
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, 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: secuencia, correlacion, señal

El análisis de la correlación de señales es una pieza clave en múltiples desarrollos de ingeniería tales como el GPS, el radar o la corrección de errores a la hora de transmitir información. Habiendo moldeado actividades tan diversas como la conducción, la cartografía o el uso de internet, el estudio de la función de autocorrelación puede derivar en nuevos desarrollos o mejoras en los ya existentes.

En este proyecto, nos centramos en la generación de nuevas secuencias pseudoaleatorias mas largas para, entre otros posibles usos, radares y sistemas GPS con mayor resolución. Para ello, se ha desarrollado un software desplegable en un nodo de supercomputación para asistir en la búsqueda de dichas secuencias.

Binary sequences and their applications

Abstract

keywords: sequence, correlation, signal

The analisis of the correlation of signals is a key piece in several engineering developments such as the GPS, the radar or the correction of errors during information transmission. Having shaped so many diverse activities like driving, cartography or internet usage, the study of the correlation function may lead to new developments or improvements in the ones that already exist.

In this project, we focus on the generation of new pseudorandom sequences for more accurate radars or GPS technologies. To do so, a new software, deployable in a supercomputer, has been developed to assist in the search of these sequences.

Contents

1 Introduction

Advances in wireless comunication and signal proccesing have drastically changed the capabilities of transmitting information. *Comentario: El siguiente párrafo lo deberías reescribir un poco, para ser más específico al trabajo. Internet está formado por varias redes y no tiene que ver en este caso aunque todo use señales. Intenta ser especifico porque publicar es mas un tema de servidores web que de comunicaciones* Physical maps have lost relevance long ago in favor of real-time position tracking systems. Who needs a physical meter when you can send a signal to measure distances accurately?. Thanks to the hard work of engineers that squeeze the capabilities of the carrier wave to transport information throughout different networks, we can even read this document in our browser.

Comentario: Escribir un párrafo para enlazar sobre que las secuencias con buenas propiedades se requieren en muchos casos. Para dar un par de ejemplos, nos centraremos en dos tecnologias : tecnicas de espectro ensanchado (spread-spectrum) y por otro lado tecnicas de localización (rádar). He pasado lo de las lineas 155- 158 aqui para hablar de aplicaciones.

- As the spreading code in direct-sequence spread spectrum(DSSS) [**?**][**?**]
  - CDMA in wireless communication[**?**]
- PRN-based radars[**?**][**?**]
  - GPS[**?**]

Spread-spectrum systems originated in military applications in order to provide capability for anti-jamming, low probability of interception and to make multiple users sharing a common channel possible (the so-called code-division multiple access (CDMA)). The most popular application were in 3G mobile cell communications, but also it is avaliable in ultra-wideband (UWB) and networks developed for law enforcement agencies can choose spread spectrum due to their security (citar Communication systems securtity, cheng, Gong). The downside is that it is not as efficient in terms of bandwidth required to send the information. Spread-spectrum signals have several forms in direct sequence (DS), frequency hopping (FH), and time hopping.

El capitulo 15 del libro de Gong sobre seguridad lo dedica a explicar porque estas tecnicas son más seguras ante atacantes que quieran escuchar con su propia antena o emitir interferencias para que nadie pueda comunicarse en ese canal. Es una las razones de la popularidad de estas secuencias. UWB es similar a Wifi, no requiere licencia para ponerte tener tu red inalámbrica y se suele utilizar para aplicaciones de sensores, localización y similares, que requieran poca velocidad de transmision. Esto es solo para que lo conozcas por encima. DS, FH, Time hopping sólo tienes que saber que son distintas formas de utilizar la secuencia para separar los usuarios la primera es por código ortogonal pero tambien se puede hacer en frecuencia y en tiempo usando las secuencias de otra manera, puedes expandirlo un poco si quieres para explicar esto.

ESTO REESCRIBELO EN TUS PALABRAS Y ENLAZALO CON TU SECCION DE PN Spreading sequences, generated by a pseudo-noise (PN) generator, used in spread- spectrum systems are pseudorandom sequences with good correlation. Especially m-sequences have been widely used in digital communications for more than 60 years, because their autocorrelation resembled white Gaussian noise processes, from which the name “pseudo- noise” comes. Spreading is accomplished by means of

a spreading sequence (or code) that is pre-shared for generation of employed pseudo- random noise (PN) sequences, which is independent of the data, called spreading sequences or chip sequences or signature sequences in different applications. At the receiver, despreading (recover- ing the original data) is accomplished by the correlation of the received spread signal with a synchronized replica of the spreading sequence used to spread the information.

A radar (radio detection and ranging) uses a radio-frequency electromagnetic signal reflected from a target to determine properties such as position, speed, etc *Comentario:Esto hay que expandirlo pero solo explicarlo muy por encima, hay que mencionar en que se basa el radar, porque son interesantes las secuencias*

Comentario: Añade aqui algo sobre GPS, porque luego lo mencionas . Una buena referencia es: Ipatov, V. P., Shebshayevic, B. V. (2010). Some proposals on signal formats for future GNSS air interface. Inside GNSS. Está en mendeley. Global possitining system is a satellite -based radionavigation owned by the United States offering geolocalization and time possitioning. They broadcast a current time and their position, using a binary sequence for synchronization. Because the wave speed is constant, the distance to an emitting satellite can be calculated and, together with the distance to three other satellite, the position of the reciever can be found solving a simple linear algebra system.

In this chapter, we will introduce the auto and crosscorrelation function for periodic binary sequences and its mathematical properties. We will also take a look at pseudorandom sequences, its properties and practical applications.

1.1 Correlation function

According to **?**], 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 we represent those phenomena 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 we are going to focus on, the normalized version is defined as follows:

Definition 1.1.1 (Normalized correlation)**.** *Given α and β two vectors of the same length n and α_i and β_i the components of the vectors:*

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

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 phenomena , for the rest of the document we are going to use the unnormalized version 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 1.1.2 (Unnormalized correlation)**.** *Given α and β two vectors of the same length n and α_i and β_i the components of the vectors:*

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

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

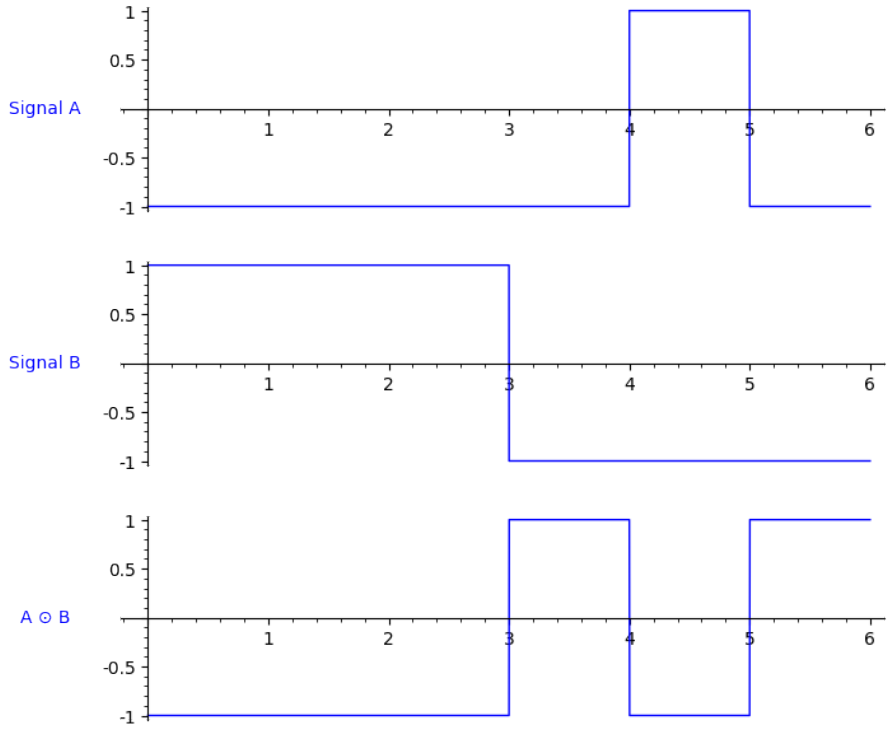

Figure 1: A graphical representation of two vectors an their pointwise product with an unnormalized correlation between them of -2 (-1 -1 -1 +1 -1 +1)

Mejor no centrarnos mucho en señales en este trabajo

As represented inFigure **??**, 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.

1.1.1 Autocorrelation function

Going on with the lecture of **?**], 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 1.1.3 (Autocorrelation)**.** *Given the function C defined in Equation* (**??**) *and n the length of the sequence S*

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

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

An example is shown in Figure **??** in which we can observe some important properties of the autocorrelation of sequences:

Theorem 1.1.1. *Given a sequence S, the autocorrelation value for* $\tau = 0$ *is:*

$$A(S)_1 = C(S,S) = \sum_{i=1}^{n} S_i^2 \tag{5}$$

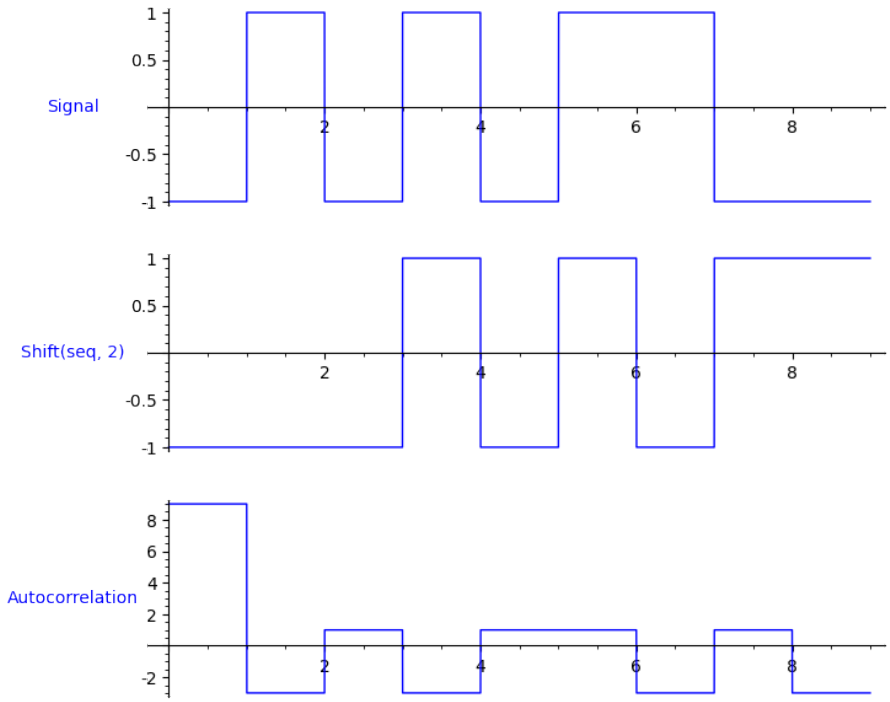

Figure 2: A graphical representation of the autocorrelation of a sequence with a shifted version of itself.

Corollary 1.1.1.1. *Given the unnormalized autocorrelation of a sequence, we can normalize it by dividing it as follows:*

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

Proof. Using Equations **??** and **??**, we can normalize **??** as follows:

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

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

Property 1.1.1. *Components of the autocorrelation vector belong to the same group as the original sequence.*

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.

1.1.2 Crosscorrelation function

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

Definition 1.1.4 (Crosscorrelation)**.** *Given C the correlation function defined in Equation* **??***, shift as the function defined in Equation* **??** *and n the length of both sequences:*

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

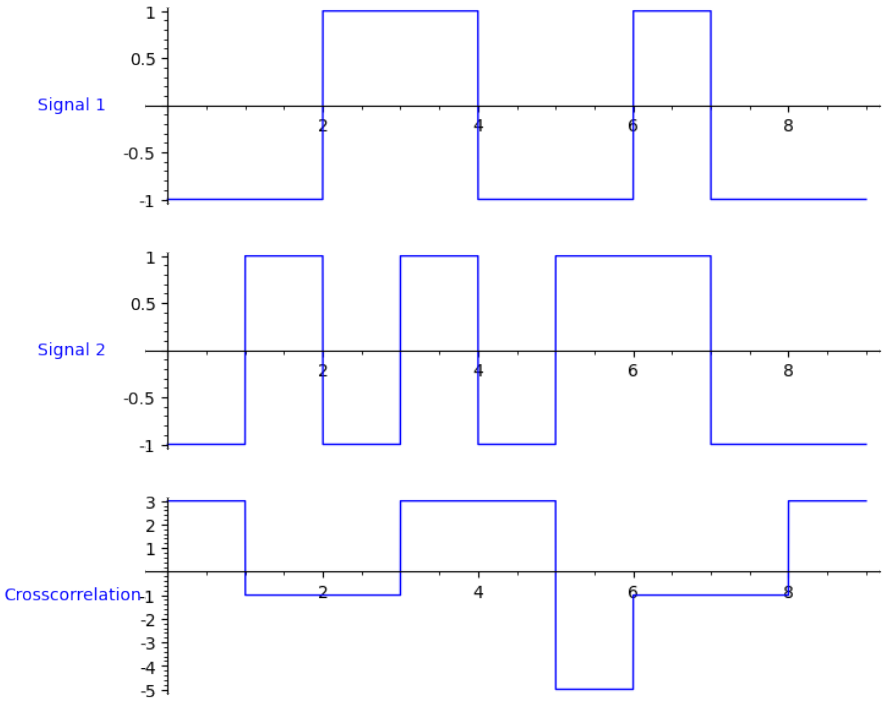

Figure 3: Two signals and it's crosscorrelation

Lemma 1.1.2. *Given a sequence S, CC the crosscorrelation function defined in Equation* **??** *and A the autocorrelation function defined in Equation* **??***:*

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

1.2 Pseudorandom noise(PRN)

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 of white noise, which is defined as vectors in which all the components are statistically independent between them.[**?**]

Even though noise in general is usually seen as an unwanted wave that limits the amount of information that can be transmited through a channel[**?**], it has some practical uses.

Property 1.2.1. *The autocorrelation of a vector of white noise equals 0 for every component where $\tau \neq 1$ [**?**]*

Taking a radar as an example, using this theorem can compute the distance just by sending a white noise signal to a target and start correlating the received signal with the original one. As the autocorrelation of white noise only has a spike when the shift is 0, that spike represents in which time instant the signal has returned. With that time instant, we can get the round-trip time and then the actual distance if we know the propagation speed of the wave.

In the case of GPS, the restrictions imposed to the noise sequence are stronger. First of all, as we will be transmiting several signals in the same frecuency, we need a set of codes with good crosscorrelation properties between them. In other words, the crosscorrelation function between two given codes must trend to 0 in every component, except when $\tau = 1$ and both codes are the same.

As noise is an statistical construct, noise measured from natural phenomena can generate sequences with poor correlation properties. As both technologies depend on this properties to work, we must find a way of creating sequences with properties similar to those of noise in an deterministic and efficient fashion.

This kind of sequences are called Pseudorandom Noise(PRN). In practical applications, PRN sequences aren't perfect noise because generating it is difficult and unnecesary. In reality, we don't need an actual 0 in every position of the autocorrelation. If we let the sequence take values in a threshold so that the system won't mistake intermidiate values with the autocorrelation spike, it will behave as expected.

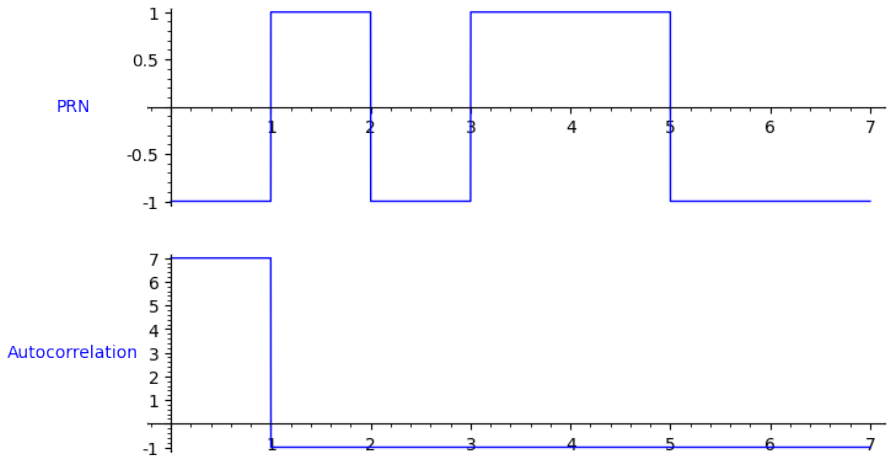

Figure 4: A pseudorandom noise sequence an it's autocorrelation function. Notice that this PRN code isn't perfect noise.

2 PRN generation

As explained in the previous chapter, PRN codes are useful in tecnologies that need properties similar to those of white noise. In this chapter, we are going to introduce some state-of-the-art tecniques in PRN generation.

2.1 Maximum Length Sequence(MLS)

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

Definition 2.1.1 (LFSR)**.** *A MLS 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 **?**], 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 MLS will always be of length of the form $2^n - 1$ where n is an arbitrary natural number.*

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

```
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 MLS 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 we might need sequences of arbitrary size in some applications. As we will 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[**?**] are a family of sequences, derived from MLS, with very important properties that are used in several applications such as wireless communication and geolocalisation. A Gold Code generator gets two MLS sequences that fulfill:

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

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

And XORs all their relative shifts generating a family of $2^n + 1$ sequences ($2^n - 1$ XORed sequences + 2 MLS 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* **??***:*

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

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 we must treat signals that are not the one we want to receive as noise.

Gold also showed a way to generate this pair of sequences, using a decimation of one MLS 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 MLS 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* **??***.*

```
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 **??**.

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

2.3 Legendre sequences

Legendre sequences, as explained in **?**], 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:*

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

We define the Legendre Sequence 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 $p \equiv 3 \bmod 4$, we can say that $LSs(p)$ has a flat autocorrelation.[**?**]*

Even though **?**] has a generalization of property **??** 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, we can derive that the distribution of Legendre Sequences is related to the Prime Number Theorem. This means that Legendre Sequences have more possible lengths than MLS 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 **?**] 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 **??**, we define the composite matrix as:*

$$CM(S,T) = \begin{bmatrix} shift(S,T_1)_1 & shift(S,T_2)_1 & \dots & shift(S,T_m)_1 \\ shift(S,T_1)_2 & shift(S,T_2)_2 & & \\ \vdots & & \ddots & \\ shift(S,T_1)_n & & & shift(S,T_m)_n \end{bmatrix} \tag{17}$$

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 **??** and **??** and the composite matrix defined at Equation **??**, we define the composite sequence 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 we are using a non-binary sequence to illustrate better the method).

We can define the correlation function for composite matrixes:

Definition 2.4.3. *Given 2 composite matrixes $M0$ and $M1$ with n rows and m columns, we define it's correlation as:*

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

We can also define the shift function for composite matrixes:

Definition 2.4.4. *Given a composite matrix M of n rows and m columns, we define the shift function as:*

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

This let's us establish the following relation:

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 we define the autocorrelation function for a composite matrix, an interesting property arises:

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

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

Notice that if we keep i constant, we get a particular component of the autocorrelation function of S.

Property 2.4.1. *We can define the autocorrelation function of the composite sequence in terms of the autocorrelation function as follows:*

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

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

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[**?**] and E.N. Gilbert [**?**] in 1965, are a set of sequences highly used in radar and sonar applications. We are going to provide the 2 definitions 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 we could compact the representation 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 **??**.

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:*

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

We say 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* **??***, we say it's a Costas array if, for any given r value, it satisfies Equation* **??***.*

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

Several construction methods have been proposed. For sake of simplicity, we are going to introduce just the Welsh construction as defined in Gilbert [**?**]:

Given a prime number p and a primitive root g of p, we can contruct a Costas array S 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 PRN 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 we are searching for periodic autocorrelation rather than aperiodic ones, these works are worth looking at them:

First of all, we are going to take a look at the definition of the aperiodic autocorrelation:

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

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

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

Definition 3.1.2. *Given a binary sequence S, we define the energy of S as:*

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

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

One of the first optimizations for this method was proposed by **?**] 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 recursivily fixing elements at the extremes of the sequence as shown in Figure **??**.

It's trivial that, when we complement a sequence, the components of the autocorrelation can be lowered by, at most, -2. Based on that, we can get a relaxation of E_{min}:

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

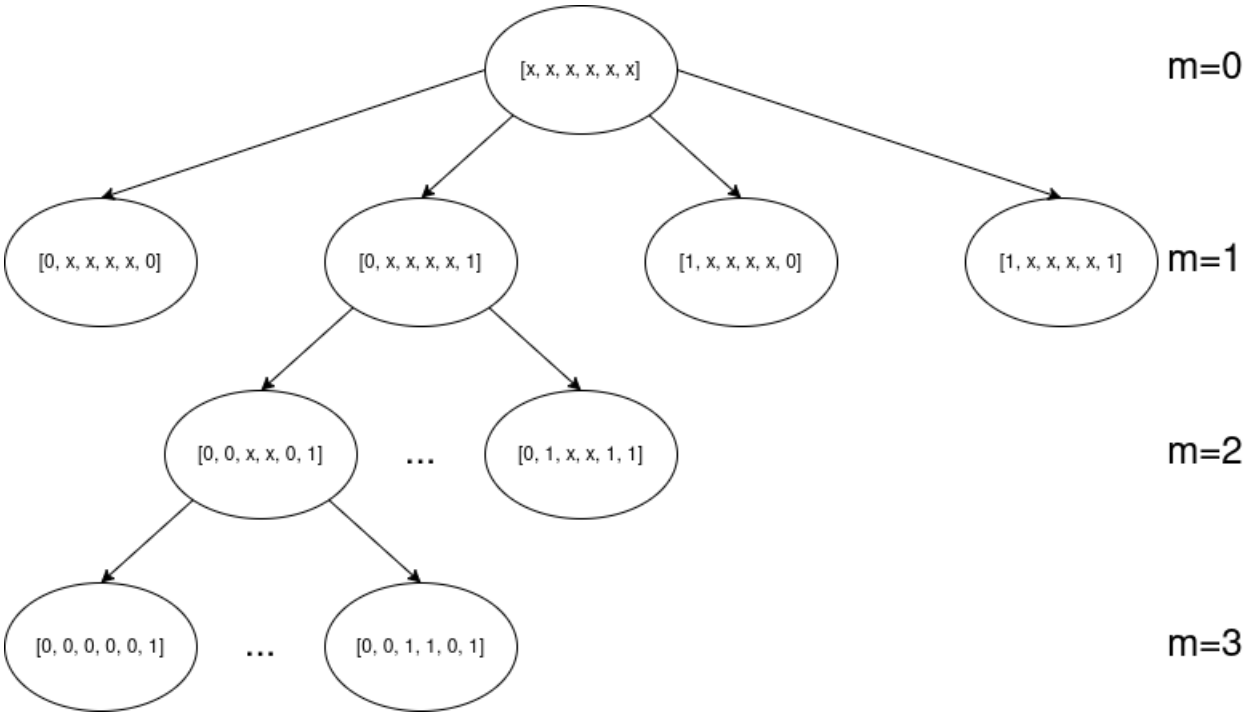

Figure 11: An example of the branching used in **?**] with $N = 6$ and where x represent unfixed values.

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, we can prune that branch reducing the amount of computation. With this algorythm, Mertens optimized sucessfully up to $N = 48$.

This work was further improved in **?**]. 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 **?**] by computing it up to $N = 85$.

3.2 Our approach

To tackle the huge complexity encountered in the previous methods, we took a different approach. 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:

- The search space for sequences of length N is reduced from $O(2^N)$ to $O(p^m)$ where $p * m = N$.

- The autocorrelation function can be optimized for the composition method (as we will see in a following chapter).
- The search space can be efficiently bounded applying the Hamming autocorrelation.
- Unfortunatly, the possible sequences that we can find by this method are limited in length. We can only get sequences of the form $p * m$ where p is prime and $gcd(p, m) = 1$.

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 we add a constant to every component to the shift sequence, we get a shifted version of the same sequence. This means that if we only computed the permutations that start with the same component, we would cover the whole search space 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* **??***, we define the Hamming autocorrelation as:*

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

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 we substitute a symbol that only appears once for another, the hamming autocorrelation won't get lower. This means that if we do a depth-in-first bounding the nodes that have a hamming autocorrelation higher than the threshold (we mean, the maximum non trivial component), we are sure that all nodes in that branch have a higher hamming autocorrelation than the threshold.

We can deduce several things from Figure **??**:

- We reduce the number of autocorrelations computed by a significant amount.
- The computation on each branch isn't balanced. This must be taken into account when we design the parallelism model.

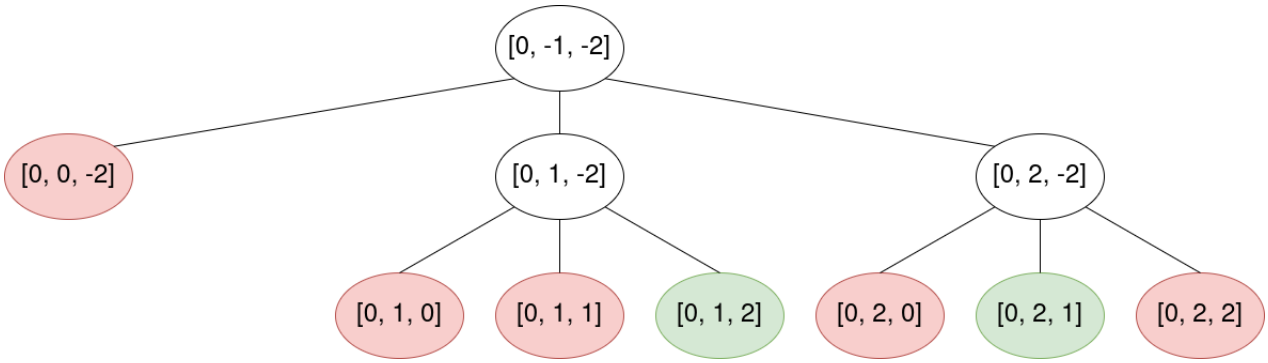

Figure 12: An example of the branch and bound algorythm with a threshold for hamming autocorrelation of 1 and a base sequence of length 3. Red nodes represent prunes and green ones final nodes in which the autocorrelation is computed and checked. Negative values represent those that haven't been initialized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

4.1 Overall description

In this section we will provide a general idea behind the software project.

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

We will summarize in a general way the functionalities we expect from this software:

- 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.
- Manage permissions.

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 we didn't design the software to take full advantage of the capabilities of these systems.

Taking this into account, we should target the software installed in those enviroments. 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 we will introduce the software requirements of our project.

4.2.1 Functional requirements

The functional requirements of the project are shown in Figure **??**.

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 **??**.

Identifier	Type	Description	Relevance
NFR01	Performance	As we are searching for sequences in a huge space, speed is a top priority to continue with the research	Very high
NFR02	Compability	As we are deploying our project in a research enviroment with different arquitectures, we must try to make it 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, we need the parts of the software to 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 casse, 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 we are working with.

As we are working with supercomputers, we can encounter a minimalist enviroment with no graph-

ical desktop. For this reason, we should focus on a command line based application 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[**?**] 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. We can waste a lot of time 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 we want a quality product and an efficient development proccess.

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, we will take a mixed approach 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"[**?**], 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 **?**] in Chapter 23, being the main difference that we don't need an oracle that predicts the value. Instead, we just check a property of the output.

A well implemented library(there are several of them, but in our case we are working with Hypothesis[**?**] 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 we choosed this type of test suites 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 we will take full advantage of this paradigm. In Figure **??**, we have an example of a property used for testing the codebase.

The problem with this paradigm is that it becomes way too complicated when we have to test methods with 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 **??**

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 we can run a validation session in which we 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 we are going to follow.

In our case, we are going to distance ourselves from 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 we can get a lot of feedback during development and fix issues earlier.

- As we are developing this project 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 we will define 2 different roles 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 we want to double check it's properties when retrieving the results from the database.

The test proccess consisted in checking that the python wrapper provided the same results as a naive implementation of the Wiener–Khinchin's algorythm. From that, the third function was tested against the first one.

At first, we tried to develop C functions with fused types. Although it favors extensibility, the compiler started to throw errors related with fused types. As the problem would have gotten worse when we developed the next parts of the program, we decided to drop support for fused types as we 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 we are interested in, the maximum Hamming autocorrelation for the prune part of the algorythm and the base sequence to use.

For this purpose, we developed a C implementation of the maximum Hamming autocorrelation 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, we defined a test based on lower and upper bounds.

In the case of the branch and bound method, we will check that all the sequences returned satisfies the specified maximum autocorrelation.

Iteration 3: Parallelism

In this iteration we focused on adapting the branch and bound algorythm to a parallel enviroment. To do that, we incrementally improved the algorythm. First, we only applied branch and bound from a given depth and we then decided to also apply it at the master's process level.

In this case, we moved completly to pure python as it isn't critical code. Most runtime of slave processes will be spent in Cython functions and, for simplicity and reliability, we 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 we focused on designing a suitable UI. To do so, we developed 2 functionalities:

- 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 passed an exhaustive process of validation to check that the user was satisfied.

5 Technology choices

In this chapter, we will take a look at the technologies used in this project, the reasons behind their adoption, pros and cons for this project and difficulties encountered during development.

5.1 SageMath

SageMath[**?**] 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, we wanted a way to share the notebooks through the cloud. We decided to work with a free version of CoCalc[**?**]. 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 pretty 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[**?**] 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 we can expect 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 because of it's performance issues. As we are developing a software with performance

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

Fortunatly, there are alternative implementations such as Jython, IronPython, etc. In our case, we are going to use Cython[**?**][**?**] as it provides a compiler to build C code with pseudo-Python. We can even call python code from C functions an viceversa, proving useful when we need a system with C level performance to operate with high-level Python libraries.

To support the decision of using Cython for the critical parts of the code, we designed some benchmarks to test the actual performance improvements. As shown in figures **??** and **??**, 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, we could just use Numpy methods to implement our algorythms (which we indeed did in the general autocorrelation function with Wiener–Khinchin's algorythm). The problem raises when we need to access the Numpy array in an undefined way and we have 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++), we can define functions that can work with diverse types depending on the input arguments. Even though in our case we will skip it as it comes with extra headaches and we are working with just integers, it's a nice feature if we would like 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 we will need 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[**?**] is an open-source relational database supported in a highly varied range of enviroments. As such, we don't depend 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, we would want a persistence system that supports well our stack of technologies. As PostgreSQL is designed to run Python code on it's SQL code, it gives us a lot of flexibility in how we want to design our application.

6 Implementation

In this chapter, we will introduce some specifications of the implementation of our solution.

6.1 General autocorrelation function

The autocorrelation function introduced in Equation **??** 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 **??**. Even though this algorythm it's simple and follows the mathematical definition, is too slow. We have to compute the correlation function for each component, leading us to a complexity of $O(n^2)$ where n it the size of the sequence with a huge constant as we have to build the shifted sequence for each component.

This constant can be improved if we avoid 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[**?**] 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[**?**], the complexity of this method lowers to $O(NlogN)$. However, its constant is still high as we need to apply 2 FFT to the sequence and apply the complex conjugate. In fact, we need to keep in mind that Fourier Transforms works with complex components while the naive approach keeps using the same type of components which makes its constant even higher than the 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 we were to use a general method for this computation, we would probably use the one based on Fourier Transforms because we will be dealing with long sequences that will compensate the big constant of this method.

However, we are dealing with sequences with the special property of having been built through the composition method. This means that we might find a non general way of computing this autocorrelation with better computational characteristics.

If we take advantage of Property **??**, we can design an algorythm with interesting properties. 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. We just need 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 we just want to check a certain property of the autocorrelation we have no need to compute the whole function.

An example implementation of this algorythm is shown in Figure **??**.

```
cdef bint c_good_composite_autocorrelation( int* autocorrelation_with_constant
                                          , 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
  cdef bint is_constant_column, is_current_shift_constant, is_constant_correlation,
                                                  is_constant_hit
  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]
      is_constant_column = affected_column == constant_offset
      current_shift  =  (positive_difference + shifts[y]) % l_signal
      is_current_shift_constant = shifts[y] == constant_offset
      is_constant_correlation = is_constant_column*is_current_shift_constant
      is_constant_hit = is_constant_column or is_current_shift_constant
      final_shift = (not is_constant_hit)*abs(current_shift-affected_column)\
                  + is_constant_hit*l_signal
      output = output \
             + (not is_constant_correlation)*autocorrelation_with_constant[final_shift]\
             + is_constant_correlation*l_signal
    if output > threshold:
      return False
  return True
```

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

6.2 Single-threaded Branch and Bound

The theoretical approach for the Branch and Bound algorithm has already explained. In this section, we introduce the actual implementation we used in the project in Figure **??**.

```
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, we 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, we need a low latency assignation of tasks. If we did use shared memory, every node would need to access memory through the ”slow” interconnection network. Instead, with MPI, we can make the proccesses to talk between them.

For the purpose of this project, we are going to work with MPICH[**?**] as the bindings of MPI4PY support it and it’s the implementation of the cluster we are working with[**?**].

In our particular problem, we decided to define a set of tasks 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 **??**. Notice that the tasks are completly unbalanced so a static scheduler wouldn't be efficient at all.

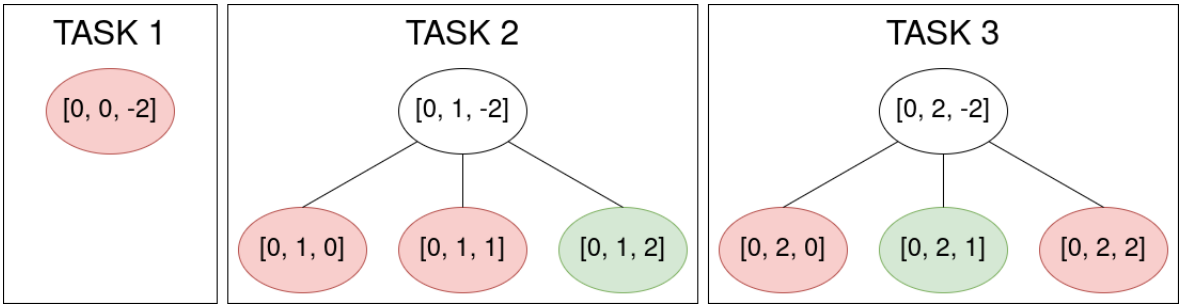

Figure 21: An example distribution of tasks for the example at Figure **??**.

This model is a good first version. However if we don't prune when assigning the tasks we will still have 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 **??** and **??**.

A third version (which we didn't implement because the second version was enough for our objective) 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, we can give the tasks 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+1): # 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)+1,
                                                              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, we will briefly discuss the design of the User Interface. As expressed in the chapter of Sofware Engineering, our primary focus is its compatibility with command lines. For that, we used a POSIX compatible interface.

First, we provided a –help option 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, 5] 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, 5, 0] 0ms
2020-08-23 19:28:42 [7] : TASK_ASSIGNED [0, 5, 3] 0ms
2020-08-23 19:28:45 [1] : EXITED
2020-08-23 19:28:45 [5] : TASK_ASSIGNED [0, 5, 2] 0ms
```

```
2020-08-23 19:28:47 [2] : EXITED
2020-08-23 19:28:48 [4] : TASK_ASSIGNED [0, 5, 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 we are using stdout, we can pipe the log 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

Some work that must be still done is summarized in this chapter:

- The persistency module must be written with PostgreSQL as we are currently dumping the results in plain-text files.
- MPI usage can be farther improved if needed, nowadays we don't buffer new tasks in the nodes so the delay of the interconnection network haven't been mitigated.
- Because of a lack of time, I couldn't use Calderon to get some benchmarks for the parallelism model. This is a shame as some statistics on the complexity of our approach are a must to publish our results.
- Even if Domingo decided to stick with Legendre sequences to keep the program simple, it would probably be interesting to explore other kinds of base sequences.
- I should try to refactor the code to be more compatible with different versions of MPI as MPI4PY support all but I had to tweak my code to work in Calderon as I had used OpenMPI in my development process.
- My code isn't portable as Python's command conventions aren't standard. In Archlinux (my own system), the commands are python and python2 while Debian (Calderon's system) uses python3 and python. As my main code wraps mpiexec, this is a pain to deal with.
- It would probably be useful to support config files as some parameters. for the command are common to all computations.
- I will have to consider if I really have to create a way to interact with the tasks and handle some kind of permissions as supercomputers seem to handle that for me.
- I will have to tidy up the code if Domingo wants to publish it as part of his investigation. Even though it's commented and it's readable, there are some parts that should be rewritten and optimized.
- We haven't expanded the literature yet so, at this stage, there's still some computations to be made.

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

The final results of the project can be sumarized as:

- I have acquired enough knowledge of the problem domain to be capable of helping to the researchers in their software needs.
- I have extended my knowledge in parallel computing by learning a new paradigm (Message Passing Parallelism).
- I have learned new tools (Cython and MPI) to fulfill the project needs.
- I have adapted my knowledge in functional programming and its robustness to develop the tests for this software.
- I have applied my knowledge in algorithmic complexity to make rational decisions on which is the best approach to a problem.
- I have overcome the limitations imposed by the current health crisis by changing my project management methodology to an agile development.
- I have learned how to manage a huge volume of scientific literature to carry out a research.
- I have started to work with a supercomputer to be capable of deploying this software and get actual results.

To conclude this report, I would like to provide some final thougts:

- After dealing with Cython, I have to say that I encountered many problems with the compiler. The GIL checker detected GIL usages in pure C code. The fused types are very cryptic. If I had to do another project with similar characteristics, I would probably write the core modules in C++ (To be able to use templates) and use Cython to create the bindings for Python.
- I recognize that I should have done some test in the supercomputer before starting the software engineering to check the availability of tools and the services that the supercomputer provides by itself.
- Even though I said I would try to develop a portable solution, as far I can tell from what I've worked so far with Calderon, i think that wasn't a realistic objective as the programs and tools available in each supercomputer seem to vary a lot.
- I have to be more methodic in my usage of GIT. Luckly I didn't need to do a rollback because my commits were huge and some of them in unstable states.

Bibliography

[] E. W. Dijkstra. GO TO statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968. http://doi.org/10.1145/362929.362947

[] E. W. Dijkstra. On a somewhat disappointing correspondence, 1987. http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1009.PDF

[] D. E. Knuth. Structured programming with **go to** statements. *ACM Comput. Surv.*, 6(4):261–301, 1974. http://doi.org/10.1145/356635.356640

[] D. E. Knuth y R. W. Floyd. Notes on avoiding 'go to' statements. *Information processing letters*, 1(4):177, 1972. http://doi.org/10.1016/0020-0190(71)90018-4

[] D. Moore, C. Musciano y M. J. Liebhaber. "'GOTO considered harmful' considered harmful" considered harmful? *Commun. ACM*, 30(5):350–355, 1987. http://doi.org/10.1145/22899.315729

[] F. Rubin. "GOTO considered harmful" considered harmful. *Commun. ACM*, 30(3):195–196, 1987. http://doi.org/10.1145/214748.315722