0.1 Introduction to Abstract Algebra

0.1.1 Groups

Throughout the lectures, probably the most important topic is the *group theory*.

As you can recall from the high school math, typically real-world processes are described using real numbers, denoted by $\mathbb{R}$. For example, to describe the position or the velocity of an object, you would rather use real numbers.

When it comes to working with computers though, real numbers become very inconvenient to work with. For instance, different programming languages might output different values for quite a straightforward operation 2.01 + 2.00. This becomes a huge problem when dealing with cryptography, which must check *precisely* whether two quantities are equal. For example, if the person's card number is N and the developed system pays from a different, but very similar card with number $N + k$ for $k \ll N$, then this system can be safely thrown out of the window.

This motivates us to work with integers (denoted by $\mathbb{Z}$), instead. This solves the problem with card numbers, but for cryptography this object is still not really suitable since it is hard to build a secure and reliable protocol exploiting pure integers.

This motivates us to use a different primitive for dealing with cryptographic systems. Similarly to programmers working with interfaces (or traits, if you are the *Rust* developer), mathematicians also use the so-called *groups* to represent objects obeying a certain set of rules. The beauty is that we do not concretize *how* operations in this set are performed, but rather state the fact that we can somehow combine elements with the pre-defined properties. We can then discover properties of such objects and whenever we apply the concrete "implementation" (spoiler, group of points on elliptic curve), these properties would still hold.

Remark. Further discussion with abstract objects should be regarded as "interfaces" which do not concretize the "implementation" of an object. It merely shows the nature of an object without going into the details.

Now, let us get dirty and define what the **group** is.

Definition 0.1. Group, denoted by $(\mathbb{G}, \oplus)$, is a set with a binary operation $\oplus$, obeying the following rules:

1. **Closure:** Binary operations always outputs an element from $\mathbb{G}$, that is $\forall a, b \in \mathbb{G} : a \oplus b \in \mathbb{G}$.
2. **Associativity:** $\forall a, b, c \in \mathbb{G} : (a \oplus b) \oplus c = a \oplus (b \oplus c)$.
3. **Identity element:** There exists a so-called identity element $e \in \mathbb{G}$ such that $\forall a \in \mathbb{G} : e \oplus a = a \oplus e = a$.
4. **Inverse element:** $\forall a \in \mathbb{G}\ \exists b \in \mathbb{G} : a \oplus b = b \oplus a = e$. We commonly denote the inverse element as $(\ominus a)$.

Quite confusing at first glance, right? The best way to grasp this concept is to consider a couple of examples.

Example. A group of integers with the regular addition $(\mathbb{Z}, +)$ (also called the *additive* group of integers) is a group. Indeed, an identity element is $e_{\mathbb{Z}} = 0$, associativity obviously holds, and an inverse for each element $a \in \mathbb{Z}$ is $(\ominus a) := -a \in \mathbb{Z}$.

Remark. Sometimes we use the term *additive group* when we mean that the binary operations in the set is addition $+$, while *multiplicative group* means that we are multiplying two numbers via $\cdot$.

Example. The multiplicative group of positive real numbers $(\mathbb{R}_{>0}, \cdot)$ is a group for similar reasons. An identity element is $e_{\mathbb{R}_{>0}} = 1$, while the inverse for $a \in \mathbb{R}_{>0}$ is defined as $\frac{1}{a}$.

Example. The additive group of natural numbers $(\mathbb{N}, +)$ is not a group. Although operation of addition is closed, there is no identity element nor inverse element for, say, 2 or 10.

One might ask a reasonable question: suppose you pick $a, b \in \mathbb{G}$. Is $a \oplus b$ the same as $b \oplus a$? Unfortunately, generally, this is not true.

For this reason, it makes sense to give a special name to a group in which the operation is commutative (meaning, we can swap the elements in the operation).

Definition 0.2. A group $(\mathbb{G}, \oplus)$ is called **abelian** if $\forall a, b \in \mathbb{G} : a \oplus b = b \oplus a$.

Example. The additive group of integers $(\mathbb{Z}, +)$ is an abelian group. Indeed, $a + b = b + a$ for any $a, b \in \mathbb{Z}$.

Example. The set of 2×2 matrices with real entries and determinant 1 (denoted by $\mathrm{SL}(2, \mathbb{R})$) is a group with respect to matrix multiplication. However, this group is not abelian! Take

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}. \tag{1}$$

Then, it is easy to verify that

$$AB = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \quad BA = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}, \tag{2}$$

so clearly $AB \neq BA$ – the elements of $\mathrm{SL}(2, \mathbb{R})$ do not commute.

Remark. Further, we will write ab instead of $a \oplus b$ and a^{-1} instead of $\ominus a$ for the sake of simplicity (and because it is more common in the literature). This is commonly called the *multiplicative notation*.

Finally, for cryptography it is important to know the number of elements in a group. This number is called the *order* of the group.

Definition 0.3. The **order** of a finite group $\mathbb{G}$ is the number of elements in the group. We denote the order of a group as $|\mathbb{G}|$.

Example. Integers modulo 13, denoted by $\mathbb{Z}_{13}$, is a group with respect to addition modulo 13 (e.g., $5 + 12 = 4$ in $\mathbb{Z}_{13}$). The order of this group is 13.

Despite the aforementioned definitions, many things are not generally obvious. For example, one might ask whether the identity element is unique. Or, whether the inverse element is unique. For that reason, we formulate the following lemma.

Lemma 0.4. Suppose $\mathbb{G}$ is a group. Then, the following statements hold:

1. The identity element is unique.
2. The inverse element is unique.
3. For all $a, b \in \mathbb{G}$ there is a unique $x \in \mathbb{G}$ such that $ax = b$.
4. If $ab = ac$ then $b = c$. Similarly, if $xy = zy$ then $x = z$.

Since this guide is not a textbook on abstract algebra, we will not prove all the statements. However, we will prove the first and second one to show the nature of the proofs in abstract algebra.

First Statement Proof. Suppose $e_1, e_2 \in \mathbb{G}$ are both identity elements. Consider $e_1 e_2$. From the definition of the identity element, we know that $e_1 e_2 = e_1$ and $e_1 e_2 = e_2$. Therefore, $e_1 = e_2$.

Second Statement Proof. Take $g \in \mathbb{G}$ and suppose $a, b \in \mathbb{G}$ are both inverses of g. By defininition,

$$ag = ga = e, \quad bg = gb = e. \tag{3}$$

Now, notice that

$$a = ae = a(gb) = (ag)b = eb = b \tag{4}$$

Thus, $a = b$.

Exercise. Prove the third and fourth statements.

0.1.2 Subgroups

When we are finally comfortable with the concept of a group, we can move on to the concept of a *subgroup*.

Suppose we have a group $(\mathbb{G}, \oplus)$. Suppose one takes the subset $\mathbb{H} \subset \mathbb{G}$. Of course, since all elements in $\mathbb{H}$ are still elements in $\mathbb{G}$, we can conduct operations between them via $\oplus$. The natural question to ask is whether $\mathbb{H}$ is a group itself. If this is the case, then $\mathbb{H}$ is called a **subgroup** of $\mathbb{G}$.

Definition 0.5. A subset $\mathbb{H} \subset \mathbb{G}$ is called a **subgroup** of $\mathbb{G}$ if $\mathbb{H}$ is a group with respect to the same operation $\oplus$. We denote this as $\mathbb{H} \leq \mathbb{G}$.

Example. Of course, not every subset of $\mathbb{G}$ is a subgroup. Take $(\mathbb{Z}, +)$. If we cut, say, 3 out of $\mathbb{Z}$ (so we get $\mathbb{H} = \mathbb{Z} \setminus \{3\}$), then $\mathbb{H}$ is not a subgroup of $\mathbb{Z}$ since an element -3 does not have an inverse in $\mathbb{H}$.

Example. Now, let us define some valid subgroup of $\mathbb{Z}$. Take $\mathbb{H} = \{3k : k \in \mathbb{Z}\}$ – a set of integers divisible by 3 (commonly denoted as $3\mathbb{Z}$). This is a subgroup of $\mathbb{Z}$, since it is closed under addition, has an identity element 0, and has an inverse for each element $3k$ (namely, $-3k$). That being said, $3\mathbb{Z} \leq \mathbb{Z}$.

These are good examples, but let us consider a more interesting example:

Example. Let $\mathbb{G}$ be a group and $g \in \mathbb{G}$. The centralizer of g is defined to be

$$C_g = \{h \in \mathbb{G} : hg = gh\} \tag{5}$$

Then, C_g is a subgroup of $\mathbb{G}$.

Exercise. Prove the lemma.

0.1.3 Cyclic Groups

Probably, cyclic groups are the most interesting groups in the world of cryptography. But before defining them, we need to know how to add/subtract elements multiple times (that is, multiplying by an integer). Suppose we have a group $\mathbb{G}$ and $g \in \mathbb{G}$. Then, g^n means multiplying (adding) g to itself n times. If n is negative, then we add g^{-1} to itself $|n|$ times. For $n = 0$ we define $g^0 = e$. Now, let us define what the cyclic group is.

Definition 0.6. Given a group $\mathbb{G}$ and $g \in \mathbb{G}$ the cyclic subgroup generated by g is

$$\langle g \rangle = \{g^n : n \in \mathbb{Z}\} = \{\dots, g^{-3}, g^{-2}, g^{-1}, e, g, g^2, g^3, \dots\}. \tag{6}$$

Example. Consider the group of integers modulo 12, denoted by $\mathbb{Z}_{12}$. Consider $2 \in \mathbb{Z}_{12}$, the group generated by 2 is then

$$\langle 2 \rangle = \{2, 4, 6, 8, 10, 0\} \tag{7}$$

Definition 0.7. We say that a group $\mathbb{G}$ is **cyclic** if there exists an element $g \in \mathbb{G}$ such that $\mathbb{G}$ is generated by g, that is, $\mathbb{G} = \langle g \rangle$.

Example. The group of integers $(\mathbb{Z}, +)$ is a cyclic group. Indeed, it is generated by 1.

0.1.4 Isomorphisms and Endomorphisms

Finally, we will define the concept of isomorphisms and endomorphisms. These are important concepts in the world of cryptography, since they allow us to compare different groups. Namely, suppose we have two groups $(\mathbb{G}, \oplus)$ and $(\mathbb{H}, \odot)$. Is there any way to state that these two groups are the same? The answer is yes, and this is done via isomorphisms.

Definition 0.8. A function $\varphi : \mathbb{G} \to \mathbb{H}$ is called an **homomorphism** if it is a function that preserves the group operation, that is,

$$\forall a, b \in \mathbb{G} : \varphi(a \oplus b) = \varphi(a) \odot \varphi(b). \tag{8}$$

Definition 0.9. An **isomorphism** is a bijective homomorphism.

Definition 0.10. If there exists an isomorphism between two groups $\mathbb{G}$ and $\mathbb{H}$, we say that these groups are isomorphic and write $\mathbb{G} \cong \mathbb{H}$.

Example. Consider the group of integers $(\mathbb{Z}, +)$ and the group of integers modulo 12 $(\mathbb{Z}_{12}, +)$. The function $\varphi : \mathbb{Z} \to \mathbb{Z}_{12}$ defined as $\varphi(x) = x \bmod 12$ is a homomorphism. Indeed:

$$\varphi(a + b) = (a + b) \bmod 12 = (a \bmod 12) + (b \bmod 12) = \varphi(a) + \varphi(b). \tag{9}$$

However, this function is not an isomorphism, since it is not bijective. For example, $\varphi(0) = \varphi(12) = 0$.

Example. Additive group of reals $(\mathbb{R}, +)$ and the multiplicative group of positive reals $(\mathbb{R}_{>0}, \cdot)$ are isomorphic. The function $\varphi : \mathbb{R} \to \mathbb{R}_{>0}$ defined as $\varphi(x) = e^x$ is an isomorphism. Indeed:

$$\varphi(a + b) = e^{a+b} = e^a \cdot e^b = \varphi(a) \cdot \varphi(b). \tag{10}$$

Thus, φ is a homomorphism. It is also injective since $e^x = e^y \implies x = y$. Finally, it is obviously onto. This means $(\mathbb{R}, +) \cong (\mathbb{R}_{>0}, \cdot)$.

One of the interesting theorems is the following.

Theorem 0.11. Suppose $\mathbb{G} = \langle g \rangle$ is a finite cyclic group, meaning $|G| = n \in \mathbb{N}$. Then, $\mathbb{G} \cong \mathbb{Z}_n$.

Idea of the proof. Define a function $\varphi : \mathbb{Z}_n \to \mathbb{G}$ as $m \mapsto g^m$. One can prove that this is an isomorphism.

Finally, we will define the concept of an endomorphism and automorphism to finish the section.

Definition 0.12. An **endomorphism** is a function $\varphi : \mathbb{G} \to \mathbb{G}$.

Definition 0.13. An **automorphism** is an isomorphic endomorphism.

Example. Given a group $\mathbb{G}$, fixate $a \in \mathbb{G}$. The map $\varphi : x \mapsto axa^{-1}$ is an automorphism.

Last two definitions are especially frequently used in Elliptic Curves theory.

0.2 Basic Number Theory

0.2.1 Primes

Primes are often used when doing almost any cryptographic computation. A prime number is a natural number $(\mathbb{Z}^+)$ that is not a product of two smaller natural number. In other words, the prime number is divisible only by itself and 1. First primes look like this: 2, 3, 5, 7, 11...

0.2.2 Deterministic prime tests

A primality test is deterministic if it outputs $True$ when the number is a prime and $False$ when the input is composite with probability 1. An example of a deterministic prime test is $TrialDivisiontest$. Here is an example implementation in Rust:

```
1   fn is_prime(n: u32) -> bool {
2       let square_root = (n as f64).sqrt() as u32;
3
4       for i in 2.. = square_root {
5           if n % i == 0 {
6               return false;
7           }
8       }
9
10      true
11  }
```

Deterministic tests often lack efficiency. For instance, even with square root optimization, the asymptotic complexity is $O(\sqrt{N})$. While further optimizations are possible, they do not change the overall asymptotic complexity.

In cryptography, N can be extremely large — 256 bits, 512 bits, or even 6144 bits. An algorithm is impractical when dealing with such large numbers.

0.2.3 Deterministic prime tests

A primality test is probabilistic if it outputs True when the number is a prime and False when the input is composite with probability less than 1. Such test is often called a pseudoprimality test. Fermat Primality and Miller-Rabin Primality Tests are examples of probabilistic primality test. Both of them use the idea of Fermat's Little Theorem:

Theorem 0.14. Let p be a prime number and a be an integer not divisible by p. Then $a^{p-1} - 1$ is always divisible by p: $a^{p-1} \equiv 1(\mod p)$

0.2.4 Fermat Primality Test

The key idea behind Fermat Primality Test: if for some a not divisible by n we have $a^{n-1} \not\equiv 1(\mod n)$

0.2.5 Prime fields

0.2.6 Modular inverse

0.3 Other Primitives

Here be vector spaces etc.

0.3.1 Some Fun: Group Implementation in Rust

In programming, we can think of a group as an interface, having a single binary operation defined, that obeys the rules of closure, associativity, identity element, and inverse element.

For that reason, we might even code a group in Rust! We will also write a simple test to check whether the group is valid and whether the group is abelian.

Trait for Group. First, we define a trait for a group. We will define a group as a trait with the following methods:

```rust
/// Trait that represents a group.
pub trait Group: Sized {
    /// Checks whether the two elements are equal.
    fn eq(&self, other: &Self) -> bool;
    /// Returns the identity element of the group.
    fn identity() -> Self;
    /// Adds two elements of the group.
    fn add(&self, a: &Self) -> Self;
    /// Returns the negative of the element.
    fn negate(&self) -> Self;
    /// Subtracts two elements of the group.
    fn sub(&self, a: &Self) -> Self {
        self.add(&a.negate())
    }
}
```

Checking group validity. Now observer the following: we get closure for free, since the compiler will check whether the return type of the operation is the same as the type of the group. However, there is no guarantee that associativity holds, and our identity element is at all valid. For that reason, we need to somehow additionally check the validity of implementation.

We propose to do the following: we will randomly sample three elements from the group $a, b, c \xleftarrow{R} \mathbb{G}$ and check our three properties:

1. $a \oplus (b \oplus c) \stackrel{?}{=} (a \oplus b) \oplus c$.
2. $a \oplus e \stackrel{?}{=} e \oplus a \stackrel{?}{=} a$.
3. $a \oplus (\ominus a) \stackrel{?}{=} (\ominus a) \oplus a \stackrel{?}{=} e$.

Additionally, if we want to verify whether the group is abelian, we can check whether $a \oplus b \stackrel{?}{=} b \oplus a$.

For that reason, for the check, we require the group to be samplable (i.e. we can randomly sample elements from the group):

```rust
/// Trait for sampling a random element from a group.
pub trait Samplable {
    /// Returns a random element from the group.
    fn sample() -> Self;
}
```

And now, our test looks as follows:

```rust
/// Number of tests to check the group properties.
const TESTS_NUMBER: usize = 100;

/// Asserts that the given group G is valid.
/// A group is valid if the following properties hold:
/// 1. Associativity: (a + b) + c = a + (b + c)
/// 2. Identity: a + e = a = e + a
/// 3. Inverse: a + (-a) = e = (-a) + a
pub fn assert_group_valid<G>()
where
    G: Group + Samplable,
{
    for _ in 0..TESTS_NUMBER {
        // Take random three elements
        let a = G::sample();
        let b = G::sample();
        let c = G::sample();

        // Check whether associativity holds
        let ab_c = a.add(&b).add(&c);
        let a_bc = a.add(&b.add(&c));
        let associativity_holds = ab_c.eq(&a_bc);
        assert!(associativity_holds, "Associativity does not hold
        ↪ for the given group");

        // Check whether identity element is valid
        let e = G::identity();
        let ae = a.add(&e);
        let ea = e.add(&a);
        let identity_holds = ae.eq(&a) && ea.eq(&a);
        assert!(identity_holds, "Identity element does not hold for
        ↪ the given group");

        // Check whether inverse element is valid
        let a_neg = a.negate();
        let a_neg_add_a = a_neg.add(&a);
        let a_add_a_neg = a.add(&a_neg);
        let inverse_holds = a_neg_add_a.eq(&e) && a_add_a_neg.eq(&e);
        assert!(inverse_holds, "Inverse element does not hold for
        ↪ the given group");
    }
}

/// Asserts that the given group G is abelian.
/// A group is an abelian group if the following property holds:
/// a + b = b + a for all a, b in G (commutativity)
pub fn assert_group_abelian<G>()
where
    G: Group + Samplable,
```

```
{
    for _ in 0..TESTS_NUMBER {
        assert_group_valid::<G>();

        // Take two random elements
        let a = G::sample();
        let b = G::sample();

        // Check whether commutativity holds
        let ab = a.add(&b);
        let ba = b.add(&a);
        assert!(ab.eq(&ba), "Commutativity does not hold for the
        ↪ given group");
    }
}
```

Testing the group $(\mathbb{Z}, +)$**.** And now, we can define a group for integers and check whether it is valid and abelian:

```
use crate::group::{Group, Samplable};
use rand::Rng;

/// Implementing group for Rotation3<f32>
impl Group for i64 {
    fn eq(&self, other: &Self) -> bool {
        self == other
    }

    fn identity() -> Self {
        0i64
    }

    fn add(&self, a: &Self) -> Self {
        self + a
    }

    fn negate(&self) -> Self {
        -self
    }
}

impl Samplable for i64 {
    fn sample() -> Self {
        let mut gen = rand::thread_rng();

        // To prevent overflow, we choose a smaller range for i64
        let min = i64::MIN / 3;
        let max = i64::MAX / 3;
        gen.gen_range(min..max)
```

```
    }
}
```

Just a small note: since we cannot generate infinite integers, we restrict the range of integers to prevent overflow. So, for the sake of simplicity, we divide the range of integers by 3, in which overflow never occurs.

And now, the moment of truth! Let us define some tests and run them:

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use group::*;

    #[test]
    fn test_integers_are_group() {
        assert_group_valid::<i64>()
    }

    #[test]
    fn test_integers_are_abelian() {
        assert_group_abelian::<i64>();
    }
}
```

Both tests pass! Now let us consider something a bit trickier.

Testing the group SO(3). We can define a group for 3×3 rotation matrices. Of course, composition of two rotation is not commutative, so we expect the abelian test to fail. However, the group is still valid! For example, there is an identity rotation matrix E, and for each rotation matrix $A \in \mathrm{SO}(3)$, there exists a rotation matrix $A^{-1} \in \mathrm{SO}(3)$ such that $AA^{-1} = A^{-1}A = E$. Finally, the associativity holds as well.

We will use the `nalgebra` library for this purpose, which contains the implementation of rotation matrices. So our implementation can look as follows:

```rust
/// A threshold below which two floating point numbers are
↪ considered equal.
const EPSILON: f32 = 1e-6;

/// Implementing group for Rotation3<f32>
impl Group for Rotation3<f32> {
    fn eq(&self, other: &Self) -> bool {
        // Checking whether the norm of a difference is small
        let difference = self.matrix() - other.matrix();
        difference.norm_squared() < EPSILON
    }

    fn identity() -> Self {
        Rotation3::identity()
    }
```

```
15
16      fn add(&self, a: &Self) -> Self {
17          self * a
18      }
19
20      fn negate(&self) -> Self {
21          self.inverse()
22      }
23  }
24
25  impl Samplable for Rotation3<f32> {
26      fn sample() -> Self {
27          let mut gen = rand::thread_rng();
28
29          // Pick three random angles
30          let roll = gen.gen_range(0.0..1.0);
31          let pitch = gen.gen_range(0.0..1.0);
32          let yaw = gen.gen_range(0.0..1.0);
33
34          Rotation3::from_euler_angles(roll, pitch, yaw)
35      }
36  }
```

Here, there are two tricky moments:

1. We cannot compare floating point numbers directly, since they might differ by a small amount. For that reason, we define a small threshold ε. We say that two matrices are equal iff the norm[1] of their difference is less than ε.
2. To generate a random rotation matrix, we generate three random angles and create a rotation matrix from these angles.

[1] one can think of norm as being the measure of "distance" between two objects. Similarly, we can define norm not only on matrices, but on vectors as well.