Facultad de Ciencias

Secuencias binarias y sus aplicaciones

Binary sequences and their applications

Trabajo de fin de grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Juan Toca Mateo
Codirector: Domingo Gómez Pérez
Codirector: Ana Isabel Gómez Pérez

septiembre de 2020

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 con 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 de 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 través 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.

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 cryptography, wireless communications and digital watermarks. Commercial applications such as GPS or military uses such as Radar have been developed and improved thanks to the search of these sequences and the analysis of their autocorrelation function and the correlation between members of the same family.

In this project, we focus on a technique for the algebraic construction of sequences where a sequence of shifts and a sequence with good properties generate a new sequence. The aim is the exhaustive search of longer sequences then the ones already present in the literature. To do so, a software intended to provide support to a designer has been developed to assist in the search of said sequences and check their properties, which can be deployed in a supercomputer. The applications of these sequences are, among others, radars and location systems with higher spatial resolution.

Contents

1 Introduction

Advances in wireless communication and signal proccessing have drastically changed the capabilities of transmitting information. Several wireless networks can share the same channel amongst several users thanks to technologies like CDMA (Coding Division Multiplexing Access). Similarly, GPS [12] satellites implementing direct-sequence spread spectrum(DSSS) can broadcast their information to provide its service. Radar technologies recovers the echo of a sent signal to calculate the distance to a target by comparing the arrival time with the emitted time. For all these applications, sequences with good properties of crosscorrelation and autocorrelation function are required. To keep things simple and just for illustration purposes, we are going to focus in just two technologies through this document: spread-spectrum and radar.

An example for the former is the direct-sequence spread spectrum technology[18][5], which is designed to share a given frequency band between multiple transmissions. To do so, DSSS employs a set of spreading sequences. A spreading sequence from this set is pre-shared between emisor and receptor, that the emisor will use to modulate its transmitted signals. Then, the receiver demodulates the signal and correlates the received sequence with the pre-shared sequence. If it finds a peak in this correlation, the received sequence is processed. Otherwise, it can not be detected. The sequences should mimic the properties of white noise to avoid an eavesdropper to be able to recover the communications, thus a set of sequences with low peak crosscorrelations is denoted as pseudonoise (PN). This method limits greatly the bandwidth of each individual transmission but improves security to jamming and interception. It is suitable to be used in applications that require with low data rate but with several emitters. In GPS, satellites broadcast their current position and timestamp and from that information the distance to the satellite can be computed. Combining the information from several satellites, the position of the receiver can be triangulated. Another example of location system would be UWB (Ultra Wide Band) based technologies.

A radar (radio detection and ranging) uses a radio-frequency electromagnetic signal reflected from a target to determine properties such as position, speed, etc. PRN-based radars[10][11] 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 peak which indicates the 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 pseudonoise sequences, its properties and practical applications.

1.1 Binary sequences

In computer science, binary symbols are normally defined by 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 as it is a natural way to represent a binary state.

However, the discussed topic takes a general mathematical approach in this work, where there is not a direct equivalence to a bit. In fact for designing sequences in k symbols, each symbol correspond to a sequence whose elements are k roots of unity in the complex numbers. With this in mind and to avoid needlessly redefining the equations found in the literature, we will stick to sequences composed by the elements 1 and -1, taken from the ring of integers $\mathbb{Z}$. Note that multiplying two sequences element by element is equivalent to the binary XOR operation.

1.2 Correlation function

According to Golomb and Gong [8], the correlation function measures how similar two phenomena are. If properly normalized, the function ranges from +1 (identical) to -1 (opposite); 0 meaning statistical independent phenomena. For sequences represented as vectors, the correlation can be conceived as the normalized dot product between those two vectors. When both sequences have the same finite length 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.

The unnormalized correlation has computational advantages over the normalized version, as can be computed using only sums and multiplications. For this reason, we recall the definition.

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 elementwise 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.

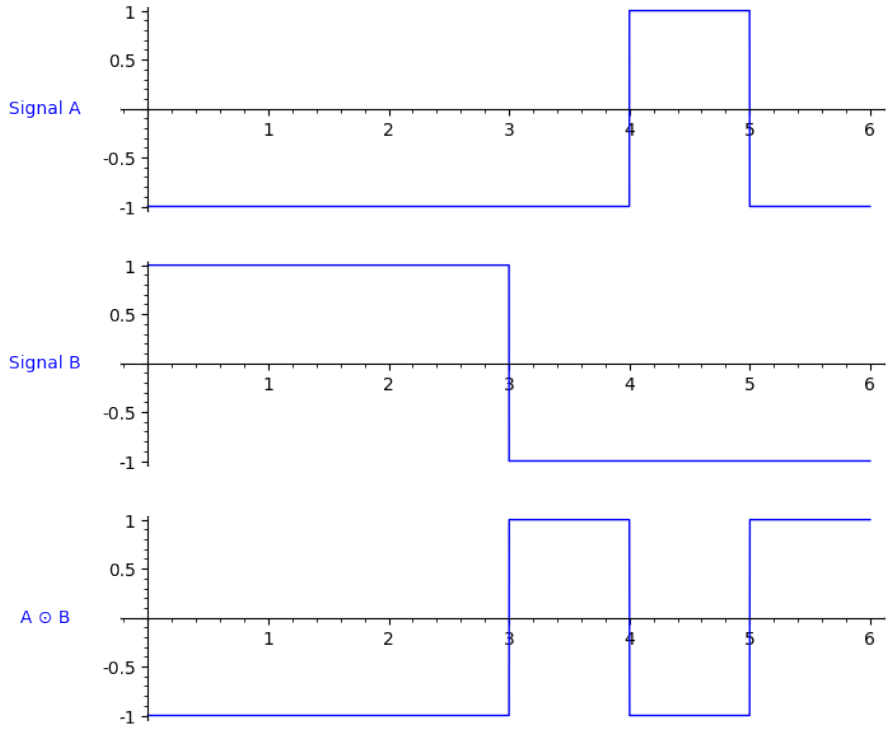

Figure 1: A graphical representation of two vectors an their pointwise product with an unnormalized correlation between them of -2 (-1 -1 -1 +1 -1 +1)

1.2.1 Autocorrelation function

Going on with the lecture of Golomb and Gong [8], 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:*

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

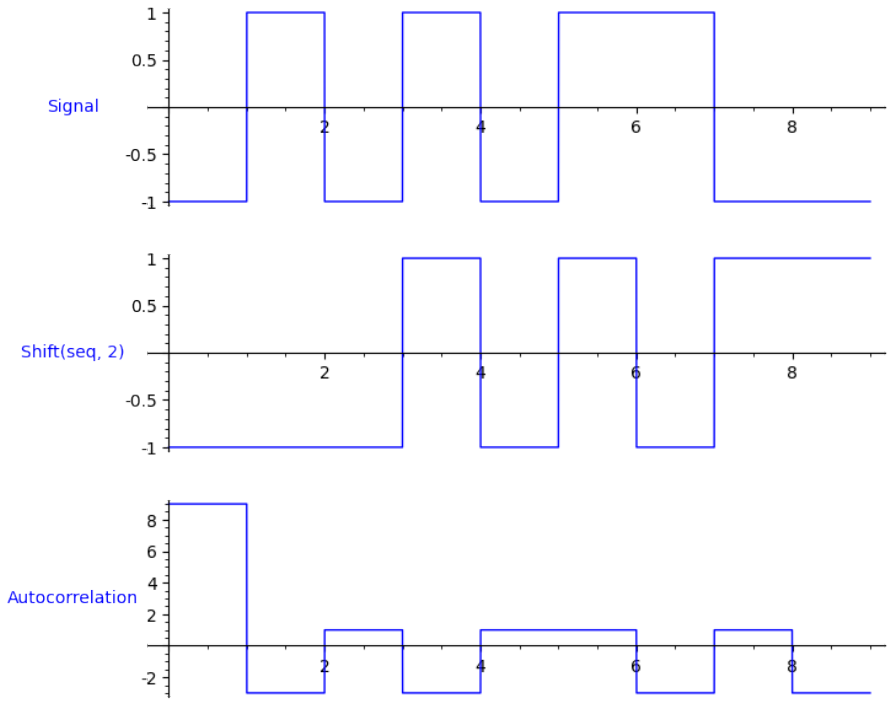

Figure 2: A graphical representation of the autocorrelation of a sequence with a shifted version of itself.

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:*

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

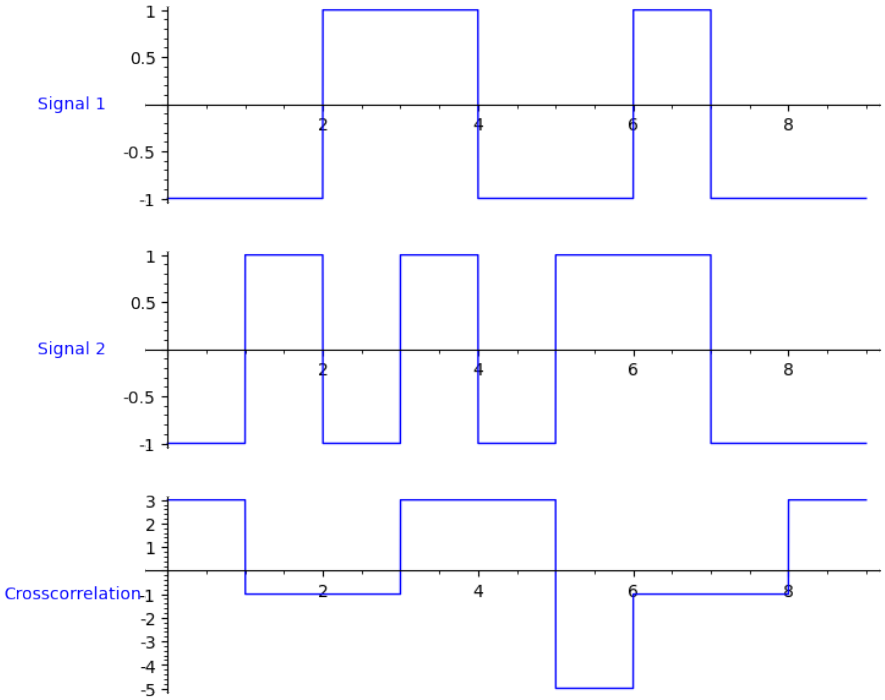

Figure 3: A graphical representation of the crosscorrelation between two sequences.

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 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[4].

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[21], 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 $\tau \neq 0$ [3].*

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 conjectured that such sequence do not exist apart from length four. That is why pseudonoise are commonly employed in practical applications. Then a threshold is defined so that the system will not mistake intermediate values with the peak in the autocorrelation.

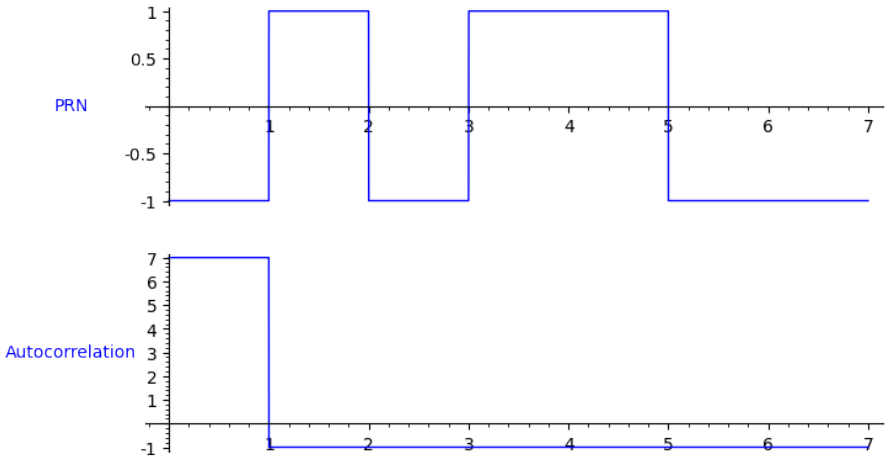

Figure 4: A pseudorandom noise sequence and its autocorrelation function. Notice that this pseudonoise sequence isn't perfect noise.

2 Pseudonoise sequence generation

As introduced previously, pseudonoise sequences are useful in technologies that require sequences with properties similar to those of white noise. In this chapter, some state-of-the-art techniques in pseudonoise sequence generation will be introduced.

2.1 Maximum Length Sequence(m-sequences)

M-sequences are a binary pseudonoise construction that was initially conceived using linear feedback shift registers (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. (See Appendix for formal details) The following properties hold for a m-sequences.

- A m-sequence will always be of length of the form $2^m - 1$ where m is an arbitrary natural number.
- A m-sequence sequence will always have an autocorrelation function such as all the components will be -1 except when $\tau = 0$

The complexity of the algorithm to construct an m-sequence of length n is $O(n)$. However, the problem is that sequences of arbitrary length might be needed in some applications.

As it will be discussed in a following chapter, the complexity of computing the autocorrelation with the Fourier Transform approach is $O(n \cdot log(n))$ so using longer length than the strictly needed has a negative direct practical consequences.

These sequences by themselves might not a be huge deal because they do not define a way to build families of sequences with low cross correlation, but they are the building blocks for other constructions, such as the Gold Codes that are used in GPS and CDMA.

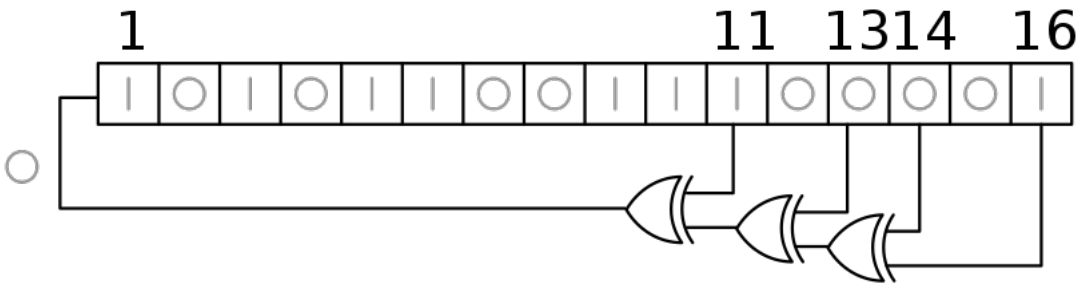

Figure 5: A diagram of a LFSR. The next symbol to be set in the register 1 is computed based on the previous state. The previous registers are moved one to the right, discarding the last register.

2.2 Gold Codes

Gold codes[7] are a family of sequences derived from m-sequences, with good properties to several applications such as wireless communication and geolocalization.

A Gold Code generator gets two m-sequences $S1$ and $S2$ of length $2^n - 1$ (n is a natural number), that fulfill that the crosscorrelation function CC (See Equation (7)) is

$$max|CC(S1, S2)| \leq 2^{\frac{n+2}{2}} \tag{9}$$

The family called Gold codes is defined by the sequence

$$\{S_1, S_2, S_1 \odot S_2, S_1 \odot shift(S_2, 1), \ldots, S_1 \odot shift(S_2, 2^n - 2)\}.$$

where $\odot$ represents the element-wise product of vectors.

Property 2.2.1. *Given any two sequences, $S1$ and $S2$, from a Gold family of sequences of length $2^n - 1$, the crosscorrelation function CC defined in Equation (7) satisfies*

$$max|CC(S1, S2)| \leq \begin{cases} 2^{\frac{n+2}{2}} + 1 & \text{if n is even} \\ 2^{\frac{n+1}{2}} + 1 & \text{if n is odd} \end{cases} \tag{10}$$

This property means that the crosscorrelation between any given pair of sequences from a Gold family is low enough to differentiate them. In practice, several devices can transmit over the same medium, then the receiver can recover the information only if it knows the corresponding Gold code. In other case it resembles noise.

Gold also proposed a way to find $S1$ and $S2$ by decimation..

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

$$S[q]_i = S_{((q \cdot i) \bmod n)}. \tag{11}$$

Property 2.2.2. *Given a m-sequence S of length $2^n - 1$, where n is odd and coprime with an integer k, the sequence pair $(S, S[2^k + 1])$ satisfies Equation (9).*

```python
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 implementation of a Gold family generator family in Python

Notice that this construction has the same problem as m-sequences, which is that only few choices of sequence length are possible . It may not be suitable for some applications.

2.3 Legendre sequences

Legendre sequences, as explained in Zierler [24], are binary sequences defined through quadratic residues. A quadratic residue modulo p is an integer number x between 0 and $p-1$ such that an integer y exists satisfying $x = y^2 \bmod p$.

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

$$LSy(n,p) = \begin{cases} 1 & \text{if n is a quadratic residue mod p} \\ -1 & \text{otherwise} \end{cases} \tag{12}$$

The Legendre sequence can be defined as:

$$LSs(p)_i = LSy(i-1,p) \text{ where } 0 \leq i \leq p-1 \tag{13}$$

```
def quadratic_residues(p):
    return set([i**2 %p for i in range(p)])

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.

Legendre sequences with length satisfying $p = 3 \bmod 4$ have flat autocorrelation, i.e. the value of the autocorrelation for any nonzero shift is -1. There is a generalization of this property by Zierler [24] but it requires the sequence to be non-binary.

As p is the length of the generated sequence, the distribution of Legendre sequences is related to the Prime Number Theorem[19]. This means that Legendre sequences have more possible lengths than in m-sequences or other constructions based on them. However, Legendre sequences have the drawback that there exists only one per sequence length.

2.4 Composition method

2.4.1 Algorithm

The composition method was introduced by Tirkel et al. [23], initially called prime arrays, uses a base sequence and a sequence of shifts to create a matrix whose columns are shifts of the base sequence 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:*

$$0 \leq T_i < n \tag{14}$$

$$gcd(n,m) = 1 \tag{15}$$

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

$$CM(S,T) = \begin{bmatrix} shift(S,T_0)_0 & shift(S,T_1)_0 & \dots & shift(S,T_{m-1})_0 \\ shift(S,T_0)_1 & shift(S,T_1)_1 & & \\ \vdots & & \ddots & \\ shift(S,T_0)_{n-1} & & & shift(S,T_{m-1})_{n-1} \end{bmatrix} \tag{16}$$

Definition 2.4.2 (Composite sequence)**.** *Given a base sequence S of length n, a shift sequence T of length m that satisfy Equations (14) and (15) and the composite matrix defined at Equation (16), the composite sequence is defined as:*

$$CS(S,T)_i = CM(S,T)_{(i \bmod m),(i \bmod n)} \tag{17}$$

The fact that the sequence has length nm and the correlation function can be deduce from the matrix is a consequence of the Chinese Remainder Theorem. This is informally called the diagonal method, because it unfolds the array following the diagonal and using the wrap around provided by the modulo operation.

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

Figure 8: Example of a computation of the composition method. For clarity purposes, the example uses a non binary base sequence.

Definition 2.4.3 (Composite matrix correlation)**.** *Given two composite matrices M and R with n rows and m columns, its correlation is defined as*

$$C(M,R) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} M_{j,i} R_{j,i}. \tag{18}$$

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

$$shift(M,\tau)_{i,j} = M_{(i-\tau \bmod m),(j-\tau \bmod n)} \tag{19}$$

This means that the following relation can be established:

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.*

$$shift(CS(S,T),\tau)_i = shift(CM(S,T),\tau)_{(i \bmod m),(i \bmod n)} \tag{20}$$

The previous corollary has an interesting implication for computing the autocorrelation function.

Corollary 2.4.0.2. *Given a composite matrix M of n rows and m columns, its autocorrelation function is defined as:*

$$A(M)_\tau = C(M, shift(M,\tau)) = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} M_{j,i}M_{(j-\tau \bmod m),(i-\tau \bmod n)} = \sum_{i=0}^{m-1}\sum_{j=0}^{n-1} shift(S,T_i)_j shift(S,T_{(i-\tau \bmod n)})_{(j-\tau \bmod m)}. \tag{21}$$

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:*

$$\sum_{j=0}^{n-1} shift(S,T_i)_j shift(S,T_{(i-\tau \bmod n)})_{(j-\tau \bmod m)} = A(S)_{|((T_j-\tau) \bmod m)-T_{(j+\tau) \bmod n}|} \tag{22}$$

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

This property is useful as a fast method for computing the autocorrelation function for composed sequences and this will be implemented for efficiency.

2.4.2 Costas arrays

Costas arrays, discovered independently by John P. Costas[1] and E.N. Gilbert [6] in 1965, are a family of sequences highly used in radar and sonar applications. The equivalent definitions from both works has been included as both will be useful for different purposes.

Definition 2.4.5 (Costas array by Costas [1])**.** *An Square matrix of size $n\times n$ is filled with 0s and 1s such that there is a single 1 in each row and column. Thus if the displacement vector is calculated by substracting all pairs of positions of the 1s, then the results are unique.*

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:

$$[3, 1, 0, 2]$$

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

$$0 \leq S_i < n \tag{24}$$

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

$$S_i - S_{i+r} \not\equiv S_j - S_{j+r} \bmod n \tag{25}$$

Definition 2.4.7 (Costas array by Gilbert [6])**.** *Given a sequence S satisfying Equation (24), it is called a Costas array if, for any given r value, it satisfies Equation (25).*

This compact representation can be feeded into the composition method as a sequence of shifts generating interesting new sequences[16].

Several construction methods for Costas arrays have been proposed. For sake of simplicity, just the Welch construction will be introduced following the work of Gilbert[6].

Given a prime number p and a primitive root g of p such as all the powers of g in $[0, p-1]$ modulo p are different from each other. A Costas array S can be constructed as follows:

$$S_i \equiv g^i \bmod p, \quad 0 \leq i < p-1 \tag{26}$$

This construction can generate sequences of length $p-1$ for a given g. 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(see Schroeder [20]).

3 Exhaustive search of binary sequences

3.1 Previous work

As shown in the previous chapter, algebraic constructions for sequences with low off-peak autocorrelation have huge constraints on the length of the generated sequences. To overcome this limitation, exhaustive searches through all possible candidates for a given length have been conducted in the past. The size of the search space in this case is $O(2^n)$, where n is the length of the binary sequence, which make their results very costly in terms of computational complexity.

For a similar case there has been studies for finding sequences with low aperiodic autocorrelation, whereas fewer works has been done in the periodic autocorrelation due to the existance of the optimal algebraic constructions, which have been sufficient for past practical purposes. In the next paragraphs we review the literature on the aperiodic autocorrelation.

First of all, aperiodic autocorrelation and the energy of a sequence are defined as follows.

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

$$A'_\tau(S) = \sum_{i=0}^{n-\tau-1} s_i s_{i+\tau} \tag{27}$$

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

$$E(S) = \sum_{k=1}^{n-1} A'_k(S)^2 \tag{28}$$

The idea is finding the sequence S belonging to a given set of possible sequences that gives the minimum posible energy through the branch and bound with objetive function

$$E_{min} = \min_{subset} \sum_{k=1}^{n-1} A'^2_k. \tag{29}$$

An improvement for this method was proposed by Mertens [15] in which he provided an algorithm with complexity $O(1.85^n)$. In his work, he applied a branch and bound approach that rules out equivalent sequences and uses an heuristic based on how complementing a single symbol of the sequence affects the autocorrelation.

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

When a symbol of the original sequence is complemented, the components of the autocorrelation can be decremented by, at most, 2. Based on that, a relaxation of E_{min} can be proposed:

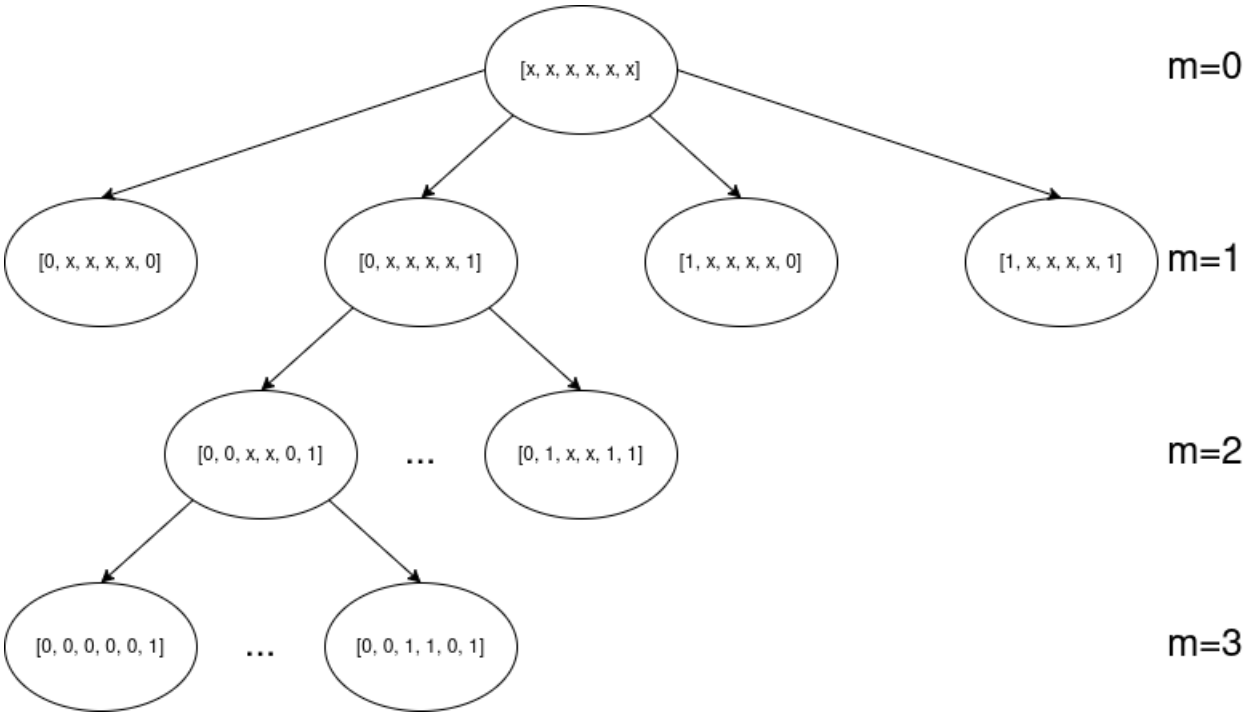

Figure 11: An example of the branching used in Mertens [15] with $n = 6$ and where x represent unfixed values.

$$E_b = \sum_{k=1}^{n-1} max\{b_k, (|A'_k| - 2f_k)^2\} \leq E_{min} \tag{30}$$

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:

$$f_k = \begin{cases} 0 & k \geq N - m \\ 2(n-m-k) & n/2 \leq k < n-m \\ n-2m & k < n/2 \end{cases} \tag{31}$$

where n is the size of the sequence to search and m the number of fixed elements at the ends 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 algorithm, Mertens succesfully computed sequences up to length $n = 48$.

This work was further improved by Packebusch and Mertens [17] combining bounds provided by different authors (Prestwich and Wiggenbrock) to create a new bound and lowering the complexity to $O(1.729^n)$ obtaining sequences of length $n = 66$. This record was broken by Leukhin et al. [14] by computing sequences up to length $n = 85$.

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 $p \cdot m = n$. This clearly means that the complexity grows slower than in previous works. In fact, the autocorrelation function can be optimized for sequences generated through the composition method as shown in Chapter 6.

The problem is that the possible lengths for the sequences are limited as m and p are required to be coprime. 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 does not explore all possible permutations and it does not ensure to find a pseudonoise sequence if it exists. To sum up, this method is a good way to construct useful sequences but should not be used to prove the non existence of pseudonoise sequences for a given length.

Given a base sequence of length p and shift sequences of length m, 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, p^m permutations. However, there are some relations between the different shift sequences that let us narrow the search space.

For example, taking into account the property:

Property 3.2.1. *Given a shift sequence S, a base sequence B and an arbitrary constant k, it exists a shift sequence S' and a constant a that fulfills:*

$$S'_0 = k \tag{32}$$

$$shift(CS(B, S), a) = CS(B, S') \tag{33}$$

It can be said that all the posible shift sequences of length m when applied in the composition method generate sequences equivalent to at least one in the subset of sequences of length m where the first component is k. This reduces the search space to $p^{(m-1)}$.

Other optimization arises from the form of the shift sequences. In general, if the symbols are repeated often, they tend 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 $shift$ function defined at Equation (3), the Hamming autocorrelation is defined as:*

$$HA(S)_\tau = \sum_{\tau=0}^{n-1} HAComponent(S_\tau, shift(S, \tau)_\tau) \tag{34}$$

where $HAComponent$ is defined as:

$$HAComponent(c1, c2) = \begin{cases} 1 & c1 = c2 \\ 0 & \text{otherwise} \end{cases} \tag{35}$$

For our branch and bound algorithm, it is 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

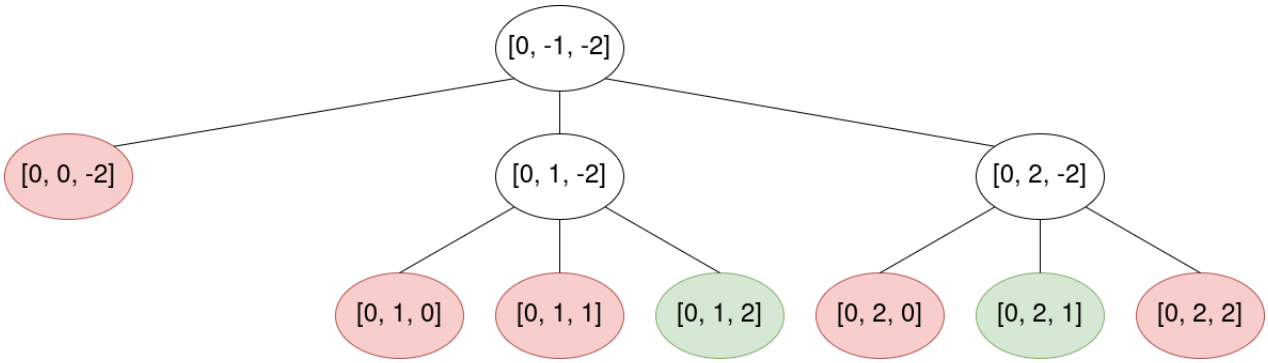

Figure 12: An example of the branch and bound algorithm with a threshold for hamming autocorrelation of 1 and a base sequence of length 3. Red nodes represent prunes and green ones final nodes in which the autocorrelation is computed and checked. Negative values represent those that haven't been initialized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

to have a higher hamming autocorrelation than the threshold.

Some properties of the algorithm 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.

4 Application development

Before starting to code an application, the process has to be planned beforehand to guarantee that the project will be successful. Every different software project has different characteristics that influence the decision on the methodology that should be followed. 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, we present the steps followed to design, construct and test an application to assist a designer to search for new binary sequences under certain requirements and to check their properties. Most of the information is mostly based on Sommerville [22] and highly inspired in Díaz [2].

4.1 Overall description

In this section we describe the general idea behind the software project and the requirements that needs to fulfill.

4.1.1 Project description

The main purpose of this project is to develop a software application to assist in the research of binary sequences with good correlation properties. The expected end-users are researchers but also designers for wireless communications systems such as radar, Wifi or GPS.

The primary challenge is to understand the problem domain to create a software as useful as possible (this first phase has been already explained in the previous chapters). Then, it is necessary to create an extensible program that assist in the search for new sequences to add to the existing scientific literature. Furthermore, a user has to be able to manage the sequences and to check and compare their properties. As this process is computationally expensive, the software is expected to offer support for parallel computing.

A summary of the functionalities that are expected from this software is provided.

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

4.1.2 User classes and characteristics

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

Researchers that will manage the search for new sequences, being capable of starting computations and modifying the database. These researchers are highly specialized personnel and are expected to be familiar with 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 performance of the software according to the specifications 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.

In the real world, both roles may overlap in the same person.

4.1.3 Operating Environment

The computation module is expected to run in highly parallelized systems such as super-computers. Most researchers work with this kind of environments, so the software would be less competitive if it would not take full advantage of the capabilities of these systems.

Taking this into account, the most common software installed in those environments should be targeted. This leads us to choose a Linux environment as most supercomputers run it. We have chosen a Python interpreter as it is a popular language in the scientific community.

4.2 Software requirements

In this section the software requirements of our project will be introduced. Interviews and document analysis have been the main source to define requirements. Later they have been validated with the intended users.

4.2.1 Functional requirements

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

Identifier	Description
FR01	The user must be capable to set the base sequence to use for the search
FR02	The user must be capable to set the size of the shift sequence to use for the search
FR03	The user must be capable of setting the maximum autocorrelation they are interested in
FR04	The user must be capable to store, recover and manage sequences.
FR05	The user must be capable to extract partial results while the computation is still running
FR06	The system administrator must be capable of changing the resources assigned to the program
FR07	The system must provide an interface that shows the progress of the computation (No need for low latency as it would conflict with NFR01)
FR08	Software must run without supervision after parameters are established.
FR09	The parameters of the load balancer must be editable by the system administrator in order to adapt to different hardware.
FR09	An system administrator must be capable of managing access privileges to the database and running computations
FR10	The system must provide a way to queue concurrent searches

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	Speed is a top priority.	Very high
NFR02	Compability	It should be made as compatible as posible with different supercomputers.	Medium
NFR03	Arquitecture	The program must take full advantage of the capabilities of supercomputers, in particular high parallelization of the system.	High
NFR04	Robustness	The program must handle errors such as data corruption, miscalculations or precision errors to avoid incorrect results.	Very high
NFR05	Robustness	The program cannot stop unexpectly and must provide a way to recover from said crashes	Medium
NFR06	Extensibility	Parts of the software should be reusable to suit each research project needs.	Medium
NFR07	Robustness	The persistence layer must be robust enough to avoid data losses.	High

Figure 14: Non-functional requirements

4.2.3 User interface requirements

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

First of all, note that the users are expected to be experienced in the use of computers, so a complex UI should not 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 environment with no graphical desktop is preferred.

For this reason, a command-line interface (CLI) is preferred over other alternatives as it requires fewer system resources, allows scripting and commands can be entered more rapidly as text. The inferface is divided in two 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[1] 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 the capability 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).

The previous discussion were translated to the following implementation of the command-line interface. 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
```

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

```
        the square root of (-l*-s)

-v : verbose mode
```

The verbose mode can log which tasks have been assigned and saves the time stamp as well. Finally it tracks the end of child processes and their idle time.

```
$ 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 standard 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 process the data of the program (one word message, well defined columns, etc.).

Verbosity is completly optional and does not impact performance when inactive. Through the experiments, this option is useful to find a good parameter configuration.

4.3 Agile development

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

In our case, this project is going to avoid a waterfall model. In this case agile development techniques adapted better to our problem for several reasons:

- The client is an active part of the project. This means that a lot of feedback can be obtained during development and issues can be fixed earlier in the development cycle.
- 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 would not be useful.

4.3.1 Role definition

Two different roles have been defined:

- Developer, which will design, program, verify and manage the software project.
- Client, which will provide the requirements for the project, as well as validating each iteration.

4.3.2 Iterations

In this subsection we will discuss how the development iterations went. A convention is that when we say a C function, we are referring to a Cython code without CPython code, in other words, functions that do not call the Python interpreter. The reader can refer to Chapter 5 in case more information on the technologies mentioned in this section is needed.

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 two first functions are not part of the actual exhaustive search algorithm, but will be useful if properties have to be checked when retrieving the results from the database.

The test process consisted in checking that the python wrapper provided the same results as a naive implementation of the algorithm based in the convolution theorem (see Section 6.1.2). 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 presented errors related with fused types. Finally 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 algorithm

In this iteration we focused on a single threaded C implementation of the branch and bound algorithm. This function receives a threshold of the maximum autocorrelation the user is interested in, the maximum Hamming autocorrelation for the prune part of the algorithm 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 can 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 check that all the returned sequences satisfied the specified maximum autocorrelation.

Iteration 3: Parallelism

In this iteration, the main focus was to adapt the branch and bound algorithm to a parallel environment. To do that, the algorithm 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. Properly based test were not written in this iteration as the code was not mature enough.

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 its 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.

4.4 Verification

One of the most important parts of software development is verifying the software. In other words, checking that the semantics of the program built are the same as the intended ones. Testing since the early stages of a project is mandatory if a quality product and an efficient development process is to be accomplish. A lot of time can be invested building a program to realize that it does not work or a fix might generate side effects that break other parts of the program and so on.

4.4.1 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 library "QuickCheck"[2], 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 is similar to the test automation explained by Sommerville [22] in Chapter 23, being the main difference that an oracle

[2] https://hackage.haskell.org/package/QuickCheck

that predicts the value is not 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[3] 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 chosen 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.

```
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

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.

4.5 Validation

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

- In early iterations, the validation process might not be as important as the verification process. This reason is the core functionality of the program is an algorithm expressed in a technical manner with little margin to misinterpretations.

[3] https://hypothesis.readthedocs.io/en/latest/

- In later iterations, the validation process gains weight in respect of the verification process as we dive into the UI design. In this case, there is more room for misunderstandings between client and developer so we must take this process into account.

Fortunately, 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 misunderstandings, desired changes, etc.

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[1] is a Python mathematical suite used in research projects as an enviroment for prototyping algorithms 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[2]. 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[3] 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 is the language the researchers are the most familiar with. This helps the project to be extended according to the user needs when it is finished.
- It is a language widely used. This means that support can be expected 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, their expertise in the language let's them explore more new concepts in the same time such as MPI or the whole domain knowledge needed for this project.

[1] https://sagemath.org/
[2] https://cocalc.com/
[3] https://python.org/

5.3 Cython

Python as a language comes in several implementations. CPython as the reference implementation has its flaws, mainly its performance issues. As the software being developed has performance constraints, using just the reference implementation is not an option.

Fortunatly, there are alternative implementations such as Jython, IronPython, etc. In our case, Cython[4][9] 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[5] or GSL[6].

One might think that, if vector operations are so efficient in CPython, it would be simpler to just use Numpy methods to implement our algorithms (which was indeed done in the general autocorrelation function with the algorithm 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 is 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

[4] https://cython.org/
[5] https://numpy.org/
[6] https://gnu.org/software/gsl/

the overhead of type checking and reference counting.

5.4 PostgreSQL

PostgreSQL[7] 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 its SQL code, it gives a lot of flexibility in how our application can be designed.

[7] https://postgresql.org/

6 Computational Analysis

In this chapter, computational aspects of the application are discussed.

6.1 Autocorrelation function implementation

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 compute all possible shifts of the sequence and then correlate them with the base sequence as shown in Figure 18. This algorithm is simple to implement and follows from the mathematical definition, however it 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 length of the sequence. The overhead due to the building of the shifted sequence weights on the performance. Some overhead could be avoided by using slice of the sequence but it will not affect the complexity.

```
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

6.1.2 Circular convolution theorem

We explore the properties of the autocorrelation function. The convolution theorem[8] gives a direct relation between between the correlation function and the Discrete Fourier Transform (DFT)

Theorem 6.1.1. *Given a sequence S and its DFT, then*

$$A(S) = DFT^{-1}[DFT\{S\} \cdot DFT\{S\}^*] \tag{36}$$

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

To compute the DFT and its inverse, the Fast Fourier Transform[13] algorithm can be implemented which would drop the complexity to $O(n \log n)$. However, now the types of the data change from integer in the naive approach to complex numbers. Then according to property 1.2.1, it will always return the same type as the original sequence. This means that the operations are more computationally expensive due to operating with complex numbers instead of integers. An example is shown in Figure 19.

```
cpdef autocorrelation(signal):

  if len(signal) == 0:
    return np.array([], dtype=np.int32)
  ft = np.fft.fft(signal)
  S = np.conj(ft)*ft

  # boilerplate
  warnings.filterwarnings(action="ignore", category=np.ComplexWarning)

  return np.round(np.fft.ifft(S)).astype(np.int32)
```

Figure 19: An example implementation of the naive autocorrelation

6.1.3 Composition method implementation

The previous discussion does not take into account the special structure of sequences generated by the composition method. It is possible to take advantage of some peculiarities to improve on the computation of the correlation.

Taking advantage of Property 2.4.1, a faster algorithm can be designed. First of all, the complexity function of this new algorithm depends on the length of the shift sequence m and the length of the base sequence p with a complexity of $O(m^2p)$. This means that when $m < \log(pm)$ this algorithm has a better complexity than the general approach.

An example implementation of this algorithm is shown in Figure 20.

```
cdef int * composite_with_autocorrelation( int* autocorrelation
                                         , int l_signal
                                         , int* shifts
                                         , int l_shifts):
    cdef int output_size = l_signal * l_shifts
    cdef int* output = <int *> malloc(output_size*sizeof(int))
    cdef int x, y, affected_column, current_shift, final_shift, positive_difference
    cdef int constant_offset = l_signal
    for x in range(output_size):
      output[x] = 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[x] = output[x] + autocorrelation_with_constant[final_shift]
    return output
```

Figure 20: The Cython implementation of the autocorrelation for the composition method.

In addition, this method is more cache friendly too as the data source of the function is smaller and it does not need to use complex number operations in binary sequences. The biggest improvement in respect of the Fourier Transform approach is that to check for the partial autocorrelation, it is possible to do it without having to complete the computation of the whole autocorrelation function as shown in Figure 21. This will be important for the brach and bound algorithm for the exhaustive search.

```
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
  cdef int constant_offset = l_signal
  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 21: The Cython implementation of the autocorrelation for the composition method that support partial computations.

6.2 Parallelism model

For the parallelism of the project, it was decided to work with MPI (Message Passing Interface). MPI is a library specification for message-passing, proposed as a de facto standard for parallel computing implementation. As the software will be deployed in a system with an interconnection network, a shared-memory parallelism would lead to a high latency in the assignation of tasks reducing performance. There are several implementations of MPI, many of which are open-source or in the public domain.

From the available implementations, a pure Python one (MPI4PY[1]) was chosen based on the easiness of the development and its compatibility with different implementations of the MPI standard.

For the purpose of this project, MPICH[2] is used as the reference MPI implementation as the bindings of MPI4PY support it and it is the implementation used in the cluster we are working with[3].

In our particular problem, a set of tasks was defined to be distributed between the different nodes. This tasks are subtrees from the search space with a given height that defines the size of the task (see Figure 22 for a diagram and Figure 23 for the implementation). Due to different sizes of the tasks, a static scheduler would not be efficient.

The preliminary version of the software implemented this model. However, if a branch is not pruned when assigning the tasks, there will still be an exponential number of tasks given by p^{m-t} where p is the length of the base sequence,m is the length of the shift sequence and t is the task size. If t is close to m, the task balancer has to deal with a few remainding task to asig, however it will not impact drastically the performance.

A second version of the algorithm applies branch and bound to generate only candidates with good Hamming properties. This rules out all branches that would led to a bad resulting autocorrelation and

[1] https://github.com/mpi4py/mpi4py
[2] https://www.mpich.org/
[3] https://web.archive.org/web/20200823200600/https://www.ce.unican.es/resource/computing/

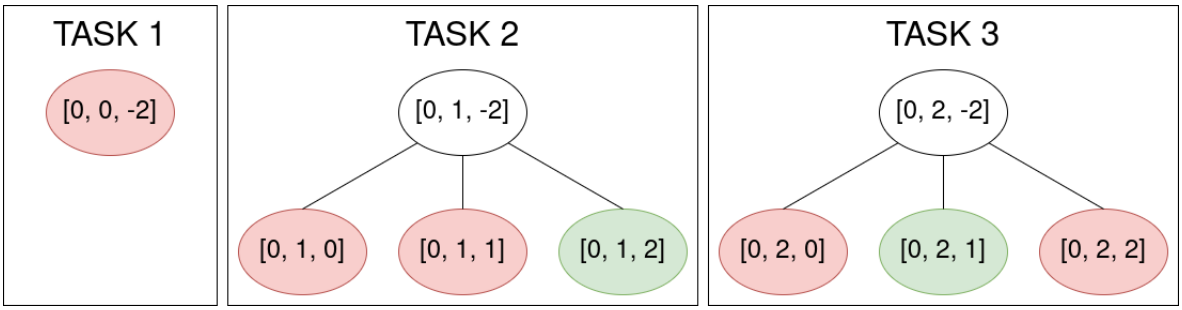

Figure 22: An example distribution of tasks for the example at Figure 12.

```
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 23: A Cython implementation of the branch and bound algorithm. Notice the amount of extra boilerplate code to achive C performance.

would not generate any useful sequences. The code implementation is shown in Figures 24 and 25.

```
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 were 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 24: 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 25: A Python implementation of an slave proccess

7 Conclusions and future work

In general, the objectives for this project were accomplished. A first version of the software has been deployed in a supercomputer to be ready to use for research. The main objective for this project was to learn how a project of scientific computation must be developed in term of needs, requirements and peculiarities.

First, the most difficult task has been to understand the problem domain. This software, in contrast with other consulting projects, requires a high level of knowledge in the state of the art of scientific research to be able to grasp the semantics of the program. This adds a huge challenge to the software engineer to ensure the correctness of the results. I can affirm with confidence that two thirds of the time was spent in learning all the concepts that were going to be used later to gather all the software requisites and semantics.

Apart from that, 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 in the future, 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 bindings for my Python code.

From a personal point of view, 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. On the other hand, I have briefly worked with a supercomputer and got a grasp on how tasks are deployed. As science-oriented development is one of the topics I am very interested in, this bachelor thesis suits in my future profesional career.

Even though we have a working prototype, there is still a lot of work to be done. First of all, the persistency part of the program can be improved to acommodate advanced queries or a way to store sequence properties. Nowadays, the search results are dumped in plain text files. In addition, the program has to be extended to cope with the diversity of base sequences of using different kinds from Legendre sequences. Additionally, there is room for improvement in the MPI usage to allow buffer assign to the task to reduce the interconnection network delay. Further work has to be done for testing the program in a production environment.

To contribute to the research on the area on binary sequences, we need to run the code for a long time to find candidate sequences, as it have been seen in the introduction the length of the sequences is of the order of 85.

As we introduced Gold codes in the report, a reader could ask why we didn't cover families of sequences in our search as they are needed to have a practical use in some applications such as DSSS. The reason behind this is that the construction of these families depend directly in the search of sequences with good autocorrelation properties. If we extend the set of known sequences with a low-peak autocorrelation, it can be used in future work to extend the literature in families of sequences.

Last but not least, the UI is still open to improvement. 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).After deployment a permissions system for the database must be implemented to allow multiple users. As it is running in an isolated environment to perform the computation, no external user has access to the data unless it is uploaded to an external server through the VPN.

Bibliography

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

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

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

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

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

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

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

[8] S. W. Golomb and G. Gong. *Signal design for good correlation: for wireless communication, cryptography, and Radar*, pages 1–9. Cambridge University Press, 2005.

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

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

[11] 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.

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

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

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

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

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

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

[18] 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

[19] M. Schroeder. *Number theory in science and communication with applications in cryptography, physics, digital information, computing, and self-similarity*, chapter 5. Springer, 2009.

[20] M. Schroeder. *Number theory in science and communication with applications in cryptography, physics, digital information, computing, and self-similarity*, page 13. Springer, 2009.

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

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

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

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

.1 An overview on finite fields

The particular type of LFSR used in m-sequences can be simulated with extensions of binary Finite Fields. A formal definition is that m-sequences are a trace mapping from an extension field $GF(p^d)$ to a base field $GF(p)$. The base field consists of the integers modulo p with p prime. In this work, we will restrict ourselves to binary sequences $p = 2$, that have attracted the most interest. As shown in Golomb and Gong [8], a sequence $a(i)$, where i is an indexing integer, is formed by taking the elements of the extension field in consecutive increasing powers of α, a primitive element, and computing the trace to the base field.

Definition .1.1 (Finite Fields)**.** *Given $E/GF(2)$, α a primitive element of E and S the resulting sequence:*

$$S_i = Tr(\alpha^i) \tag{37}$$

The base field consists of the integers modulo p with p prime. If β is an element of $GF(p^d)$ then

$$Tr^d(\beta) = \sum_{i=0}^{d-1} \beta^{p^i}. \tag{38}$$

```
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 26: An example of a posible implementation of m-sequences