MSc Thesis

Evaluation of Interpolation Methods with Application to Shamir's Secret Sharing

Aristomenis Tressos

Piraeus - August, 2023

Dpt. of Digital Systems, University of Piraeus
Digital Systems Security
Supervisor: Prof. Dr. Christos Xenakis

Acknowledgements

This thesis is dedicated to my parents, Sofia and Dimitris. I would like to express my gratitude to them for their mental and financial assistance throughout my academic journey. It would certainly be very much harder without their assistance.

Abstract

One of the most fundamental components of cryptography is the Shamir's Secret Sharing Scheme (SSSS) that enables the distribution of a secret among multiple parties in a secure manner. An integral aspect of SSS involves the reconstruction of the original secret through interpolation techniques, which remain crucial for maintaining the scheme's effectiveness. The main focus of this thesis is the evaluation of various interpolation methods within the context of SSSS. The primary objective is to comprehensively assess the performance and suitability of distinct interpolation methods, elucidating their respective strengths and weaknesses. By conducting a series of meticulous experiments and analyses, this study examines the behavior of interpolation methods under different scenarios. The findings reveal that the optimal choice of an interpolation method hinges on the specific characteristics of each use case, emphasizing the need for a judicious selection process.

Contents

List of Figures

Chapter 1

Introduction

The early days of asymmetric cryptography took place during the late 1970s, which was characterized by the introduction of the Diffie-Hellman Key Exchange protocol [1]. The introduction of asymmetric cryptography marked a significant advancement in the field, as it effectively addressed the challenge of distributing a single symmetric key to all participants within a cryptosystem. The goal was accomplished by the possession of a distinct set of keys by each party, consisting of a private key and a public key. In the same year, the widely recognized RSA cryptosystem was introduced, enabling users to encrypt and transmit messages in a secure manner [2]. Alongside the advancement of both technology and technical resources, there has been a growing demand for larger cryptographic keys in order to enhance security. As a result, the efficiency of encrypting and decrypting messages using public-key cryptography has been undermined.

In order to address the problem at hand, hybrid cryptosystems were developed, wherein a combination of symmetric and asymmetric cryptography techniques were employed to achieve different goals. The encryption and decryption procedures were executed using symmetric algorithms, such as AES and DES whereas the distribution of the symmetric master key was accomplished through the utilization of asymmetric key-sharing protocols, such as Diffie-Hellman. Modern technologies, including the SSL protocol, also employ this approach. In 1979, Adi Shamir, a renowned cryptographer, introduced a novel scheme for distributing secrets [3], which was considered an alternative to the Diffie-Hellman cryptographic protocol. The Shamir Secret Sharing scheme is based upon the principles of threshold cryptography, wherein the core idea focuses on the notion that the generation of a secret remains undisclosed unless a quorum of the parties acts together to combine their knowledge. The secret is distributed through what we call shares, and the quorum can reveal it using *interpolation*.

The objective of this thesis is to conduct a comprehensive analysis of several interpolation methods and evaluate their performance, with the aim of drawing conclusions on the most preferable solution for secret sharing. This assessment is conducted using experimental procedures, wherein each approach is executed with varying parameters and qualities. This document is accompanied by a GitHub repository [4] that includes all the source files utilized in the development of the experiments and the interpolation methods. The main contribution

of this thesis is to present a complete practical and theoretical overview of the available options for achieving efficient interpolation in the Shamir Secret Sharing scheme.
The present thesis is organized in the following manner:

- The first chapter includes the necessary mathematical background knowledge for interpreting the experiments.
- The second chapter provides an overview of the various interpolation methods that will be examined, along with an examination of their respective characteristics and potential optimization techniques.
- After establishing a solid foundation of fundamental knowledge, chapter three is devoted to the experiments, which serve as the main topic of this thesis. Each use case is assigned its own section, with the findings presented as diagrams and tables.
- In the fourth and last chapter, a discussion takes place, providing a concise overview of the obtained results, presenting the derived conclusions, and putting forward possible directions for further research.

Chapter 2

Preliminaries

Before diving deeper into the concepts of secret sharing, polynomial interpolation and key distribution, some background knowledge is necessary. This chapter will provide an introduction to the definitions and notation that will be utilized in subsequent chapters. A unique aspect of this thesis, in comparison to other relevant resources, is the inclusion of SageMath code blocks beside each definition. This feature enhances the reader's understanding and comprehension of each subject. The main motivation for this aspect lies in the notion that an in-depth understanding of a subject is best achieved when one possesses both theoretical and practical knowledge.

2.1 Finite Fields

Let us examine the set of real numbers, denoted as $\mathbb{R}$. How many elements does it contain? The answer is quite simple; infinitely many. For each given real number n, it is always possible to find $n + 1$. Therefore, it may be concluded that the set of real numbers $\mathbb{R}$ has an infinite cardinality.

On the other hand, a finite field, which is also known as a Galois Field or GF, is a set with finite cardinality.From now on, the term "order" shall be used to refer to the cardinality of a set. A fundamental example of a finite field is $GF(p)$ (or $\mathbb{F}_p$), which represents the set of integers modulo p, where p denotes a prime number.

Birkhoff and Mac Lane (1977)

Theorem 2.1. Every finite field has prime power order. [5]

This implies that the order of a finite field can be either a prime number p or a power of a prime p^n.

Finite fields consist of two fundamental operations: addition and multiplication. Additive groups are generated through the application of addition, while multiplicative groups are formed through the application of multiplication. In the subsequent sections of this thesis, our

focus will solely be on multiplicative groups. Specifically, we will refer to the multiplicative group of integers modulo n as $\mathbb{Z}_n^\times$. Multiplicative groups do not contain the element 0 as the inverse $\frac{1}{0} = 0^{-1}$ does not exist. As a result, the order of the multiplicative group of $\mathbb{F}_p$ is $p-1$.

```
sage: F = GF(101)
sage: F(0)^(-1)
ZeroDivisionError: inverse of Mod(0, 101) does not exist
```

Additive Inverse

Definition 1. The additive inverse of an element $x \in \mathbb{F}p$ is an element $-x$ such that $x + (-x) = 0 \pmod{p}$.

Multiplicative Inverse

Definition 2. The multiplicative inverse of an element $x \in \mathbb{F}p$ is an element x^{-1} such that $xx^{-1} = 1 \pmod{p}$.

Keep in mind that x^{-1} is just notation and does not represent the fraction $\frac{1}{x}$ as it would not be an element of $\mathbb{F}_p$. For example, let the field $\mathbb{F}_{101}$ and suppose we want to calculate 31^{-1}. We are looking for a number s.t. when multiplied by 31 we get 1, the identity element. By applying any method of our choice, we get that $31^{-1}\ mod\ 101 = 88$ which can be verified with Sage.

```
sage: F = GF(101)
sage: F(31)^(-1)
88
sage: 31 * 88 % 101 == 1
True
```

2.1.1 Order of field elements

Apart from the order of $\mathbb{F}_p$, the elements of it have also their own order which is defined as follows. For simplicity, we will denote the order of an element x as $|x|$.

Additive Order of an element

Definition 3. The additive order of an element $x \in \mathbb{F}_p$ is the smallest number $k \in \mathbb{F}_p$ such that $xk \equiv 0 \pmod{p}$

Multiplicative Order of an element

Definition 4. The multiplicative order of an element $x \in \mathbb{F}_p$ is the smallest number $k \in \mathbb{F}_p$ such that $x^k \equiv 1 \pmod{p}$

For example, suppose we want to find the multiplicative order of $13 \in \mathbb{F}_{2803}$. By using any method of our choice, we get that $|13| = 934$.

```
sage: p = 2803
sage: F = GF(p)
sage: F(13).multiplicative_order()
934
sage: pow(13, 934, p) == 1
True
```

2.1.2 Lagrange's Theorem

The most naive approach to calculate $|13|$ is to compute $13^k \; \forall \; k \in [1, p-1]$ until 1 is found.

```
p = 2803
F = GF(p)
for k in range(1, p):
    if pow(13, k, p) == 1:
        print(k)
        break
```

Listing 1: Finding the order of an element with the naive approach.

Output:

```
934
```

Nevertheless, this approach does not scale for higher values of p. A slightly optimized approach utilizes Lagrange's Theorem that is defined below.

Lagrange's Theorem (1770-71)

Theorem 2.2. If G is a finite group and $\mathbb{H}$ is a subgroup of $\mathbb{G}$, then the order of $\mathbb{H}$ divides the order of $\mathbb{G}$. [6]

In simple words, it states that the order of every element of a group $\mathbb{G}$ divides $|\mathbb{G}|$. Thus, $|13|$ should be a divisor of $|\mathbb{F}_{2803}| = 2802$. The divisors are : $[1, 2, 3, 6, 467, 934, 1401, 2802]$.

$$
\begin{aligned}
13^{1} &\pmod{2803} = 13 \\
13^{2} &\pmod{2803} = 169 \\
13^{3} &\pmod{2803} = 2197 \\
13^{6} &\pmod{2803} = 43 \\
13^{467} &\pmod{2803} = 2802 \\
13^{934} &\pmod{2803} = 1 \ (repeats\ from\ now\ on) \\
13^{935} &\pmod{2803} = 13 \ (13^{1}) \\
13^{936} &\pmod{2803} = 169 \ (13^{2}) \\
13^{937} &\pmod{2803} = 2197 \ (13^{3}) \\
13^{938} &\pmod{2803} = 531 \ (13^{4})
\end{aligned}
$$

```
divs = divisors(p-1)
for div in divs:
    if pow(13, div, p) == 1:
        print(div)
        break
```

Listing 2: Finding the order of an element using Lagrange's Theorem.

Output:

```
934
```

The proposed method involves verifying if 13 raised to each divisor yields a result of 1, which is significantly more efficient compared to the conventional approach. The time complexity depends entirely on finding the divisors of $p - 1$.

2.1.3 Group Generators and Cyclic Groups

As previously discussed, $|x|$, where $x \in \mathbb{F}_p$, is the number of unique elements that x can generate raised to the elements of the group $1, 2, 3, ..., p - 1$. For example, $|13|$ can generate 934 distinct elements in $\mathbb{F}_{2803}$. Following the computation of 13^{934}, a recurring pattern of the same 934 values will be observed. Therefore, it may be stated that the element 13 *generates a total of 934 elements within the group*.

The elements that generate all $p - 1$ elements of the group, are called **generators** or **primitive roots** and the groups generated by a generator are called **cyclic groups**. The order

of generators is always equal to $p - 1$. For example, $g = 7$ is a generator of $\mathbb{F}_{11}$ because it generates the entire $\mathbb{Z}_{11}^{\times}$.

$$\begin{aligned}
7^1 &\pmod{11} = 7 \\
7^2 &\pmod{11} = 5 \\
7^3 &\pmod{11} = 2 \\
7^4 &\pmod{11} = 3 \\
7^5 &\pmod{11} = 10 \\
7^6 &\pmod{11} = 4 \\
7^7 &\pmod{11} = 6 \\
7^8 &\pmod{11} = 9 \\
7^9 &\pmod{11} = 8 \\
7^{10} &\pmod{11} = 1
\end{aligned}$$

Thus, $|7| = 11 - 1 = 10$. Similarly, we show that $g = 3$ is **not** a generator of $\mathbb{Z}_{11}$.

$$\begin{aligned}
3^1 &\pmod{11} = 3 \\
3^2 &\pmod{11} = 9 \\
3^3 &\pmod{11} = 5 \\
3^4 &\pmod{11} = 4 \\
3^5 &\pmod{11} = 1 \\
3^6 &\pmod{11} = 3 \\
3^7 &\pmod{11} = 9 \\
3^8 &\pmod{11} = 5 \\
3^9 &\pmod{11} = 4 \\
3^{10} &\pmod{11} = 1
\end{aligned}$$

3 generates only five elements so $|3| = 5$.

Again, we can find generators or check whether an element is a generator more quickly by making use of Lagrange's Theorem. If an element g is a generator, then we know that $p-1$ is the first number for which $g^{p-1} \equiv 1 \pmod{p}$. We could do the optimization using the divisors of $p-1$ but we will present an alternative method that involves working with the distinct factors of $p - 1$, denoted as f_0, f_1, f_2, and so on.

Considering the previous example, suppose we want to find a generator of $\mathbb{F}_{2803}$. Factoring 2802 yields the distinct factors $[2, 3, 467]$. According to Lagrange's Theorem, the possible order of any element can be one of these factors or any product of those. If an element g is a generator, then $g^k \neq 1 \; \forall \; k \neq p - 1$. Now, instead of calculating the divisors, we could raise g to the powers of $\frac{2802}{2} = 3 \cdot 467$, $\frac{2802}{3} = 2 \cdot 467$ and $\frac{2802}{467} = 2 \cdot 3$ and check whether g is a generator by asserting that the result is > 1.

Starting off with $g = 2$,

$$2^{\frac{2802}{2}} = 2^{1401} = 2802 \pmod{2803}$$
$$2^{\frac{2802}{3}} = 2^{934} = 2389 \pmod{2803}$$
$$2^{\frac{2802}{467}} = 2^{6} = 64 \pmod{2803}$$

Notice that no power of 2, other than 2^{2802}, yields 1. This means that we found a generator immediately with our first try. The following Sage script outputs all the generators of F_{2803}.

```
p = 2803
factors = [2, 3, 467]
for g in range(2, p-1):
    if all(pow(g, (p-1)//f, p) != 1 for f in factors):
        print(g)
```

Listing 3: Finding all generators of F_{2803}

Output:

```
2
11
12
18
20
21
29
...
2797
2799
```

2.2 Roots of Unity

This section is essential in understanding the motivation behind the Fast (FFT) and the Discrete Fourier Transform (DFT). Later on, we will examine how FFT works and how it utilizes the properties of the roots of unity and the primitive roots of unity to optimize polynomial interpolation. These properties will be described below.

2.2.1 The set $\mathbb{C}$

Generally, the n^{th} *root of unity* is a number ω such that,

$$\omega^n = 1$$

For example, suppose we are working in $\mathbb{C}$ and we want to find the 4^{th} roots of unity. This is equivalent to solving the following equation,

$$\omega^4 = 1$$

,
After reordering, we get,

$$(\omega^2 - 1)(\omega^2 + 1) = 0$$
$$(\omega - 1)(\omega + 1)(\omega^2 + 1) = 0$$

The solutions are $\{1, -1, i, -i\}$, where $i = \sqrt{-1}$. These are also the 4^{th} roots of unity in $\mathbb{C}$.

Primitive n^{th} root of unity

Definition 5. A **primitive** n^{th} root of unity is a number ω that is solution of the equation $\omega^n = 1$ but **not** a solution of $\omega^m = 1$, for all $0 < m < n$.

Taking the example above, let's see whether the 4^{th} roots of unity we found above are primitive 4^{th} roots of unity.

For $m = 1$, we get the trivial solution $\{1\}$. This means that 1 is not a primitive 4^{th} root of unity. Next, for $m = 2$,

$$\omega^2 = 1$$
$$(\omega - 1)(\omega + 1) = 0$$

We find that the solutions are $\{1, -1\}$. Therefore neither 1 nor -1 are 4^{th} primitive roots of unity. Finally, for $m = 3$,

$$\omega^3 = 1$$
$$(\omega - 1)(\omega^2 + \omega + 1) = 0$$

We find that the solutions are $\{1, -\frac{1+i\sqrt{3}}{2}, \frac{-1+i\sqrt{3}}{2}\}$.
We conclude that only i and $-i$ are 4^{th} primitive roots of unity as they are not solutions of the above equations.

2.2.2 The set $\mathbb{F}_p$

It turns out that n^{th} roots of unity do not exist only in the set $\mathbb{C}$ but also in finite fields $\mathbb{F}_p$. Let $n, p > 1$. It is trivial to prove this theorem with the help of Lagrange's theorem. If the primitive n^{th} root of unity exists, it can be calculated as:

$$\omega = g^{\frac{p-1}{n}} \pmod{p}$$

where g is a primitive root (generator) of $\mathbb{F}_p$.

From the previous subsection we know that n^{th} roots of unity satisfy $\omega^n = 1$ which is easy to show:

$$\omega^n = g^{\frac{p-1}{n}n} = g^{p-1} = 1 \pmod{p}$$

Primitive n^{th} roots of unity can be a convenient way to specify elements of a group that have order n. Consider the following example. We will set p to be a safe prime. Recall that a safe prime p is a prime number written in the form $2q + 1$, where q is another prime number. The reason for such a choice is the ease of factorization of $p-1$. Let the safe prime $p = 2*11+1 = 23$. Then $|\mathbb{F}_{23}| = 23 - 1 = 22$. Thus we know that the order of any element in this group will be $2, 11$ or 22. In fact, the only elements that can have order 2 are 1 and $p - 1 = 22$ since $1^2 = 1 \pmod{23}$ and $(23 - 1)^2 = (-1)^2 = 1 \pmod{23}$. Equivalently, the only elements that are 2-nd primitive roots of unity are 1 and -1.

```
p = 23
for i in range(1, p):
    if pow(i, 2, p) == 1:
        print(i)
```

Listing 4: Finding the 2^{nd} roots of unity.

Output:

```
1
22
```

Based on the factors of $p-1$, we deduce that there can be only 2^{nd}, 11^{th} and 22^{th} primitive roots of unity. Let us consider the task of determining a primitive 11th root of unity, which may also be expressed as finding an element inside $\mathbb{F}_{23}$ with order 11. The first step is to find a generator of the group. For an element g to be considered a generator, we require $g^2 \neq 1$ and $g^{11} \neq 1$ (all operations are performed modulo 23).

Starting with $g = 2$.

$$2^1 = 2$$
$$2^{11} = 1$$

The order of $g = 2$ is 11 so it is not a generator.

We continue with $g = 3$.

$$3^1 = 3$$
$$3^{11} = 1$$

Next, $g = 5$.

$$5^1 = 5$$
$$5^{11} = 22$$
$$5^{22} = 1$$

We found that $g = 5$ is a generator because its order is $p - 1 = 22$. A primitive 11^{th} root of unity can be calculated as follows:

$$\omega = g^{\frac{22}{11}} = g^2 = 5^2 = 25 = 2 \pmod{23}$$

The order of the element 2 is 11 which means that it can generate a subgroup of $\mathbb{F}_{23}$ that consists of 11 elements in total. These elements are also primitive 11^{th} roots of unity. Below we list them,

$$\{2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}\} = \{2, 4, 8, 16, 9, 18, 13, 3, 6, 12\}$$

2.2.3 Properties

This section will demonstrate the symmetric features of the (primitive) n^{th} roots of unity in finite fields, which are connected to their periodicity. The experiments and findings will be provided specifically for n being a power of two. It is important to acknowledge that a primitive nth root of unity $\omega \in \mathbb{F}_p$ has an order of n, indicating that it forms a subgroup with an order of n. As an illustration, the 2^{nd} roots of unity constitute a subgroup with a cardinality of 2, so indicating the existence of two 2^{nd} roots of unity in total.

To start with, the following planes are presented which show the $2^{nd}, 4^{th}$ and 8^{th} roots of unity.

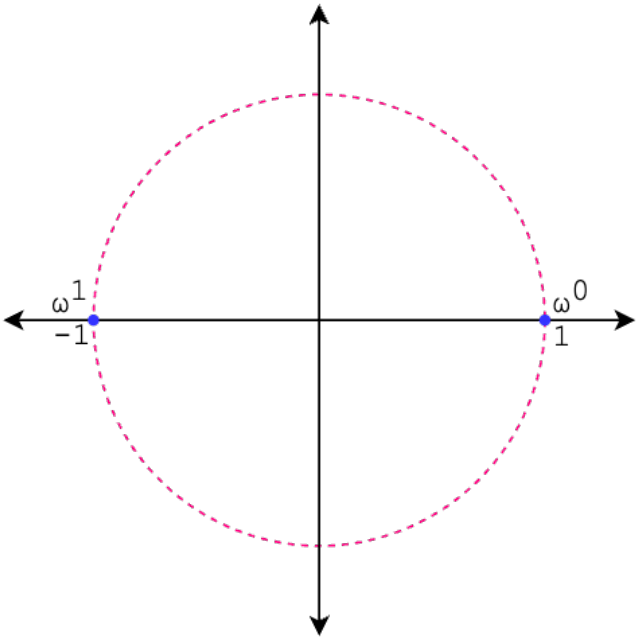

Figure 2.1: The primitive 2^{nd} roots of unity.

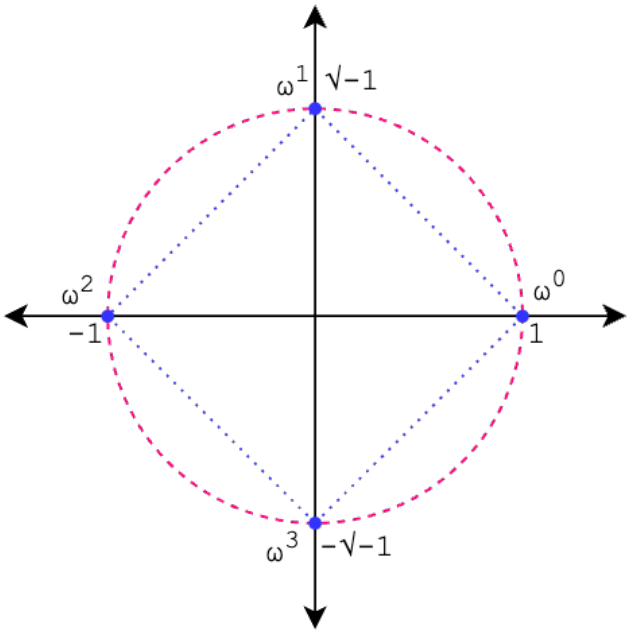

Figure 2.2: The primitive 4^{th} roots of unity.

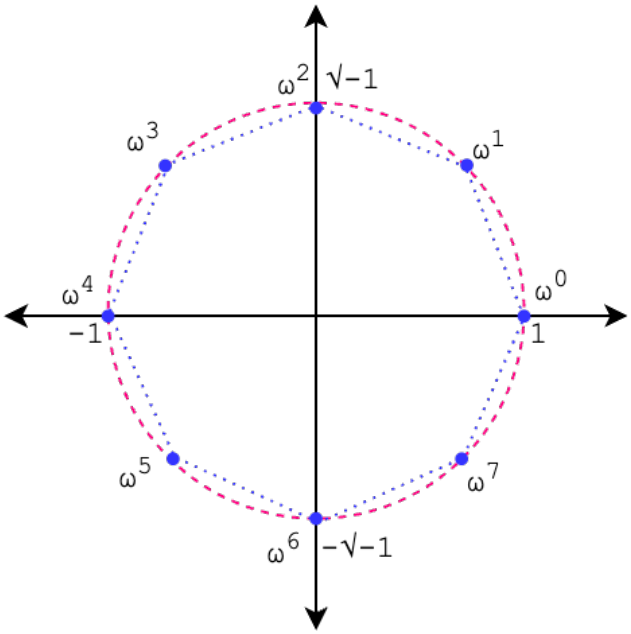

Figure 2.3: The primitive 8^{th} roots of unity.

As we work in finite fields, $\sqrt{-1}$ is equivalent to $\sqrt{p-1}$.

The 4^{th} roots of unity will be used as an example to explain the distances between roots of unity. From ω^0, we land to the next root of unity ω^1 by moving $\frac{2\pi}{4} = \frac{\pi}{2}$ [1] radians on the circle. In general, ω^i is the point that corresponds to the radian $\dfrac{2\pi i}{4} = \dfrac{i\pi}{2}$ on the circle. Similarly, for the 8^{th} roots of unity, ω^i is the point that corresponds to the radian $\dfrac{2\pi i}{8} = \dfrac{i\pi}{4}$ on the circle.

Generally, for an n^{th} root of unity ω, we get ω^i by moving $\dfrac{2\pi i}{n}$ radians on the circle. Taking the 8^{th} roots of unity as an example, it holds that $\omega^0 = -\omega^4, \omega^1 = -\omega^5, \omega^2 = -\omega^6$ and $\omega^3 = -\omega^7$. Generally, when $n = 2^k$ it holds that:

$$\omega^i = \omega^{i+\frac{n}{2}} \pmod{p}$$

for $0 \leq i < \frac{n}{2}$. If $i > n$, then ω^i can be reduced by $\omega^{i \pmod{n}}$.

At a high level, this feature is preserved due to the cancellation of the common factor of 2 between the numerator and denominator, resulting in the emergence of periodicity. If n is not a power of two, the previously mentioned relation may not apply since the denominator is not divided by the numerator's factor of 2. The aforementioned property may be succinctly expressed using the below code snippet.

```
n = 8
q = 50411
p = 403289 # n * q + 1
F = GF(p)
```

[1] There is an analogy between 2π and the order of $\mathbb{F}_p$ as both values indicate that a full circle was performed.

```
g = 3 # generator
omega = F(pow(g, (p-1)//n, p)) # 8th root of unity
print(all([omega**i == omega**(i+n//2)] for i in range(n//2)))
```

```
True
```

Another property that is already mentioned, by calculating ω a subgroup of order n is constructed. The elements of this subgroup are $\omega^0, \omega^1, \ldots, \omega^{n-1}$ with each of them being an n^{th} root of unity too.

2.3 Polynomial Evaluation and Interpolation

Although polynomial evaluation and polynomial interpolation may appear similar, they are fundamentally different from each other.

2.3.1 From Evaluation to Interpolation

The utilisation of polynomial interpolation is of significant importance in secret sharing systems, and the subsequent chapters feature experiments that are all centred around the interpolation of polynomials utilising different methodologies. Although the fundamental notion stays unchanged, interpolation can be defined within the specific context of cryptography, as is the case in our study.

Data interpolation is a widely applicable technique that may be utilised in several domains, including the real numbers ($\mathbb{R}$), rational numbers ($\mathbb{Q}$), finite fields ($\mathbb{F}_p$), and others. In the field of cryptography, our focus will mostly be on interpolation performed over finite fields, as these fields are frequently used in cryptographic protocols, like Shamir Secret Sharing.

Polynomial Interpolation

Definition 6. Let the set of $n+1$ data points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$ with each x_i being unique and $x_i, y_i \in \mathbb{F}p$. The process of determining the n-degree polynomial F such that $F(x_i) = y_i$ is called **polynomial interpolation**.

In simple words, determining a polynomial F is equivalent to determining its coefficients $a_0, \ldots, a_{n-1} \in \mathbb{F}p$ such that $F(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_0$. [7]

Knowing the polynomial F, the following theorem ensures that it is the only one that interpolates the specific data points. [8] More formally:

Uniqueness of the interpolating polynomial

Theorem 2.3. Given a set of $n+1$ points $(x_1, y_1), \ldots, (x_{n+1}, y_{n+1})$ with each x_i being unique, there exists a unique n-degree polynomial that interpolates these points.

Let's look at an example. We will use the following SageMath script.

```
# initialize GF(p) and the corresponding polynomial ring
p = 331
PR = PolynomialRing(GF(p), 'x')
points = [(72, 123), (64, 224), (289, 247), (139, 66), (297, 180), (137,
↪  260), (32, 322), (271, 211), (282, 293), (199, 223)]
F = PR.lagrange_polynomial(points)
for (x,y) in points:
    # verify that F(x) = y
    assert F(x) == y
```

Listing 5: A basic example of polynomial interpolation.

The code above terminates successfully with no error so the interpolation worked successfully. The function *lagrange_polynomial* interpolates the given points using the Lagrange's Interpolation Method which will be described in the next chapter. For now, consider it as a method that calculates the interpolating polynomial from a given set of points.

The interpolated polynomial is:

$$F(x) = 177x^9 + 171x^8 + 24x^7 + 36x^6 + 85x^5 + 221x^4 + 122x^3 + 325x^2 + 146x + 68$$

and we could represent it using just the list of its coefficients:

$$[68, 146, 325, 122, 221, 85, 36, 24, 171, 177]$$

2.4 Shamir Secret Sharing

After establishing the foundational information necessary for comprehending secret sharing schemes and interpolation methods, we can now delve into a more detailed examination of these concepts. The central focus of this thesis is to the optimisation and comparative analysis of different interpolation algorithms. It is important to comprehend the use of interpolation in cryptography and the reason for doing such comparisons. One widely utilised application is the Shamir Secret Sharing Scheme (SSSS). To enhance clarity, we shall denote it as SSS. In reality, SSS is a (k, n)-threshold scheme (see Chapter 1) so we know the following:

– There is a secret s that is distributed among n users.

– The secret can be computed by any subset of $k \leq n$ users.

– No group of $k-1$ users is enough to compute the secret.

Consider the last statement above and recall that the interpolating polynomial of degree $k-1$ can be determined by at least k points. This is the key point to understand the security of SSS. *At least k users are needed to determine a $(k-1)$-degree polynomial*.

2.4.1 Initialization

The dealer, who is considered to be a reliable entity, performs the subsequent procedures to initialise the Shamir secret sharing scheme.

– Selects a random prime number p so that all operations are performed in the field $\mathbb{F}_p$.

– It generates a $(k-1)$-degree polynomial P with k coefficients $a_i \in \mathbb{F}_p$.

$$P(x) = \sum_{i=0}^{k-1} a_i x^i = a_0 + a_1 x + a_2 x^2 + ... + a_{k-1} x^{k-1} \pmod{p}$$

The secret s can be represented as the constant term a_0.

2.4.2 Secret Distribution

The secret is distributed via *shares*. A share is just a tuple $(i, P(i))$. From now on, we will denote the share $(i, P(i))$ as (x_i, y_i). Next, the dealer performs the following:

– The dealer sends the shares $(x_i, y_i)\ \forall\ 1 \leq i \leq n$ to each one of the n users.

To prevent the user from obtaining the secret represented by the constant term a_0, the value of i is initialised to 1 and not to 0. While the distribution of a total of n shares occurred, it is important to note that only k shares are required for the purpose of polynomial interpolation.

2.4.3 Interpolation

The secret a_0 can be obtained by the k users through the exchange of their shares. Polynomial interpolation, as previously mentioned, offers a viable approach to achieve this objective. In conventional practise, the Lagrange interpolation technique has been commonly employed. However, this thesis aims to investigate, analyse, and evaluate other interpolation methods, with the objective of optimising their performance.

2.4.4 A numerical example

Initialization Phase

First thing is to define the threshold. Let us consider a cryptosystem with a total of $n = 10$ participants. In order to access the secret s, it is required that a minimum of $k = 6$ persons are involved. The interpolation process can be performed by any subset of users consisting of 6, 7, 8, 9, or 10 individuals.

Once the system has been established, the trustworthy dealer proceeds to choose a random prime number, say $p = 101$ and selects the value $s = 39$ as the secret. Then, it generates a 5-degree polynomial with the coefficients being elements of $\mathbb{F}_{101}$. The polynomial $P(x)$ is expressed as:

$$P(x) = 39 + 71x + 24x^2 + 67x^3 + 82x^4 + 13x^5$$

Notice that s is the constant term a_0 of P.

Distribution Phase

The dealer distributes the shares (x_i, y_i) to the n users.

$$\begin{aligned}(x_1, y_1) &= (1, 94)\\ (x_2, y_2) &= (2, 16)\\ (x_3, y_3) &= (3, 59)\\ (x_4, y_4) &= (4, 10)\\ (x_5, y_5) &= (5, 42)\\ (x_6, y_6) &= (6, 52)\\ (x_7, y_7) &= (7, 9)\\ (x_8, y_8) &= (8, 100)\\ (x_9, y_9) &= (9, 68)\\ (x_{10}, y_{10}) &= (10, 65)\end{aligned}$$

Any subset of $k = 6$ is enough to obtain the secret. Below you can see some of these subsets.

$$\begin{aligned}&\{n_1, n_2, n_3, n_4, n_5, n_6\}\\ &\{n_2, n_9, n_3, n_4, n_1, n_8\}\\ &\{n_5, n_8, n_1, n_9, n_7, n_2\}\end{aligned}$$

We will randomly choose the second subset for the interpolation phase.

Interpolation phase

Thus the users from the second subgroup gather their shares and proceed to the polynomial interpolation phase in which they will eventually reveal the constant term a_0 and obtain the secret.

The task is to recover the polynomial P that interpolates the points $\{(2, 16), (9, 68), (3, 59), (4, 10), (1, 94), (8, 100)\}$. Similarly to [6], we apply the Lagrange interpolation method using the SageMath software. In the next chapter, we will dive deeper into the internals of this method.

```
sage: PR = PolynomialRing(GF(101), 'x')
sage: points = [(2, 16), (9, 68), (3, 59), (4, 10), (1, 94), (8, 100)]
sage: PR.lagrange_polynomial(points)
13*x^5 + 82*x^4 + 67*x^3 + 24*x^2 + 71*x + 39
```

Listing 6: An example of an interpolating polynomial.

Users computed the correct polynomial and they are able to obtain the secret $a_0 = s = 39$.

We can see that if we try to interpolate fewer shares the reconstructed polynomial is completely different.

```
sage: PR.lagrange_polynomial(points[1:])
3*x^4 + 45*x^3 + 50*x^2 + 37*x + 60
```

Five shares result in a 4-degree polynomial to be interpolated while P is of degree 5. However, by including more shares in the interpolation the reconstructed polynomial is still the same.

```
sage: PR.lagrange_polynomial(points + [(5, 42])
13*x^5 + 82*x^4 + 67*x^3 + 24*x^2 + 71*x + 39
```

The picture below shows the process of how Shamir Secret Sharing works at a high level.

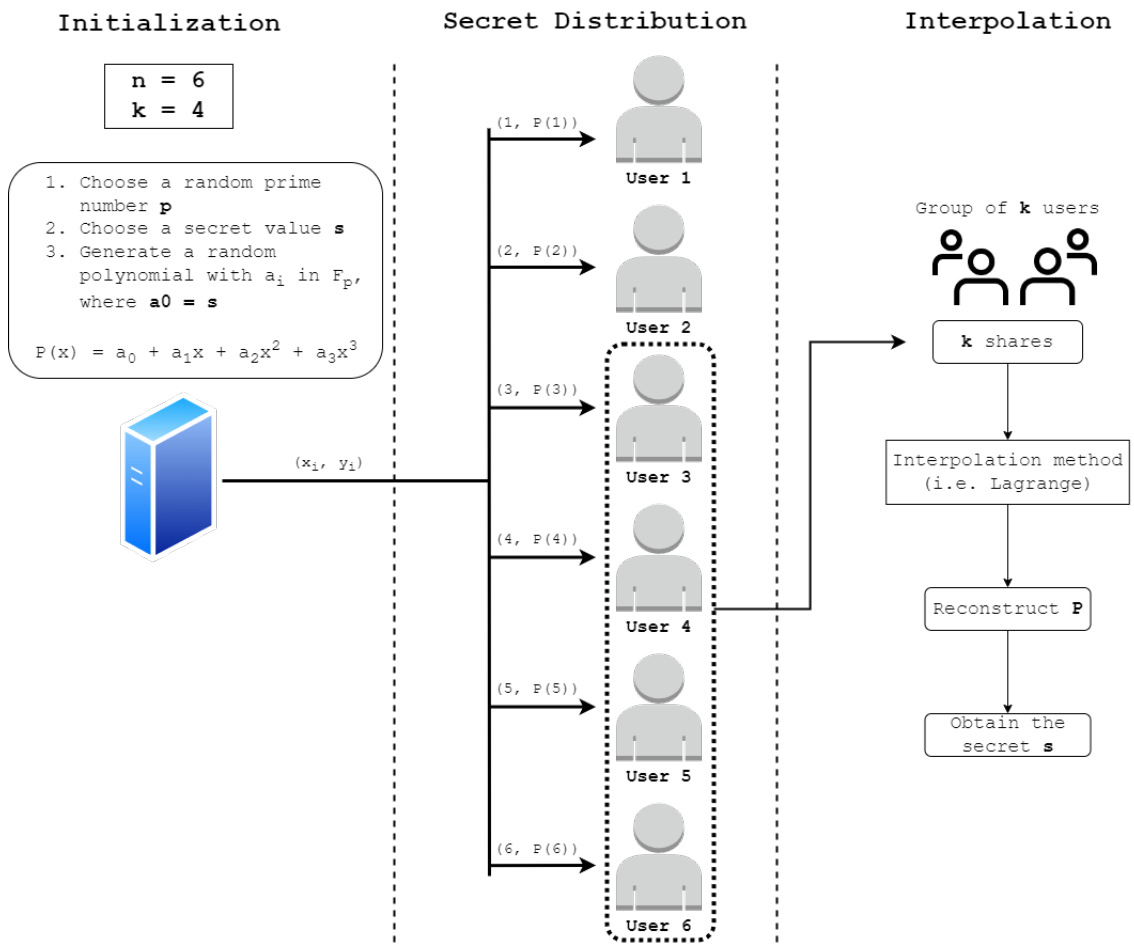

Figure 2.4: The Shamir Secret Sharing Scheme

Chapter 3

Polynomial Interpolation Methods

This chapter will provide a description of several interpolation methods. The purpose is to facilitate a comparison of their space and time complexity, as well as their overall performance, in subsequent sections of this thesis.

3.1 From a Linear Algebra Perspective

It is essential to firstly understand the algebraic nature of polynomials so that the concept of interpolation is easier to digest. Let's see what we get when we substitute the shares $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), ..., (x_{11}, y_{11})\}$ into the polynomial $P(x) = a_0 + a_1x + a_2x^2 + ... + a_{10}x^{10} \ (mod\ p)$.

$$\begin{aligned} a_0 + a_1x_1 + a_2x_1^2 + ... + a_{10}x_1^{10} &= y_1 \\ a_0 + a_1x_2 + a_2x_2^2 + ... + a_{10}x_2^{10} &= y_2 \\ a_0 + a_1x_3 + a_2x_3^2 + ... + a_{10}x_3^{10} &= y_3 \\ &\vdots \\ a_0 + a_1x_{11} + a_2x_{11}^2 + ... + a_{10}x_{11}^{10} &= y_{11} \end{aligned}$$

With the help of linear algebra, we can write these equations using matrices and vectors.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{10} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{10} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{11} & x_{11}^2 & \cdots & x_{11}^{10} \end{bmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{10} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{11} \end{pmatrix}$$

This is a linear system of equations with the 11 coefficients $a_0, a_1, ..., a_{10}$ being unknown. To ensure the existence of a *distinct* solution for this system, it is necessary to have a minimum of 11 linear equations that incorporate the variables in question. Each share represents a different relation and the number of shares corresponds to the number of the unknown variables in the system. As a result, we are certain that there will be a unique solution for the system.

Thinking about it algebraically, it should now make more sense why fewer shares would not result in the correct interpolated polynomial.

Solving this system is equivalent to solving for the coefficient vector. To solve for this vector, recall that the equation $A\mathbf{x} = B$ has solution $\mathbf{x} = A^{-1}B$, where $\mathbf{x}$ is the unknown vector and A, B the known matrices. Thus, we can solve for the coefficient vector with techniques like Gaussian Elimination as follows:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{10} \end{pmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{10} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{10} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{11} & x_{11}^2 & \cdots & x_{11}^{10} \end{bmatrix}^{-1} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{11} \end{pmatrix}$$

3.1.1 The Vandermonde Matrix

It is worth to mention that a matrix of the form:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{10} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{10} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{11} & x_{11}^2 & \cdots & x_{11}^{10} \end{bmatrix}$$

is also known as a **Vandermonde matrix** and is characterised by the property that the i^{th} row of such matrices represents a geometric progression of x_i. Returning to the interpolation problem, the computation of the coefficient vector involves the multiplication of the vector y_i with the inverse of the Vandermonde Matrix, which is created from the elements x_i.

3.1.2 Example

Let us examine an illustrative case. Consider a scenario where there are 11 parties who are each given a single share. These shares are intended to be used to rebuild a 10-degree polynomial, denoted as P, with the ultimate goal of obtaining a secret value, denoted as s. All mathematical operations are performed within the finite field $mathbbF_{829}$.

$$\{(1, 718), (2, 329), (3, 216), (4, 225), (5, 184), (6, 662), (7, 174), (8, 632), (9, 264), (10, 555), (11, 117)\}$$

We will apply the algebraic method to compute the unknown coefficient vector. Thus, we calculate the following:

$$\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{10} \end{pmatrix} = \begin{bmatrix} 1 & 1 & 1^2 & \cdots & 1^{10} \\ 1 & 2 & 2^2 & \cdots & 2^{10} \\ 1 & 3 & 3^2 & \cdots & 3^{10} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 11 & 11^2 & \cdots & 11^{10} \end{bmatrix}^{-1} \cdot \begin{pmatrix} 718 \\ 329 \\ 216 \\ \vdots \\ 117 \end{pmatrix}$$

We will use Sage to calculate the coefficient vector. Luckily, it comes with a handy function that computes the Vandermonde matrix given the vector $(x_1, x_2, x_3, ..., x_{11})$.

```
sage: p = 829
sage: shares = [(1, 718), (2, 329), (3, 216), (4, 225), (5, 184), (6,
↪  662), (7, 174), (8, 632), (9, 264), (10, 555), (11, 117)]
sage: A = Matrix.vandermonde([x for x, _ in shares], GF(p))
sage: A
[  1   1   1   1   1   1   1   1   1   1   1]
[  1   2   4   8  16  32  64 128 256 512 195]
[  1   3   9  27  81 243 729 529 758 616 190]
[  1   4  16  64 256 195 780 633  45 180 720]
[  1   5  25 125 625 638 703 199 166   1   5]
[  1   6  36 216 467 315 232 563  62 372 574]
[  1   7  49 343 743 227 760 346 764 374 131]
[  1   8  64 512 780 437 180 611 743 141 299]
[  1   9  81 729 758 190  52 468  67 603 453]
[  1  10 100 171  52 520 226 602 217 512 146]
[  1  11 121 502 548 225 817 697 206 608  56]
sage: B = vector([y for _, y in shares], GF(p))
sage: A^(-1) * B
(417, 590, 472, 566, 650, 175, 318, 709, 209, 295, 462)
```

Listing 7: Interpolation using Gaussian Elimination and the Vandermonde Matrix.

We managed to reconstruct the polynomial:

$$P(x) = 417+590x+472x^2+566x^3+650x^4+175x^5+318x^6+709x^7+209x^8+295x^9+462x^{10}$$

and consequently the secret $a_0 = s = 417$.

3.2 Lagrange Interpolation

This approach is one of the most popular and commonly used techniques for data interpolation. The polynomial representation in the Lagrange method differs from that of the algebraic approach that discussed in the preceding section, however the interpolating polynomials are equivalent in both cases. We will refer to the polynomial interpolated by the Lagrange method as the **Lagrange Polynomial**.

Let the set of n points $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$. Then the $n-1$-degree Lagrange polynomial is given by the following formula:

$$L(x) = y_1 l_1(x) + y_2 l_2(x) + ... + y_n l_n(x) = \sum_{i=1}^{n} y_i l_i(x) \tag{3.1}$$

where $l_i(x)$ is the i-th element of the **Lagrange Basis Polynomials**:

$$B = \{l_1(x), l_2(x), ..., l_n(x)\}$$

The first few l_i are given by the following formulas:

$$l_1(x) = \frac{(x-x_2)(x-x_3)\dots(x-x_n)}{(x_1-x_2)(x_1-x_3)\dots(x_1-x_n)} = \prod_{j=2}^{n} \frac{x-x_j}{x_1-x_j}$$

$$l_2(x) = \frac{(x-x_1)(x-x_3)\dots(x-x_n)}{(x_2-x_1)(x_2-x_3)\dots(x_2-x_n)} = \prod_{j=1,\ j\neq 2}^{n} \frac{x-x_j}{x_2-x_j}$$

In the general case, $l_i(x)$ can be computed as follows [8]:

$$l_i(x) = \frac{(x-x_1)(x-x_2)\dots(x-x_{i-1})(x-x_{i+1})\dots(x-x_n)}{(x_i-x_1)(x_i-x_2)\dots(x_i-x_{i-1})(x_i-x_{i+1})\dots(x_i-x_n)} = \prod_{j=1,\ j\neq i}^{n} \frac{x-x_j}{x_i-x_j} \tag{3.2}$$

At this point, it's necessary to define two base cases for the Lagrange basis polynomials so that we can explain why the formulas above work.

$$l_i(x_j) = \begin{cases} 1 & , \ i = j \\ 0 & , \ i \neq j \end{cases}$$

$l_i(x_j)$ is also known as the **Kronecker Delta** function δ_{ij}.

For the interpolating polynomial P, the shares (x_i, y_i) must satisfy $P(x_i) = y_i$. Thus the Lagrange polynomial L interpolates the data because:

$$L(x_j) = \sum_{i=1}^{n} y_i l_i(x_j) = \sum_{i=1}^{n} y_i \delta_{ij} = y_j$$

For example:

$$L(x_1) = \sum_{i=1}^{n} y_i \delta_{i1} = y_1\delta_{11} + y_2\delta_{21} + ... + y_n\delta_{n1} = y_1 * 1 + y_2 * 0 + ... + y_n * 0 = y_1$$

$$L(x_2) = \sum_{i=1}^{n} y_i \delta_{i2} = y_1\delta_{12} + y_2\delta_{22} + ... + y_n\delta_{n2} = y_1 * 0 + y_2 * 1 + ... + y_n * 0 = y_2$$

The division operation required for determining l_i is the most computationally intensive component of Lagrange interpolation. As a means of optimisation, it is possible to enhance the computational efficiency by performing the multiplication of the numerator and denominator separately, followed by a singular division operation, rather than doing n divisions individually.

```
def L(i, X):
    Lpoly = 1
    for j in range(len(X)):
        if j != i:
            numer = x - X[j]
            denom = X[i] - X[j]
            Lpoly *= (numer / denom)
    return Lpoly
```

Listing 8: Unoptimized calculation of Lagrange Basis polynomial.

The next section provides a concise overview of the optimisation process. The initial code snippet demonstrates the computation of l_i without optimisation, accompanied with the total time needed for the interpolation process.

This function finishes in the following time (seconds):

```
25.76
```

The next code snippet shows the same results but for the optimized calculation of l_i.

```
def L(i, X):
    numer = 1
    denom = 1
    for j in range(len(X)):
        if j != i:
            numer *= (x - X[j])
            denom *= (X[i] - X[j])
    return numer / denom
```

Listing 9: Optimized calculation of Lagrange Basis polynomial.

```
19.12
```

3.2.1 Example with a small-degree polynomial

Suppose the dealer of the secret sharing scheme constructs the following polynomial defined over $\mathbb{F}_{89}$:

$$P(x) = 27x^4 + 52x^3 + 71x^2 + 75x + 73$$

The dealer distributes the shares to the users $(1, 31), (2, 20), (3, 78), (4, 1), (5, 55)$ and they want to obtain the secret 73 using the Lagrange interpolation method. First, the Lagrange

basis polynomials have to be calculated. Based on (3.2) we get: [1]

$$l_1(x) = \frac{(x-2)(x-3)(x-4)(x-5)}{(1-2)(1-3)(1-4)(1-5)} = 26x^4 + 81x^3 + 66x^2 + x + 5$$
$$l_2(x) = \frac{(x-1)(x-3)(x-4)(x-5)}{(2-1)(2-3)(2-4)(2-5)} = 74x^4 + 17x^3 + 5x^2 + 3x + 79$$

Similarly, we calculate l_3, l_4, l_5.

$$l_3(x) = 67x^4 + 86x^3 + 79x^2 + 25x + 10$$
$$l_4(x) = 74x^4 + 76x^3 + 8x^2 + 25x + 84$$
$$l_5(x) = 26x^4 + 7x^3 + 20x^2 + 35x + 1$$

Notice that the computation of $l_i(x)$ requires polynomial division in the final step. Thus the Lagrange polynomial can be calculated as:

$$L(x) = \sum_{i=1}^{n} y_i l_i(x) = 27x^4 + 52x^3 + 71x^2 + 75x + 73$$

3.2.2 Efficient constant term reconstruction

After providing an explanation of the inner workings of Lagrange interpolation, it is now imperative to shift our attention towards the practical application in secret sharing schemes. The primary objective of this method is for users to obtain the shared secret by combining their individual shares. In reality, their sole interest is in the constant term $a_0 = P(0)$ of the polynomial, leading them to disregard the remaining coefficients. Therefore, instead of evaluating the function $L(x)$, the function is evaluated at $x = 0$, denoted as $L(0)$.

$$L(0) = \sum_{i=1}^{n} y_i l_i(0)$$

where $l_i(0)$ are now defined as:

$$l_1(0) = \frac{(0-2)(0-3)(0-4)(0-5)}{(1-2)(1-3)(1-4)(1-5)} = 5$$
$$l_2(0) = \frac{(0-1)(0-3)(0-4)(0-5)}{(2-1)(2-3)(2-4)(2-5)} = 79$$

and so on.

It is seen that the values of $l_i(0)$ are obtained by the division of constants, resulting in a constant value. This demonstrates that this technique is considerably more efficient compared to the conventional approach of computing the Lagrange basis polynomials, which involves polynomial division. Therefore, it may be concluded:

$$L(0) = \sum_{i=1}^{n} y_i l_i(0) = 31 * 5 + 20 * 79 + 78 * 10 + 1 * 84 + 55 * 1 = 73$$

[1] Since we work in $\mathbb{F}_p$, division is equivalent to multiplication with the multiplicative inverse and negative numbers $-k$ are equivalent to $-k \ (mod\ p)$.

which is indeed the shared secret.

Generally, the computation of the Lagrange basis polynomials requires the application of the distributive property in the numerator which corresponds to $2d$ multiplications, where d the degree of the polynomial. For the efficient approach, it is reduced just to d constant multiplications.

3.3 Improved and Barycentric Lagrange Interpolation

As the names imply, these methods are similar to the Lagrange Interpolation method but optimized so that they scale for larger values of n.

Normally, the Lagrange method exhibits a computational complexity of $\mathcal{O}(n^2)$ operations for determining the basis polynomials l_i and evaluating the polynomial. This characteristic renders the approach suboptimal for computational tasks and practical implementations. Furthermore, the addition of every new point necessitates the complete recomputation of all n^2 processes. We are looking for different representations of the Lagrange polynomial, aiming at improving the computational efficiency of Lagrange method.

3.3.1 Improved Lagrange Interpolation

Recall how the Lagrange Basis polynomial l_i is defined:

$$l_i(x) = \prod_{j=1,\ j\neq i}^{n} \frac{x - x_j}{x_i - x_j}$$

Let $l(x)$ be the numerator of l_i but with the i-th term included in the product.

$$l(x) = \prod_{j=1}^{n} (x - x_j)$$

Then the numerator of l_i can be rewritten as:

$$l_i(x) = \frac{l(x)}{x - x_i}$$

Let the new definition of the **weight** to be defined as the inverse of the denominator of l_i:

$$w_i = \frac{1}{\prod_{j=1, j\neq i}^{n} (x_i - x_j)}$$

Finally, the i-th Lagrange polynomial l_i can be rewritten as:

$$l_i(x) = \frac{l(x)}{x - x_i} w_i$$

By substituting the above into the Lagrange polynomial (3.1) we get:

$$L(x) = \sum_{i=1}^{n} y_i l_i(x) = \sum_{i=1}^{n} y_i \frac{l(x)}{x - x_i} w_i$$

The value $l(x)$ is independent of the sum counter i and therefore can be brought outside of the sum:

$$L(x) = l(x) \sum_{i=1}^{n} \frac{w_i}{x - x_i} y_i \tag{3.3}$$

This method requires $\mathcal{O}(n^2)$ operations to precompute the weights w_j and then $\mathcal{O}(n)$ for the evaluation of L which is a major improvement compared to the $\mathcal{O}(n^2)$ operations for calculating l_j using the standard Lagrange form. The polynomial l can also be precomputed in $\mathcal{O}(n)$ operations independently from the sum which produces the dominant operational complexity.

The formula (3.3) is also known as the **Improved Lagrange** Interpolation formula. [9]

3.3.2 Barycentric Lagrange Interpolation

The Barycentric Lagrange Interpolation formula is similar to the improved interpolation formula but contains more improvements and more specifically, it avoids the evaluation of $l(x)$. For any x, it holds that:

$$\sum_{i=1}^{n} l_i(x) = 1$$

because the constant function $B(x) = 1$ is the unique polynomial that interpolates the points $\{(x_1, 1), (x_2, 1), (x_3, 1), ..., (x_n, 1)\}$. Thus, the formula (3.6) can be simplified by setting $L(x) = \dfrac{L(x)}{B(x)}$.

$$L(x) = \frac{l(x) \sum_{i=1}^{n} \dfrac{w_i}{x - x_i} y_i}{l(x) \sum_{i=1}^{n} \dfrac{w_i}{x - x_i}}$$

where the denominator is the Lagrange interpolating polynomial for the polynomial B. Cancelling out the polynomials l we get:

$$L(x) = \frac{\sum_{i=1}^{n} \dfrac{w_i}{x - x_i} y_i}{\sum_{i=1}^{n} \dfrac{w_i}{x - x_i}} \tag{3.4}$$

This is known as the true form of the **Barycentric Interpolation** formula.

It is trivial to observe that the numerator and the denominator look really similar. Indeed, the numerator's value may also be utilised to calculate the denominator, so significantly reducing the computing complexity. It is significant to observe that the difficulty of including a new node x_{n+1} in the interpolation is influenced by the following factors:

– The computation of the n weights, $w_1, w_2, ..., w_n$. These weights have to be divided with $x_i - x_{n+1}$ which leads to $\mathcal{O}(n)$ operations totally.

– The computation of the new weight w_{n+1} which requires another $\mathcal{O}(n)$ operations.

Therefore, the total update cost for the Barycentric interpolation is $\mathcal{O}(n)$ while the cost for the standard Lagrange method is $\mathcal{O}(n^2)$. [9]

3.3.3 A numerical example

Having explained how the Barycentric Interpolation formula is derived, an example with a small-degree polynomial will be shown. Let the polynomial:

$$P(x) = 97x^4 + 86x^3 + 111x^2 + x + 88$$

and five shares : $\{(1, 109), (2, 34), (3, 35), (4, 9), (5, 126)\}$ defined over $\mathbb{F}_{137}$. Assuming the polynomial is unknown, we will interpolate these points using the Barycentric method that was presented above. Since all computations will be performed in finite fields, multiplication with the inverse replaces division. Our first step should be the computation of the weights w_i. Starting off with w_1 we get:

$$w_1 = \frac{1}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)(x_1 - x_5)} = \frac{1}{(1-2)(1-3)(1-4)(1-5)} = \frac{1}{24} = 40$$

Next, w_2 can be calculated as:

$$w_2 = \frac{1}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)(x_2 - x_5)} = \frac{1}{-6} = 114$$

Similarly, we calculate the rest weights and finally:

$$w_1 = 40$$
$$w_2 = 114$$
$$w_3 = 103$$
$$w_4 = 114$$
$$w_5 = 40$$

Now, by looking at (3.7) we can see that the sum $\sum_{i=1}^{n} \frac{w_i}{x - x_i}$ appears in both the numerator and the denominator. Therefore we can compute the numerator and reuse the inverse of the denominator of the resulting fraction to compute the denominator of (3.7). So:

$$\sum_{i=1}^{5} \frac{w_i}{x - x_i} y_i = \frac{w_1 y_1}{x-1} + \frac{w_2 y_2}{x-2} + \cdots + \frac{w_5 y_5}{x-5} = \frac{97x^4 + 86x^3 + 111x^2 + x + 88}{x^5 + 122x^4 + 85x^3 + 49x^2 + 17}$$

This quantity can be used as the numerator of (3.4). Notice that the numerator is the interpolating polynomial P. That is because the denominators of the compound fraction cancel out:

$$P(x) = \frac{\frac{97x^4 + 86x^3 + 111x^2 + x + 88}{x^5 + 122x^4 + 85x^3 + 49x^2 + 17}}{\frac{1}{x^5 + 122x^4 + 85x^3 + 49x^2 + 17}} = 97x^4 + 86x^3 + 111x^2 + x + 88$$

Therefore we ended up with the interpolating polynomial P.

3.4 Newton Interpolation

Another contribution to the field of polynomial interpolation is attributed to Isaac Newton. After the so-called Lagrange Interpolating Polynomial, we will discuss about the **Newton Interpolating Polynomials**.

Given $n+1$ points $\{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\}$[2], the n-degree Newton interpolating polynomial N is calculated as follows:

$$N(x) = \sum_{i=0}^{n} d_i n_i(x)$$

where $n_i(x)$ is also known as the i-th **Newton basis polynomial** and is given by the following product:

$$n_i(x) = \prod_{j=0}^{i-1} (x - x_j)$$

with $n_0(x) = 1$.

It is observed that N may be expressed as a linear combination of the basis polynomials n_i with their coefficients being d_i[3]. The computation of these coefficients is accomplished by a technique referred to as *divided differences*. Although the Newton and Lagrange interpolation methods may appear similar at first glance, it is important to note that they possess distinct characteristics and should be selected based on their suitability for a certain use case. In subsequent analysis, we will explore the significant benefit the divided differences provide to the Newton technique in particular applications. First, let's describe how does the divided differences method work.

3.4.1 The Divided Differences method

Let's rewrite the polynomial N with the basis polynomials expanded. [8]

$$N(x) = d_0 + d_1(x - x_0) + d_2(x - x_0)(x - x_1) + \cdots + d_n(x - x_0)(x - x_1)\dots(x - x_n) \tag{3.5}$$

We know that the shares (x_i, y_i) satisfy $N(x_i) = y_i$. Thus, by substituting to (3.3) we have:

$$N(x_0) = d_0 = y_0$$
$$N(x_1) = d_0 + d_1(x_1 - x_0) = y_1$$

Notice that we can solve for d_1 using the last equation:

$$d_1 = \frac{y_1 - d_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0}$$

[2]Setting the share index to 0 does not necessarily imply that $x_0 = 0$. It denotes the first share.

[3]d_i have nothing to do with the polynomial coefficients.

We calculate d_2 in a similar manner:

$$N(x_2) = d_0 + d_1(x_2 - x_0) + d_2(x_2 - x_0)(x_2 - x_1) = y_2$$

With some rearrangement, we get:

$$d_2 = \frac{\frac{y_2 - y_1}{x_2 - x_1} - \frac{y_1 - y_0}{x_1 - x_0}}{x_2 - x_0} \tag{3.6}$$

Notice how each value d_i is written in terms of d_{i-1}. This observation is fundamental to understand the advantage of the Newton method that will be discussed later. The divided differences method is characterized by a recursive behavior so we can generalize the formula for d_n as:

$$d_n = \frac{\frac{y_n - y_{n-1}}{x_n - x_{n-1}} - d_{n-1}}{x_n - x_0} \tag{3.7}$$

For a more formal definition, let $D[x_0, x_1]$ be the divided differences table computed over x_0 and x_1.:

$$D[x_0, x_1] = \frac{y_1 - y_0}{x_1 - x_0}$$

Thus we can rewrite (3.4) using the recursive formula below.

$$D[x_0, x_1, x_2] = \frac{D[x_1, x_2] - D[x_0, x_1]}{x_2 - x_0}$$

and more generally:

$$D[x_0,\ x_1,\ x_2,\ \ldots,\ x_n] = \frac{D[x_1, x_2,\ \ldots,\ x_n] - D[x_0,\ x_1,\ \ldots,\ x_{n-1}]}{x_n - x_0}$$

3.4.2 Faster computation for new shares

As mentioned above, to compute the Newton polynomial, one have to calculate the divided differences $d_0, d_1,\ \ldots,\ d_n$ between the shares (x_i, y_i). From the relation (3.5), we know that d_n can be written in terms of d_{n-1}. Equivalently, the divided difference d_{n+1} is expressed in terms of d_n which means that instead of performing two recursive steps in the numerator, one can reuse the n previously computed divided differences for computing the $n+1$-th divided difference. By doing that, the only thing left to be calculated is $D[x_1, x_2,\ \ldots,\ x_n]$ as $D[x_0,\ x_1,\ \ldots,\ x_{n-1}]$ is pre-computed.

Speaking in a cryptographic context and more specifically in the context of SSS, suppose ten users want to interpolate their shares to obtain the secret. They compute the divided differences $d_0, d_1, , \ldots,\ d_9$ and successfully reconstruct $N(x)$. Suddenly, a new user joins the secret sharing scheme and wants to know the secret too. Normally, the eleven users would have to compute the divided differences all over again but due to the properties of divided differences, this is not mandatory. With the new addition, the divided differences $d_0, d_1, , \ldots,\ d_9$ remain unchanged and the new divided difference d_{10} can be calculated based on those.

In contrast, by looking at the calculation of the Lagrange Basis Polynomial l_i (3.2) it's clear that with the addition of new shares, the basis polynomials have to be recomputed all over again which is something that makes Newton interpolation more suitable whenever newAs previously stated, in order to compute the Newton polynomial, it is necessary to calculate the divided differences $d_0, d_1, \ \ldots, \ d_n$ based on the given data points (x_i, y_i). Based on equation (3.7), we know that the expression for d_n may be expressed in terms of d_{n-1}. Similarly, the divided difference d_{n+1} can be formulated in relation to d_n, so allowing for the reuse of the n previously calculated divided differences in the computation of the $n+1$-th divided difference, instead of executing two recursive steps in the numerator. After performing the aforementioned steps, the only remaining calculation is the determination of $D[x_1, x_2, \ \ldots, \ x_n]$, since $D[x_0, \ x_1, \ \ldots, \ x_{n-1}]$ has already been computed in prior.

In the cryptography field, particularly within the framework of Secret Sharing Schemes (SSS), let us consider a scenario where 10 users seek to collectively reconstruct their individual shares in order to gain the secret. The divided differences $d_0, d_1, \ldots, d_9$ are computed and used to correctly rebuild the function $N(x)$. Assumt that at some point, an additional participant enrols in the secret sharing protocol and wants to obtain the secret too. Typically, the computation of divided differences would need to be repeated for each of the eleven users. However, due to the inherent characteristics of divided differences, this is not mandatory. After the introduction of the new addition, the divided differences $d_0, d_1, \ldots, d_9$ remain unaltered, while the computation of the new divided difference d_{10} may be derived from these existing values.

In contrast, upon examining the computation of the Lagrange Basis Polynomial l_i (3.2), it becomes evident that the introduction of new shares necessitates the recomputation of the basis polynomials from scratch. This characteristic renders Newton interpolation more appropriate in scenarios where new parties join the secret sharing scheme. users join the secret sharing scheme.

3.4.3 A simple example

Suppose the dealer of the secret sharing scheme constructs the following polynomial defined over $\mathbb{F}_{179}$:

$$P(x) = 62x^5 + 55x^4 + 14x^3 + 93x^2 + 165x + 16$$

The dealer distributes the shares to the users $(1, 47), (2, 114), (3, 125), (4, 78), (5, 162), (6, 142)$ and they want to obtain the secret 16 using the Newton interpolation method. The first step is

to calculate the divided differences $d_0, d_1, d_2, d_3, d_4, d_5$. [4]

$$
\begin{aligned}
d_0 &= N(x_0) = y_0 = 47 \\
d_1 &= \frac{y_1 - d_0}{x_1 - x_0} = \frac{114 - 47}{2 - 1} = 67 \\
d_2 &= \frac{\frac{y_2 - y_1}{x_2 - x_1} - d_1}{x_2 - x_0} = \frac{\frac{125 - 114}{2 - 1} - 67}{3 - 1} = \frac{-56}{2} = 151
\end{aligned}
$$

Similarly we compute the rest divided differences.

$$
\begin{aligned}
d_3 &= 119 \\
d_4 &= 90 \\
d_5 &= 62
\end{aligned}
$$

Now we calculate the Newton polynomial as:

$$N(x) = 47 + 67(x-1) + 151(x-1)(x-2) + \cdots + 62(x-1)(x-2)(x-3)(x-4)(x-5)$$

We can use Sage to do some algebra and derive the final form of N.

```
sage: p = 179
sage: P.<x> = PolynomialRing(GF(p))
sage: 47 + 67*(x-1) + 151*(x-1)*(x-2) + 119*(x-1)*(x-2)*(x-3) +
 ↪  90*(x-1)*(x-2)*(x-3)*(x-4) + 62*(x-1)*(x-2)*(x-3)*(x-4)*(x-5)
62*x^5 + 55*x^4 + 14*x^3 + 93*x^2 + 165*x + 16
```

Listing 10: Computation of the Newton polynomial.

Indeed, we can verify that $N(x) = P(x)$ thus the users obtain the secret $s = 16$.

3.5 Discrete and Fast Fourier Transform in Finite Fields

As already discussed, the purpose of this thesis is to compare the performance of the interpolation methods described in this chapter. Prior to conducting any experiment, it is crucial to provide a concise overview of the backdrop for each algorithm. The final technique mentioned in the enumeration is interpolation through the use of the Fast Fourier Transform (FFT), which is a derivative of the Discrete Fourier Transform (DFT). Complex numbers are frequently employed as the values in Fast Fourier Transform (FFT) and Discrete Fourier Transform (DFT). This thesis aims to explore the generalisation of the aforementioned approaches and their application to rings and finite fields. A comprehensive understanding of the foundational concepts discussed in the previous chapter is necessary for comprehending this particular section.

[4] Again, we work in $\mathbb{F}_p$ so division is equivalent to multiplication with the multiplicative inverse and negative numbers $-k$ are equivalent to $-k \ (mod\ p)$.

3.5.1 Discrete Fourier Transform over Finite Fields

Until now, we represent shares as the tuple (x_i, y_i), where x_i is an integer and $y_i = P(x_i)$. Usually, x_i is the ID of each shareholder so it increases by one each time. Then, as already discussed, the sequence of $\{y_0, y_1, \ldots, y_{n-1}\}$ is used to compute the interpolating polynomial. As the name implies, the Fourier Transform methods transform this sequence to another sequence and more specifically to:

$$\{F_0, F_1, \ldots, F_{n-1}\}$$

where $F_i = P(\omega^i)$ and $\omega \in \mathbb{F}_p$ is a primitive n^{th} root of unity. Therefore the polynomial is now computed by the shares:

$$(\omega^i, P(\omega^i)) = (\omega^i, F_i)$$

Speaking with linear algebra terms, recall the Vandermonde matrix that was formed by $x_0, x_1, \ldots, x_{n-1}$. [10] [11]

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

DFT transforms the above linear relation to the following using:

$$\begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{n-1} \end{pmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)^2} \end{bmatrix} \cdot \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Notice that in the transformed relation, $x_i = \omega^i$. We shall call the Vandermonde matrix of $1, \omega, \omega^2, \ldots, \omega^{n-1}$, the **DFT matrix**. Alternatively, we can write this algebraically as:

$$F_k = \sum_{i=0}^{n-1} a_i \omega^{ki}$$

With some rearrangement, we deduce the inverse DFT (IDFT) formula which aims at recomputing the coefficients a_i.

$$a_i = \frac{1}{n} \sum_{k=0}^{n-1} F_k \omega^{-ik}$$

This formula follows from the symmetric properties of the primitive n^{th} roots of unity that were discussed in section 2.2.3. In fact, the existence of these properties is what makes the Fast Fourier Transform interpolation faster than the rest methods and most essentially the core motivation behind Fourier Transform. Equivalently, IDFT can be represented using matrix multiplication too.

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{n-1} \end{pmatrix}$$

Similarly to the DFT matrix, we will denote the matrix with the inverse powers of ω as the **IDFT matrix**.[11]

3.5.2 Fast Fourier Transform over Finite Fields

One can notice that computing IDFT matrix, is similar to computing the Vandermonde matrix that was presented at section 3.1.1 which requires $\mathcal{O}(n^2)$ operations.

The DFT method can in fact be calculated faster, making it a Fast Fourier Transform. The trick for this is based on the symmetric properties of the n^{th} roots of unity and the algorithm presented by Cooley and Tukey (CT) in their paper. [10].

Cooley & Tukey Algorithm

The use of this algorithm reduces the complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(nlogn)$ operations. The core idea is that it reduces the initial DFT problem to smaller DFT problems of half size, recursively using divide-and-conquer. [12] For the sake of demonstration and showcasing the algorithm's optimal performance, this section will choose the number of shares n to be a power of two; $n = 2^k$. [5] The same method can be generalized for other values of n too, such as $n = p^k$ or $n = p_1 p_2 p_3 \dots p_k$ but these choices don't highlight the big advantage of the CT algorithm. Let the following polynomial P.

$$P(x) = \sum_{i=0}^{n-1} a_i x^i$$

P can be splitted into two polynomials; one with the even-indexed coefficients of P, say P_0, and one with the odd-indexed coefficients, say P_1.

$$\begin{aligned} P(x) &= (a_0 + a_2 x^2 + \ \dots\ a_{n-2} x^{n-2}) + (a_1 x + a_3 x^3 + \ \dots\ + a_{n-1} x^{n-1}) \\ P(x) &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} x^{2i} + \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} x^{2i+1} \\ P(x) &= \sum_{i=0}^{\frac{n}{2}-1} a_{2i} (x^2)^i + x \sum_{i=0}^{\frac{n}{2}-1} a_{2i+1} (x^2)^i \end{aligned}$$

Notice that the first sum operand is the $(n/2)$-degree polynomial P_0 evaluated at x^2 and the other operand is the $(n/2)$-degree polynomial P_1 evaluated at x^2. Therefore we get that P can be expressed as:

$$P(x) = P_0(x^2) + xP_1(x^2) \tag{3.8}$$

[5] Sometimes we might refer to the number of shares as the size of the FFT.

We repeat this procedure for each of P_0 and P_1 until the FFT size is 1 and cannot be splitted furthermore. Below there is a symbolic example when $n = 8$.

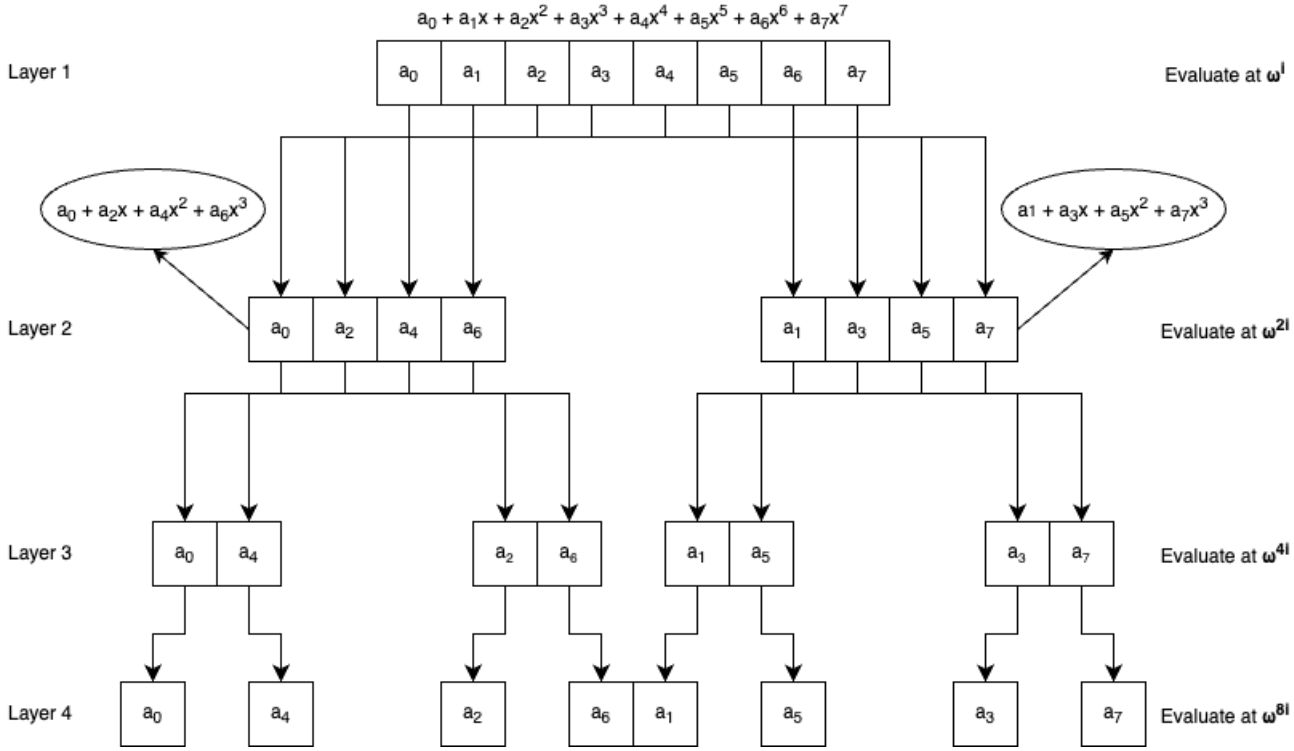

Figure 3.1: CT Algorithm - The Polynomial Divide-and-Conquer Part for $n = 8$.

Notice that by substituting x^2 into the two polynomials of layer 2 and multiplying the right-branch polynomial by x, we get the polynomial P of layer 1. Similarly, this relation holds at every layer (see (3.8)).

Therefore the original problem of evaluating P at ω^i [6], has been reduced to evaluating the layer 2 polynomials at $(\omega^i)^2 = \omega^{2i}$.

Suppose we want to evaluate the original polynomial at ω^i; that is: $P(\omega^i) = a_0 + a_1\omega^i + a_2\omega^{2i} + \ \ldots\ + a_7\omega^{7i}$. Let's see how (3.8) can be utilized to recursively reduce the evaluation problem into smaller-sized FFTs.

$$P(\omega^i) = P_0(\omega^{2i}) + \omega^i P_1(\omega^{2i}) \quad (layer\ 1)$$

To evaluate P_0 and P_1 at ω^{2i} we compute:

$$P(\omega^{2i}) = P_0(\omega^{4i}) + \omega^{2i} P_1(\omega^{4i}) \quad (layer\ 2) \tag{3.9}$$

Notice how polynomials in layer 2 are reduced to degree $n/2$. Finally, to evaluate P_0 and P_1 at ω^{4i} we compute:

$$P(\omega^{4i}) = P_0(\omega^{8i}) + \omega^{4i} P_1(\omega^{8i}) \quad (layer\ 3) \tag{3.10}$$

[6]The original problem is the computation of the Fourier Transformed shares $(\omega^i, (P(\omega^i))$. The FFT size at layer 1 is 8 while the FFT size for each polynomial in the 2^{nd} layer is 4 and so on.

Evaluating the smaller-sized FFTs recursively and substituting the evaluations into the polynomial of the above layer, we eventually obtain the shares of the original polynomial.

$$\{(\omega^0, P(\omega^0)), (\omega^1, P(\omega^1)),\ \ldots,\ (\omega^7, P(\omega^7))\}$$

3.5.3 A numerical example

In this section, only the polynomial evaluation step will be performed. The polynomial interpolation part will be shown in the next section in which we will discuss about Inverse Fast Fourier Transform (IFFT).

Let the safe prime $p = 2^3 * 29 + 1 = 233$. Consider the finite field $\mathbb{F}_p$ with generator $g = 3$. Suppose we have the polynomial:

$$P(x) = 25 + 180x + 118x^2 + 193x^3 + 144x^4 + 78x^5 + 230x^6 + 157x^7$$

and we want to calculate the $n = 8$ shares $(\omega^0, P(\omega^0)), (\omega^1, P(\omega^1)),\ \ldots,\ (\omega^7, P(\omega^7))$, where $\omega = g^{\frac{p-1}{n}} \pmod p = 3^{29} \pmod{233} = 221$ is an 8^{th} primitive root of unity. The naive $\mathcal{O}(n^2)$ approach is to substitute each power of omega into P. However, the FFT approach will be followed that computes these shares in $\mathcal{O}(nlog_2n)$ time. First, the polynomial is splitted as shown in Figure 3.1. Below, some layers will be evaluated to understand how the recursive algorithm works. Starting with layer 4, the 0-degree polynomial $P(x) = a_i$ is constant and it evaluates to a_i. Thus,

$$\begin{aligned} a_0 &\Rightarrow P(\omega^{8*0}) = 25 \\ a_4 &\Rightarrow P(\omega^{8*0}) = 144 \\ &\vdots \\ a_7 &\Rightarrow P(\omega^{8*0}) = 157 \end{aligned}$$

Now, in layer 3, the FFT size is 2 so we will evaluate the polynomials in the left branch at two powers of ω: $\omega^{4*0} = 1$ and $\omega^{4*1} = \omega^4 = 232$. [7]

$$\begin{aligned} [25, 144] &\Rightarrow P(\omega^{4*0}) = P_0(\omega^{8*0}) + \omega^{4*0}P_1(\omega^{8*0}) = 25 + 144 = 169 \\ [25, 144] &\Rightarrow P(\omega^{4*1}) = P_0(\omega^{8*1}) + \omega^{4*1}P_1(\omega^{8*1}) = 25 - 144 = 114 \quad [8] \\ [118, 230] &\Rightarrow P(\omega^{4*0}) = \ \ldots\ = 118 + 230 = 115 \\ [118, 230] &\Rightarrow P(\omega^{4*1}) = \ \ldots\ = 118 - 230 = 121 \\ &\vdots \end{aligned}$$

FFT size being 2 means that we need to evaluate the polynomials at two shares; $[169, 114]$ for the polynomial $[25, 144]$ and $[115, 121]$ for the polynomial $[118, 230]$. Similarly, we proceed

[7] The formula (3.10) is used here.

[8] Due to the symmetric properties of the primitive n^{th} roots of unity we know that:

$$\omega^i = -\omega^{\frac{n}{2}+i} \pmod p$$

with layer 2. The size of the FFT is 4 so the polynomial in the left branch will be evaluated at four powers of ω: $\omega^{2*0}, \omega^{2*1} = 144, \omega^{2*2} = 232$ and $\omega^{2*3} = 1$:[9]

$$\begin{aligned}
[25, 118, 144, 230] &\Rightarrow P(\omega^{2*0}) = P_0(\omega^0) + \omega^0 P_1(\omega^0) = 169 + 115 = 51 \\
[25, 118, 144, 230] &\Rightarrow P(\omega^{2*1}) = P_0(\omega^4) + \omega^2 P_1(\omega^4) = 114 + 144 * 121 = 63 \\
[25, 118, 144, 230] &\Rightarrow P(\omega^{2*2}) = P_0(\omega^8) + \omega^4 P_1(\omega^8) = 169 - 115 = 54 \\
[25, 118, 144, 230] &\Rightarrow P(\omega^{2*3}) = P_0(\omega^{12}) + \omega^6 P_1(\omega^{12}) = P_0(\omega^4) - \omega^2 P_1(\omega^4) = \\
&= 114 - 144 * 121 = 165
\end{aligned}$$

Thus the evaluations of the polynomial $[25, 118, 144, 230]$ are:

$$\begin{aligned}
P(\omega^0) &= 51 \\
P(\omega^2) &= 63 \\
P(\omega^4) &= 54 \\
P(\omega^6) &= 165
\end{aligned}$$

By doing the same for the right branch, we can finally evaluate the original polynomial at the powers of ω^i and therefore compute the shares:

$$(1, 193), (221, 7), (144, 87), (136, 91), (232, 142), (12, 119), (89, 21), (97, 6)$$

The following SageMath code proves that these shares are valid.

```
sage: [(omega**i, P(omega**i)) for i in range(n)]
[(1, 193),
 (221, 7),
 (144, 87),
 (136, 91),
 (232, 142),
 (12, 119),
 (89, 21),
 (97, 6)]
```

Listing 11: Computation of the FFT shares using the root of unity ω.

Therefore by setting $i = 4$ and $n = 8$, we get that:

$$\omega^4 = -\omega^{4+4} = -\omega^8 = -1 \pmod{p}$$

[9] The formula (3.9) is used here.

Note that at the original polynomial P, the evaluations $P(\omega^0)$ and $P(\omega^4)$ are symmetric. So as $P(\omega^1)$ and $P(\omega^5)$ and so on. Therefore iterating $n/2$ times and performing 2 evaluations each time is enough since the rest powers of ω are related to the first $n/2$. In the first iteration, $P(\omega^0)$ with $P(\omega^4)$ will be computed, in the second, $P(\omega^1)$ with $P(\omega^5)$, in the third, $P(\omega^2)$ with $P(\omega^6)$ and in the last, $P(\omega^3)$ with $P(\omega^7)$.

3.5.4 Inverse Fast Fourier Transform

The Inverse Fourier Transform is the problem of determining the left hand side coefficient vector of the following relation:

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \frac{1}{n} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega^{-1} & \omega^{-2} & \cdots & \omega^{-(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{-(n-1)} & \omega^{-2(n-1)} & \cdots & \omega^{-(n-1)^2} \end{bmatrix} \cdot \begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{n-1} \end{pmatrix}$$

The algorithm is similar to that of the standard Fourier Transform but now the Vandermonde matrix consists of negative powers of ω so the task is to apply the FFT that was described above but this time evaluating at negative powers. As a final step, IFFT divides the inner product of each multiplied vector with n to obtain the coefficients. Continuing our previous example, we attempt to reconstruct the original polynomial P that made up the shares:

$$(1, 193), (221, 7), (144, 87), (136, 91), (232, 142), (12, 119), (89, 21), (97, 6)$$

First, the following polynomial is formed:

$$S(x) = 193 + 7x + 87x^2 + 91x^3 + 142x^4 + 119x^5 + 21x^6 + 6x^7$$

The coefficient of x^i equals to the evaluation of P at ω^i.

Then the process is similar to that one for computing the shares; the Cooley-Tukey algorithm is applied with the only difference being the evaluation at negative powers of ω. More specifically, polynomials at layer 4 are now evaluated at ω^{-8i}, layer 3 at ω^{-4i}, layer 2 at ω^{-2i} and layer 1 at ω^{-i}. At the end of the FFT, we should be left with the following evaluations:

$$\{200, 42, 12, 146, 220, 158, 209, 91\}$$

However these are not our original coefficients, we still have to divide by n. Doing that, we get:

$$\{25, 180, 118, 193, 144, 178, 230, 157\}$$

which are indeed the coefficients of the original polynomial P. The values above can be verified with the following SageMath code.

```
sage: F = GF(p)
sage: PR.<x> = PolynomialRing(F)
sage: shares = [193, 7, 87, 91, 142, 119, 21, 6]
sage: S = PR(shares)
sage: evals = [S(omega**(-i)) for i in range(n)] ; evals
[200, 42, 12, 146, 220, 158, 209, 91]
sage: coeffs = [ev / n for ev in evals] ; coeffs
[25, 180, 118, 193, 144, 78, 230, 157]
```

Listing 12: IFFT - Proof of Correctness.

3.5.5 Time Complexity Analysis

Let $T(n)$ be a function that calculates the time the FFT method takes to interpolate n shares. We will try to compute the time complexity of the FFT method.

From Fig. (3.1), we see that at each layer, the original problem is splitted into two subproblems of size $\frac{n}{2}$. Moreover, each layer, requires $n-1$ additions and $n-1$ multiplications but to make calculations easier, we round to n. That is; T can be written recursively as:

$$T(n) = 2 \cdot T(\frac{n}{2}) + 2 \cdot n \tag{3.11}$$

At the second call of the algorithm, the problem is splitted in two subproblems of size $\frac{n}{2}$. By substituting n with $\frac{n}{2}$, we compute $T(\frac{n}{2})$ as:

$$T(\frac{n}{2}) = 2 \cdot T(\frac{n}{4}) + n$$

and substitute it back to (3.11):

$$\begin{aligned} T(n) &= 2 \cdot (2 \cdot T(\frac{n}{4}) + n) + 2 \cdot n = \\ &= 2^2 \cdot T(\frac{n}{2^2}) + 2 \cdot 2 \cdot n \end{aligned}$$

Similarly, for the third call:

$$T(n) = 2^3 \cdot T(\frac{n}{2^3}) + 2 \cdot 3 \cdot n$$

After k calls, the algorithm will have reached the bottom layer (in our case layer 4) in which the time cost to solve the FFT is constant (i.e. $T(1)$). Therefore we deduce that $\frac{n}{2^k} \to 1$. Solving for k we get $2^k = n \Rightarrow k = log_2(n)$. Then, after k calls, we have:

$$\begin{aligned} T(n) &= 2^k \cdot T(1) + 2 \cdot k \cdot n = \\ &= n \cdot T(1) + 2 \cdot log_2(n) \cdot n = \\ &= \mathcal{O}(2 \cdot n \cdot log_2(n) + n) = \\ &= \mathcal{O}(n \cdot log_2(n)) \end{aligned}$$

This completes the proof of the FFT time complexity. It is important to note that dividing a problem in halves per call, is an indicator that the algorithm has logarithmic nature.

Chapter 4

Experiments and Results

After analysing the necessary mathematical foundations, this section will present all the aforementioned information implemented with Python using SageMath 9.8. This paper will primarily focus on presenting a selection of experiments and their corresponding results related to interpolation methods. These experiments aim to demonstrate the impact of various factors on the effectiveness and performance of each approach. The rationale for this contribution stems from the dearth of existing literature on the subject when considering its integration with secret sharing schemes.

In conclusion, in our GitHub repository, we present code that calculates FFT for $n = 2^k, 3^k$. The algorithm can be extended to cases when n is a power of any prime number, i.e. $n = p^k$ but it is out of the scope of this thesis so we skip it. More specifically, the experiments conducted and the respective findings will be structured and presented as follows:

- The efficiency and the performance of the FFT method will be measured and compared to unoptimized standard Lagrange, Newton and Barycentric Lagrange for increasing number of shares. Due to computational overhead, results of standard Lagrange will be presented seperately with smaller number of shares without loss of generality.

- The Newton's method will be compared to the Barycentric Lagrange's under the context of adding new shares to the Shamir Secret Sharing scheme. The choice of these two methods rises from their capability of precomputing some values for n users which can be reused when a new user is added to the scheme. The FFT method won't be tested here cause of its limitations regarding the number of shares n. Also, standard Lagrange is omitted since it has no precomputation advantages and the basis polynomials have to be recomputed each time from scratch.

- Finally, we utilize the optimized formulas of the standard Lagrange, Newton, Barycentric and compare them to FFT. In secret sharing schemes, using interpolation to interpolate the entire polynomial is not efficient as all the coefficients except for the constant term a_0 remain unused. This use case shows the efficiency of each algorithm when only a_0 is reconstructed.

The source code for reproducing the results of the following experiments, can be found in our GitHub. [4]

4.1 Domain Parameters

Since these interpolation methods will be applied to SSS over finite fields, we have to define the finite field we will work with. The domain parameters are some fixed system-wide parameters that will be used for all the experiments. Due to the limitations of FFT, the parameters are affected as following:

- The first step is to construct the finite field $\mathbb{F}_p$. As FFT does not work with a random p, we need to construct it manually. To compute the primitive n^{th} roots of unity, the division $\frac{p-1}{n}$ has to be an integer, or in other words, n must divide $p-1$. We select p to be a safe prime of the form $p = n_{max}q + 1$, where q is a random 28-bit prime number. We set the value $n_{max} = 2^{100}$ as an upper bound as in our use cases, n will never be that large. Therefore the value of p is:

  $$p = 2^{100} * 135783853 + 1 = 172126482756751667503106328023403593729$$

  The size of p is arbitrary as it does not affect the performance of the interpolation methods.

- Another important domain parameter that is crucial mostly for FFT is the group generator g. We can find g using any method that was described in the previous sections. Let $g = 3$.

In each use case, a 128-bit AES key will be set as the constant term of the polynomial and will be treated as the secret s. The task is to distribute this secret among the users of the secret-sharing scheme so that eventually they all agree on the same key.

4.2 Comparing FFT against unoptimized Lagrange and Newton

4.2.1 Setting up the environment

Regarding the first use case, due to the FFT limitations, we must adjust some additional settings.

- The number of shares n should be a power of two. More specifically, we set $n_{min} = 2^7$ and $n_{max} = 2^{13}$ as these choices highlight the differences in performance better.

- The same value for ω cannot be used for each different n as then, the Cooley-Tukey algorithm would require slight modifications which is not preferred for our purposes. This issue could be solved if the roots of unity were precomputed but in this use case we work with the unoptimized versions of each algorithm.

- For the same reason as above, for each value of n, the interpolating polynomial's degree should be $n - 1$.

As a result, we cannot fix the interpolating polynomial and an arbitrary $(n-1)$-degree polynomial has to be generated for each n.

Method \ n	128	256	512	1024	2048	4096	8192
Newton	0.0363	0.1922	0.4111	1.7848	6.3193	25.5859	124.896
Barycentric	0.1241	0.334	0.8126	2.2027	6.9594	26.1571	123.0452
FFT	0.0032	0.0045	0.0133	0.0288	0.0527	0.1269	0.2577

Table 4.1: Performance of FFT and unoptimized Newton, Barycentric.

Each table cell represents how many seconds taken for the interpolation to finish.

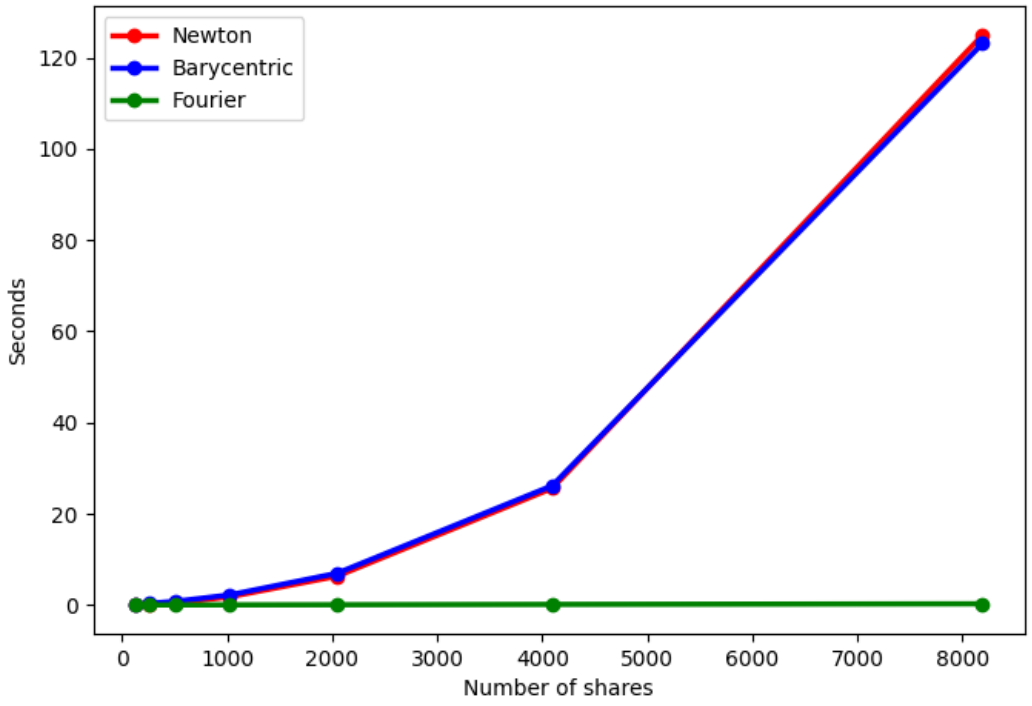

Figure 4.1: Unoptimized Newton, Barycentric and FFT diagram.

The diagram shows that before $n = 4096$, Newton and Barycentric perform almost identically but then Newton's time complexity starts to increase rapidly. These results justify the theoretical complexities which were discussed above. The difference between $\mathcal{O}(nlog_2(n))$ and $\mathcal{O}(n^2)$ has become clear as the graph of the FFT is logarithmic and closer to constant $\mathcal{O}(1)$ and its rate of change is infinitesimal.

For the standard Lagrange, we set $n_{min} = 100$ and $n_{max} = 1000$ and increase n by 100 in each iteration. Since there are no FFT limitations, we can fix the 5-degree polynomial:

```
F(x) = 1041582906288760482479960851109266032 62*x^5 +
↪  5095337582493038089285243387204145310 1*x^4 +
↪  9169005641289508455928477487225920554 3*x^3 +
↪  1332459484411800228004461289446100454 62*x^2 +
↪  7461290490139153796031110805513264762 5*x +
↪  4128697931302607515564642177456070891 2
```

The results are shown below.

n	Interpolation Time (s)
100	0.4622
200	2.0773
300	5.6790
400	10.2764
500	19.4573
600	27.8461
700	41.8228
800	61.8564
900	84.2031
1000	110.2638

Table 4.2: Performance of unoptimized standard Lagrange.

As one can see, by the time $n = 1000$, Lagrange already takes about 2 minutes to finish while the other algorithms take at most 4 seconds.

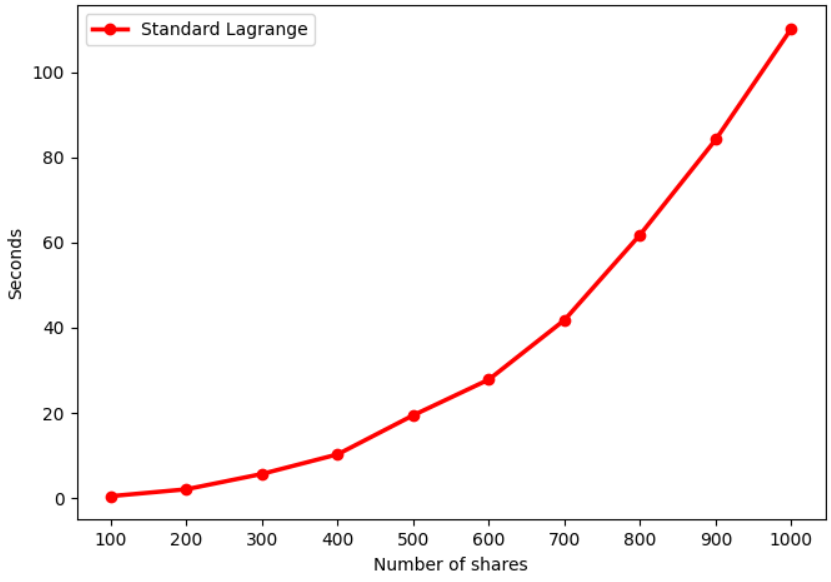

Figure 4.2: Unoptimized Standard Lagrange diagram.

4.3 Evaluating Newton and Barycentric Lagrange performance for new shares

The motive of doing this comparative analysis is to determine the most suitable interpolation method for implementation in a practical secret sharing application. To provide more clarification, let us consider an application that use secret sharing as a means of enabling key exchange. This application is originally composed of a total of 100 users. The Newton and Barycentric techniques utilise precomputed divided differences and weights of length 100. These precomputed values are utilised for each new user added to the system, eliminating the necessity of recalculating everything from scratch.

To illustrate the concept, we assign a value of $n = 3000$ to the variable representing the number of shares. The precomputed values are saved and subsequently augmented with a fresh share until $n = 3010$. Let us consider the scenario in which no values have been precomputed. Presented below are the outcomes of the Newton and Barycentric methods when divided differences and weights are not precomputed. It should be noted that the optimized version of these methods is utilized, which focuses on recovering only the constant term a_0 rather than the entire polynomial.

Method \ n	3000	3001	3002	3003	3004	3005	3006	3007	3008	3009	3010
Newton	7.2775	8.8466	8.9825	10.1798	9.3954	9.9933	10.0459	9.8957	10.6419	9.408	10.0066
Barycentric	3.7977	4.2422	3.8622	3.8829	4.0821	3.9942	3.9467	4.0356	4.1143	3.9807	4.0146

Table 4.3: Performance of Newton and Barycentric without precomputed values.

The diagram is presented below.

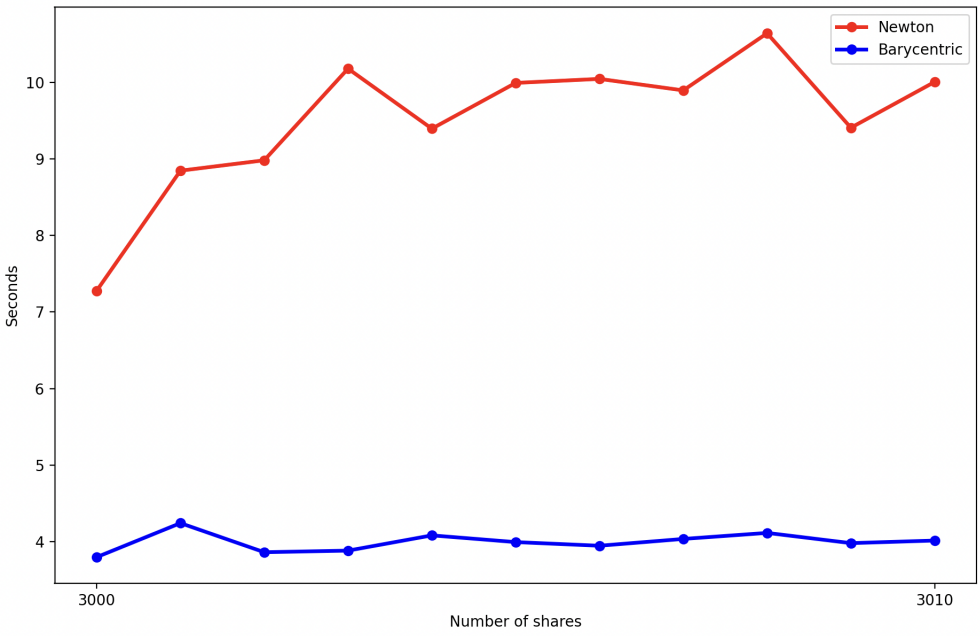

Figure 4.3: Newton and Barycentric without precomputed values.

While both methods perform significantly better than standard Lagrange, Barycentric is clearly faster - something that was expected after our discussions in the previous chapter. Now, let us see the results of Newton and Barycentric with the same number of shares and when precomputed values are used.

Method \ n	3000	3001	3002	3003	3004	3005	3006	3007	3008	3009	3010
Newton	7.5535	0.0995	0.1071	0.0905	0.1031	0.1095	0.1163	0.0928	0.1035	0.0929	0.1121
Barycentric	3.8343	0.0097	0.0099	0.0098	0.0104	0.6093	0.0097	0.0096	0.0102	0.0096	0.0098

Table 4.4: Performance of Newton and Barycentric with precomputed values.

The diagram is presented below.

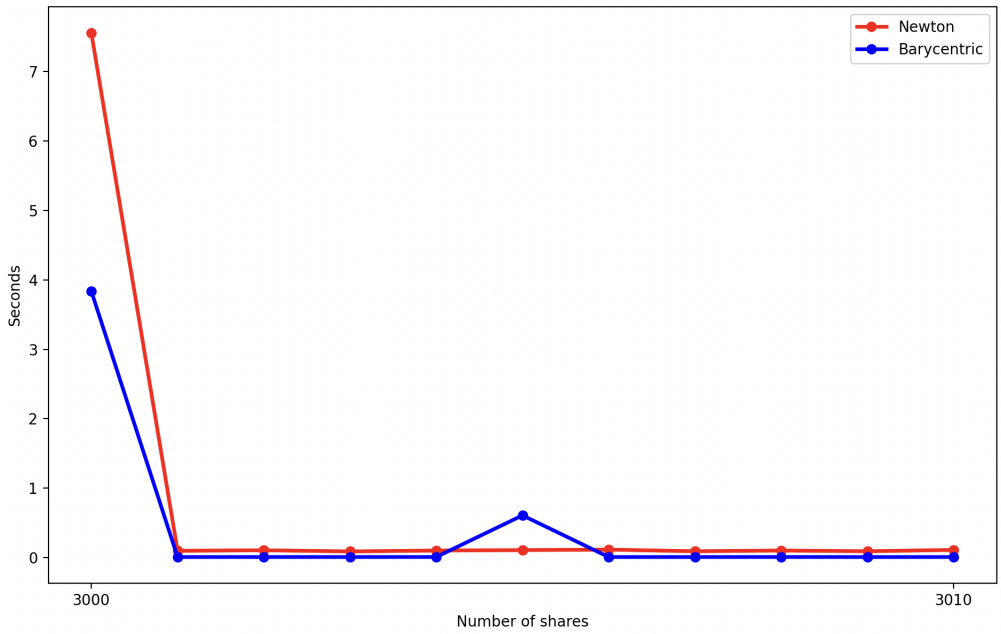

Figure 4.4: Newton and Barycentric with precomputed values.

The peak of both methods occurs at the first value of n. The interpolation with $n + 1$ shares makes use of the precomputed n values, the $n + 2$ shares make use of the precomputed $n + 1$ and so on. The difference in performance is enormous with Barycentric still having a lead over Newton. It is safe to conclude that in any case, it is preferable to apply Barycentric interpolation for a secret sharing application that has gradually increasing number of users.

4.4 Comparing FFT against optimized Lagrange and Newton

The final use case tests the performance of the aforementioned methods when they are optimized to interpolate only the constant term, instead of the entire polynomial. Based on the

results, we are able to discuss and come to conclusions regarding the choice of the right interpolation method for the secret sharing. Again, FFT is part of the case so we set $n_{min} = 2^9$ and $n_{max} = 2^{14}$.

Method \ n	512	1024	2048	4096	8192	16384
Standard Lagrange	0.7805	1.9506	7.8885	32.4151	146.6758	588.1504
Newton	0.3668	1.5633	5.8963	23.9422	108.0476	580.7062
Barycentric	0.159	0.6072	2.4219	10.3676	45.716	163.3128
FFT	0.0152	0.0266	0.0541	0.1152	0.2332	0.5194

Table 4.5: Performance of FFT and optimized standard Lagrange, Newton and Barycentric.

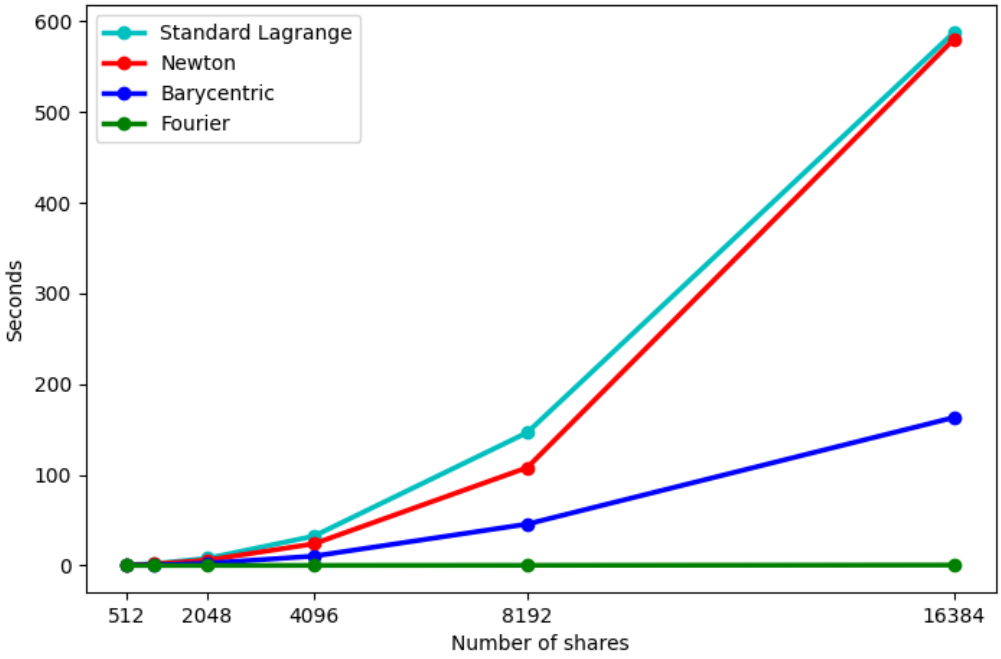

Figure 4.5: Optimized methods performance.

As expected, the complexity of the FFT method dominates the rest with a significant difference.

Chapter 5

Discussion

Having done a benchmarking analysis, it is possible to draw reliable conclusions on the most suitable interpolation method for each specific application. The response depends on the specifics of the application. In the scenario when the number of shares remains constant, it is possible to establish a fixed value that is a power of two and consistently apply the Fast Fourier Transform (FFT) algorithm. However, if the number of shares increases, the FFT approach becomes inapplicable, and alternative approaches such as Optimised Newton and optimised Barycentric Lagrange should be applied. In any case, the complexity of the standard Lagrange method increases rapidly, making it inappropriate for larger applications where many users have to interact or chat with each other.

In summary, the first and third use cases demonstrate the significant difference between the $\mathcal{O}(n \log n)$ Fast Fourier Transform (FFT) algorithm and the remaining $\mathcal{O}(n^2)$ techniques. Nevertheless, it should be noted that the FFT method may not necessarily be the most optimal approach to use. The Cooley-Tukey algorithm's applicability to the second use case is limited by its constraints on the number of shares. The use of the Fast Fourier Transform (FFT) when the value of n is an arbitrary integer, rather than a prime power, has not been studied in this thesis. There are other methods as well such as the Prime Factor Algorithm (PFA, also known as Good Thomas Algorithm) that applies FFT using the Chinese Remainder Theorem when $n = n_1 n_2 n_3 \dots$ with n_i being coprime. Additionally, there exist Rader's method and the split-radix algorithm, which are both very scalable and versatile while maintaining equivalent speed to the Cooley-Tukey algorithm. However, due to their high level of implementation complexity, it is not advisable to use them for secret sharing applications.

Furthermore, the inclusion of a comprehensive, generic Fast Fourier Transform (FFT) implementation capable of accommodating any arbitrary value of n is a potential field of investigation for future work, as this thesis does not primarily focus on the FFT algorithm. However, the repository of this thesis contains SageMath implementations for the case when $n = 2^k$ and 3^k. Last but not least, by comprehending the cyclical characteristics of roots of unity, one may deduce their behaviour for powers of other prime numbers as well.

Chapter 6

Conclusion

In conclusion, this thesis has endeavored to synthesize and highlight its primary contributions. Foremost among these is the comprehensive evaluation of interpolation methods, which represents the central focus of this work. In addition to this significant contribution, we recognize an auxiliary but noteworthy facet: the provision of accompanying code snippets aimed at enhancing the practical understanding of the discussed concepts.

The impetus for this research stemmed from a personal aspiration to create a comprehensive reference, explaining cryptographic concepts not just in theoretical terms but also in practical implementation. This thesis has presented a unique opportunity to fulfill this need. As previously mentioned, the culmination of this endeavor will be the availability of source code that implements all of the discussed interpolation methods.

Furthermore, this work has undertaken an examination of some interpolation techniques, delving into the intricacies of their inner workings. The subsequent presentation of results from various use cases was carefully designed to accentuate critical factors influencing the performance of each method.

Looking forward, there exist promising avenues for future research. One notable prospect involves an in-depth experimental evaluation of the Fast Fourier Transform (FFT) method, which serves as the foundation for several related algorithms. Evaluating these methods within the context of a Secret Sharing scheme, such as Shamir's Secret Sharing (SSS), would be a valuable direction for future investigations. Finally, as an additional research aspect, there are FFT techniques, like the Radix-2 FFT, which offer a more generalized approach than the Cooley-Tukey algorithm without sacrificing speed and performance. The development and integration of a versatile FFT algorithm within secret sharing schemes have the potential to significantly enhance their efficiency and computational speed.

Bibliography

[1] Whitfield Diffie and Martin E. Hellman. *New directions in cryptography*. IEEE, Nov. 1976.

[2] R. L. Rivest, A. Shamir, and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. ACM, Feb. 1978.

[3] Adi Shamir. *How to share a secret*. ACM, Nov. 1979.

[4] Tressos Aristomenis. Github repository : "MSc-Thesis-src", 2023.

[5] Garrett Birkhoff and Saunders Mac Lane. *A survey of modern algebra*. Macmillan Publishing Co., Inc., 4th edition, 1977.

[6] Joseph A. Gallian. *Contemporary Abstract Algebra*. Cengage Learning, 9th edition, Jan. 2016.

[7] Joachim Von Zur Gathen and Jürgen Gerhard. *An Algorithm for the Machine Calculation of Complex Fourier Series*. Cambridge University Press, May 1999.

[8] Mridul Aanjaneya. *Polynomial, Lagrange, and Newton Interpolation*. Rutgers University, Nov. 2017.

[9] Jean-Paul Berrut and Lloyd N. Trefethen. *Barycentric Lagrange Interpolation*. ACM, Jan. 2004.

[10] J.M Pollard. *The Fast Fourier Transform in a Finite Field*. AMS, Apr. 1971.

[11] R. Fateman. *The (finite field) Fast Fourier Transform*. University of California, Spring 2000.

[12] James W. Cooley and John W. Tukey. *An Algorithm for the Machine Calculation of Complex Fourier Series*. AMS, May 1965.