Travail préalable à l'implémentation

1. Tirage aléatoire des matrices
   But : tirer $s \leftarrow$ Gaussienne ; $e \leftarrow$ Gaussienne (de petites tailles), $A \leftarrow$ matrice uniforme à coefficients dans Z/qZ, output $A.s + e$.

```
use nalgebra::{DMatrix, DVector};
use rand_distr::{Normal, Distribution, Uniform};

alexis
fn main() {
    let mut random_gen = rand::thread_rng();

    let taille_s :i32 = 10; // Taille de s
    let taille_e :i32 = 10; // Taille de e
    let taille_a :i32 = 10; // Nombre de lignes de A

    //modulus
    let q: i32 = 44;
    let q_f64: f64 = q as f64;

    //Standard deviation
    let alpha :f64 = (2.0 * std::f64::consts::PI).sqrt()*q_f64.powf( n: -0.75);
    let sigma :f64 = (q_f64 * alpha) / (2.0 * std::f64::consts::PI).sqrt();

    //Loi de distribution pour les vecteurs s, e et a
    let loi_uniforme_matrix = Uniform::from(1.. < q);
    let loi_normal_vect_e = Normal::new(0.0, sigma).unwrap();
    let loi_normal_vect_s = Normal::new(0.0, 10.0).unwrap();
```

```rust
    //Vecteurs s
    let vect_s: Vec<f64> = (0..< taille_s)
        .map(|_| ((loi_normal_vect_s.sample(&mut random_gen))as f64).round().abs() % q as f64) : impl Iterator<Item=f64>
        .collect();

    //Vecteurs e
    let vect_e: Vec<f64> = (0..< taille_e)
        .map(|_| ((loi_normal_vect_e.sample(&mut random_gen))as f64).round().abs() % q as f64) : impl Iterator<Item=f64>
        .collect();


    //Creation objet DVector via les vecteurs s et e
    let s = DVector::from_vec(vect_s);
    let e = DVector::from_vec(vect_e);

    //Matrice A
    let mut vecteur_a : Vec<Vec<f64>> = Vec::new();

    //
    for _ in 0..< taille_a {
        let row: Vec<f64> = (0..< taille_s)
            .map(|_| (loi_uniforme_matrix.sample(&mut random_gen) % q) as f64) : impl Iterator<Item=f64>
            .collect();
        vecteur_a.push(row);
    }
```

```rust
    for _ in 0..< taille_a {
        let row: Vec<f64> = (0..< taille_s)
            .map(|_| (loi_uniforme_matrix.sample(&mut random_gen) % q) as f64) : impl Iterator<Item=f64>
            .collect();
        vecteur_a.push(row);
    }
    let a = DMatrix::from_vec(taille_a, taille_s, vecteur_a.into_iter() : impl Iterator<Item=Vec<...>>
        .flatten() : impl Iterator<Item=f64>
        .collect());


    //Calcul de A.s + e
    let result = &a * &s + &e;

    println!("s : {}", s);
    println!("e : {}", e);
    println!("A : {}", a);
    println!("A.s + e: {}", result);
}
```

2. <u>Reconstruction d'un polynôme à partir de shares</u>
   <u>But :</u> Pour n = 1 et q pair :
   - Input un secret s
   - Tirer au hasard P = a_0 + s.X de degré au plus n tel que le coefficient de degré n de P est égal à s.
   - Output le vecteur de shares [s1:=P(0)=a0, s2:= P(1)=a0+a1].
   - Pour reconstruire P à partir de ce vecteur de 2 shares : lui appliquer la forme linéaire : R(X):=(projection sur coeff de degré 2)o(Lagrange)(X) = s2 – s1.

```rust
use std::str::FromStr;
use rand::distributions::Sample;


alexis +2
fn main() {

    let threshold :usize = 2;
    let modulo :i64 = 1024;
    let secret :i64 = 55;
    let nb_shares :i64 = 2;


    4 usages  alexis +1
    pub fn generator_polynome(modulo_number: i64, zero_value: i64, threshold: usize) -> Vec<i64> {
        //genere les distributions
        let mut range = rand::distributions::range::Range::new(0, modulo_number-1);
        let mut rng = rand::OsRng::new().unwrap();

        //On genere les coefficients du polynome
        let mut value_vector: Vec<i64> = (0..=threshold)
            .map(|_| range.sample(&mut rng)) :impl Iterator<Item=?>
            .collect();
        println!("{}", range.sample(&mut rng)
        );

        //On met le secret au degre voulu
        value_vector[threshold] = zero_value;
        return value_vector;
    }

    1 usage  alexis +1
    pub fn share_generation(coefficients: &[i64], point: i64, modulo: i64) -> i64 {
        //Prends les coeff du x au plus grand au plus petit
        let mut reversed_coefficients :Rev<Iter<i64>> = coefficients.iter().rev();

        let haut_deg :i64 = *reversed_coefficients.next().unwrap();
        let tail :Rev<Iter<i64>> = reversed_coefficients;

        //Modulo Q ou pas sur la valeur des shares?
        //tail.fold(haut_deg, |partial, coef| (partial * point + coef) % modulo)
        tail.fold(haut_deg, f: |partial :i64, coef :&i64| (partial * point + coef) % modulo)

    }


    //n = 1
    4 usages  alexis +1
```

```rust
    pub fn polynome_evaluation_coeff(nb_shares:i64,  coefficients: &[i64], modulo : i64) -> Vec<i64> {
        //Generation les shares pour chaque evaluation
        (0.. < nb_shares )
            .map(|point :i64 | share_generation(coefficients, point as i64, modulo)) :impl Iterator<Item=i64>
            .collect()

    }

    let polynome :Vec<i64> = generator_polynome(modulo, secret, threshold: threshold-1);
    println!("Coeff polynome (1,x,x^2,...) -> {:?}", polynome);
    let shares :Vec<i64> = polynome_evaluation_coeff(nb_shares, &polynome, modulo);
    println!("Shares %q -> {:?}", shares);



    ////////////////Seconde partie intermediaire - fonction R //////////////////
    /**
    fn projection_sur_a2(shares: &[i64]) -> f64 {
        let x = [0.0, 1.0, 2.0]; // Points correspondant aux évaluations du polynôme
        //y[0] = share[0], y[1] = share[1], y[2] = share[2]
        let y = [shares[0] as f64, shares[1] as f64, shares[2] as f64];

        let a2 = y[0] / ((x[0] - x[1]) * (x[0] - x[2])) +
            y[1] / ((x[1] - x[0]) * (x[1] - x[2])) +
```

```rust
            y[2] / ((x[2] - x[0]) * (x[2] - x[1]));
        a2
    }
    **/

    1 usage  alexis
    pub fn projection_sur_a2(shares: &[i64]) -> f64 {
        return (shares[1] - shares[0]) as f64;
    }


    let test_gab :f64 = projection_sur_a2(&shares);
    println!("Coefficient récupéré de x^2: {}", test_gab);
```

3. <u>A partir des $i^{\text{èmes}}$ shares de secrets partagés, trouver le $i^{\text{ème}}$ share de l'image de ces secrets par L linéaire</u>

<u>But :</u> Déterminer une fonction $(s_i,u_i,v_i) \mapsto$ i-ème share de L(s,u,v), où

- s, u, v sont des secrets partagés, c'est-à-dire que chaque machine i a les 3 shares s_i, u_i, v_i
- L : $(s,u,v) \mapsto as+bu+cv+d$ est une forme linéaire.

```rust
    ////////////////Troisième partie intermediaire //////////////////


    let polynome_a :Vec<i64> = generator_polynome(modulo, secret, threshold: threshold-1);
    let polynome_b :Vec<i64> = generator_polynome(modulo, secret, threshold: threshold-1);
    let polynome_c :Vec<i64> = generator_polynome(modulo, secret, threshold: threshold-1);
```

```rust

    let shares_a: Vec<i64> = polynome_evaluation_coeff( nb_shares: 3, &polynome_a, modulo);
    let shares_b: Vec<i64> = polynome_evaluation_coeff( nb_shares: 3, &polynome_b, modulo);
    let shares_c: Vec<i64> = polynome_evaluation_coeff( nb_shares: 3, &polynome_c, modulo);
    println!("Shares A %q -> {:?} {:?} {:?}", shares_a, shares_b, shares_c);

    /**
    fn l(s_i: &[i64], u_i: &[i64], v_i: &[i64], a: i32, b: i32, c: i32, d: i32) -> Vec<i64> {
        let mut result = Vec::new();
        for i in 0..s_i.len() {
            let s = s_i[i];
            let u = u_i[i];
            let v = v_i[i];
            let valeur  = a as i64 * s + b as i64 * u + c as i64 * v + d as i64;
            result.push(valeur);
        }

        result
    }
    **/

    1 usage  alexis
    pub fn l(s_i: &[i64], u_i: &[i64], v_i: &[i64], a: i32, b: i32, c: i32, d: i32) -> Vec<i64> {
        let mut result: Vec<i64> = Vec::new();
```

```rust
        for i: usize in 0..s_i.len() {
            let s: i64 = s_i[i];
            let u: i64 = u_i[i];
            let v: i64 = v_i[i];
            let valeur: i64 = a as i64 * s + b as i64 * u + c as i64 * v + d as i64;
            result.push(valeur);
        }
        result
    }


    1 usage  alexis
    fn share_l_i(s_i: i64, u_i: i64, v_i: i64, a: i32, b: i32, c: i32, d: i32) -> i64 {
        let result: i64 = a as i64 * s_i + b as i64 * u_i + c as i64 * v_i + d as i64;
        result
    }



    let l_result: Vec<i64> = l(&shares_a, &shares_b, &shares_c, a: 2, b: 3, c: 4, d: 5);
    let tt: i64 = share_l_i( s_i: shares_a[2], u_i: shares_b[2], v_i: shares_c[2], a: 2, b: 3, c: 4, d: 5);
    println!("L(s,u,v) est :{:?}", l_result);
    println!("i-ème share de L(s,u,v) est :{}", tt);
```

```rust



}
```