Finite field library and counting points over finite field

Computing project – Part 1

Jujian Zhang, Yuan Yang and Diego Chicharro

LSGNT

9 December 2021

Abstract

In the first part of our project, we built finite field library, by finding all irreducible polynomials of degree n over the prime field $\mathbb{F}_p$, and then we compute the number of points of a given finite field $\mathbb{F}_{p^n}$ and a given elliptic curve $y^2 = x^3 + ax + b$.

Define *modp* element

First we defined the class *modp* element in Python

```python
# should we make a class for integer mod p
class modp(object):
    def __init__(self, p : int, n : int):
        self.p = p
        self.n = n % p

    def __repr__(self):
        return f"{self.n} mod {self.p}"
```

And operations(+,-,*,/) between them.

```python
def __add__(self, other : modp):
    if self.p != other.p :
        raise ValueError
    return modp(self.p, (self.n + other.n) % self.p)

def __neg__(self):
    return modp(self.p, (-self.n))

def __sub__(self, g : modp):
    return self + (-g)

def __mul__(self, g: modp):
    if self.p != g.p :
        raise ValueError
    return modp(self.p, (self.n * g.n) % self.p)
def __truediv__(self, g: modp):
    if self.p !=g.p :
        raise ValueError
    if g.n% self.p ==0 :
        raise ZeroDivisionError
    return modp(self.p, (self.n * g.n**(self.p-2))%self.p)
```

Caution: need to define what is 'equal'!

We need to compare if two modp elements are equal.

```python
def __eq__(self, g : modp):
    if self.p != g.p :
        raise ValueError
    return (self.n - g.n) % self.p == 0
```

DEFINE *MODP POLYNOMIALS*.

Then we can define polynomials with modp element coefficients, it contains degree, and a coefficients dictionary.

```python
class polynomial_modp(object):
    def __init__(self, p, coeff : dict):
        self.p = p
        self.coeff = {k :coeff[k] for k in coeff.keys() if coeff[k] != modp(p, 0)}
        # remove coefficients that are zero
        if self.coeff == {}:
            self.deg = 0
        else :
            self.deg = max(self.coeff.keys())

    def lc(self) -> modp : #leading coeff
        return self.coeff.get(self.deg, modp(self.p, 0))
```

Notice that we removed all zero coefficient, and we keep the leading coefficient.

Again, define basic operations between modp polynomials

Again, we define (+,-,*) operations between modp polynomials.

```python
    def __add__(self, g : polynomial_modp):
        new_coeff  = dict()
        for i in range(self.deg + g.deg + 1):
            new_coeff[i] = self.coeff.get(i, modp(self.p, 0)) + g.coeff.get(i, modp(self.p,0))
        return polynomial_modp(self.p, new_coeff)

    def __neg__(self):
        return polynomial_modp(self.p, {k : -v for k, v in self.coeff.items()})

    def __sub__(self, g : polynomial_modp):
        return self + (-g)

    def __mul__(self, g : polynomial_modp):
        new_coeff = dict()
        for i in range(self.deg + g.deg + 1):
            new_coeff[i] = sum((self.coeff.get(j, modp(self.p, 0)) * g.coeff.get(i - j, modp(self.p, 0)) for j i
        return polynomial_modp(self.p, new_coeff)
```

FLOOR DIVISION AND MODULO OPERATION

We need floor division.

```python
def __floordiv__(self, g : polynomial_modp):
    # long division
    lc1 = self.lc()
    if lc1.n == 0 :
        return polynomial_modp(self.p, {})
    lc2 = g.lc()
    if lc2.n == 0:
        raise ZeroDivisionError
    lc3 = lc1 / lc2
    h = polynomial_modp(self.p, {self.deg - g.deg : lc3})

    if self.deg < g.deg :
        return polynomial_modp(self.p, {})
    elif self.deg == g.deg :
        return h
    else :
        return h + (self - h * g) // g
```

And the residue polynomial.

```python
def __mod__(self, g : polynomial_modp):
    return self - (self // g) * g
```

Define two *lists* of polynomials!

Define all polynomials of degree no more than *n*

```
def all_polynomial_upto_deg_n(p : int, n : int) -> list[polynomial_modp] :
    return [polynomial_modp(p, {k : modp(p, c[k]) for k in range(n+1) if c[k] != 0}) for c in product(*[[i for i in
```

Define all *monic* polynomials of degree no more than *n*

```
def all_polynomial_upto_deg_n_with_leading_coeff_1(p : int, n : int) -> list[polynomial_modp]:
    return list(filter(lambda f : f.lc() == modp(f.p, 1), all_polynomial_upto_deg_n(p, n)))
```

Check if a polynomial is composite or irreducible!

Define two functions to check if it is composite or irreducible.

```
def is_composite(p : polynomial_modp) -> bool :
    d = p.deg

    for f in all_polynomial_upto_deg_n(p.p, d-1):
        if f.deg > 0 and ((p % f) == polynomial_modp(p.p,
            # print(f"factor is {f}")
            return True
    return False

def is_irreducible(p : polynomial_modp) -> bool : return
```

Generate all irreducible polynomial of degree no more than n

```
for f in all_polynomial_upto_deg_n_with_leading_coeff_1(2,3):
    if is_irreducible(f) == 1:
        print(f)
```

```
x mod 2
1 mod 2
1+x^2+x^3 mod 2
1+x mod 2
1+x+x^3 mod 2
1+x+x^2 mod 2
```

Find an irreducible polynomial.

```
def a_monic_irreducible_modp_polynomial_of_deg_n(p: int, n:int):
    for f in all_polynomial_upto_deg_n_with_leading_coeff_1(p,n):
        if f.deg == n and is_irreducible(f) == 1:
            return f
```

Have a test!

```
print(a_monic_irreducible_modp_polynomial_of_deg_n(7,2))
```

```
1+x^2 mod 7
```

Application: Counting points on a given elliptic cueve

Given a prime number p, degree n, coefficients a, b, we count the number of $\mathbb{F}_{p^n}$ points of the elliptic curve $y^2 = x^3 + ax + b$.

```
def counting_points_of_elliptic_curves(p:int, a: int, b: int, n:int):
    # counting the number of points of the elliptic curve y^2=x^3+ax+b over the finite field F_{p^n}
    f = a_monic_irreducible_modp_polynomial_of_deg_n(p,n)
    k = 0
    polya = polynomial_modp(p,{0:modp(p,a)}) # the polynomial a
    polyb = polynomial_modp(p,{0:modp(p,b)}) # the polynomial b
    print(f,polya,polyb)
    for g in all_polynomial_upto_deg_n(p,n-1):
        for h in all_polynomial_upto_deg_n(p,n-1):
            if ((g*g-h*h*h-polya*h-polyb) % f)== polynomial_modp(p,{}):
                k=k+1
    return k+1 # add the infinity point
```

Test!

```
print(counting_points_of_elliptic_curves(3,1,-1,4))
#  counting the number of points of the elliptic curve y^2=x^3+x-1 over F_{3^4}. It's 64.
```

```
64
```

Thank you!