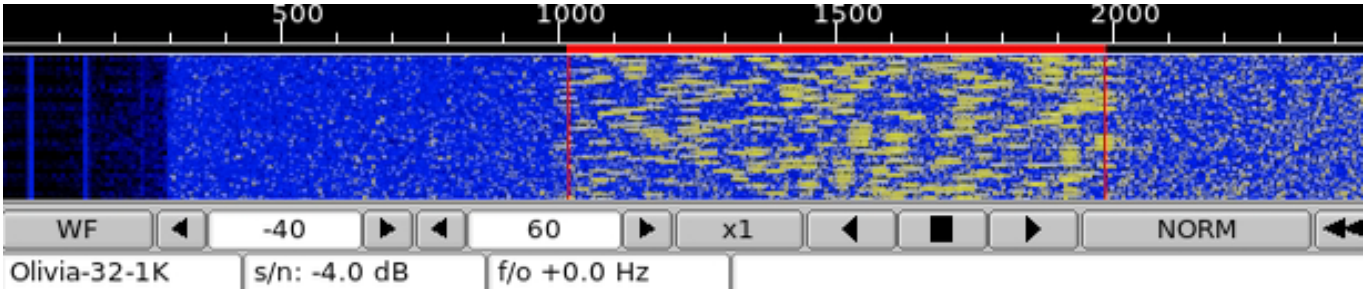


Figure 23: Spectrum of the modulated signal received via FM channel in UHF radio frequency.

The text was copied perfectly despite the negative signal/noise ratio, of -4 dB, which indicates a total power of the background noise (of the white type) of more than double that of the signal. The situation is similar to that of the previous test with background white noise at -30 dBFS.

Collecting the results in a table (Table 1) it is highlighted how the implemented implementation provides the best performance on white type noise; however it remains usable with excellent results even on structured type noise provided that the power of the disturbance does not exceed approximately double that of the emitted signal.

Noise type	Emission power	Signal to noise ratio	# blocks transmitted	# blocks with errors	% blocks with errors
Tipologia di rumore	**Potenza di emissione**	**Rapporto segnale/ rumore**	**# blocchi trasmessi**	**# blocchi con errori**	**% blocchi con errori**
nessuno	0	25,1 dB	29	0	0%
-40 dBFS (bianco)	0	10,3 dB	29	0	0%
-30 dBFS (bianco)	0	-3 dB	29	0	0%
-30 dBFS (bianco)	-3 dB	-6,3 dB	29	0	0%
-30 dBFS (bianco)	-6 dB	-10,2 dB	29	0	0%
-30 dBFS (bianco)	-10 dB	-11,8 dB	29	11	37,9%
-30 dBFS (musicale)	0	6,8 dB	29	3	10,3%
-30 dBFS (parlato)	0	8,5 dB	29	2	6,9%
rumore di fondo UHF	500 mW all'antenna	-4 dB	32	0	0%

Table 1: Summary table of the experimental tests carried out.

Chapter 5

Conclusions

With regard to Olivia modulation, it has been possible to quantitatively appreciate its remarkable qualities of resistance to various types of noise, confirming it as one of the most robust communication methods for amateur radio purposes, for which it was created. The rather inefficient use of the signal bandwidth, as well as the very low modulation speed limit its practical usefulness in everyday communications. However, the relative complexity and heterogeneity of the stages that compose it makes it a good case study to establish the flexibility and adequacy of a numerical calculation implementation system.

The Python ecosystem, consisting of the language itself and the very complete collection of freely available libraries, has evolved over time from a simple scripting system to a real platform that can be used for the most diverse purposes. Suitable for both rapid and intuitive modeling and for optimized and definitive implementation, it lends itself very well tonumerical calculation, and is superior to other comparable systems for the availability of the source code, created collaboratively by contributors from all over the world and therefore verifiable in its functionality by the scientific community in general, and for the absence of license costs that are prohibitive for many. These strengths are rapidly giving it a dignity generally recognized in the academic field and make it, in addition to a computer language, an effective communication language for sharing algorithms and their formal specification.

Olivia modulation is among the most complex among those now classic used in the amateur radio world because it is composed of many stages and makes use of various non-trivial signal processing concepts. It is noteworthy the agility and clarity with which it was possible to create an implementation in Python, obtaining a clear and effectively almost self-documenting source code, which could easily be used as a formal specification.

A comparison with the reference implementation in C++ makes it clear how the level of abstraction guaranteed by Python allows to understand at a quick glance what the code intends to do, reducing to a minimum the parts of boilerplate code supporting and managing the most exquisitely IT issues related to the execution of the program.

Thanks to the wide range of platforms that support Python, the prepared code can be easily used and integrated into other software of the most diverse type, such as smartphone apps or embedded systems such as M2M modems or telemetry without great reconversion efforts.

The benefits seen in this case study are also applicable to implementations of modulations or calculation algorithms of greater commercial and practical utility and confirm how the transition from more traditional development systems to Python can be a winning choice for both the academic and industrial worlds.

Bibliography and sitography

[LIT-1] F. Xiong - *Digital Modulation Techniques (2nd edition),* Artech House Telecommunications Library, 2006

[LIT-2] E. O. Brigham - *The Fast Fourier Transform and its applications*, Prentice-Hall, 1988

[LIT-3] S. Agaian, H. Sarukhanyan, K. Egiazarian, J. Astola - *Hadamard Transforms*, SPIE Press, 2011

[ARRL-1]: *The Draft Specification For The Olivia HF Transmission System.* http://www.arrl.org/olivia .

[GITH-1]: Repository contenente il codice dell'implementazione di riferimento, archiviato dal sito originale. https://github.com/mrbostjo/Olivia_MFSK

[RFC-4648]: The Base16, Base32, and Base64 Data Encodings. https://tools.ietf.org/html/rfc4648

[WIKI-1]: Immagine da https://upload.wikimedia.org/wikipedia/commons/thumb/3/39/Fsk.svg/800px-Fsk.svg.png . CC BY-SA 3.0.

[WIKI-2]: Immagine da https://en.wikipedia.org/wiki/Zero_crossing#/media/File:Zero_crossing.svg . Pubblico dominio.

[WIKI-3]: Immagine da https://en.wikipedia.org/wiki/Discrete_Fourier_transform#/m edia/File:Fourier tr ansform,_Fourier series ,_DTFT,_DFT.svg . CC0.

[WIKI-4]: Immagine da https://en.wikipedia.org/wiki/File:DIT-FFT-butterfly.png . CC BY 3.0.

[WIKI-5]: Immagine da https://en.wikipedia.org/wiki/Fast_Walsh %E2%80%93Hadamard_transform#/media/ File:Fast_walsh hadamard transform_8.svg . CC BY-SA 3.0.

[WIKI-6]: Immagine da

https://upload.wikimedia.org/wikipedia/commons/a/af/Gray_code_reflect.png .
Pubblico dominio.