Field Operations on a finite field and elliptic curve operaions

Kiran Deep Ghosh (Crs2310)

December 2, 2024

Contents

1 Introduction

1.1 Overview of field operations in a finite field

Modular arithmetic is a system of arithmetic for integers where numbers "wrap around" upon reaching a certain value called the modulus. In the context of finite fields, modular arithmetic is essential for defining operations like addition, subtraction, multiplication, and inversion that satisfy field properties.

Finite Field Basics

A finite field (also known as a Galois field) is a set of elements that:

- Is finite (has a fixed number of elements).
- Forms a field, meaning: Addition, subtraction, multiplication, and division (except by zero) are defined. These operations satisfy the properties of commutativity, associativity, and distributivity. There is an additive identity (0) and a multiplicative identity (1).

A common example is the finite field $\mathbf{Z_p}$ where p is a prime number. The elements of $\mathbf{Z_p}$ are integers $\{0, 1, 2, ..., p-1\}$, and operations are performed modulo p.

Modular Arithmetic Operations

1. Addition, $(a + b)$ mod p
2. Subtraction, $(a - b)$ mod p
3. Multiplication, $(a \cdot b)$ mod p
4. Multiplicative Inverse(Division), To divide a by b, compute:$(a \cdot b^{-1})$ mod p where b^{-1} is moduler inverse of b, satisfying $(b \cdot b^{-1})$ mod p =1

1.2 Purpose of the Project

The purpose of this project is to design and implement an efficient framework for performing modular arithmetic in a finite field defined by a large prime p, and subsequently apply these operations to build the foundation for elliptic curve cryptography (ECC). This project aims to explore the mathematical and computational principles behind modular arithmetic and elliptic curves, bridging the gap between theory and practice.

Key Objectives:

1. **Develop Modular Arithmetic Operations in Finite Fields:**
    - Efficiently handle addition, subtraction, multiplication, and modular reduction in a finite field of large prime p.
    - Implement Barrett Reduction to optimize modular arithmetic, replacing division operations with efficient multiplication and shifts, making it suitable for large-scale cryptographic applications.
2. **Enable Elliptic Curve Computations:**
    - Use the developed modular arithmetic framework to implement elliptic curve operations, such as point addition and scalar multiplication.
    - Provide a basis for exploring elliptic curves over finite fields, which are central to modern cryptography.

To achieve the objectives of this project, the following implementation strategy is adopted:

1.3 Define Finite Field Operations

- Implement the basic modular arithmetic operations over a finite field defined by a large prime p="e92e40ad6f281c8a082afdc49e1372659455bec8ceea043a614c835b7fe9eff5".

Steps:

1. Design functions for addition, subtraction, multiplication, and modular inversion modulo p.
2. Use Barrett Reduction for efficient modular reduction to replace costly division operations.
3. Optimize these functions for large numbers, ensuring scalability for cryptographic applications.

Outcome: A complete set of arithmetic operations that adhere to the properties of a finite field , forming the foundation for elliptic curve computations.

2 Implemen Barrett Reduction

- **Objective:** Optimize modular reduction to make arithmetic operations faster.
- **Barrett reduction Algorithm:** Barrett reduction finds z mod p for given positive integers z and p. The quotient $\lfloor z/p \rfloor$ is estimated using less-expensive operations involving powers of a suitably-chosen base b (e.g., $b = 2^L$ for some L which may depend on the modulus but not on z). A modulus-dependent quantity $\lfloor b^{2k}/p \rfloor$ must be calculated, making the algorithm suitable for the case that many reductions are performed with a single modulus.

Barrett reduction Algorithm
INPUT: p, b $\geq$ 3, k = $\lfloor logbp \rfloor + 1$, $0 \leq z < b^{2k}$, and $= \lfloor b^{2k}/p \rfloor$.
OUTPUT: z mod p.
1. $\hat{q} \leftarrow \lfloor \lfloor z/b^{k-1} \rfloor \cdot \mu / b^{k+1} \rfloor$
2. $r \leftarrow$ (z mod b^{k+1}) - ($\hat{q} \cdot p$ mod b^{k+1})
3. If r ¡ 0 then $r \leftarrow r + b^{k+1}$.
4. While $r \geq p$ do: $r \leftarrow r - p$.
5. Return(r).

2.1 Key Components in the Implementation

The barrett function implements modular reduction using Barrett reduction. Here are a few points about the code, and clarification:

1. Code Flow Overview:
   - The product $Z * mu$ is computed and shifted to isolate the higher bits relevant to the division (via $field\_mult$ and the Q array).
   - A secondary multiplication with p ($Q * p$) approximates the value of Z/p.
   - The subtraction $Z - B$ refines this to calculate the remainder.
   - If the remainder is still greater than or equal to p, repeated subtraction is applied.
2. Functions Used:

- *field_mult* presumably performs a polynomial or field multiplication.
- *subs* presumably performs subtraction of arrays.
- *compare* checks if one array is larger than the other.

3. Approximate the Quotient $\lfloor z/p \rfloor$.
   - Objective: Use the precomputed value μ to estimate $Q = \lfloor z/p \rfloor$.
   - Multiply the higher-order bits of Z (starting from Z + 1 to ignore the least significant bits) with μ. This gives an intermediate product A.
   - Shift the product A to isolate the most significant bits. These bits represent the approximate quotient Q.
4. Compute the Product $Q * p$ which approximate $\lfloor z/p \rfloor * p$
   - Multiply the quotient Q with the modulus p to get another intermediate product B.
   - This product B represents the largest multiple of p that is less than or equal to Z.
5. Compute the Remainder $Z - B$.
   - Use a subtraction routine (subs) to subtract the lower bits of B from the corresponding bits of Z, starting from their higher-order regions (Z + 9 and B+9).
   - Store the result in the *result* array.
6. Ensure the Remainder is Canonical ($result < p$)
   - Objective: After subtraction, the result might still be larger than p. Apply a final correction to ensure $0 \leq result < p$.
   - Compare result with p using the compare function.
   - If $result \geq p$, subtract p iteratively until $result < p$.

3 Introduction to Modular Exponentiation

Modular exponentiation computes $H = G^n \text{modp}$, where G is the base, n is the exponent, and p is the modulus. This operation is fundamental in many cryptographic algorithms,

including RSA, Diffie-Hellman, and Elliptic Curve Cryptography. The $left\_to\_right$ (L2R) binary exponentiation method processes the bits of the exponent n from the most significant bit (MSB) to the least significant bit (LSB). This approach balances efficiency and memory requirements while ensuring the correctness of the modular computation.

1. Input Parameters

    - G (Base): An array representing the base G for exponentiation. It is typically a number or polynomial smaller than the modulus p.
    - n (Exponent): An array representing the exponent n. The function iterates over each bit of n, starting from the most significant bit.
    - p (Modulus): The modulus for the modular exponentiation operation.
    - mu (Precomputed Value for Barrett Reduction): The value $\mu = \lfloor 2^k/p \rfloor$ precomputed for efficient modular reduction using the Barrett reduction algorithm.
    - H (Result): The output array where the final result G^n modp is stored.

The algorithm begins by initializing the result H to 1, represented as an array. It then iterates through each bit of the exponent n from the most significant bit (MSB) to the least significant bit (LSB). For each bit, it performs a square operation by computing $H = H^2$ modp, using field_mult for squaring and barrett for modular reduction. If the current bit is 1, the algorithm performs an additional multiplication step, $H = H \cdot G$modp, again using field_mult and barrett. This "square-and-multiply" process ensures efficient computation of G^n modp by reducing unnecessary operations and keeping intermediate results within bounds.

4 Introduction to Elliptic Curve Point Addition

Point addition is a fundamental operation in elliptic curve cryptography (ECC). It combines two points, $P_1 = (x_1, y_1)$,$P_2 = (x_2, y_2)$, on an elliptic curve to produce a third point $P_3 = (x_3, y_3)$. This operation is central to scalar multiplication, the building block for cryptographic schemes such as ECDSA (Elliptic Curve Digital Signature Algorithm) and ECDH (Elliptic Curve Diffie-Hellman). The rules of addition depend on whether the points are distinct, identical, or if one of them is the point at infinity.

The provided *point_addition* function implements point addition in modular arithmetic, ensuring all operations are performed modulo p, the prime defining the finite field.

5 Key Steps in the Algorithm

1. **Handle Edge Cases:** If $y_2 = y_1$, adjust y_2 modulo p. And if $x_2 = x_1$, adjust x_2 modulo p. These adjustments ensure the algorithm avoids undefined or edge cases, such as division by zero.

2. **Calculate the Slope** λ Compute the numerator $y_2 - y_1$ and the denominator $x_2 - x_1$. Perform modular inversion of the denominator using *exponent_l2r* with $q = p - 2$ (as $a^{-1} \equiv a^{p-2}$ mod p). Square the result to prepare for modular division, effectively computing λ mod p.

3. **Compute the x-coordinate of** p_3**:** Calculate $x_3 = \lambda^2 - x_1 - x_2$ modp. Ensure modular consistency by adjusting intermediate results and handling cases where $x_3 = x_1$ or $x_3 = x_2$.

4. **Compute the y-coordinate of** p_3**:** Calculate $y_3 = \lambda(x_1 - x_3) - y_1$ modp. Use modular arithmetic to ensure all results remain within the bounds of p.

5. **Store the Result:** Save x_3 and y_3 in the output structure p_3.

6 Point Addition on Elliptic Curve

Point addition in elliptic curve cryptography (ECC) is the operation of summing two distinct points$P_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ on an elliptic curve to produce a third point $p_3 = (x_3, y_3)$. This operation involves computations over a finite field, ensuring modular arithmetic is consistently applied.

The provided implementation of the $point_addition$ function is optimized for finite field arithmetic using modular subtraction, field multiplication, and modular reduction (via Barrett reduction). This version uses pointers for the output enhancing efficiency in certain contexts.

1. **Calculate the Slope (**λ**):** Compute the numerator $y_2 - y_1$ using $modular_subs$ and store it in *temp*. Compute the denominator $x_2 - x_1$ using $modular_subs$ and store it in *temp*1. Use *exponent_l2r* to compute the modular inverse of $x_2 - x_1$, storing the result in temp3. Multiply the numerator $y_2 - y_1$ by the modular inverse using $field_mult$ and reduce the result with barrett. This gives λ mod p, stored in temp.

2. **Compute x_3:** Square λ to compute λ^2 storing the intermediate result in $temp1$. Subtract x_1 and x_2 from λ^2 using $modular\_subs$ to compute $x_3 = \lambda^2 - x_1 - x_2 \text{modp}$. Store the result in $P_3.x$.

3. **Compute y_3:** Use $x_3 = \lambda(x_1 - x_3) - y_1$ to calculate the y coordinate of the resulting point p^3. Adjust results using modular arithmetic to ensure $y_3 \in [0, p-1]$.

4. **Store the Result:** Assign the calculated x_3 and y_3 to the output structure P_3.

7 Point Doubling on Elliptic Curve

Point doubling is a critical operation in elliptic curve cryptography (ECC), where a single point $P_1 = (x_1, y_1)$ is "added to itself" to produce $P_3 = 2P_1 = (x_3, y_3)$. This operation is essential for scalar multiplication, the process of computing kP by repeatedly doubling and adding points. In ECC, point doubling involves several steps performed within a finite field, including modular arithmetic, modular inversion, and field multiplications.

The function *point_doubling* implements this operation over a finite field defined by a prime p. It uses modular arithmetic to ensure all computations stay within the field and avoids edge cases such as division by zero.

1. **Compute $3x_1^2 + a$:** Square x_1 to get x_1^2. Triple and add the curve parameter a to compute $_{num} = 3x_1^2 + a$ modulo p. Adjust intermediate results using modular arithmetic to ensure they remain within the field.

2. **Compute $2y_1$:** Double y_1 to get the denominator $2y_1$ mod p. If $2y_1 = 0$ modp, the doubling operation is invalid as it results in the point at infinity.

3. **Compute $\lambda = \frac{3x_1^2+a}{2y_1}$:** Use $exponent_l 2r$ to compute the modular inverse of $2y_1$. Multiply the numerator $3x^2 + a$ with the inverse of $2y_1$, followed by a Barrett reduction to find λ modp.

4. **Compute x_3:** Use $x_3 = \lambda^2 - 2x_1$ mod p, to calculate the x coordinate of the resulting point p_3. Handle edge cases and adjust the result using modular arithmetic to ensure $x_3 \in [0, p-1]$.

5. **Compute y_3:** Compute $x_1 - x_3$ using $modular\_subs$, storing the result in temp3. Multiply λ with $x_1 - x_3$ using $field\_mult$, and reduce the result modulo p. Subtract y_1 from the result to compute $y_3 = \lambda(x_1 - x_3) - y_1$ modp. Store the result in $p_3.y$.

6. **Store the Result:** Assign the computed x_3 and y_3 to the output structure P_3.

8 Conclusion

Key aspects of this project included implementing foundational operations such as point addition, point doubling, and scalar multiplication, which form the backbone of ECC. These operations were meticulously designed to handle modular arithmetic over a finite field, ensuring precision and consistency in computations. Techniques like Barrett reduction and modular inversion using Fermat's Little Theorem were employed to optimize performance and maintain the correctness of results within the field.

The integration of modular arithmetic within the finite field F_p ensures the cryptographic strength of ECC. Each operation, from modular addition and subtraction to field multiplication and modular inversion, adheres to strict mathematical principles, preventing vulnerabilities and maintaining the cryptographic integrity of the system. These techniques enable the secure and efficient computation of critical processes such as key generation, encryption, and digital signature creation.

Overall, the project demonstrates the practical implementation of ECC, showcasing its ability to provide high-security guarantees with efficient resource utilization. By focusing on finite field modular arithmetic and its application in ECC, this work contributes to the understanding and adoption of secure cryptographic protocols in contemporary and future systems. The results reinforce the critical role of ECC in advancing secure communications and ensuring data integrity across a wide range of applications.

9 References:

- Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone, Handbook of Applied Cryptography, CRC Press, 1996.
- Neal Koblitz, "Elliptic Curve Cryptosystems," Mathematics of Computation, 1987.
- William Stallings, Cryptography and Network Security: Principles and Practice, Pearson Education.