PRE week3

작성자 : 김주현

작성일 : 2023년 7월 20일

👉 RWLE (Ring Learning with Errors)

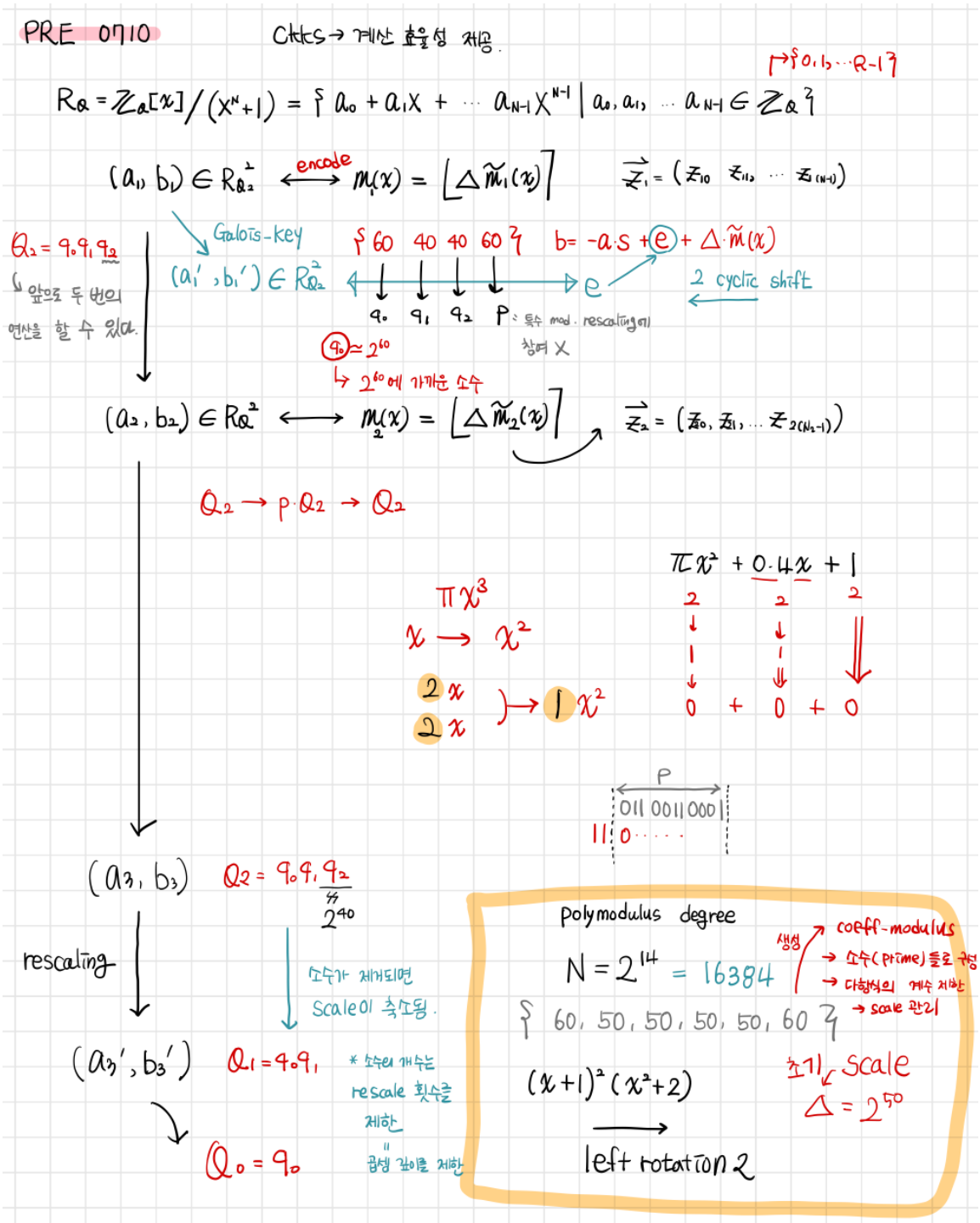
- 암호학에서 사용되는 중요한 기술 중 하나로, 간단히 말해 노이즈가 추가된 선형 방정식 시스템
- RWLE는 환에서 다항식 곰셈과 오차 처리를 통해 이루어지며, 계산적으로 어렵다는 특성을 활용해 안전한 암호 시스템을 구축

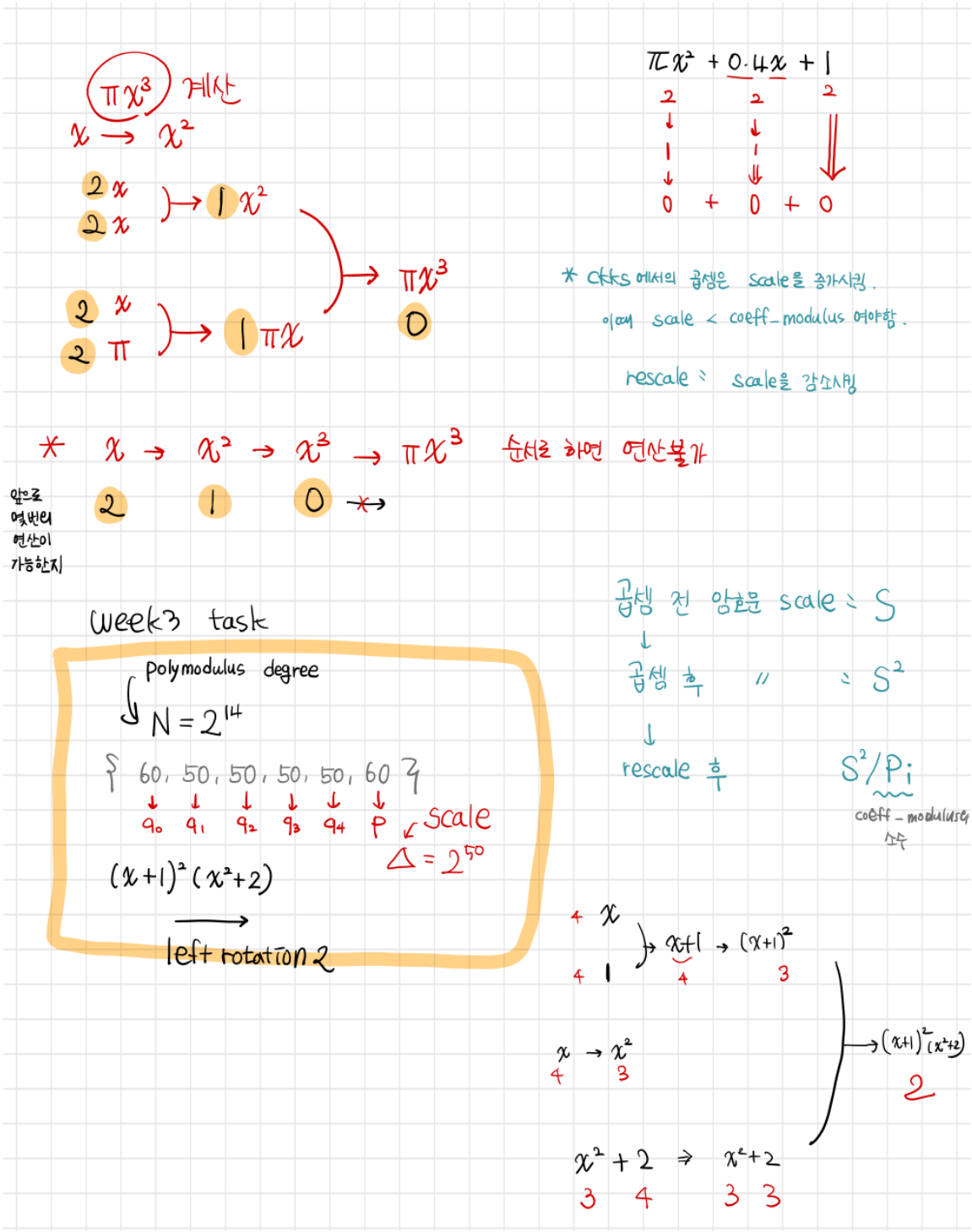
Ring(환) : 숫자들의 집합. 일반적으로 원형구조로 표현됨. 예를 들어 원형 시계에서 1,2,3,..11, 12, 1, 2, 3 과 같은 숫자들이 반복되는 구조가 환이다.

- 데이터를 암호화하여 해독하지 못하도록 보호하며, 다른 알고리즘과 비교했을 때 수학적으로 강력하며 보안성이 높다

▼ RLWE의 암호화 및 복호화

- ✍️암호화 : RLWE에서 메세지를 암호화하려면, 먼저 Ring 위의 랜덤한 값을 선택하고, LWE문제에서의 오류를 추가, 그런 다음 이 값과 공개키를 사용해 메세지를 암호화
- ✍️복호화 : 해당 암호문을 받은 수신자는 비밀키를 사용해 해독 → 원래 메세지를 얻음


PRE 이야기
CKKS → 계산 효율성 제공.
$R_Q = \mathbb{Z}_Q[x]/(x^N+1) = \{a_0 + a_1 x + \cdots a_{N-1}x^{N-1} \mid a_0, a_1, \cdots a_{N-1} \in \mathbb{Z}_Q\}$
↳ {0, 1, ··· Q-1}
$(a_1, b_1) \in R_{Q_2}^2$ ⟷ encode $m_1(x) = \lfloor \Delta \tilde{m}_1(x) \rceil$ $\vec{z}_1 = (z_{10}\ z_{11}\ \cdots\ z_{1(N-1)})$
$Q_2 = q_0 q_1 q_2$
↳ 앞으로 두 번의 연산을 할 수 있다.
Galois-key
$(a_1', b_1') \in R_{Q_2}^2$
{60 40 40 60}
q_0 q_1 q_2 P : 특수 mod. rescaling 시 참여 X
$b = -a \cdot s + e + \Delta \cdot \tilde{m}(x)$
2 cyclic shift
$q_0 \simeq 2^{60}$
↳ 2^{60}에 가까운 소수
$(a_2, b_2) \in R_Q^2$ ⟷ $m_2(x) = \lfloor \Delta \tilde{m}_2(x) \rceil$ $\vec{z}_2 = (z_{20}, z_{21}, \cdots z_{2(N_2-1)})$
$Q_2 \to p \cdot Q_2 \to Q_2$
πx^3
$x \to x^2$
$2x$ $2x$ → $1x^2$
$\pi x^2 + 0.4x + 1$
2 2 2
0 + 0 + 0
P
011 0011 000
1 0 . . . . .
(a_3, b_3) $Q_2 = q_0 q_1 q_2$ ≈ 2^{40}
rescaling
소수가 제거되면 scale이 축소됨.
(a_3', b_3') $Q_1 = q_0 q_1$
$Q_0 = q_0$
* 소수의 개수는 rescale 횟수를 제한
= 곱셈 깊이를 제한
Polymodulus degree
$N = 2^{14} = 16384$
생성 → coeff-modulus
→ 소수(prime)들로 구성
→ 다항식의 계수 저장
→ scale 관리
{60, 50, 50, 50, 50, 60}
초기 scale $\Delta = 2^{50}$
$(x+1)^2(x^2+2)$
left rotation 2



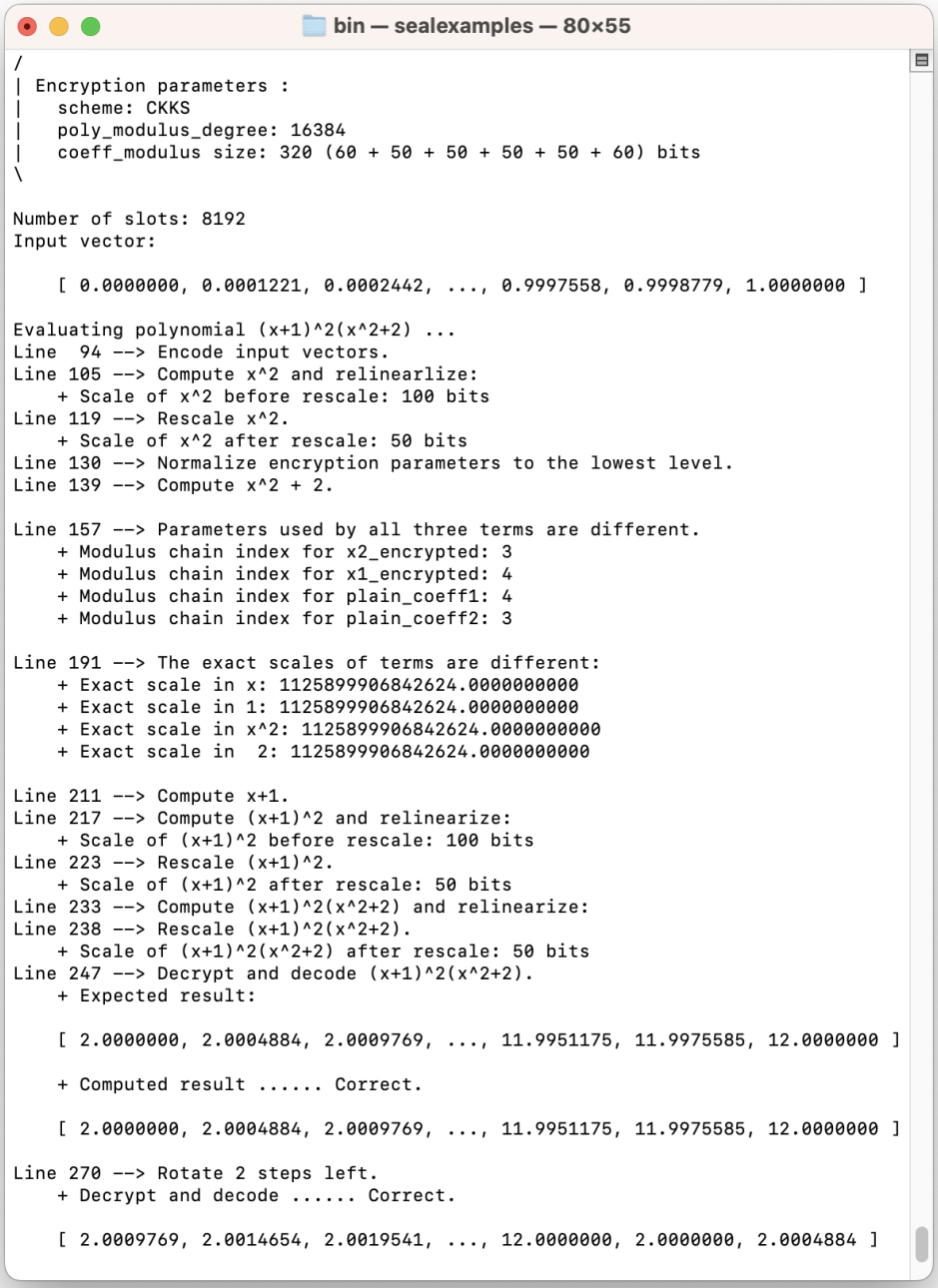

```
// Copyright (c) Microsoft Corporation. All rights reserved.
// Licensed under the MIT license.

#include "examples.h"

using namespace std;
using namespace seal;

void example_ckks_task()
{
    print_example_banner("Example: CKKS task");
    //이 예제에서는 암호화된 부동 소수점 입력 데이터 x에 대해 다음 다항 함수를 평가합니다
```

```
/*
In this example we demonstrate evaluating a polynomial function

    (x+1)^2(x^2+2)

on encrypted floating-point input data x for a set of 4096 equidistant points
in the interval [0, 1]. This example demonstrates many of the main features
of the CKKS scheme, but also the challenges in using it.

We start by setting up the CKKS scheme.
*/


EncryptionParameters parms(scheme_type::ckks);
/*
해당 코드는 CKKS 스키마를 설정하기 위해 EncryptionParameters 객체를 생성하는 부분입니다.
CKKS 스키마를 사용하려면 EncryptionParameters 객체를 생성하고 스키마 유형을 scheme_type::ckks로 설정해야 합니다.*/


size_t poly_modulus_degree = 16384; //2^14
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 50, 50, 50, 50, 60 }));
//CoeffModulus::Create 함수로 적절한 크기의 소수를 생성.
//coeff_modulus의 선택은 암호 연산의 효율성과 결과의 정확성에 영향을 미칩니다.

/*
We choose the initial scale to be 2^50. At the last level, this leaves us
60-40=20 bits of precision before the decimal point, and enough (roughly
10-20 bits) of precision after the decimal point. Since our intermediate
primes are 40 bits (in fact, they are very close to 2^50), we can achieve
scale stabilization as described above.
*/
double scale = pow(2.0, 50);

SEALContext context(parms);
print_parameters(context);
cout << endl;

KeyGenerator keygen(context);
auto secret_key = keygen.secret_key();
PublicKey public_key;
keygen.create_public_key(public_key);
RelinKeys relin_keys;
keygen.create_relin_keys(relin_keys);
GaloisKeys gal_keys;
keygen.create_galois_keys(gal_keys);
Encryptor encryptor(context, public_key);
Evaluator evaluator(context);
Decryptor decryptor(context, secret_key);

CKKSEncoder encoder(context);
size_t slot_count = encoder.slot_count();
cout << "Number of slots: " << slot_count << endl;

vector<double> input;
input.reserve(slot_count);
double curr_point = 0;
double step_size = 1.0 / (static_cast<double>(slot_count) - 1);
for (size_t i = 0; i < slot_count; i++)
{
    input.push_back(curr_point);
    curr_point += step_size;
}
cout << "Input vector: " << endl;
print_vector(input, 3, 7);

cout << "Evaluating polynomial (x+1)^2(x^2+2) ..." << endl;

/*
We create plaintexts for 2 and 1 using an overload of CKKSEncoder::encode
that encodes the given floating-point value to every slot in the vector.
scale(2^50) 값을 이용해 2와 1을 encode.
ex 3.14 * 1000(scale) = 3140 소수는 다루기 어려움. 정수로 바꾸는 작업
*/
```

```
    Plaintext plain_coeff2, plain_coeff1;
    encoder.encode(2, scale, plain_coeff2);
    encoder.encode(1, scale, plain_coeff1);

    //x
    Plaintext x_plain;
    print_line(__LINE__);
    cout << "Encode input vectors." << endl;
    encoder.encode(input, scale, x_plain);
    Ciphertext x1_encrypted;
    encryptor.encrypt(x_plain, x1_encrypted);

    /*
    we first compute x^2 and relinearize. However, the scale has now grown to 2^100.
    scale : 2^100
    */
    Ciphertext x2_encrypted; //x^2
    print_line(__LINE__);
    cout << "Compute x^2 and relinearlize:" << endl;
    evaluator.square(x1_encrypted, x2_encrypted);
    evaluator.relinearize_inplace(x2_encrypted, relin_keys);
    cout << "    + Scale of x^2 before rescale: " << log2(x2_encrypted.scale()) << " bits" << endl;

    /*
    Now rescale; in addition to a modulus switch, the scale is reduced down by
    a factor equal to the prime that was switched away (50-bit prime). Hence, the
    new scale should be close to 2^50. Note, however, that the scale is not equal
    to 2^50: this is because the 50-bit prime is only close to 2^50.
    //x 일 때 Q = q0*q1*q2*q3*q4
    //곱셈 후 x^2의 Q = q0*q1*q2*q3
    */
    print_line(__LINE__);
    cout << "Rescale x^2." << endl;
    evaluator.rescale_to_next_inplace(x2_encrypted);
    cout << "    + Scale of x^2 after rescale: " << log2(x2_encrypted.scale()) << " bits" << endl;

    x2_encrypted.scale() = pow(2.0, 50);

    /*
    x^2: level 3
    2  : level 4  -> level 3 (lowest level로 맞춰주는 작업)
        */
    print_line(__LINE__);
    cout << "Normalize encryption parameters to the lowest level." << endl;
    parms_id_type last_parms_id = x2_encrypted.parms_id();
    evaluator.mod_switch_to_inplace(plain_coeff2, last_parms_id);

    /*
    (x^2 + 1)
    x^2 and 2 ciphertexts are now compatible and can be added.
    */
    print_line(__LINE__);
    cout << "Compute x^2 + 2." << endl;

    Ciphertext encrypted_result2;
    evaluator.add_plain_inplace(x2_encrypted, plain_coeff2);


    /*
    x^2+1
    Now we would hope to compute the sum of two terms(x^2 and 1). However, there is
    a serious problem: the encryption parameters used by all three terms are
    different due to modulus switching from rescaling.

    Encrypted addition and subtraction require that the scales of the inputs are
    the same, and also that the encryption parameters (parms_id) match. If there
    is a mismatch, Evaluator will throw an exception.
    */
    cout << endl;
    print_line(__LINE__);
    cout << "Parameters used by all three terms are different." << endl;
    cout << "    + Modulus chain index for x2_encrypted: "
         << context.get_context_data(x2_encrypted.parms_id())->chain_index() << endl;
    cout << "    + Modulus chain index for x1_encrypted: "
```

```
        << context.get_context_data(x1_encrypted.parms_id())->chain_index() << endl;
    cout << "    + Modulus chain index for plain_coeff1: "
         << context.get_context_data(plain_coeff1.parms_id())->chain_index() << endl;
    cout << "    + Modulus chain index for plain_coeff2: "
         << context.get_context_data(plain_coeff2.parms_id())->chain_index() << endl;
    cout << endl;

    /*
    Let us carefully consider what the scales are at this point. We denote the
    primes in coeff_modulus as P_0, P_1, P_2, P_3, P_4 in this order.
    After the computations
    above the scales in ciphertexts are:

        - Product (x+1)^2 has scale 2^100 and is at level 3
        - We rescaled down to scale 2^100/P_4

        - Product x^2 has scale 2^100 and is at level 3
        - We rescaled down to scale 2^100/P_4

        - To compute (x^2 + 2), 2 must be at level 3. But 2 is now at level 4
        - Rescale 2 (level 4 -> level 3)
        - (x^2 + 2)  is at level 3.

        - Product (x+1)^2 and (x^2+2) has scale (2^100/P_4)^2
        - We rescaled it down to scale (2^100/P_4)^2/P_3 and level 2;

    Although the scales of all three terms are approximately 2^50, their exact
    values are different, hence they cannot be added together.
    */
    print_line(__LINE__);
    cout << "The exact scales of terms are different:" << endl;
    ios old_fmt(nullptr);
    old_fmt.copyfmt(cout);
    cout << fixed << setprecision(10);
    cout << "    + Exact scale in x: " << x1_encrypted.scale() << endl;
    cout << "    + Exact scale in 1: " << plain_coeff1.scale() << endl;
    cout << "    + Exact scale in x^2: " << x2_encrypted.scale() << endl;
    cout << "    + Exact scale in  2: " << plain_coeff2.scale() << endl;

    cout << endl;
    cout.copyfmt(old_fmt);




    /*
    (x + 1)
    x and 1ciphertexts are now compatible and can be added.
    */
    print_line(__LINE__);
    cout << "Compute x+1." << endl;
    evaluator.add_plain_inplace(x1_encrypted, plain_coeff1);

    //(x+1)^2 연산 후 rescale. 결과는 result1_encrypted에
    Ciphertext result1_encrypted;
    print_line(__LINE__);
    cout << "Compute (x+1)^2 and relinearize:" << endl;
    evaluator.square(x1_encrypted, result1_encrypted);
    evaluator.relinearize_inplace(result1_encrypted, relin_keys);
    cout << "    + Scale of (x+1)^2 before rescale: " << log2(result1_encrypted.scale()) << " bits" << endl;

    print_line(__LINE__);
    cout << "Rescale (x+1)^2." << endl;
    evaluator.rescale_to_next_inplace(result1_encrypted);
    cout << "    + Scale of (x+1)^2 after rescale: " << log2(result1_encrypted.scale()) << " bits" << endl;


    //(x+1)^2(x^2+2) 연산 후 rescale. 결과는 result1_encrypted에
    //result1_encrypted : (x+1)^2
    //x2_encrypted : (x^2+2)
    Ciphertext result_encrypted;
    print_line(__LINE__);
    cout << "Compute (x+1)^2(x^2+2) and relinearize:" << endl;
    evaluator.multiply_inplace(result1_encrypted, x2_encrypted);
```

```
    evaluator.relinearize_inplace(result1_encrypted, relin_keys);

    print_line(__LINE__);
    cout << "Rescale (x+1)^2(x^2+2)." << endl;
    evaluator.rescale_to_next_inplace(result1_encrypted);
    cout << "    + Scale of (x+1)^2(x^2+2) after rescale: " << log2(result1_encrypted.scale()) << " bits" << endl;

    /*
    First print the true result.
    */
    Plaintext plain_result;
    print_line(__LINE__);
    cout << "Decrypt and decode (x+1)^2(x^2+2)." << endl;
    cout << "    + Expected result:" << endl;
    vector<double> true_result;
    for (size_t i = 0; i < input.size(); i++)
    {
        double x = input[i];
        true_result.push_back((x+1)*(x+1)*(x*x +2));
    }
    print_vector(true_result, 3, 7);

    /*
    Decrypt, decode, and print the result.
    */

    decryptor.decrypt(result1_encrypted, plain_result); //복호화
    vector<double> result;
    encoder.decode(plain_result, result); //디코딩
    cout << "    + Computed result ...... Correct." << endl;
    print_vector(result, 3, 7);

    Ciphertext rotated;
    Plaintext plain;
    print_line(__LINE__);
    cout << "Rotate 2 steps left." << endl;
    evaluator.rotate_vector(result1_encrypted, 2, gal_keys, rotated);
    cout << "    + Decrypt and decode ...... Correct." << endl;
    decryptor.decrypt(rotated, plain);
    vector<double> result2;
    encoder.decode(plain, result2);
    print_vector(result2, 3, 7);

    /*
    While we did not show any computations on complex numbers in these examples,
    the CKKSEncoder would allow us to have done that just as easily. Additions
    and multiplications of complex numbers behave just as one would expect.
    */
}
```


bin — sealexamples — 80×55
/
| Encryption parameters :
| scheme: CKKS
| poly_modulus_degree: 16384
| coeff_modulus size: 320 (60 + 50 + 50 + 50 + 50 + 60) bits
\
Number of slots: 8192
Input vector:
[0.0000000, 0.0001221, 0.0002442, ..., 0.9997558, 0.9998779, 1.0000000]
Evaluating polynomial (x+1)^2(x^2+2) ...
Line 94 --> Encode input vectors.
Line 105 --> Compute x^2 and relinearlize:
+ Scale of x^2 before rescale: 100 bits
Line 119 --> Rescale x^2.
+ Scale of x^2 after rescale: 50 bits
Line 130 --> Normalize encryption parameters to the lowest level.
Line 139 --> Compute x^2 + 2.
Line 157 --> Parameters used by all three terms are different.
+ Modulus chain index for x2_encrypted: 3
+ Modulus chain index for x1_encrypted: 4
+ Modulus chain index for plain_coeff1: 4
+ Modulus chain index for plain_coeff2: 3
Line 191 --> The exact scales of terms are different:
+ Exact scale in x: 1125899906842624.0000000000
+ Exact scale in 1: 1125899906842624.0000000000
+ Exact scale in x^2: 1125899906842624.0000000000
+ Exact scale in 2: 1125899906842624.0000000000
Line 211 --> Compute x+1.
Line 217 --> Compute (x+1)^2 and relinearize:
+ Scale of (x+1)^2 before rescale: 100 bits
Line 223 --> Rescale (x+1)^2.
+ Scale of (x+1)^2 after rescale: 50 bits
Line 233 --> Compute (x+1)^2(x^2+2) and relinearize:
Line 238 --> Rescale (x+1)^2(x^2+2).
+ Scale of (x+1)^2(x^2+2) after rescale: 50 bits
Line 247 --> Decrypt and decode (x+1)^2(x^2+2).
+ Expected result:
[2.0000000, 2.0004884, 2.0009769, ..., 11.9951175, 11.9975585, 12.0000000]
+ Computed result ..... Correct.
[2.0000000, 2.0004884, 2.0009769, ..., 11.9951175, 11.9975585, 12.0000000]
Line 270 --> Rotate 2 steps left.
+ Decrypt and decode ..... Correct.
[2.0009769, 2.0014654, 2.0019541, ..., 12.0000000, 2.0000000, 2.0004884]