Lagrange: design of a low-level mathematical library for polynomial interpolation over Galois finite fields

Alan Cao
alan@codemuch.tech

May 2018

Abstract

Mathematical operations and algorithms have important for the fields of computation and cryptography. The design and implementation of mathematical libraries in programming languages have become a topic of interest. More and more developers seek to write critical code without wasting time in using large libraries that have a need for auditing and review, or use pre-existing libraries that integrate unused features, presenting performance setbacks. This paper introduces an implementation of a library for one of these mathematical concepts, Lagrange polynomial interpolation, over $Gf(256)$. We introduce the mathematical theory behind polynomial interpolation, and define finite fields and their role in polynomial interpolation. Finally, we implement a mathematical library in the C programming language, deriving source code from previously designed cryptography libraries and discuss the performance and other improvements we can implement.

1 Introduction

The implementation of mathematical operations in modern computing has become prevalent in many sub-fields and concentrations. The power of multi-core processors and random-access memory has enabled high-level programming that can compute complexity in the matter of milliseconds. With the

ease of mathematical computation, it is no wonder that modern cryptography has become so integrated with the fields of mathematics, relying on the properties of numbers to create innovative algorithms and cryptosystems. These can range from hashing algorithms, PRNGs, and other important secure operations.

This paper aims to examine Lagrange polynomial interpolation at a technical approach, rather than one of theory and numerical analysis. Polynomials have been seen to be surprisingly useful and powerful in the fields of computer science, as they display properties that can be applicable to a variety of systems. Polynomial interpolation is one of them, where deconstructed points of a polynomial on a Cartesian xy-coordinate plan can be used to reconstruct the original polynomial. One notable example of practicality of polynomial interpolation can be seen in PolyPasswordHasher [2], which is a password storage scheme designed around Shamir Secret Sharing

When expressing mathematical concepts such as polynomials, one often thinks about operations with rational numbers, in the field of $\mathbb{Q}$. However, rather than using conventional infinite number sets, we will be introducing the concept of *Galois finite fields* instead. As seen later in **Section 2**, several restrictions are placed on Lagrange polynomial interpolation, and Galois finite fields will be used to lift them.

In the context of computational complexity in relation to cryptography, it is always important to consider the performance of computer algorithms. This is often measured through *time* (i.e **big O-notation**), where the quicker the execution time is, the better the algorithm becomes. As mentioned above, modern computation have already yielded fast and abstracted[1] machines. This means that even if an algorithm has been implemented to be much slower and inefficient, it would still be indifferent for users. However, when placing it in the context of cryptography in a modern age of powerful machines, this becomes critical for security and usability. For example, key-distribution servers for large-scale websites must be able to use the quickest and most efficient computational operations to process authentication requests, whether it be sign-ups (key creation) sign-outs (key storage) and logging in (key integrity and validity authentication).

With that said, one important factor to consider when dealing with efficient implementations is the usage of low-level and systems programming

[1] referring to the high-level of abstractions placed on top of programs and computational operations

languages, in this case, C. C is a programming language that works with userspace system calls that directly interact with kernelspace, placing it at the near-bottom of the programming language abstraction hierarchy. Since C provides important features such as rapid compilation time and native bitwise operations, it will be utilized throughout the paper.

2 Lagrange Polynomial Interpolation

Lagrange polynomial interpolation is defined as a technique of reconstructing a polynomial by utilizing derived points of a Cartesian *xy*-coordinate plane. This process involves the creation of *Lagrange polynomials*, which is represented by this formal theorem:

$$L(x) := \sum_{j=0}^{y} y_j l_j(x)$$

where $l_j(x)$ is represented by:

$$l_j(x) := \prod_{0 \leq m \leq k} \frac{x - x_m}{x_j - x_m} = \frac{x - x_0}{x_j - x_0} ... \frac{x - x_{j-1}}{x_j - x_{j-1}} * \frac{x - x_{j+1}}{x_j - x_{j+1}} ... \frac{x - x_k}{x_j - x_k}$$

where $m \neq j$ and $0 \leq j \leq k$.

Let's examine the process that it takes for a polynomial to be destructed into points, and then constructed through Lagrange interpolation.

2.1 Create data points

The polynomial to be constructed:

$$f(x) := 2x^2 + 5x + 8$$

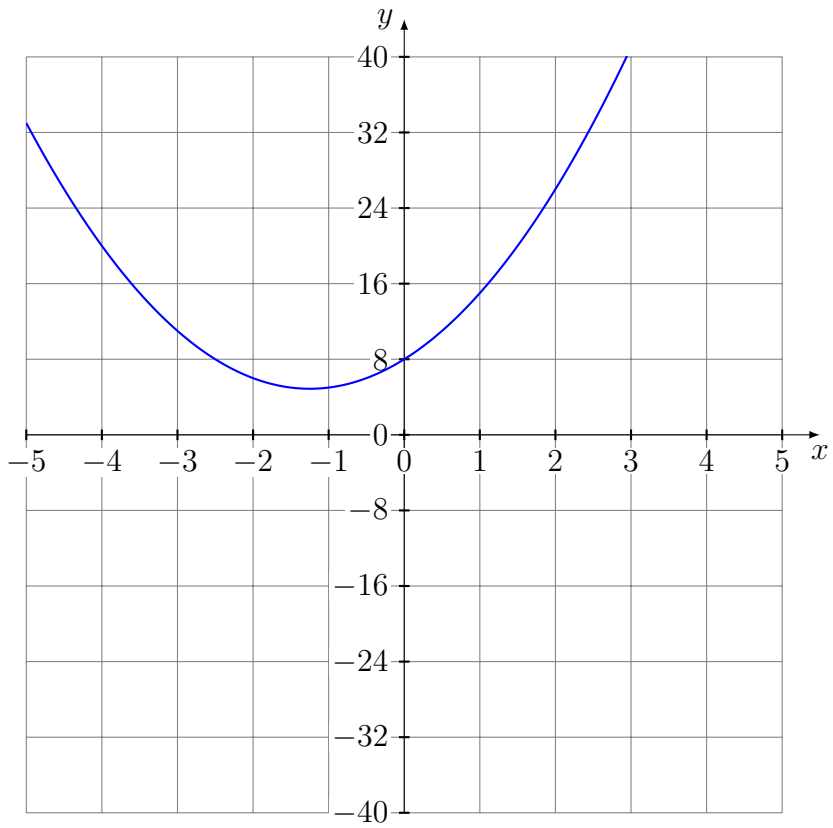

Figure 1. A graph of polynomial $2x^2 + 5x + 8$

Since the polynomial possesses a degree d of 2, it will require $d + 1$, or 3 data points for reconstruction. Data points are selected, where $x \neq 0$. Notice that that the y value for $x = 0$ is the constant for the polynomial, and therefore cannot be used.

Here are the data points that we will be using:

$$[(1, 15), (2, 26), (3, 41)]$$

2.2 Create Lagrange Polynomial

By using our defined theorem and our constructed data points, we can reconstruct the coefficients by creating our Lagrange polynomial.

$$L(x) := (15 * \frac{x-2}{1-2} * \frac{x-3}{1-3}) + (26 * \frac{x-1}{2-1} * \frac{x-3}{2-3}) + (41 * \frac{x-1}{3-1} * \frac{x-2}{3-2})$$

We obtain $2x^2 + 5x + 8$ after creating our interpolating polynomial, and simplifying it. However, to retrieve just the coefficients, x can be substituted for 0. The resulting simplification would produce $2 + 5 + 8$, the coefficients of our original polynomial.

3 Galois Fields

While implementing polynomial interpolation in a $\mathbb{Q}$ field provides countless opportunities for computational operations, it becomes inefficient and even insecure when delving into sensitive fields such cryptography [1]. Therefore, we introduce the idea of *finite fields*.

A finite field , or more commonly known as a **Galois Field** represents a finite set of numerical values. When dealing with daily arithmetic operations, we often rely on the $\mathbb{Q}$ field, which comprises of every rational number. However, Galois fields are represented by p^n elements, where p is a prime. Therefore, the set of all possible elements for a Galois field of p^n is 0 to $p^n - 1$.

When working with a Galois field, arithmetic operations (addition, multiplication and their inverses) can still be implemented as if the set was comprised of normal numbers, but their results are only found within the respective field. This means that a reduction modulo of p must occur.

With that stated, we can represent a traditional number as a polynomial, where the coefficients of the polynomial corresponds with a bit of the number. For example, the number 3 in $\mathbb{Q}$ can be represented as 011_2, and $0x2 + 1x + 1$ (or simply, $x+1$ in $Gf(2^8)$. This places constraints over how the polynomial is implemented. The degree of the polynomial is less than n, and the coefficients are in the field of $Gf(p)$. For example, a polynomial over the field of $Gf(2^8)$ would have a degree less than our equal to 7, with coefficients that are {0, 1}.

When performing polynomial multiplication, a reduction modulo of an irreducible polynomial $P(x)$ is performed at the end. However, we will not be doing that in our library implementation, as we will see later when implementing *lookup tables*.

4 Implementation

Now, let's create an implementation of polynomial interpolation over $Gf(256)$, or $Gf(2^8)$ as a library. Core functions and data structures will only be examined, but the FULL source code is available open-sourced[2].

4.1 Lagrange Construction and Reconstruction Operations

The first step is creating a preemptive validation function, $new_polynomial()$ where the coefficients of a polynomial an are passed as an array[3]. The degree is supplied, the polynomial is constructed, and for each supplied x-value, a y-value is calculated and returned along with it.

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

/* macro for retrieving size for a fixed-sized array */
#define SIZEOF(coefficients) sizeof(coefficients) /
 ↪  sizeof(coefficients[0])

typedef uint8_t u8;

/* lagrange_t type, which will be heap-allocated and freed at
 ↪  the end of execution */
typedef struct {
    u8 * coefficients;
    u8 degree;
} lagrange_t;

/* Since C does not have support for tuples or similar
 ↪  associative types, we take advantage of a typedef-ed
 ↪  struct */
typedef struct {
```

[2] https://github.com/ex0dus-0x/lagrange

[3] where we assume that the polynomial is in standard form, such that the terms are ordered by degree.

```
18      u8 x;
19      u8 y;
20  } lagrange_coordinate;
21
22  /* Create a new polynomial for lagrange polynomial
    ↪  interpolation */
23  lagrange_t *
24  lagrange_new(u8 coefficients[], u8 degree);
25
26  /* Create a tuple-like type for xy coordinate points */
27  lagrange_coordinate *
28  lagrange_create_point(u8 x, u8 y);
29
30  /* Implement if user chooses to reconstruct a polynomial with
    ↪  a set of coordinates. */
31  void
32  lagrange_reconstruct(lagrange_t * polynomial,
    ↪  lagrange_coordinate * coordinates[], u8 size);
33
34  /* Implement if a user chooses to test a created polynomial
    ↪  with a set of coordinates */
35  void
36  lagrange_test(lagrange_t * polynomial, lagrange_coordinate *
    ↪  coordinates[], u8 size);
```

Note that the definition of the `SIZEOF(..)` macro. Since C automatically deferences a fixed-size array passed as an argument to a pointer, we are unable to retrieve the size of all the elemnts. As a result, `SIZEOF(..)` is used to find the size before function execution.

The implementation above of `lagrange_new()` simply creates a new heap-allocated `lagrange_t`. This new type is introduced for several purposes:

1. Test against *(x,y)* points (hence, providing a preset number of coefficients)

2. Reconstruct coefficients (hence, initializing the structure with NULLs)

Once a `lagrange_t` type is defined, the user can now harness `lagrange_reconstruct()` in order to perform full polynomial interpolation, or use `lagrange_test()` in order to test a set of coordinates against a pre-defined `lagrange_t` type.

These functions are implemented with error-checking to ensure that the criteria for Galois-field arithmetic are met for polynomials.

4.2 Finite field helpers

We introduce a set of helper functions important for the arithmetic computation in $Gf(256)$. These helper functions are written based on the Rijndael implementation, or more commonly known as **AES**, the Advanced Encryption Standard. Since the Galois field limits the integers we can use, all operations will continue to use `u8` types.

As explained earlier, addition and subtraction are inverses, and can therefore be represented as two simple XOR operations.

```
/* Addition of two numbers in Gf(256) */
u8 galois_add(u8 a, u8 b) {
    return a^b;
}

/* Subtraction of two numbers in Gf(256) */
u8 galois_subtract(u8 a, u8 b) {
    return a^b;
}
```

Multiplication and division are both harder operations, and previous various implementations call for bit manipulation, which can easily become overwhelming and even insecure. As a result, we implement look-up tables, which provide a way for the function to efficiently retrieve a value when multiplying or dividing, without bit manipulation, which can result in a timing attack if this library is implemented for cryptography purposes. This method is prominently featured in the **Rijndael** standard, or more commonly known as **AES**, specifically in the *Galois Counter Mode*, or GCM [3].

```
1 /* Lookup tables are define as static consts */
2
3 u8 galois_multiply(u8 a, u8 b){
4     /* perform other necessary error-checking operations */
5     return GALOIS_EXP_TABLE[(GALOIS_LOG_TABLE[a] +
      ↪  GALOIS_LOG_TABLE[b]) % 255 ];
```

```
}

u8 galois_divide(u8 a, u8 b){
    /* perform other necessary error-checking operations */
    return GALOIS_EXP_TABLE[( 255 + GALOIS_LOG_TABLE[a] -
     ↪  GALOIS_LOG_TABLE[b]) % 255 ];
}
```

Finally, we define the function for actual mathematical polynomial interpolation.

```
/* iteratively calculates each term, and stores results in
 ↪  pointer */
void
compute_lagrange(lagrange_coordinate * points[], u8 size, u8 *
 ↪  result_coefficients);
```

4.3 Example Usage

With the library implementation complete, we can now examine how easy it is to reconstruct a polynomial using our specification.

```
#include "lagrange.h"

int main(){
  lagrange_coordinate * one = (lagrange_coordinate *)
   ↪  lagrange_create_point(1, 4);
  lagrange_coordinate * two = (lagrange_coordinate *)
   ↪  lagrange_create_point(2, 15);
  lagrange_coordinate * three = (lagrange_coordinate *)
   ↪  lagrange_create_point(3, 40);
  lagrange_coordinate * four = (lagrange_coordinate *)
   ↪  lagrange_create_point(4, 85);

  lagrange_coordinate * test_set[4] = { one, two, three, four
   ↪  }; // three coordinate points
  u8 test_coefficients[4] = { 1, 1, 1, 1 };              //
   ↪  degree of 3

```

```
12    lagrange_t * testpoly = lagrange_new(test_coefficients,
      ↪  SIZEOF(test_coefficients));
13    lagrange_test(testpoly, test_set, SIZEOF(test_set)); //
      ↪  SUCCESS
14  }
```

5 Conclusions

We have presented a library implementation for Lagrange polynomial interpolation within Galois-256 finite fields. We took a look at how polynomial interpolation worked, and how finite fields worked alongside it, as it is incredibly useful feature in computation, especially cryptography. After understanding the theory, we finally demonstrated how to implement a simple-to-use library in the C Programming Language.

Still a lot of work may be done to improve efficiency and usability of the library. Benchmarking can be used to test the run-time of Galois lookup operations. Unit testing can be used to confirm validity of results from polynomial interpolation. Despite this, we hope that this library can help provide developers efficiently work with polynomial interpolation operations without excessive overhead or wasted time.

References

[1] Daniel V. Bailey and Christof Paar. "Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography". In: *Journal of cryptology* 14.3 (2001), pp. 153–176. URL: http://www.ei.ruhr-uni-bochum.de/media/crypto/veroeffentlichungen/2010/08/08/baileypaaroefjc.pdf.

[2] Justin Cappos and Santiago Torres. *PolyPasswordHasher: Protecting Passwords In The Event Of A Password File Disclosure*. Tech. rep. 2014. URL: https://password-hashing.net/submissions/specs/PolyPassHash-v1.pdf.

[3] Sam Trenholme. *AES' Galois field*. URL: http://www.samiam.org/galois.html. (accessed: 05.04.2018).