ECC is just a subgroup of Algebra Curve over The Finite Field [Note and Draft]

Ryan J. Kung
ryankung@ieee.org

October 8, 2017

Contents

1 The Elliptic Curves Group

2 Pubic-key cryptography

"The mathematics of public-key cryptography uses a lot of group theory. Different cryptosystems use different groups, such as the group of units in modular arithmetic and the group of rational points on elliptic curves over a finite

field. This use of group theory derives not from the "symmetry" perspective, but from the efficiency or difficulty of carrying out certain computations in the groups. Other public-key cryptosystems use other algebraic structures, such as lattices."' [2]

2.1 Definition [1]

A set $\mathbb{G} = a, b, c, ...$ is called a group, if tehere exists a group addition $(+)$ connecting the elements in $(\mathbb{G}, +)$ in the following way:

(1) $a, b \in \mathbb{G} : \ c = a + b \in \mathbb{G}$ (closure)
(2) $a, b, c \in \mathbb{G} : (a + b)c = a(b + c)$ (associativity)
(3) $\exists e \in \mathbb{G} : a + e = e, \forall a \in \mathbb{G}$ (identity / neutral element)
(4) $\forall a \in \mathbb{G}, \exists b \in \mathbb{G} : a + b = e, i.e., b \equiv -a$ (inverse)
if a group obey axiom (1,2), it is a SemiGroup;
if a group obey axiom (1,2,3), it is a monadid;
if a group obey axiom (1,2,3,4) and the axiom of commutatativity$(a + b = b + a)$, it is a Abelian Group

2.2 Abstract Implementation of Groups

abstract.py

```
class Groupoid(metaclass=ABCMeta):

    __slots__ = ()

    def __init__(self, v):
        self.value = v

    @abstractmethod
    def op(self, g: 'Group') -> 'Group':
        pass

    def __eq__(self, b) -> bool:
        return self.value == b.value

    def __add__(self, g: 'Group') -> 'Group':
        assert isinstance(g, type(self))
        res = self.op(g)
        assert isinstance(res, type(self))
        return res

    def __repr__(self):
        return "%s::%s" % (
            type(self).__name__,
```

```
            self.value
        )

    def __str__(self):
        return str(self.value)


class SemiGroup(Groupoid):

    __slots__ = ()

    @abstractmethod
    def op(self, g: 'Group') -> 'Group':
        '''
        The Operator for obeying axiom `associativity` (2)
        '''
        pass


class Monoid(SemiGroup):
    __slots__ = ()

    @abstractproperty
    def identity(self):
        '''
        The value for obeying axiom `identity` (3)
        '''
        pass

    def __matmul__(self, n):
        return double_and_add_algorithm(
            getattr(n, 'value', n), self, self.identity)


class Group(Monoid):
    __slots__ = ()

    @abstractmethod
    def inverse(self, g: 'Group') -> 'Group':
        '''
        Implement for axiom `inverse`
        '''
        pass

    def __sub__(self, g: 'Group') -> 'Group':
        '''
```

```
        Allow to reverse op via a - b
        '''
        return self.op(g.inverse())

    def __neg__(self) -> 'Group':
        return self.inverse()
```

2.3 Concrete EC Group

```
class EllipticCurveGroup(Group):
    # for y^2 = x^3 + A * x + B
    A = abstractproperty()
    B = abstractproperty()

    def op(self, g):
        if g.value == 0:
            return self
        field = self.value[0].__class__

        if self.value[0] != g.value[0]:
            m = (self.value[1] - g.value[1]) / (self.value[0] - g.value[0])
        if self.value[0] == g.value[0]:
            m = (field(3) * self.value[0] * self.value[0] +
                 field(self.A)) / (field(2) * self.value[1])
        r_x = (m * m - self.value[0] - g.value[0])
        r_y = (self.value[1] + m * (r_x - self.value[0]))
        return self.__class__((r_x, -r_y))

    def inverse(self):
        return self.__class__((self.value[0], -self.value[0]))

    @property
    def identity(self):
        # The abstract zero of EC Group
        return self.__class__(0)
```

3 The Finite Field

3.1 Definition

A field is any set of elements that satisfies the field axioms for both addition and multiplication and is a commutative division algebra.

Field Axioms [6] are generally written in additive and multiplication pairs:
(1) $(a + b) + c = a + (b + c)$; $(ab)c = a(bc)$ (associativity)
(2) $a + b = b + a$; $ab = ba$ (Commutativity)
(3) $a(b + c) = ab + ac$; $(a + b)c = ac + bc$ (distributivity)

(4) $a + 0 = a = 0 + 1$; $(a.1 = a = 1.a)$ (identity)
(5) $a + (-a) = 0 = (-a) + a$; $aa^{-1} = 1 = a^{-1}aifa \neq 0$ (inverses)

3.2 Abstract Implementation of Fields

```
class Field(Group):
    __slots__ = ()

    @abstractmethod
    def sec_op(self, g: 'Group') -> 'Group':
        '''
        The Operator for obeying axiom `associativity` (2)
        '''
        pass

    @abstractmethod
    def sec_inverse(self) -> 'Group':
        '''
        Implement for axiom `inverse`
        '''
        pass

    @abstractmethod
    def sec_identity(self):
        pass

    def __invert__(self):
        return self.sec_inverse()

    def __mul__(self, g: 'Group') -> 'Group':
        '''
        Allowing call associativity operator via A * B
        Strict limit arg `g` and ret `res` should be subtype of Group,
        For obeying axiom `closure` (1)
        '''
        res = self.sec_op(g)
        assert isinstance(res, type(self)), 'result shuould be %s' % type(self)
        return res

    def __truediv__(self, g: 'Group') -> 'Group':
        return self.sec_op(g.sec_inverse())
```

3.3 The Finite Field

A finite field is, A set with a finite number of elements. An example of inite field is the set of integers modulo p, where p is a prime number, which can be generally note as $\mathbb{Z}/p$, $GF(p)$ or $\mathbb{F}_p$.

3.4 Implementation of Finite Field

```
class FiniteField(Field):

    P = abstractproperty()

    @property
    def identity(self):
        return self.__class__(0 % self.P)

    @property
    def sec_identity(self):
        return self.__class__(1 % self.P)

    def inverse(self):
        return self.__class__(-self.value % self.P)

    def sec_inverse(self):
        gcd, x, y = extended_euclidean_algorithm(self.value, self.P)
        assert (self.value * x + self.P * y) == gcd
        if gcd != 1:
            # Either n is 0, or p is not prime number
            raise ValueError(
                '{} has no multiplicate inverse '
                'modulo {}'.format(self.value, self.P)
            )
        return self.__class__(x % self.P)

    def op(self, n):
        return self.__class__((self.value + n.value) % self.P)

    def sec_op(self, n):
        return self.__class__((self.value * n.value) % self.P)
```

4 Finite Field Functor

4.1 Definition of Category

There are three laws that categories need to follow:

(1) if $f :\rightarrow b, g : b \rightarrow c$ *and* $h : c \rightarrow d$ *then* $h \circ (g \circ f) = (h \circ g) \circ f$ (associativity)

(2) $\forall object_x, \exists morphism 1_x : x \rightarrow x$ such as $\forall morphism\ f : a \rightarrow b : \exists 1_b \circ f = f = f \circ 1_a$ (identity)

(3) if $f : B \rightarrow C$ *and* $g : A \rightarrow B : h : A \rightarrow C$ such that $h = f \circ g$ (closed under the composition operation)

4.2 Functor of Finite Field

In mathematics, a functor is a type of mapping between categories arising in category theory. Functors can be thought of as homomorphisms between categories. In the category of small categories, functors can be thought of more generally as morphisms. [7]

Thus, when discussing about Integer Field and Prime Finite Field, we can simply redefined the implementation of Finite Field above as a Functor which defined $int \rightarrow FinitePrimeSet$ as $fmap$.

5 Composed

5.1 Scala Multiplication

For reasoning the subgroup of EC Group, we introduce the scala multiplication here. The Scala Multiplication can be easily implement with 'Double as Multi Alogorithm' like this:

```
def double_and_add_algorithm(n, x, init):
    """
    Returns the result of n * x, computed using
    the double and add algorithm.
    """
    def bits(n):
        """
        Generates the binary digits of n, starting
        from the least significant bit.
        bits(151) -> 1, 1, 1, 0, 1, 0, 0, 1
        """
        while n:
            yield n & 1
            n >>= 1

    result = init
    addend = x

    for bit in bits(n):
        if bit == 1:
            result = addend + result
```

```
        addend = addend + addend

    return result


def schoof_algorithm(p, a, b):
    return p + 1 - frobenius_trace(EllipticCurve(FiniteField(p), a, b))
```

And we can add the property to the *Monoid*, and denoted with symbol @.

```
class Monoid(SemiGroup):
    __slots__ = ()

    @abstractproperty
    def identity(self):
        '''
        The value for obeying axiom `identity` (3)
        '''
        pass

    def __matmul__(self, n):
        return double_and_add_algorithm(
            getattr(n, 'value', n), self, self.identity)
```

5.2 Group over Finite Field

With the Functor of Finite Field, we can easily calculate the ECG over Finite Fields like this.

```
a = EllipticCurveGroupN710((FiniteField97(11), (FiniteField97(10))))
b = EllipticCurveGroupN710((FiniteField97(87), (FiniteField97(27))))
res = EllipticCurveGroupN710((FiniteField97(74), (FiniteField97(41))))
assert (a + b) == res
```

5.3 SubGroup

And by researching about the ECG over Finite Field, we can easily find out the Scala Multiplication on it is actually a subgroups (the + between nP always obey the *closurerule*.)

```
  def test_cyclic_subgroup():
    a = EllipticCurveCyclicSubgroup36(1)
    b = EllipticCurveCyclicSubgroup36(2)
    assert a + b == EllipticCurveCyclicSubgroup36(3)
    assert EllipticCurveGroup0203(
        (FiniteField97(3), FiniteField97(6))
```

```
) @ (a + b) == EllipticCurveGroup0203(
    (FiniteField97(80), FiniteField97(87))
)
a = EllipticCurveCyclicSubgroup36(1)
b = EllipticCurveCyclicSubgroup36(6)

assert EllipticCurveGroup0203(
    (FiniteField97(3), FiniteField97(6))
) @ (a + b) == EllipticCurveGroup0203(
    (FiniteField97(80), FiniteField97(10))
)

a = EllipticCurveCyclicSubgroup36(1)
b = EllipticCurveCyclicSubgroup36(9)

assert EllipticCurveGroup0203(
    (FiniteField97(3), FiniteField97(6))
) @ (a + b) == EllipticCurveGroup0203(0)
```

References

[1] Roland Winkler, Aug 2011, Introduction to Group Theory, http://niu.edu/rwinkler/teaching/group-11/g-lecture.pdf

[2] Why is group theory important, http://www.math.uconn.edu/ kconrad/-math216/whygroups.html

[3] Bartosz Milewski, Category Theory for Programmers, https://bartoszmilewski.com/2014/10/28/category-theory-for-programmers-the-preface/

[4] Wikipedia, Category Theory, https://en.wikipedia.org/wiki/Haskell/Category_theory

[5] Wikibooks/Haskell, Category Theory, https://en.wikibooks.org/wiki/Haskell/Category_theory

[6] Weisstein, Eric W. "Field Axioms." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/FieldAxioms.html

[7] Wikipedia, Functor, https://en.wikipedia.org/wiki/Functor