
UC
UNIVERSIDAD
DE CANTABRIA

Agradecimientos

Me gustaría agradecer este proyecto principalmente a mi director de TFG que me ha brindado la oportunidad de hacer un TFG con un gran componente de investigación. Ha tenido, junto con Andrew Tirkel, la paciencia necesaria para explicarme los entresijos de su investigación para poder desarrollar satisfactoriamente este proyecto.

Sin embargo, no podemos olvidar a todas aquellas personas que me han facilitado en mayor o en menor medida el desarrollo de este TFG. Muchas gracias a Raúl Nozal por haberme dado una idea general del modelo de paralelismo que debía usar para desplegar mi software en un supercomputador. Sinceros agradecimientos a Jose Ángel Herrero por darme acceso al nodo de supercomputación Calderón en pleno Agosto estando de vacaciones. Le agradezo a David Herreros haberme proporcionado apuntes sobre MPI y el haberme respondido a algunas dudas sobre su funcionamiento. Por otro lado, me gustaría agradecer a mi amigo William Britton Ott por ayudarme a revisar el inglés de algunas partes del TFG. Por último, los consejos y experiencias de Judith González sobre como llevar a cabo el TFG me han facilitado mucho la labor.

Entrando ahora en el terreno personal, me gustaría agradecer de nuevo a David Herreros y a David Gragera por acompañarme desde el instituto hasta acabar mi carrera universitaria y por el apoyo que me han brindado. A mi familia por proporcionarme la posibilidad de poder desarrollar mis ambiciones académicas.

También me gustaría agradecer a aquellas personas que me han apoyado en mi necesidad de dejar a un lado temporalmente otras responsabilidades para centrarme en este proyecto, en especial a mis compañeros de la sectorial.

Por último, y de una manera muy general, agradecimientos a los gigantes sobre cuyos hombros se han cimentado todos los conocimientos aquí expuestos.

Secuencias binarias y sus aplicaciones

Resumen

palabras clave: secuencias binarias, baja autocorrelación, búsqueda exhaustiva, ramificación y poda

Generar familias de secuencias con correlación acotada entre otras propiedades es de interés en diversas áreas como criptografía, comunicaciones inalámbricas y marcas de agua digitales. Aplicaciones comerciales como el GPS, o usos militares como el Radar se han desarrollado y mejorado a partir de la búsqueda de estas secuencias mediante el estudio de su función tanto de autocorrelación como la correlación entre los miembros de una misma familia.

En este proyecto, nos centramos en una técnica para la construcción algebraica de secuencias donde, a traves de una secuencia de desplazamientos y una secuencia con buenas propiedades, se genera una nueva secuencia. El objetivo es la búsqueda exhaustiva de secuencias de mayor longitud que las ya existentes en la literatura. Para este fin, se ha desarrollado un software de apoyo al diseñador con capacidad de ser desplegado en un nodo de supercomputación para asistir a la búsqueda de dichas secuencias y la comprobación de sus propiedades. La finalidad de estas secuencias, entre otros posibles usos, son radares y sistemas de localización con mayor resolución espacial y temporal.

Binary sequences and their applications

Abstract

keywords: binary sequences, low autocorrelation, exhaustive search, branch-and-bound

The generation of families of sequences with a bounded correlation among other properties is of interest in several fields such as cyptography, wireless comunications and digital watermarks. Commercial application such as GPS or military uses such as radar have been developed and improved thanks to the search of these sequences by the research on their autocorrelation function and the correlation between members of the same family.

In this project, we focus on a technique for the algebraic contruction of sequences where, through a sequence of shitfs and a sequence with good properties, a new sequence is generated. The objective is the exhaustive search of longer sequences that the ones already shown in the literature. To do so, a support software for the designer has been developed with the ability to be deployed in a supercomputer to assist in the search of said sequences and in checking their properties. The applications of this sequences are, among others, radars and location systems with higher spatial and temporal resolution.

Contents

1 Introduction

Advances in wireless comunication and signal proccesing have drastically changed the capabilities of transmitting information. Several wireless networks can operate in the same channel thanks to technologies like CDMA. Similarly, GPS[22] satellites using direct-sequence spread spectrum(DSSS) can broadcast their information to provide it's service. Long distances can be measured with the help of radar technologies. This technology depends on analyzing the eco of a sent signal to calculate the distance to a target.

For several developments on signal proccesing, sequences with good properties of crosscorrelation and autocorrelation function are needed to work. To keep things simple and just for ilustration purposes, we are going to focus in just two technologies through this document: spread-spectrum and radar.

The direct-sequence spread spectrum technology[28][15] is designed to fraction a given frecuency band between multiple transmissions. To do so, DSSS designs a set of spreading sequences designed to appear noise to other receptors. In other words, a set of pseudonoise sequences with low peak crosscorrelations between them. A spreading sequence from this set is pre-shared between emisor and receptor. Then, the receiver correlates the received sequence with a synchronized version of the pre-shared sequence. If it founds a peak in this correlation, the received sequence is processed. Otherwise, it's detected as noise and discarded. This method limits greatly the bandwith of each individual transmission but avoids time division and improves security to jamming. This system is suitable to be used in applications with low volume of information transmision but with several emitter such as GPS. In GPS, satellites broadcast their current position and timestamp and from that the distance to the sattelite can be computed. Having the signal of some more, the position of the receiver can be triangulated.

A radar (radio detection and ranging) uses a radio-frequency electromagnetic signal reflected from a target to determine properties such as position, speed, etc. There are several versions of it but the ones interesting for this project are PRN-based radars[20][21]. These radars exploit the characteristics of pseudonoise sequences to compute the round trip time of the sent signal using the autocorrelation function. When the signal returns to the receiver, the correlation between the original signal and the received one will have a correlation spikes which indicates the exact time instant to consider as the arrival time.

In this chapter, the auto and crosscorrelation function for periodic binary sequences and its mathematical properties will be introduced as well as pseudorandom sequences, its properties and practical applications.

1.1 Binary sequences

Before diving into the matter, it is needed to point out the specific definition of binary sequences for this field:

In computer science, binary symbols are normally defined with the set $\{0, 1\}$ to be used in boolean

logic and arithmetics. This approach is useful when dealing with the inner workings of a computer or its theoretical constructs.

However, the discussed topic in this work takes a mathematical approach. In fact, most parts of the theory exposed here are just specific versions of general definitions for complex numbers. With this in mind and to avoid redefining needlessly the equations, we will stick to sequences composed by the elements 1 and -1, taken from the ring of integers.

1.2 Correlation function

According to Golomb and Gong [18], the correlation function measures how similar two phenomena are. If properly normalized, the function ranges from +1 (identical) to -1 (opposite); 0 meaning completly unrelated phenomena. If those phenomena are represented as vectors, the correlation can be conceived as the normalized dot product between those 2 vectors. In the discrete case where both sequences have the same length, the one in which this project focus on, the normalized version is defined as follows:

Definition 1.2.1 (Normalized correlation)**.** *Given α and β two vectors of the same length n and α_i and β_i the components of the vectors:*

$$C(\alpha,\beta) = \frac{(\alpha\cdot\beta)}{|\alpha||\beta|} = \frac{\sum_{i=0}^{n-1}\alpha_i\beta_i}{(\sum_{i=0}^{n-1}\alpha_i^2)^{\frac{1}{2}}(\sum_{i=0}^{n-1}\beta_i^2)^{\frac{1}{2}}}. \tag{1}$$

Notice that in this vector representation:

- Orthogonal vectors have a correlation value of 0 .
- Vectors with the same direction and orientation have a correlation value of 1.
- Vectors with the same direction but opposite orientation have a correlation value of -1.

Even though the normalized version is a good way to grasp the concept of the degree of similarity between two phenomen, for the rest of the document the unnormalized version is going to be used unless it is stated. This definition of the correlation function have several advantages for our research as it is simpler and carries the same amount of information while saving us some computation resources and complexity on our theoretical analysis. The unnormalized correlation is defined as:

Definition 1.2.2 (Unnormalized correlation)**.** *Given α and β two vectors of the same length n and α_i and β_i the components of the vectors:*

$$C(\alpha,\beta) = (\alpha\cdot\beta) = \sum_{i=0}^{n-1}(\alpha\odot\beta)_i = \sum_{i=0}^{n-1}\alpha_i\beta_i \tag{2}$$

where "$\odot$" represents the pointwise product of vectors.

As represented in Figure 1, unnormalized correlation can be performed using only integer arithmetics, multiplication and addition, thus becoming easier to implement using the resources avaliable on a digital device.

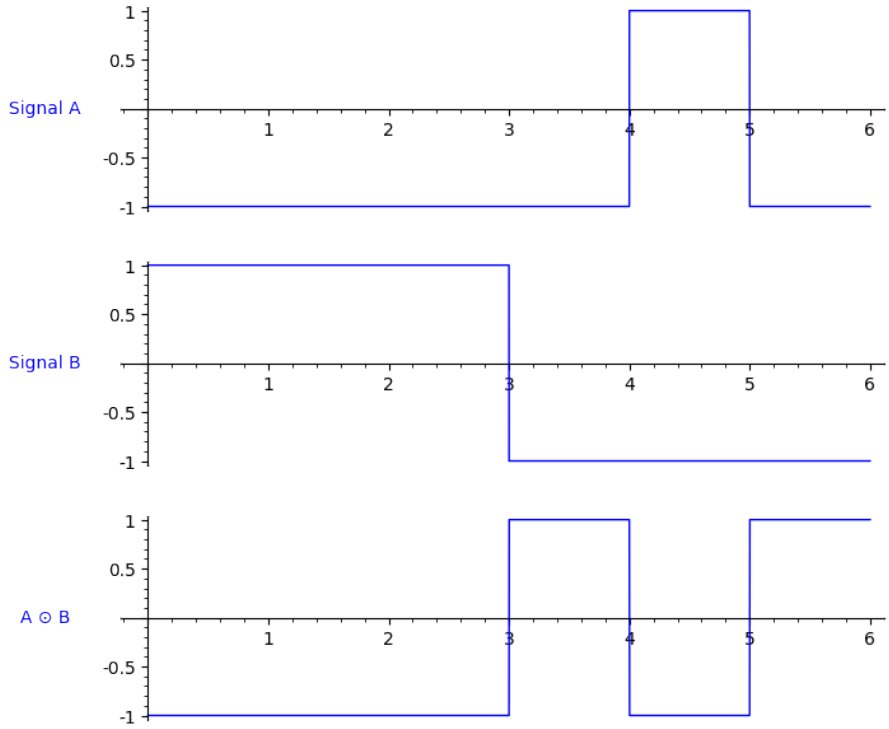

Figure 1: A graphical representation of two vectors an their pointwise product with an unnormalized correlation between them of -2 (-1 -1 -1 +1 -1 +1)

1.2.1 Autocorrelation function

Going on with the lecture of Golomb and Gong [18], the autocorrelation function is a measure of how the correlation behaves if, for a given sequence, a circular shift is applied and then correlated with the original sequence for every possible shift. It is defined for periodic sequences as follows:

Definition 1.2.3 (Autocorrelation)**.** *Given the function C defined in Equation (2) and n the length of the sequence S*

$$shift(S,\tau)_i = S_{(i+\tau) \bmod n} \tag{3}$$

$$A(S)_\tau = C(S, shift(S,\tau)) = \sum_{i=0}^{n-1} S_i S_{(i+\tau) \bmod n} \tag{4}$$

An example is shown in Figure 4 in which some important properties of the autocorrelation of sequences can be observed:

Theorem 1.2.1. *Given a sequence S, the autocorrelation value for $\tau = 0$ is:*

$$A(S)_0 = C(S,S) = \sum_{i=0}^{n-1} S_i^2 \tag{5}$$

Corollary 1.2.1.1. *Given the unnormalized autocorrelation of a sequence, it can be normalized by dividing it as follows:*

$$A'(S)_\tau = \frac{A(S)_\tau}{A(S)_0} \tag{6}$$

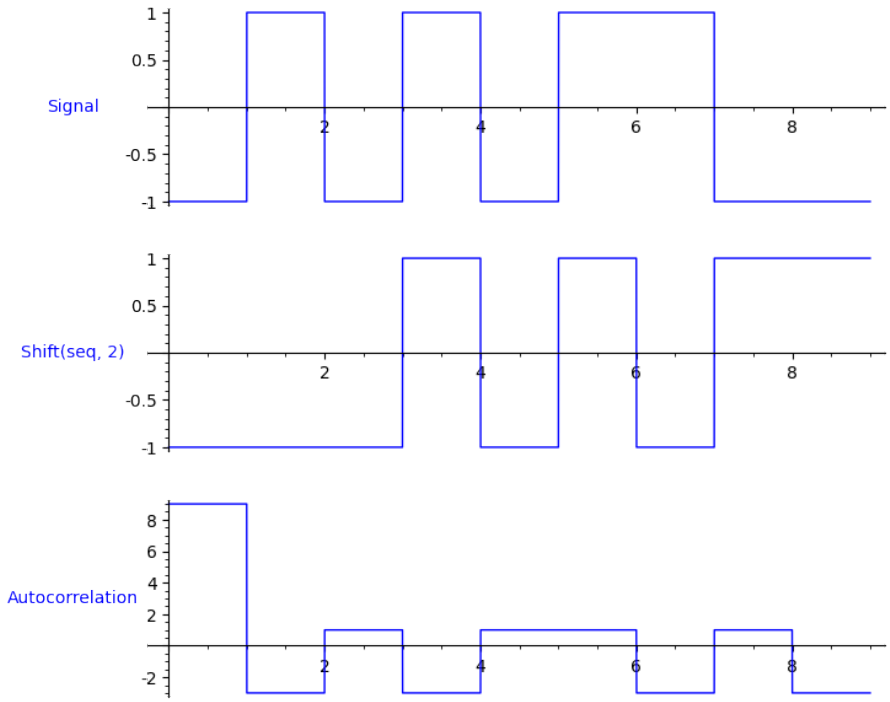

Figure 2: A graphical representation of the autocorrelation of a sequence with a shifted version of itself.

Proof. Using Equations (1) and (4), Equation (4) can be normalized as follows:

$$A'(S)_\tau = C'(S, shift(S,\tau)) = \frac{C(S, shift(S,\tau))}{(\sum_{i=0}^{n-1} S_i^2)^{\frac{1}{2}}(\sum_{i=0}^{n-1} S_{i+\tau}^2)^{\frac{1}{2}}} = \frac{A(S)_\tau}{\sum_{i=0}^{n-1} S_i^2} = \frac{A(S)_\tau}{A(S)_0}$$

Keep in mind that, even though S_i^2 and $S_{i+\tau}^2$ aren't the same element, the elements of the shifted version are the same as the original sequence so the total sum is the same. □

Corollary 1.2.1.2. *Given the autocorrelation of a sequence, $A(S)_0$ will always be the maximum value of the autocorrelation.*

Property 1.2.1. *If the components of the original sequence belong to the same ring, the components of its autocorrelation belong to that same ring.*

Even though this seems a naive property, this will prove useful when we introduce the algorithm based in the Fourier Transform to compute the autocorrelation function.

1.2.2 Crosscorrelation function

The crosscorrelation function measures how a sequence correlates with all the posible shifts of another sequence. This function is useful to analyze if two signals can be mistaken one for another by a receiver when delays in time occur.

Definition 1.2.4 (Crosscorrelation)**.** *Given C the correlation function defined in Equation (2), shift as the function defined in Equation (3) and n the length of both sequences:*

$$CC(S1, S2)_\tau = C(S1, shift(S2,\tau)) = \sum_{i=0}^{n-1} S1_i S2_{(i+\tau) \bmod n} \tag{7}$$

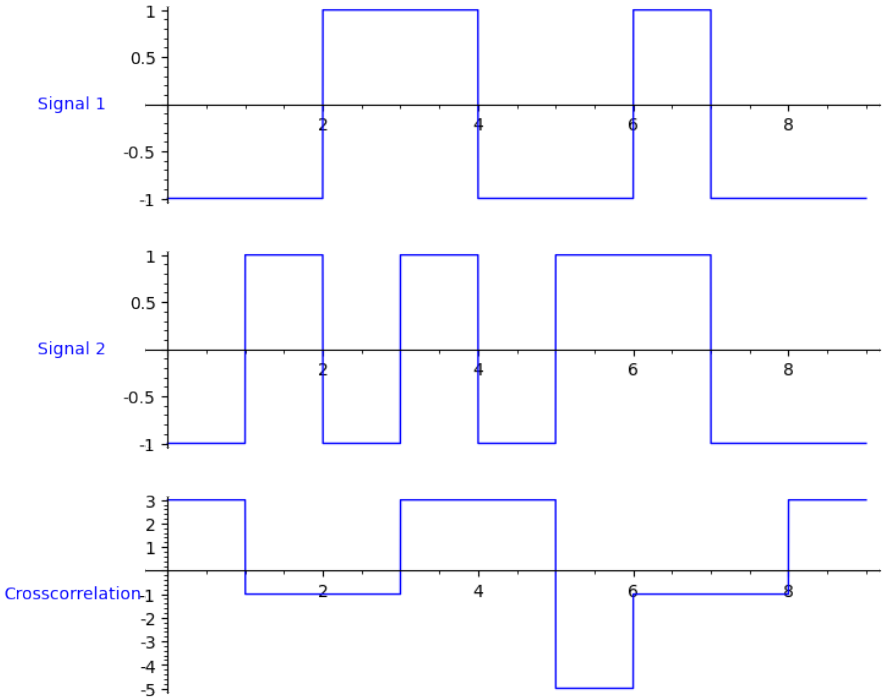

Figure 3: A graphical representation of the crosscorrelation between two sequences.

Definition 1.2.5. *Given a sequence S, the crosscorrelation function (CC) can be defined in Equation (7) with the autocorrelation function A defined in Equation (4):*

$$CC(S, S) = A(S) \tag{8}$$

1.3 Pseudorandom noise (PN)

Noise have a different meaning depending on the field of study in which is used. In our case we are going to work with random vectors, which are defined as vectors whose components are realizations of independent and uniform distributed variables[14].

Even though noise in general is usually seen as an unwanted phenomena that limits the amount of information that can be transmitted through a channel[29], it is useful to study its properties, such as

Property 1.3.1. *The expected value of the autocorrelation of a random vector is zero for every component where $\tau \neq 0$ [13].*

Taking a radar as an example, using this property the distance can be computed just by sending a random sequence in the direction of a target and start correlating the received signal with the original one. As the autocorrelation of random vector is different from zero only when the shift is zero, the peak in the autocorrelation gives in which time instant the signal has returned to the receiver. With that time instant, the round-trip time can be computed and then the actual distance using the propagation speed of the wave.

In the case of GPS, the restrictions imposed to the random sequence are stronger. First of all, as several signals will be transmitted in the same frequency, a set of sequences with good auto and crosscorrelation properties between them is needed. In other words, the maximum crosscorrelation function between two given sequences must trend to zero in every component, except when $\tau = 0$ and

both sequences are the same.

Random sequences do not guarantee good properties of correlation, even noise measured from natural phenomena can generate sequences with poor correlation properties. As stated before, important technologies depend on sequences with these properties, therefore, it is necessary to develop methods that are efficient to create sequences with properties similar to those of random in a deterministic and efficient fashion.

This kind of sequences are called Pseudo Noise (PN). Although for most applications, the off peak autocorrelation and crosscorrelation should be equal to zero, it is conjetured that such sequence do not exist apart from length four. That is why pseudo noise are use in practical applications. Then a threshold is defined so that the system will not mistake intermediate values with the peak in the autocorrelation.

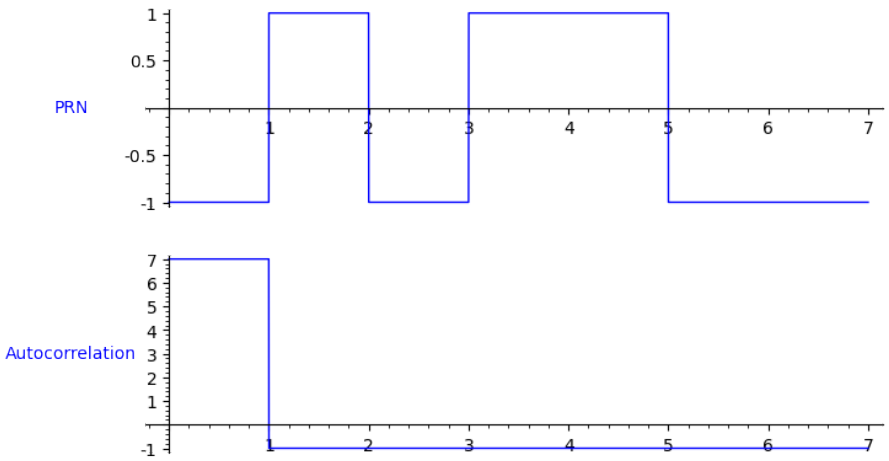

Figure 4: A pseudorandom noise sequence an it's autocorrelation function. Notice that this pseudonoise sequence isn't perfect noise.

2 Pseudonoise generation

As explained in the previous chapter, pseudonoise sequences are useful in tecnologies that need properties similar to those of white noise. In this chapter, some state-of-the-art tecniques in pseudonoise generation will be introduced.

2.1 Maximum Length Sequence(m-sequences)

M-sequences are an exponential binary pseudonoise construction that was initially concived using linear feedback shift registers(LFSR). The particular type of LFSR used in m-sequences can be simulated with extensions of binary Finite Fields. The definitions are:

Definition 2.1.1 (LFSR)**.** *A m-sequence is a binary sequence generated by an LFSR that, given an initial state different from 0, it cycles between all posible states except 0.*

Which, as shown in Golomb and Gong [18], is equivalent to:

Definition 2.1.2 (Finite Fields)**.** *Given $E/GF(2)$, α a primitive element of E and S the resulting sequence:*

$$S_i = trace(\alpha^i) \tag{9}$$

Property 2.1.1. *A m-sequence will always be of length of the form $2^n - 1$ where n is an arbitrary natural number.*

Property 2.1.2. *A m-sequence sequence will always have an autocorrelation function such as all the components will be -1 except when $\tau = 0$*

```
def maximal_sequence(n):
    f.<alpha> = GF(2**n)
    g = f.primitive_element()
    r = vector([(g^i).trace () for i in range(f.order()-1)]).change_ring(ZZ)
    return vector([x*2 - 1 for x in r])
```

Figure 5: An example of a posible implementation of m-sequences

Notice that, even though the construction is exponential, the complexity of the algorithm is $O(n)$ when n is the size of the sequence. The problem is that sequences of arbitrary size might be needed in some applications. As it will be discussed see in a following chapter, the complexity of computing the autocorrelation with the Fourier Transform approach is $O(n * log(n))$ so using longer sequences than needed has a direct impact on the performance of the system.

This sequences by themselves might not a be huge deal because they don't define a way to build families of well crosscorrelated sequences, but they are the building blocks for other contructions, such as the Gold Codes used in GPS and CDMA.

2.2 Gold Codes

Gold codes[17] are a family of sequences, derived from m-sequences, with very important properties that are used in several applications such as wireless communication and geolocalisation. A Gold Code generator gets two m-sequences sequences that fulfill:

Property 2.2.1. *Given two m-sequences that can generate Gold Codes, $S1$ and $S2$, of length $2^n - 1$, CC the crosscorrelation function defined in Equation (7):*

$$max|CC(S1, S2)| \leq 2^{\frac{n+2}{2}} \tag{10}$$

And XORs all their relative shifts generating a family of $2^n + 1$ sequences ($2^n - 1$ XORed sequences + 2 m-sequences).

Property 2.2.2. *Given any two sequences, $S1$ and $S2$, from a Gold family of sequences of length $2^n - 1$, CC the crosscorrelation function defined in Equation (7):*

$$max|CC(S1, S2)| \leq \begin{cases} 2^{\frac{n+2}{2}} + 1 & \text{if n is even} \\ 2^{\frac{n+1}{2}} + 1 & \text{if n is odd} \end{cases} \tag{11}$$

This property means that the crosscorrelation between any given pair of sequences from a Gold family is low enough to differentiate them. This proves useful when several devices are transmiting in the same frecuency and signals that are not the one being received must be treated as noise.

Gold also showed a way to generate this pair of sequences, using a decimation of one m-sequence.

Definition 2.2.1 (Decimation)**.** *Given a sequence S of length n, a decimation by q of S is defined as:*

$$S[q]_i = S_{((q \cdot i) \bmod n)} \tag{12}$$

Property 2.2.3. *Given a m-sequence S of length $2^n - 1$ where n is odd and and a coprime of n named k, the sequence pair $(S, S[2^k + 1])$ number fulfills Equation (10).*

```
def gold_code(maximal1, maximal2):
    r = [maximal1, maximal2]
    for x in range(len(maximal2)):
        r.append(hadamard_product(maximal1, displace(maximal2, x)))
    return r

def decimation(sequence, k):
    l = []
    for x in range(len(sequence)):
        l.append(sequence[(x*k) % len(sequence)])
    return vector(l)

def gold_with_decimation(n):
    s1 = maximal_sequence(n)
    s2 = decimation(s1, 3) # 3 = (2^k) + 1 when k is 1(1 is coprime with every number)
    return gold_code(s1, s2)
```

Figure 6: An example implementation of a generation of a family of gold sequences relying in the example at Figure 5.

Notice that this construction has the same problem as m-sequences. It's an exponential contruction so it might not be enough for some applications.

2.3 Legendre sequences

Legendre sequences, as explained in Zierler [32], are binary sequences defined through quadratic residues as follows:

Definition 2.3.1. *Let p be an odd prime and the function "Legendre Symbol" be:*

$$LSy(n,p) = \begin{cases} 1 & \text{if n is a quadratic residue mod p} \\ -1 & \text{otherwise} \end{cases} \tag{13}$$

The Legendre Sequence can be defined as:

$$LSs(p)_i = LSy(i-1,p) \text{ where } 1 < i \leq p \tag{14}$$

```
def legendre_symbol(a, p):
    residues = quadratic_residues(p)
    m = a % p
    if m in residues:
        return 1
    else:
        return -1

def legendre_sequence(p):
    return vector([legendre_symbol(a, p) for a in range(0, p)])
```

Figure 7: An example implementation of the generation of a Legendre sequence.

Some Legendre Sequences have interesting autocorrelation properties:

Property 2.3.1. *Given an odd prime p such as $p \equiv 3 \bmod 4$, $LSs(p)$ has a flat autocorrelation.[32]*

Even though Zierler [32] has a generalization of property 2.3.1 to all Legendre Sequences, it requires the introduction of a third symbol making the sequence non-binary.

As p is the variable defining the size of the generated sequence, the distribution of Legendre Sequences is related to the Prime Number Theorem. This means that Legendre Sequences have more possible lengths than in m-sequences or other exponential contructions. However, Legendre Sequences have the drawback that there is only one per sequence length.

2.4 Composition method

2.4.1 Algorythm

The composition method (introduced by Tirkel et al. [31] as Prime Arrays) uses a base sequence and a sequence of shifts to create a matrix of sequence components as follows:

Definition 2.4.1 (Composite matrix)**.** *Given a base sequence S of length n and a sequence of integers T of length m such that:*

$$0 \leq T_i < n \tag{15}$$

$$gcd(n,m) = 1 \tag{16}$$

Given the $shift$ function defined in Equation (3), the composite matrix is defined as:

$$CM(S,T) = \begin{bmatrix} shift(S,T_0)_0 & shift(S,T_1)_0 & \dots & shift(S,T_{m-1})_0 \\ shift(S,T_0)_1 & shift(S,T_1)_1 & & \\ \vdots & & \ddots & \\ shift(S,T_0)_{n-1} & & & shift(S,T_{m-1})_{n-1} \end{bmatrix} \tag{17}$$

In other words, each column represents a shift of the base sequence defined by the sequence of shifts.

Definition 2.4.2 (Composite sequence)**.** *Given a base sequence S of length n and a sequence of integers T of length m that fulfill Equations (15) and (16) and the composite matrix defined at Equation (17), the composite sequence is defined as:*

$$CS(S,T)_i = CM(S,T)_{(i \bmod m),(i \bmod n)} \tag{18}$$

$$S = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$
$$T = \begin{bmatrix} 0 & 2 & 1 & 4 & 3 \end{bmatrix}$$
$$CM(S,T) = \begin{bmatrix} 0 & 3 & 4 & 1 & 2 & 4 \\ 1 & 4 & 0 & 2 & 3 & 0 \\ 2 & 0 & 1 & 3 & 4 & 1 \\ 3 & 1 & 2 & 4 & 0 & 2 \\ 4 & 2 & 3 & 0 & 1 & 3 \end{bmatrix}$$
$$CS(S,T) = [0\ 4\ 1\ 4\ 1\ 4\ 1\ 0\ 2\ 0\ 2\ 0\ 2\ 1\ 3\ 1\ 3\ 1\ 3\ 2\ 4\ 2\ 4\ 2\ 4\ 3\ 0\ 3\ 0\ 3]$$

Figure 8: Example of a computation of the composition method(note that a non-binary sequence is used to illustrate better the method).

Definition 2.4.3 (Composite matrix correlation)**.** *Given 2 composite matrixes $M0$ and $M1$ with n rows and m columns, it's correlation is defined as:*

$$C(M0, M1) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} M0_{j,i} M1_{j,i} \tag{19}$$

Definition 2.4.4 (Composite matrix shift)**.** *Given a composite matrix M of n rows and m columns, the shift function is defined as:*

$$shift(M,\tau)_{i,j} = M_{(i-\tau \bmod m),(j-\tau \bmod n)} \tag{20}$$

This means that the following relation can be stablished:

Corollary 2.4.0.1. *Getting a shift of the composite sequence is equivalent to applying a shift to the matrix and then extracting the corresponding sequence.*

$$shift(CS(S,T),\tau)_i = shift(CM(S,T),\tau)_{(i \bmod m),(i \bmod n)} \tag{21}$$

When the autocorrelation function is defined for a composite matrix, an interesting property arises:

Definition 2.4.5. *Given a composite matrix M of n rows and m columns, it's autocorrelation function is defined as:*

$$\begin{aligned} A(M)_\tau = C(M, shift(M,\tau)) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} M_{j,i} M_{(j-\tau \bmod m),(i-\tau \bmod n)} = \\ \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} shift(S,T_i)_j shift(S, T_{(i-\tau \bmod n)})_{(j-\tau \bmod m)} \end{aligned} \tag{22}$$

Notice that if i is kept constant, a particular component of the autocorrelation function of S is obtained.

Property 2.4.1. *The autocorrelation function of the composite sequence can be defined in terms of the autocorrelation function as follows:*

$$\sum_{j=0}^{n-1} shift(S,T_i)_j shift(S,T_{(i-\tau \bmod n)})_{(j-\tau \bmod m)} = A(S)_{|((T_j-\tau) \bmod m)-T_{(j+\tau) \bmod n}|} \tag{23}$$

$$A(M)_\tau = \sum_{i=0}^{m-1} A(S)_{|((T_j-\tau) \bmod m)-T_{(j+\tau) \bmod n}|} \tag{24}$$

This property will prove useful in our software project as it's a fast method of computing the autocorrelation function.

2.4.2 Costas arrays

Costas arrays, discovered independently by John P. Costas[11] and E.N. Gilbert [16] in 1965, are a set of sequences highly used in radar and sonar applications. Both definitions will be provided as both will be useful for different purposes:

Definition 2.4.6 (Costas array(Costas))**.** *Square matrix of size n×n filled with 0s and 1s such that there aren't more than multiple 1s in each row or column and that every displacement vector is distinct from the rest.*

This definition is used in several deployments of sonar and radar to generate systems with a good ambiguity function, in other words, tolerant to the Doppler effect.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Figure 9: An example of a Costas array

Notice that the representation can be compacted by just having a list of the rows in which each 1 lives:

$$[4, 2, 1, 3]$$

Figure 10: The compact representation of the Costas array of Figure 9.

This representation is equivalent to the definition of a Costas array provided by Gilbert:

Definition 2.4.7 (Distinct difference permutations)**.** *Given a sequence of integers S of length n such that:*

$$0 \leq S_i < n \tag{25}$$

It is said that S is a distinct difference permutation r apart if, for any given pair (S_i, S_j), satisfies:

$$S_i - S_{i+r} \not\equiv S_j - S_{j+r} \bmod n \tag{26}$$

Definition 2.4.8 (Costas array(Gilbert))**.** *Given a sequence S satisfying Equation (25), it's called a Costas array if, for any given r value, it satisfies Equation (26).*

This compact representation can be feeded into the composition method as a sequence of shifts generating interesting new sequences[26].

Several construction methods have been proposed. For sake of simplicity, just the Welsh construction will be introduced which is defined in Gilbert[16] as:

Given a prime number p and a primitive root g of p, a Costas array S can be constructed as follows:

$$S_i \equiv g^i \bmod p \tag{27}$$

Notice that this contruction can generate sequences of a prime length as the Legendre Sequences. However, it can generate several sequences for a given length. As the number of possible sequences depend on the number of primitive elements of the finite field of order p, the number of possible costas arrays for a given length using this construction is $\phi(p-1)$ where ϕ is the Euler's totient function.

3 Exhaustive pseudonoise search

3.1 Previous work

As shown in the previous chapter, arithmetic methods for finding sequences with a low off-peak autocorrelation have huge constraints on the length of the generated sequences. To overcome this limitation, exhaustive searches through the possible permutations have been conducted in the past. This exhaustive searchs have, if not properly optimized, a search space of $O(2^n)$, where n is the length of the binary sequence, which make their results very limited by computational complexity.

Some developments have been conducted in the past for aperiodic autocorrelation optimization. Even though this project searches for periodic autocorrelation rather than aperiodic ones, it's worth taking a look at these works.

First of all, aperiodic autocorrelation will be defined to understand the optimizations proposed in these papers:

Definition 3.1.1 (Aperiodic autocorrelation)**.** *Given a binary sequence S, its aperiodic autocorrelation is defined as:*

$$A'_\tau(S) = \sum_{i=0}^{N-\tau-1} s_i s_{i+\tau} \tag{28}$$

All these works focus on finding a sequence S that minimizes:

Definition 3.1.2. *Given a binary sequence S, the energy of S is defined as:*

$$E(S) = \sum_{k=1}^{N-1} A'_k(S)^2 \tag{29}$$

$$E_{min} = \min_{subset} \sum_{k=1}^{N-1} A'^2_k \tag{30}$$

One of the first optimizations for this method was proposed by Mertens [25] in which he provided an algorythm with a complexity of $O(1.85^n)$. In his work, he applied a branch and bound algorythm that rules out the equivalent sequences and sets a minimum bound to the autocorrelation based on how complementing a single symbol of the sequence affects the autocorrelation.

First of all, the recursion is done by picking a sequence and recursively fixing elements at the extremes of the sequence as shown in Figure 11.

It's trivial that, when a symbol of the original sequence is complemented, the components of the autocorrelation can be lowered by, at most, -2. Based on that, a relaxation of E_{min} can be proposed:

$$E_b = \sum_{k=1}^{N-1} max\{b_k, (|A'_k| - 2f_k)^2\} \leq E_{min} \tag{31}$$

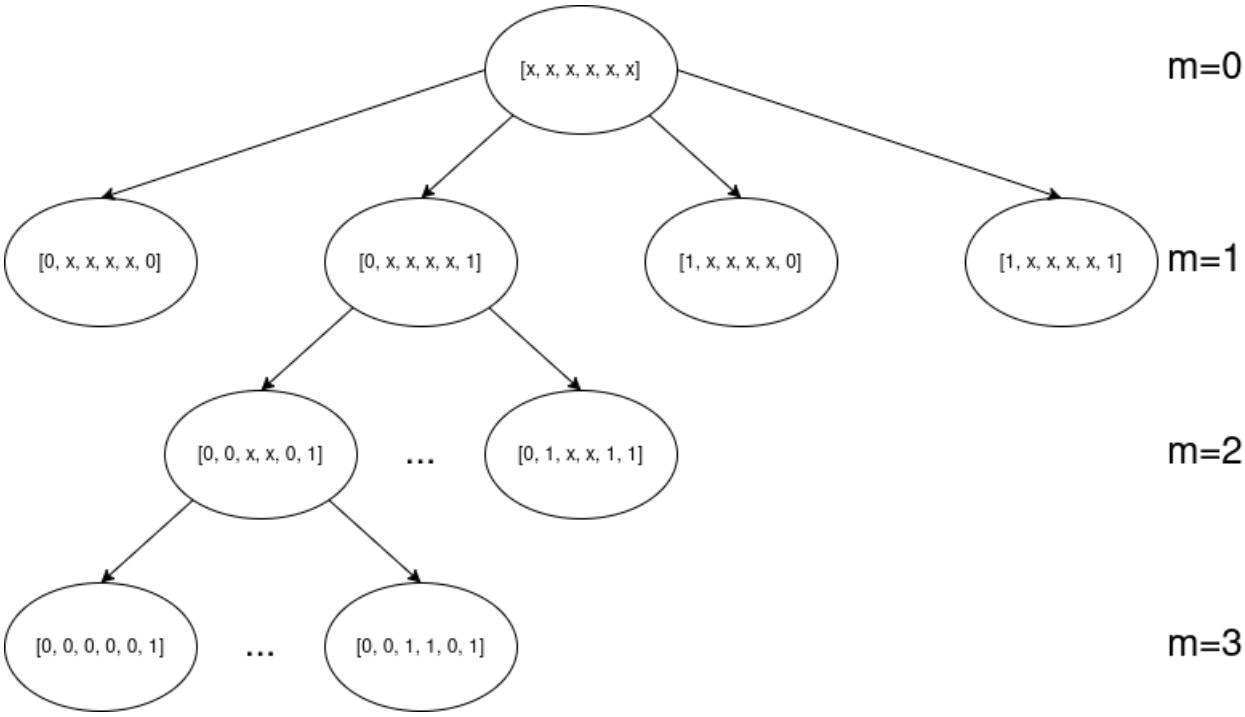

Figure 11: An example of the branching used in Mertens [25] with $N = 6$ and where x represent unfixed values.

where A'_k is the autocorrelation of an arbitrary sequence, $b_k = (N - k) \bmod 2$ the minimum possible value for $|A'_k|$ and f_k the number of unfixed elements in A'_k given by:

$$f_k = \begin{cases} 0 & k \geq N - m \\ 2(N - m - k) & N/2 \leq k < N - m \\ N - 2m & k < N/2 \end{cases} \tag{32}$$

where N is the size of the sequence to search and m the number of fixed elements at the extremes of the sequence.

If E_b is greater than the best candidate for E_{min} so far, that branch can be pruned reducing the amount of computation. With this algorythm, Mertens optimized sucessfully up to $N = 48$.

This work was further improved in Packebusch and Mertens [27]. In this paper, they review 2 bounds provided by different authors (Prestwich and Wiggenbrock) and combine them to create a new bound that lowers the complexity to $O(1.729^N)$, solving the LABS problem up to $N = 66$. This record was broken by Leukhin et al. [24] by computing it up to $N = 85$.

3.2 Our approach

To tackle the huge complexity encountered in the previous methods, a different approach was taken. Instead of dealing with the combinatorial explosion of all the possible binary sequences of length N, we decided to work with a smaller set consisting on all the possible sequences which can be constructed through the composition method with Legendre base sequences.

This approach haves some pros and cons. First of all, the search space is reduced from $O(2^N)$ to $O(p^m)$ where $p * m = N$. This clearly means that the complexity grows much more smoothly than in previous works. In fact, the autocorrelation function can be optimized for sequences generated through

the composition method as shown in a following chapter.

The problem is that the possible sizes for the sequences are limited as n and p are required to fulfill $gcd(p, m) = 1$. Even though it means that these method cannot find optimal sequences for all lengths, the restriction is looser than the non-exhaustive methods. Apart from that, this method doesn't explore all possible permutations and it doesn't ensure to find a pseudonoise sequence if it exists. To sum up, this method has been proven useful through examples as a good way to construct useful sequences but shouldn't be used to prove the non existence of pseudonoise sequences for a given length.

Given a base sequence of size n and a length m for the shift sequences, our program needs to find all the shift sequences that generate a composite sequence with a good autocorrelation.

This means that the search space are all the posible permutations of the shift sequence, in other words, n^m permutations. However, there are some relations between the different shift sequences that let us narrow the search space.

For example, if a constant is added to every component to the shift sequence, shifted version of the same sequence is obtained. This means that if only the permutations that start with the same component are computed, the whole search space would be covered as any other permutations would just be shifts of one permutation from the computed set. This optimization narrows our search space to $n^{(m-1)}$.

Other optimization arises from the form of the shift sequences. In general, if the symbols are repeated often, they trend to generate higher autocorrelation spikes or periods inside the composite sequence. This concept can be easily expressed with the Hamming autocorrelation function:

Definition 3.2.1 (Hamming autocorrelation)**.** *Given a sequence S of length n and the function $shift$ defined at Equation (3), the Hamming autocorrelation is defined as:*

$$HA(S)_\tau = \sum_{\tau=0}^{n-1} HAComponent(S_\tau, shift(S, \tau)_\tau) \tag{33}$$

where $HAComponent$ is defined as:

$$HAComponent(c1, c2) = \begin{cases} 1 & c1 = c2 \\ 0 & \text{otherwise} \end{cases} \tag{34}$$

For our branch and bound algorythm, it's important to note that if a symbol that only appears once is substituted for another, the hamming autocorrelation won't get lower. This means that if a depth-in-first bounding of the nodes that have a hamming autocorrelation higher than the threshold (we mean, the maximum non trivial component) is performed, all nodes in that branch are ensured to have a higher hamming autocorrelation than the threshold.

Some properties of the algorythm can be deduced from Figure 12. First of all, the number of autocorrelations computed can be reduced by a significant amount. However, the computation on each branch isn't balanced. This must be taken into account when the parallelism model is designed.

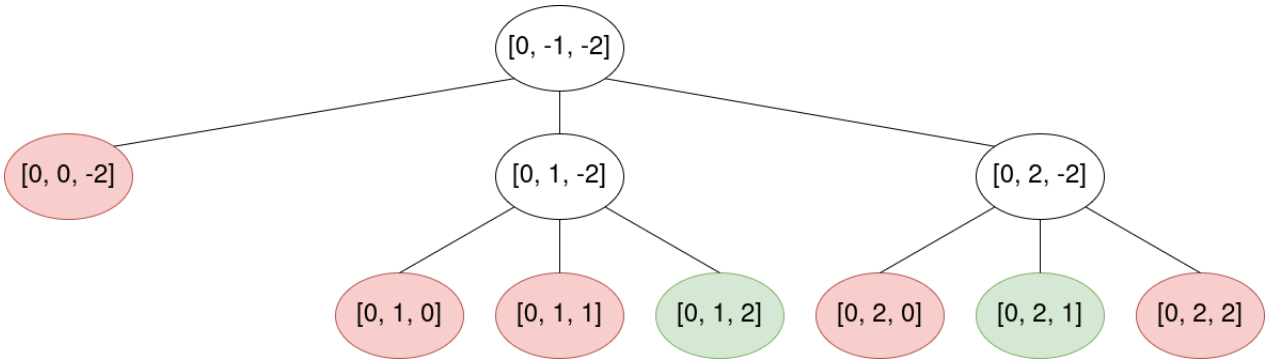

Figure 12: An example of the branch and bound algorythm with a threshold for hamming autocorrelation of 1 and a base sequence of length 3. Red nodes represent prunes and green ones final nodes in which the autocorrelation is computed and checked. Negative values represent those that haven't been initialized yet.

4 Software Engineering

Before starting to develop our software blindly, a good engineer has to plan beforehand how the proccess will be. Every different project has different characteristics that influence the methodology that should be taken. Technologies, procedures, testing, time schedules, client meetings... All of them must be taken into account to prevent extra work, bad quality software or client dissatisfaction. In this chapter, most of the information is taken directly from Sommerville [30] or highly inspired in Díaz [12].

4.1 Overall description

In this section a general idea behind the software project will be provided.

4.1.1 Project description

This project is developed to assist in the research of binary sequences with good autocorrelation functions. These sequences are used in applications such as radar, wifi or GPS.

The primary focus will be understanding the problem domain to create a software as useful as possible. Then create an extensible program that implements an exhaustive search to look for new sequences to add to the existing scientific literature.

4.1.2 Product functions

The functionalities that are expected from this software will be summarized in a general way:

- Perform computations in search for sequences.
  - Obtain real-time information of the computation proccess.
  - Change dedicated resources to computation.
- Manage the set of found sequences.
  - Query by different parameters of the sequences.
  - Handle found sequences manually.
  - Plot sequence properties.

4.1.3 User classes and characteristics

Users of this project can be divided in 2 main groups:

Researchers that will control the search for new sequences, being capable of starting computations and modifying the database. These researchers are highly specialized personnel and are expected to

understand common abbreviations in the field, as well as knowing which parameters are more promising to get a successful computation.

System administrators that will optimize the application according to the specificiations of the system in which it is deployed. System administrators are specialized professionals which know how to run benchmarks, maximize performance and understand advanced concepts of parallel computing. They are expected to be capable of optimizing the software with a manual that explains the inner workings of the program in a technical jargon.

In the real world, both roles might overlapp in the same person.

4.1.4 Operating Enviroment

The computation module is expected to run in highly parallelized systems such as super-computers. This is the enviroment in which researchers work, so it would be a shame if the software wouldn't take full advantage of the capabilities of these systems.

Taking this into account, the software installed in those enviroments should be targeted. This leads us to assume a Linux enviroment(as most supercomputers run it) and a Python interpreter(as it's a popular language in the scientific community).

4.2 Software requirements

In this section the software requirements of our project will be introduced.

4.2.1 Functional requirements

The functional requirements of the project are shown in Figure 13.

Identifier	Description
FR01	The researcher must be capable to set the base sequence to use for the search
FR02	The researcher must be capable to set the size of the shift sequence to use for the search
FR03	The researcher must be capable of setting the maximum autocorrelation he is interested in
FR04	The researcher must be capable to extract the results while the computation is still running (this doesn't mean that the data must be avalaible with a low latency)
FR05	The system administrator must be capable of changing the resources assigned to the program
FR06	The system must provide an interface that shows the progress of the computation (again, there's no need for low latency as it would conflict with NFR01)
FR07	Once established the parameters, the software must run without needing supervision of any user
FR08	The parameters of the load balancer must be editable by the system admin (as different machines might need different values)
FR09	An administrator must be capable of managing privileges for accesing the database and running computations
FR10	The system must provide a way to queue different searches to compute in succesion

Figure 13: Functional requierements

4.2.2 Non-functional requirements

The non-functional requirements of the project are shown in Figure 14.

Identifier	Type	Description	Relevance
NFR01	Performance	As sequences are being searched in a huge space, speed is a top priority to continue with the research	Very high
NFR02	Compability	As our project is being deployed in a research enviroment with different arquitectures, it should be made as compatible as posible in the case it must be changed to another node	Medium
NFR03	Usability	This software is expected to be used by specialized researchers and it isn't an interactive application, so the time spent dealing with the application by users is low. Interface shouldn't be a priority	Very low
NFR04	Arquitecture	The program must take full advantage of the capabilities of a supercomputer, in particular the high degree of parallelization of the system	High
NFR05	Robustness	The program must not produce errors. Corruption of data, miscalculations or precision errors must not be tolerated as it would screw up the whole result	Very high
NFR06	Robustness	The program cannot have memory leaks. It's expected to run for a long time and a memory leak can cause a crash. It can be fixed by restarting memory leaked threads without affecting the end result	Medium
NFR07	Extensibility	Since the program is used as part of a research, the parts of the software should be reusable in case the research shows a new posible use for the project as part of a new development	Medium
NFR08	Data availability	The availability of the data isn't a main concern as the project doesn't aim to be an interactive platform	Low
NFR09	Robustness	The persistence layer must be robust enough to avoid data loses since it is costly to produce	High

Figure 14: Non-functional requirements

4.2.3 User interface requirements

UI design is an important topic of software engineering as the success of a project is related directly to the users experience and how they relate to the software.

First of all, note that the users are supposed to be experienced in the use of computers, so a complex UI shouldn't be a problem. In this case, even though the easier the better, our development has a huge constraint on UI design that should be taken into account: the special type of OS this project will be deployed in.

As the main focus are supercomputers, a minimalist enviroment with no graphical desktop is to be expected. For this reason, a command line based application is preferred with 2 different main sections:

- Application launcher (resource allocation, parallelism model, etc.)
- Runtime interaction with the system (tasks management, database queries, etc.)

As most supercomputers run UNIX-based systems, our application should follow the POSIX[3] standard on the way it treats arguments. It should follow conventions such as the use of flags such as –help or –verbose and providing a man page.

It will also provide a way to store the configuration of the system in case the application must be restarted quickly (mainly platform specific configuration such as parameters of the load balancer).

4.3 Verification

One of the most important parts of software development is verifing the software. In other words, checking that the semantics of the program built are the same as the intended ones. A lot of time can be invested building a program to realize that it doesn't work in the last moment and waste a lot of time trying to fix it. That fix might generate side effects that break other parts of the program and so on. Testing since the early stages of a project is mandatory if a quality product and an efficient development proccess is to be accomplish.

4.3.1 Unit tests

Unit testing is the smallest piece of test suite in a project. There exist several approaches in the literature such as white box and black box testing. In our project, a mixed approach will be taken depending on the situation:

Property based tests

Property-based testing is a not so well known type of black box testing that is built around the idea of defining properties of functions instead of test cases. Originally implemented by the Haskell's libray "QuickCheck"[5], this paradigm excels at generating huge volumes of test cases with just some extra lines of code leading to improvements on the coverage over the search space. It's similar to the test automation explained by Sommerville [30] in Chapter 23, being the main difference that an oracle that predicts the value isn't needed. Instead, just a property of the output is checked.

A well implemented library(there are several of them, but in our case we are working with Hypothesis[2] since our project is built in Python) should be capable of applying most well practices of black box testing, such as edge cases, all pairs, etc.

The main reason why this type of test suites were choosen is that all the properties are already defined in this document and can be used straightforward as test cases. In fact, as most of our functions are static and pure, the generator will be very simple so the tests will take full advantage of this paradigm. In Figure 15, there is have an example of a property used for testing the codebase.

The problem with this paradigm is that it becomes way too complicated when the tested methods have side effects, IO, state machines, etc. As this kind of systems usually depend on complex rules to build the generator of all the components involved in this systems. For these kind of tests, we will rely on the old method of designing test cases by hand.

```
import Cython_lib.SignalProccesing as SP
import unittest as ut

from hypothesis import given
import hypothesis.strategies as st
import hypothesis.extra.numpy as hnp

class TestAutocorrelation(ut.TestCase):

    def generator(self, data):
        small_int = st.integers(min_value=0, max_value=25)
        signal_length = data.draw(small_int)
        return data.draw(hnp.arrays(np.int64, signal_length, elements=st.integers(
                                                    min_value=-1, max_value=1)))

    @given(st.data())
    def test_maximum_value(self, data):
        """
        Test that checks the maximum value of the autocorrelation
        """
        signal = self.generator(data)
        auto = SP.autocorrelation(signal)
        if not len(signal):
            return
        assert max(auto) == auto[0]
        assert max(auto) < len(auto)+1
```

Figure 15: An example test for Corollary 1.2.1.2

4.4 Validation

The validation proccess in this project depends highly in the iteration in which we are:

- In early iterations, the validation proccess might not be as important as the verification proccess. This is because the core functionality of the program is an algorythm expressed in a technical manner with little margin to misinterpretations.
- In later iterations, the validation proccess gains weight in respect of the verification proccess as we dive into the UI design. In this case, there is more room for missunderstandings between client and developer so we must take this proccess into account.

Fortunatly, we are working with an agile mindset so a validation session can be performed often so that the developers introduce the new features to our client. Then, he can try out the features and point out missunderstandings, desired changes, etc.

4.5 Agile development

One of the most important tasks to do before starting a project is deciding which project management model the project will follow.

In our case, this project is going to avoid a waterfall model (even though it is the taught model for branches of our degree not focused in software engineering) favoring agile development techniques adapted to our problem for several reasons:

- The client is an active part of the project. This means that a lot of feedback can be obteined during development and fix issues earlier.

- As this project is being developed in the middle of a health crisis, the availability of project resources cannot be predicted. This means that having a rigid schedule planned too ahead of time wouldn't be useful.

4.5.1 Role definition

In the development of this project, 2 different roles will defined assigned to 2 distinct people:

- Developer which will design, program, verify and manage the software project. In this case, this person corresponds to the author of this report (Juan Toca).
- Client which will provide the requirements for the project, as well as validating each iteration. In this case, this person corresponds to the director of this project (Domingo Pérez).

4.5.2 Iterations

In this subsection we will discuss how the development iterations went.

Some conventions: When we say a C function, we are referring to a Cython code without CPython code, in other words, functions that don't call the Python interpreter.

Iteration 1: Composite autocorrelation

In this iteration, the development was focused on developing an efficient way of computing the autocorrelation function of a base sequence with a given shift sequence. 3 versions were developed:

- A pure C function that given the autocorrelation of the base sequence and the shift sequence computes the autocorrelation.
- A wrapper Python function for the previous function which computes the autocorrelation of the given base sequence and passes it to the C function.
- A pure C function which checks if the maximum component of the composite autocorrelation exceeds the threshold provided.

The 2 first functions aren't part of the actual exhaustive search algorithm, but will be useful if it's properties must be checked when retrieving them results from the database.

The test proccess consisted in checking that the python wrapper provided the same results as a naive implementation of the algorythm based in the convolution theorem. From that, the third function was tested against the first one.

At first, C functions with fused types were developed. Although it favors extensibility, the compiler started to throw errors related with fused types. As the problem would have gotten worse when the next parts of the program were developed, it was decided to drop support for fused types as it was only expected to work with integers of 32 bits.

Iteration 2: Branch and bound algorythm

In this iteration we focused on a single threaded C implementation of the branch and bound algorythm. This function receives a threshold of the maximum autocorrelation the user is interested in, the maximum Hamming autocorrelation for the prune part of the algorythm and the base sequence to use.

For this purpose, a C implementation of the maximum Hamming autocorrelation was developed and an implementation of Legendre sequences to be used as base sequences (more types of base sequences might be added in the future, but this one was explicitly asked by the client).

The test designed for Legendre sequences exploits its flat autocorrelation to check the functions consistency. In the case of Hamming's autocorrelation, a test based on lower and upper bounds was designed.

In the case of the branch and bound method, it will be checked that all the sequences returned satisfied the specified maximum autocorrelation.

Iteration 3: Parallelism

In this iteration, the main focus was to adapt the branch and bound algorythm to a parallel enviroment. To do that, the algorythm was incrementally improved. At first, branch and bound was only applied from a given depth and then it was decided to also apply it at the master's process level.

In this case, this iteration was implemented in pure python as it isn't critical code. Most runtime of slave processes will be spent in Cython functions and, for simplicity and reliability, it was decided to stick to Python.

Tests in this iteration were made in a more manual fashion, running the code and checking that all results were coherent. This was done like this because the code being tested was completly impure and property based tests wouldn't have a worthy coverage to effort relationship to consider writing them.

Iteration 4: User Interface

In this iteration a suitable UI for the program was designed. To do so, two functionalities were implemented:

- Support for command line arguments to initialize tasks.
- A verbose mode to get statistics of the program.

Command line parameters complies to POSIX's standard and informs the user of possible errors in the input, while the verbose mode logs events with it's corresponding times to debug the performance of the computation.

Again, the testing of this module was purely manual because of all the IO involved. The user interface was shown and explained to the client to receive their approval.

5 Technology choices

In this chapter, the technologies used in this project will be discussed as well as the reasons behind their adoption, pros and cons for this project and difficulties encountered during development.

5.1 SageMath

SageMath[6] is a Python mathematical suite used in research projects as an enviroment for prototyping algorythms or math concepts in general.

In our case, it served as a junction point between a mathematician that is used to express ideas in math expressions and a developer that is used to understanding concepts by making them work. Apart from that, it was also useful at generating some figures for this document.

For our use case, a way to share the notebooks through the cloud was needed. We decided to work with a free version of CoCalc[8]. Even though it served the purpose of sharing code without the need of using a repository, I have to say that in terms of other services such as running the notebooks was very dissapointing. For low demanding tasks it performs well, but for bigger computations I had to copy the code and run it locally. In future projects, I might try out the payed version (as it has tons of features) or other alternatives.

5.2 CPython

Python[4] programming language is one of the most used languages in scientific enviroments, as well as in system integration developments.

In our case, the usage of Python serves 2 different purposes:

- It's the language the researchers are the most familiar with. This helps the project because it's finished, they can extend the software for their needs as they want.
- It's a language widely used. This means that it can expected support from a lot of platforms while having a high-quality bunch of libraries to work with.

In addition, the developer has a solid experience with the stack of technologies around Python which shouldn't be understated. Furthermore, his profiency in the language let's him explore more new concepts in the same time such as MPI or the whole domain knowledge needed for this project.

5.3 Cython

Python as a language comes in several implementations. CPython as the reference implementation has it's flaws, mainly it's performance issues. As the software being developed has performance con-

straints, using just the reference implementation is not an option.

Fortunatly, there are alternative implementations such as Jython, IronPython, etc. In our case, Cython[1][19] was the chosen one as it provides a compiler to build C code with pseudo-Python. Python code can be called from C functions an viceversa, proving useful when a system with C level performance to operate with high-level Python libraries is needed.

To support the decision of using Cython for the critical parts of the code, some benchmarks were developed to test the actual performance improvements. As shown in figures 16 and 17, Cython benefits a lot from tasks that requires iterations, but when using vector arithmetics with Numpy the performance impact drops. This is because behind the scenes Numpy functions are just Python wrappers for C functions so the heavy computation is done in C.

CPython:	318.65 seg	100.0 %
Cython:	12.64 seg	3.9 %
C:	12.33 seg	3.8 %

Figure 16: Results of a benchmark of a long iteration.

CPython:	8.66 seg	100.0 %
Cython unoptimized:	8.64 seg	099.7 %
Cython optimized:	8.26 seg	095.3 %
Cython GSL:	8.24 seg	095.1 %

Figure 17: Results of a benchmark of vector operations using Numpy or GSL.

One might think that, if vector operations are so efficient in CPython, it would be simpler to just use Numpy methods to implement our algorythms (which was indeed done in the general autocorrelation function with the algortyhm based on the convolution theorem). The problem raises when it is needed to access the Numpy array in an undefined way by the library and to code a loop to compute a function (the composite autocorrelation for example).

Cython provides native support for Numpy arrays, letting us access them with a C level performance. In fact, as it supports fused types (the equivalent to templates in C++), functions that can work with diverse types depending on the input arguments can be defined. Even though in our case it will be skipped as it comes with extra headaches and only integers are needed for the purposes of this project, it's a nice feature if it was needed to extend Sage to fully support our research field.

One important thing to take into account when developing with Cython is that C types allocated in the heap (Cython supports raw C vectors and data structures from C++ std) doesn't have automatic memory management. This will make the debugging tougher as it will be needed to look for memory leaks. In contrast, Numpy arrays do support automatic memory management at the cost of the overhead of type checking and reference counting.

5.4 PostgreSQL

PostgreSQL[10] is an open-source relational database supported in a highly varied range of enviroments. As such, there is no dependency on a particular Linux distribution to deploy (for example,

Oracle only supports RedHat).

Even if the database isn't as fast as Oracle or other databases, our application isn't constraint by IO (as the generation of values to store in the database depends directly in a costly computation). In our case, a persistence system that supports well our stack of technologies is preferred. As PostgreSQL is designed to run Python code on it's SQL code, it gives a lot of flexibility in how our application can be designed.

6 Implementation

This chapter introduces some specifications of the implementation of our solution.

6.1 General autocorrelation function

The autocorrelation function introduced in Equation (4) can be implemented in several ways:

6.1.1 Naive approach

The naive approach for the autocorrelation function consists in computing all the displacements of the sequence and then their correlation with the base sequence as shown in Figure 18. Even though this algorythm it's simple and follows the mathematical definition, is too slow. The correlation function for each component have to be computed, leading to a complexity of $O(n^2)$ (where n is the size of the sequence) with a huge constant as building the shifted sequence for each component is needed.

This constant could be improved by avoiding building the shifts (just using slices of the array), but the complexity would stay the same.

6.1.2 Circular convolution theorem

The other option is to step in the world of mathematical properties. Fortunatly, there exists the convolution theorem[18] that lets us express the autocorrelation function in terms of Fourier Transforms as:

Theorem 6.1.1. *Given a sequence S and the Discrete Fourier Transform(DFT):*

$$A(S) = DFT^{-1}[DFT\{S\} \cdot DFT\{S\}^*] \tag{35}$$

where $DFT\{S\}^*$ represents the complex conjugate of $DFT\{S\}$.

Notice that, using the Fast Fourier Transform[23], the complexity of this method lowers to $O(NlogN)$. However, its constant is still high as applying 2 FFT to the sequence and the complex conjugate is needed. In fact, keep in mind that Fourier Transforms work with complex components (however, according to Property 1.2.1, it will always return the same type as the original sequence) while the naive approach keeps using the same type as the components which makes its constant even higher than the optimized naive approach.

```
def displace(vec, offset):
    return vector(list(vec[offset:])+list(vec[:offset]))

def slow_naive_autocorrelation(vec):
    return [correlation(vec, displace(vec, x)) for x in range(len(vec))]
```

Figure 18: An example implementation of the naive autocorrelation

6.1.3 Specific solution for the composition method

If a general method for this computation were used, the one based on the convolution theorem would be preferred because the program is expected to deal with long sequences that will compensate the big constant of this method.

However, the sequences this program is dealing with has the special property of having been built through the composition method. This means that a non general way of computing this autocorrelation with better computational characteristics exploting the pecularities of this construction might exist.

Taking advantage of Property 2.4.1, an algorythm with interesting properties can be designed. First of all, the complexity function depends on the size of the shift sequence. Being m the length of the shift sequence and n the length of the composite sequence, the resulting algortyhm has a complexity of $O(nm)$. This means that when $m < log(n)$ this algorythm has a better complexity than the Fourier Transform's approach.

In addition, this algorythm has a better constant. It only needs to iterate once through the autocorrelation sequence. This method is more cache friendly too as the data source of the function is smaller and it doesn't need to use complex operations in binary sequences.

But the biggest improvement in respect of the Fourier Transform is that the complexity of a partial result of size p is $O(mp)$ while the convolution theorem requires $O(nlog(n))$ for a partial result. In practice this means that, if just checking a certain component of the autocorrelation is needed, there is no need to compute the whole function.

An example implementation of this algorythm is shown in Figure 19.

```
cdef bint c_good_composite_autocorrelation( int* autocorrelation
                                          , int l_signal
                                          , int* shifts
                                          , int l_shifts
                                          , int threshold):
  cdef int output_size = l_signal * l_shifts
  cdef int output
  cdef int x, y, affected_column, current_shift, final_shift, positive_difference
  for x in range(1, output_size):
    output = 0
    positive_difference = l_signal - (x%l_signal)
    for y in range(l_shifts):
      affected_column =  shifts[(y+x)%l_shifts]
      current_shift  =  (positive_difference + shifts[y]) % l_signal
      final_shift = abs(current_shift-affected_column)
      output = output + autocorrelation[final_shift]
    if output > threshold:
      return False
  return True
```

Figure 19: The Cython implementation of the composite autocorrelation. Notice that branchless programming is used to improve performance.

6.2 Single-threaded Branch and Bound

The theoretical approach for the Branch and Bound algorithm has been already explained. In this section, just the actual implementation that was used in the project is shown in Figure 20.

```
cdef void get_list_of_good_shifts( int* autocorrelation
                                 , int  sequence_length
                                 , int* initial_shift # Shift sequence
                                 , int  shift_length
                                   # Index at which the recursion must start
                                 , int  fixed_shift_offset
                                 , int  hamming_upper_limit
                                 , int  correlation_upper_limit
                                   # linked list at which we must store
                                   # the results
                                 , list[int*]* sequence_list):
  cdef int x, hamming, new_shift_offset
  cdef bint result
  cdef int* stored_sequence
  if fixed_shift_offset < shift_length: # Recursive case
    for x in range(0, sequence_length):
      initial_shift[fixed_shift_offset] = x # We fix a new component on the shift
      new_shift_offset = fixed_shift_offset +1 # We move the pointer
      hamming = SP.c_max_hamming_autocorrelation(initial_shift, shift_length)
      if hamming < hamming_upper_limit: # If we don't prune, we keep on with
                                        # the recursion
        get_list_of_good_shifts( autocorrelation
                               , sequence_length
                               , initial_shift
                               , shift_length
                               , new_shift_offset
                               , hamming_upper_limit
                               , correlation_upper_limit
                               , sequence_list)
    # After recursion, we unitialized the new component
    initial_shift[fixed_shift_offset] = -fixed_shift_offset
  else: # Base case
    result = SP.c_good_composite_autocorrelation( autocorrelation
                                                , sequence_length
                                                , initial_shift
                                                , shift_length
                                                , correlation_upper_limit)
    if result: # If the sequence has good properties, we store it
      stored_sequence = <int*> malloc(shift_length*sizeof(int))
      for x in range(0, shift_length):
        stored_sequence[x] = initial_shift[x]
      sequence_list.push_front(stored_sequence)
```

Figure 20: A Cython implementation of the branch and bound algorythm. Notice the amount of extra code to achive C performance.

6.3 Parallelism model

For the parallelism of the project, it was decided to work with MPI. This model was implemented in pure Python as there was no need for a high performance in this part of the software (as the time spent in this code is already minimal).

However, a low latency assignation of tasks is needed. If shared memory was used, every node would need to access memory through the ”slow” interconnection network. Instead, with MPI, proccesses to talk between them.

For the purpose of this project, MPICH[9] is used as the bindings of MPI4PY support it and it’s the implementation of the cluster we are working with[7].

In our particular problem, a set of tasks was defined to distribute between the different nodes. This tasks are subtrees from the search space with a given height that defines the size of the task. An example is show in Figure 21. Notice that the tasks are completly unbalanced so a static scheduler wouldn't be efficient at all.

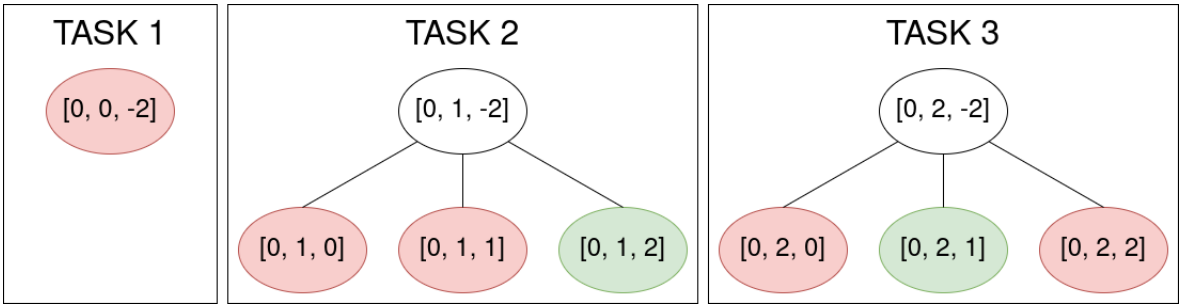

Figure 21: An example distribution of tasks for the example at Figure 12.

This model is a good first version. However, if a prune isn't performed when assigning the tasks, there will still be an exponential number of tasks (specifically, n^{l-t} where n is the length of the base sequence, l the length of the shift sequence and t the task size). This shouldn't be a problem if t is close to l, but this will raise an issue with the task balancer as there wouldn't be enough tasks to balance the load.

A second version of the algorithm applies the branch and bound algorythm in the master proccess to generate only tasks with good Hamming properties. This rules out all tasks that would instantly result in a bad autocorrelation and wouldn't generate any useful sequences. The code implementation is shown in Figures 22 and 23.

A third version (which wasn't implemented because the second version was enough for our objectives) could be done by using the Hamming as an heuristic to determine the size of the task. Instead of using a fixed size for the subtrees, the tasks to can be given to the process at hand based on the Hamming autocorrelation of the root of the task.

```
def master( polling_delay # Time between polls
          , shift_length # Length of the shift sequence
          , task_size # Height of the sub trees
          , hamming # Max hamming autocorrelation
          , max_value # Length of the base sequence
          , verbose):
    requests = []
    arr = np.array([-x for x in range(shift_length)], dtype=np.int32)
    # Task iterator
    task_iter = task_iterator(arr, 1, task_size, max_value, hamming)
    try:
        for x in range(1, num_proccess): # Initialize first task
            requests.append(comm.issend(shift_to_int(next(task_iter), task_size,
                                                     max_value), dest=x, tag=11))
    except StopIteration: # If there are not enough tasks
        kill_slaves()
        print("The number of tasks found where too low for the number of threads, "\
              "consider lowering the size of the task or assigning less cores")
        exit(0)


    for task in task_iter: # Iterate over all the tasks
        exit_var = False
        while not exit_var:
            sleep(polling_delay)
            i, b, msg = MPI.Request.testany(requests)
            if b and i >= 0: # If there is a finished task
                # assign new task
                requests[i] = comm.issend(shift_to_int(task, task_size, max_value), dest=
                                                     i+1, tag=11)
                exit_var = True
    # Tell the slaves to exit
    kill_slaves()


def task_iterator(arr, current_offset, task_size, max_value, hamming):
    it = len(arr) - task_size
    ham_auto = SP.max_hamming_autocorrelation(arr) < hamming
    if ham_auto: # Prune
        if current_offset == it: # base case
            yield arr
        else: # recursive case
            for x in range(max_value): # For all possible shifts
                arr[current_offset] = x
                # Go one step deeper
                yield from task_iterator(arr, current_offset+1, task_size, max_value,
                                                     hamming)
            arr[current_offset] = -current_offset
```

Figure 22: A Python implementation of the master proccess

```
def slave( base_sequence
         , sequence_length # Length of shifts sequences
         , task_size # Height of the sub trees
         , hamming_upper_limit
         , correlation_upper_limit
         , verbose):
    exit_var = False
    while not exit_var:
        # Recieve task
        t = clock_gettime_ns(CLOCK_PROCESS_CPUTIME_ID)
        data = comm.recv(source=0, tag=11)
        if data != -1: # If it's an actual task
            # Compute the task
            seq = int_to_shift_sequence(data, sequence_length, len(base_sequence),
                                                        task_size)
            if verbose:
                elapsed = int((clock_gettime_ns(CLOCK_PROCESS_CPUTIME_ID) - t)/1000000)
                log("TASK_ASSIGNED " + str(list(seq[:len(seq)-task_size])) + " " + str(
                                                            elapsed) + "ms")
            r = bb.py_get_list_of_good_shifts( base_sequence, hamming_upper_limit
                                        , correlation_upper_limit, seq, task_size)
            # Store the results
            store_sequences(r, rank)
        else: # If there's no more tasks
            if verbose:
                log("EXITED")
            # Exit
            exit_var = True
```

Figure 23: A Python implementation of an slave proccess

6.4 UI implementation

Last but not least, the design of the User Interface will be briefly discussed. As expressed in the chapter of Sofware Engineering, our primary focus is its compatibility with command lines. For that, a POSIX compatible interface was created.

First, a –help option was developed to list all the flags:

```
$ python main.py --help
usage: python main.py [option ...]

Options and arguments:
-n : number of threads to use(must be compatible with your MPI enviroment)
     Defaults to the MPI configuration default

-p : delay between polls in the master thread(higher values will make the
     slaves to wait more until the next task, lower values will increase
     CPU usage of master)

-s : length of the base sequence(in this version must be a prime number to
     generate Legendre sequences. Other values have undefined behaviour)

-l : length of shift sequences(this option must be coprime with value
     of s, other values have undefined behaviour)

-t : size of the task for each thread(this option must be lower than the
     value provided by -l, other values have undefined behaviour)

-h : maximum hamming autocorrelation allowed(this option must be a
     positive integer other values have undefined behaviour) Defaults to -l

-c : maximum autocorrelation we are interested in(this option must be a
     positive integer other values have undefined behaviour) Defaults to
     the square root of (-l*-s)

-v : verbose mode
```

Verbose mode logs which tasks have been assigned and at which time stamp, as well as logging the end of slave proccesses and the idle time of slaves:

```
$ python main.py -s 5 -l 23 -t 20 -c 7 -h 3 -v
2020-08-23 19:27:49 [1] : TASK_ASSIGNED [0, 0, 1] 9ms
2020-08-23 19:27:49 [3] : TASK_ASSIGNED [0, 0, 3] 10ms
2020-08-23 19:27:49 [5] : TASK_ASSIGNED [0, 0, 4] 6ms
2020-08-23 19:27:49 [7] : TASK_ASSIGNED [0, 1, 1] 17ms
2020-08-23 19:27:49 [6] : TASK_ASSIGNED [0, 1, 0] 3ms
2020-08-23 19:27:49 [2] : TASK_ASSIGNED [0, 0, 2] 0ms
                        .
                        .
                        .
2020-08-23 19:28:41 [1] : TASK_ASSIGNED [0, 4, 0] 0ms
2020-08-23 19:28:42 [7] : TASK_ASSIGNED [0, 4, 3] 0ms
2020-08-23 19:28:45 [1] : EXITED
2020-08-23 19:28:45 [5] : TASK_ASSIGNED [0, 4, 2] 0ms
```

```
2020-08-23 19:28:47 [2] : EXITED
2020-08-23 19:28:48 [4] : TASK_ASSIGNED [0, 4, 4] 0ms
2020-08-23 19:28:49 [6] : EXITED
2020-08-23 19:28:52 [3] : EXITED
2020-08-23 19:28:55 [5] : EXITED
2020-08-23 19:28:56 [7] : EXITED
2020-08-23 19:28:57 [4] : EXITED
```

Notice that, as stdout is used, the log can be piped to a file. The output format is designed in a way that eases the use of utilities such as awk to proccess the data of the program (one word message, well defined columns, etc.).

Verbosity is completly optional and doesn't impact performance when inactive. It's particularly useful when tweaking the parameters to get the best configuration.

7 Future work

Even though we have a working prototype that can perform the algorythm, there's still a lot of work to be done:

First of all, the persistency part of the program needs a complete rewrite. Nowadays, the search results are dumped in plain text files and have no kind of advanced queries or a way to store sequence properties.

The program has a design limitation on the diversity of base sequences that can be used for the computation. It would be interesting to be capable of using different kinds from Legendre sequences. This leads us to the topic of a whole refactor of the code if Domingo wants to publish it as part as the research project he is doing. Some parts can be written more clearly and the MPI usage isn't the most efficient one. For example, all the tasks are assigned unbuffered which means that the interconnection network delay hasn't been mitigated at all.

Apart from that, the program haven't been tested in a production environment. Some incompatibility problems must be fixed between the development enviroment in which, among other things, OpenMPI was used rather than MPICH and I wasn't able to compile MPI4PY in the supercomputer(I have to say that I hadn't much time to do so because of limitations out of my control). Needless to say that this means we haven't extended the existing literature on the topic yet.

To finish my contribution to Domingo's research, I need to develop a complexity analysis of the algorythm in a similar fashion to the previous works of exhaustive research. To do so, we need to run the code for a long time to gather enough data to estimate the said complexity.

Last but not least, the UI needs some tweaking too. We should provide the possibility of storing the parameters in a configuration file as some of them will be shared between most of the tasks (for example, the number of processes used). I should also review if the program really needs a permissions system for the database. As its running in an isolated enviroment to perform the computation, no external user has access to the data unless it's uploaded to an external server through the VPN(which would be a terrible practice and it would result in another different app).

This seems like a really long lists of TODOs. Keep in mind that this Bachelor thesis is a part of a research project that isn't yet finished. I think I've accomplish the goals that my director had for this thesis.

8 Conclusions

In general, I think I can say I've accomplished my personal objectives for this project. My main objective was to learn how a project of scientific computation must be developed: its needs, requirements and peculiarities.

First, and probably the most difficult task for a developer is to understand the problem domain. This kind of sofware, in contrast with other consulting projects, requires a high level of knowledge in a scientific research to be able to grasp the semantics of the program. This adds a huge challenge to the software engineer, as they must have a close relation with the client to ensure the correctness of the program. I can affirm with confidence that two thirds of the time was spent in learning all the concepts that where going to be used later to gather all the software requisites and semantics.

Apart from that, new coding challenges arised. The deployment on a supercomputer needs a paradigm shift in program design. Shared-memory parallelism is no longer an option as we are working with several computers connected by a "slow" interconnection network. Instead, a message-passing based approach is preferred with its added complexities and new ways of structuring the program.

As the program was supposed to be extendable by the researchers, I had to find a way to create a code as familiar for them as possible. This lead me to develop a Python program (for its extensibility and readability) with critical code written Cython to get a competent performance. Looking backwards, I think that using Cython instead of C++ was an error as the compiler gave me some troubles. If I have to participate in a similar project, I will probably write all critical code directly on C++ and use Cython as an easy way to create the bndings for my Python code.

One of the things I enjoyed the most of this project is how well a functional programming mindset adapts to this particular problem. Being used to the pureness and robustness of FP languages such as Haskell, code can be written more effectively by exporting their features to this project.

COVID-19 has shown me one of the big problems of waterfall-based project management: its low fault tolerance. If I hadn't used an agile development method with more flexibility, I would probably have not been capable reaching this progress to this date. Instead, I would have been waiting for the situation to change to allocate the projects resources.

Last but not least, I've briefly worked with a supercomputer and got a grasp on how tasks are deployed. There is still a lot of work to be done and I should have started to work with it earlier to develop a program more compatible with its arquitecture as I didn't expect to be so different from my development enviroment but... we learn from mistakes.

To sum up, we can say that this project has been a good first approach to scientific computing and high performance software.

Bibliography

[1] Cython.
https://cython.org/

[2] Hypothesis library.
https://hypothesis.readthedocs.io/en/latest/

[3] Posix.
https://web.archive.org/web/20200804110243/https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

[4] Python.
https://www.python.org/

[5] Quickcheck library.
https://hackage.haskell.org/package/QuickCheck

[6] Sagemath.
https://sagemath.org/

[7] Calderon computer.
https://web.archive.org/web/20200823200600/https://www.ce.unican.es/resource/computing/

[8] Cocalc.
https://cocalc.com/

[9] Mpich.
https://www.mpich.org/

[10] Postgresql.
https://www.postgresql.org/

[11] J. P. Costas. Medium constraints on sonar design and performance. Technical report.

[12] M. Díaz. Development of a game about management of software projects. Bachelor's thesis, Universidad de Cantabria, 2019.

[13] D. Everett. Periodic digital sequences with pseudonoise properties. *G.B.C. Journal*, 33(3), 1966.

[14] J. A. Fessler. On transformations of random vectors. Technical Report 314, The University of Michigan, Aug. 1998.

[15] P. G. Flikkema. Spread-spectrum techniques for wireless communication. *IEEE Signal Processing Magazine*, 14(3):26–36, 1997.

[16] E. N. Gilbert. Latin squares which contain no repeated digrams. *SIAM Review*, 7(2):189–10, 04 1965.

[17] R. Gold. Optimal binary sequences for spread spectrum multiplexing (corresp.). *IEEE Transactions on Information Theory*, 13(4):619–621, 1967.

[18] S. W. Golomb and G. Gong. *Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar*. Cambridge University Press, 2005.

[19] P. Herron. *Learning Cython Programming*. Packt Publishing, 2013.

[20] S. W. Hogberg and J. Si. Decimating pseudorandom noise receiver. *IEEE Transactions on Aerospace and Electronic Systems*, 35(1):338–343, 1999.

[21] E. J. Holder, D. Aalfs, B. M. Keel, and M. Dorsett. A comparison of prn and lfm waveforms and processing in terms of the impact on radar resources. In *2001 CIE International Conference on Radar Proceedings (Cat No.01TH8559)*, pages 529–532, 2001.

[22] V. P. Ipatov and B. V. Shebshayevich. Some proposals on signal formats for future gnss air interface. 2010.

[23] J. W. T. James W. Cooley. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[24] A. Leukhin, V. Bezrodnyi and Y. Kozlova. Labs problem and ground state spin glasses system. *EPJ Web Conf.*, 132:02013, 2017.

[25] S. Mertens. Exhaustive search for low-autocorrelation binary sequences. *Journal of Physics A: Mathematical and General*, 29(18), Sep. 1996.

[26] O. Moreno and A. Tirkel. New optimal low correlation sequences for wireless communications. Jun. 2012.

[27] T. Packebusch and S. Mertens. Low autocorrelation binary sequences. *Journal of Physics A: Mathematical and Theoretical*, 49(16):165001, mar 2016.

[28] I. Poole. Direct sequence spread spectrum: the basics. https://web.archive.org/web/20200724125605/https://www.electronics-notes.com/articles/radio/dsss/what-is-direct-sequence-spread-spectrum.php

[29] C. E. Shannon. Communication in the presence of noise. *Proceedings of the IEEE*, 86(2):447–457, 1998.

[30] I. Sommerville. *Software Engineering*. Addison-Wesley, 8 edition, 2007.

[31] A. Z. Tirkel, C. F. Osborne and T. E. Hall. Steganography - applications of coding theory. Jul. 1997.

[32] N. Zierler. Legendre sequences. Technical report, Massachusetts Institute of Technology, May. 1958.