```
void example_ckks_basics()
{    print_example_banner("Example: CKKS Basics");
```

**이 예제의 목적**: - 계산하고자 하는 함수 : $f(x) = \pi x^3 + 0.4x + 1$

- 암호화된 데이터(x) : [0,1]을 $2^{12}$로 쪼갠 precision 가진 float 데이터를 암호화.
- CKKS 로 해야 실수 다룰 수 있다.

**암호문의 scale** : precision 결정하는 파라미터

- 곱셈연산하면 늘어난다.
- 암호문 scale 은 coeff.mod 길이보다 항상 작아야 한다. 안그럼 저장할 공간부족.

**rescaling 이 무엇인지** (scale: 암호문의 scale 로 클수록 precision↑ )

- CKKS 가 제공하는 rescale 으로 scale 을 줄이거나 확장시 안정화시킴.
- 연산 중간 중간에 coeff.mod 의 마지막 소수를 제거, 암호문길이를 줄여서 빠르게 계산하게 해주는 예제 3 의 mod_switch_to()와 비슷하다.
- 정확히, rescale 은 coeff.mod 마지막소수(p)를 제거하면서 scale→scale/p 로 줄게함.
- 피연산자인 두 암호문들의 scale 이 맞아야한다. float 계산생각 $2^{20}+0.1=2^{20}$

**scale 값설정?** scale 값은 coeff.mod 의 소수들과 비슷하게한다.(반드시는 아니고)

- 초기 scale =s ≈ p → (곱셉 1 회)scale=$s^2$ →(rescale)scale=$s^2/p \approx s$ →....
- 따라서, mul.depth(D)만큼 연산해도 scale=s 유지
- 최종 s 는 마지막 소수 p 보다는 작아야 pre-decimal-point 값 저장가능.

- 따라서, **rescale 횟수==coeff.mod 소수갯수 ∝ multiplicative depth(D)(곱셈연산횟수)**
- 보통 scale 을 control 하고 싶어하므로 rescale 기능있는 CKKS 많이씀.

➔ **good strategy** (1) 1st prime of coeff.mod 길이를 60bit 로 하여 precision(복호시)최대 되게함.
(2) special_prime(last)길이도 60bit 로 해서 가장 큰 prime 과 크기 같을수 있게함.
(3) 나머지 D-2 개의 소수들은 60bit 보단 작게, 그들끼리는 같게함.

기본세팅 : **[1] Enc파라미터 [2] Context [3]키 [4]암호기 [5]Evaluator [6]복호기**

**[1] Encryption 파라미터 생성 : + CKKS 의 scale 설정**

- poly.mod.deg(N)=8192=$2^{13}$ : 유효!
- coeff.mod : CoeffModulus::Create 로 비트길이{60,40,40,60}인 소수 4 개의 곱 (찾아줌)
  - coeff.mod 비트길이상한(@N=$2^{13}$)=218 >Σ(bit-length of primes)=200 : 유효!
  - special_prime 크기가 가장큰 소수와 같을 수 있도록 비트길이 설정됨 :유효! plaintext.
- scale = 40bits ($2^{40}$) < 200bits :유효!
  - 소수들의 크기와 비슷하게 설정해서 rescale 할 때 scale=s 유지.
  - (최종 p 길이=60bit)-(scale=40bit)=20bit : pre-decimal-point precision 충분하다.
  - after-decimal-point precision 도 10~20bit 로 충분함 (?)

```
    EncryptionParameters parms(scheme_type::ckks);
    size_t poly_modulus_degree = 8192;
    parms.set_poly_modulus_degree(poly_modulus_degree);
    parms.set_coeff_modulus(CoeffModulus::Create(poly_modulus_degree, { 60, 40, 40, 60 }));
    double scale = pow(2.0, 40);
```

**[2] Context 생성**

```
    SEALContext context(parms);
    print_parameters(context);
```

```
| Encryption parameters :
        |   scheme                : CKKS
        |   poly_modulus_degree: 8192
        |   coeff_modulus size   : 200 (60 + 40 + 40 + 60) bits
```

**[3] 키생성**

```
  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);
```

**[4] 암호기 생성**

```
    Encryptor encryptor(context, public_key);
```

**[5] Evaluator 생성**

```
    Evaluator evaluator(context);
```

**[6] 복호기 생성**

```
    Decryptor decryptor(context, secret_key);
```

메세지(1)→평문(1) : 메세지(실수/복소수 벡터)→ plain 로 batch 화( string 등으로 encoding)한다.

① 일단 CKKSEncoder 생성하면 slot_counter=N/2=4096 으로 저장된다.

② 메세지(실수)벡터 x=input= (input.reserve 뭐임) [0,1]을 $2^{12}$로 나눠채움. AAA.AAAAAAA 형태로 프린트

③ f(x)의 계수들 {π,0.4,1.0}을 {plain_coeff}로 & x=input 을 x_plain 으로 엔코딩한다.

```
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);
Plaintext plain_coeff3, plain_coeff1, plain_coeff0;
encoder.encode(3.14159265, scale, plain_coeff3);
encoder.encode(0.4, scale, plain_coeff1);
encoder.encode(1.0, scale, plain_coeff0);
Plaintext x_plain;
print_line(__LINE__);
cout << "Encode input vectors." << endl;
encoder.encode(input, scale, x_plain);
```

```
Number of slots: 4096
Input vector:
    [ 0.0000000, 0.0002442, 0.0004884, ..., 0.9995116, 0.9997558, 1.0000000 ]
Line 129 --> Encode input vectors.
```

평문→암호 : Encryptor( plain )≡encrypted, fresh 암호벡터의 scale=40 확인.

- BFV 와 같은 방식으로 plain vector(x_plain)를 x1_encrypted 벡터로 암호화한다.
- 나머지 x3_enc/x1_enc.coeff3/enc_result 등은 연산 결과암호문들을 저장할 것이다.

```
Ciphertext x1_encrypted;
encryptor.encrypt(x_plain, x1_encrypted);
Ciphertext x3_encrypted;
Ciphertext x1_encrypted_coeff3;
Ciphertext encrypted_result;
```

연산 : $\pi X^3+0.4X+1.0$= plain_coeff3•$x_3$_encryted + plain_coeff1•$x_1$_encrypted + plain_coeff0 where X = encrypted

① x3_enc = X•X →곱셈뒤엔 반드시 "relinearize" →scale 체크→rescale→scale 체크

② x1_enc_coeff3 = π•X →곱셈뒤엔 반드시 "relinearize" →scale 체크→rescale→scale 체크

③ x3_enc = $X^2$ • πX →곱셈뒤엔 반드시 "relinearize" →scale 체크→rescale→scale 체크

④ x1_enc = 0.4•X →곱셈뒤엔 반드시 "relinearize" →scale 체크→rescale→scale 체크

```
print_line(__LINE__);    cout << "Evaluating polynomial PI*x^3 + 0.4x + 1 ..." << endl;
    cout << "Compute x^2 and relinearize:" << endl;
    evaluator.square(x1_encrypted, x3_encrypted);
    evaluator.relinearize_inplace(x3_encrypted, relin_keys);
    cout << "      + Scale of x^2 before rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
    evaluator.rescale_to_next_inplace(x3_encrypted);
    cout << "      + Scale of x^2 after rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
print_line(__LINE__);    cout << "Compute and rescale PI*x." << endl;
    evaluator.multiply_plain(x1_encrypted, plain_coeff3, x1_encrypted_coeff3);
    cout << "      + Scale of PI*x before rescale: " << log2(x1_encrypted_coeff3.scale()) << " bits" << endl;
    evaluator.rescale_to_next_inplace(x1_encrypted_coeff3);
    cout << "      + Scale of PI*x after rescale: " << log2(x1_encrypted_coeff3.scale()) << " bits" << endl;
print_line(__LINE__);    cout << "Compute, relinearize, and rescale (PI*x)*x^2." << endl;
    evaluator.multiply_inplace(x3_encrypted, x1_encrypted_coeff3);
    evaluator.relinearize_inplace(x3_encrypted, relin_keys);
    cout << "      + Scale of PI*x^3 before rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
    evaluator.rescale_to_next_inplace(x3_encrypted);
    cout << "      + Scale of PI*x^3 after rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
print_line(__LINE__);    cout << "Compute and rescale 0.4*x." << endl;
    evaluator.multiply_plain_inplace(x1_encrypted, plain_coeff1);
    cout << "      + Scale of 0.4*x before rescale: " << log2(x1_encrypted.scale()) << " bits" << endl;
    evaluator.rescale_to_next_inplace(x1_encrypted);
    cout << "      + Scale of 0.4*x after rescale: " << log2(x1_encrypted.scale()) << " bits" << endl;
```

```
Evaluating polynomial PI*x^3 + 0.4x + 1 ...
Line 140 -->Compute x^2 and relinearize:          before rescale: 80 bits    + after rescale: 40 bits
Line 165 -->Compute and rescale PI*x.             before rescale: 80 bits    + after rescale: 40 bits
Line 180 -->Compute, relinearize, rescale (PI*x)*x^2.   before rescale: 80 bits    + after rescale: 40 bits
Line 192 --> Compute and rescale 0.4*x.           before rescale: 80 bits    + after rescale: 40 bits
```

- x3_enc=$x^2$ 와 x1_enc_coeff3=πx 의 scale 이 rescale 덕분에 같음? 곱셈(x3_enc = $\pi x^3$)이 가능하다.
- D=4(Lv3~0) 이고 암호문은 Lv2 가 최상위이니, rescale 1 회한 x3_enc=$x^2$ 와 x1_enc_coeff3=πx 는 Lv1 에 있다. 이들을 다시 곱하고 rescale 한 결과인 x3_enc=$\pi x^3$ 는 Lv0 에 있다.
- rescale 결과는 40bit 에 가깝지 정확히 $2^{40}$ 아니다.

⑤ 세 항들의 level/scale 일치하는지 확인

- 이제 x3_enc=πx$^3$ , x1_enc=0.4x, plain_coeff0=1.0 을 더해야 한다.
-(문제) 이 세 term 들은 각각 Lv0, Lv1,Lv2 에 있어서 Enc 파라미터가 다르다.
- • /+연산은 피연산암호문들의 scale 뿐만 아니라 parms_id 도 같아야 한다. 아니면 예외처리됨.
- primes in coeff_modulus : $P_0$, $P_1$, $P_2$, $P_3$ (**$P_3$ =special prime 으로 rescale 관여안함)**
  - * **x•x** : 80@level 2; **π•x** :80@level 2; --->(rescaled) $2^{80}/P_2$@level 1;
  - * **x²•πx** : $(2^{80}/P_2)^2$; --->(rescaled) $(2^{80}/P_2)^2/P_1$@level0;
  - * **0.4•x** : $2^{80}$@level2; --->(rescaled) $2^{80}/P_2$@level 1;
  - ***1** : $2^{40}$@level 2;

→scale 이 40bit 로 같아보이지만 정확히 일치하지도 않다. level 도 scale 도 불일치상태.

```
print_line(__LINE__);
    cout << "Parameters used by all three terms are different." << endl;
    cout << "      + Modulus chain index for x3_encrypted: "
         << context.get_context_data(x3_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_coeff0: "
         << context.get_context_data(plain_coeff0.parms_id())->chain_index() << endl;
    cout << endl;
```

```
Line 209 --> Parameters used by all three terms are different.
    + Modulus chain index for x3_encrypted: 0
    + Modulus chain index for x1_encrypted: 1
    + Modulus chain index for plain_coeff0: 2
```

```
print_line(__LINE__);
    cout << "The exact scales of all three terms are different:" << endl;
    ios old_fmt(nullptr);
    old_fmt.copyfmt(cout);
    cout << fixed << setprecision(10);
    cout << "      + Exact scale in PI*x^3: " << x3_encrypted.scale() << endl;
    cout << "      + Exact scale in  0.4*x: " << x1_encrypted.scale() << endl;
    cout << "      + Exact scale in      1: " << plain_coeff0.scale() << endl;
    cout << endl;
    cout.copyfmt(old_fmt);
```

```
Line 237 --> The exact scales of all three terms are different:
    + Exact scale in PI*x^3:  1099512659965.7514648438
    + Exact scale in  0.4*x:  1099511775231.0197753906
    + Exact scale in      1:  1099511627776.0000000000
```

⑥ fix the problem!!!

[scale match]

- $P_1$, $P_2$ 비슷하니까 SEAL 이 $P_1$=$P_2$ 로 인식하게 lie!
  - 그러면 **x²• πx** 항의 최종 scale = $2^{40}$ 은 단순히 이 term 을 $2^{120}/(P_2^2 \cdot P_1) \approx 1$ 로 스케일링한 것을 의미한다. 이는 눈에띄는 error 없도록 한다.
- **1** 을 scale= $2^{80}/P_2$, 로 엔코딩하여 **0.4•x** 에 곱해서 rescale 하여 Lv0 으로 내릴 수도 있음.
- 여기서는 가장 간단한 방법 즉, **x²• πx** 와 **0.4•x** 스케일 바꾸는 방식을 택함.

[level match]

- rescale 없이 mod_switch_to()로 죄다 x3_enc 의 parms_id 즉 Lv0 으로 보내버리면 됨.

```
    print_line(__LINE__);
    cout << "Normalize encryption parameters to the lowest level." << endl;
    parms_id_type last_parms_id = x3_encrypted.parms_id();
    evaluator.mod_switch_to_inplace(x1_encrypted, last_parms_id);
    evaluator.mod_switch_to_inplace(plain_coeff0, last_parms_id);
```

```
Line 262 --> Normalize scales to 2^40.
Line 273 --> Normalize encryption parameters to the lowest level.
```

⑦ 이제 더하자.

```
print_line(__LINE__);
    cout << "Compute PI*x^3 + 0.4*x + 1." << endl;
    evaluator.add(x3_encrypted, x1_encrypted, encrypted_result);
    evaluator.add_plain_inplace(encrypted_result, plain_coeff0);
    Plaintext plain_result;
```

```
Line 282 --> Compute PI*x^3 + 0.4*x + 1.
```

결과 복호화 : 결과(encrypted result )→복호(plain result )→디코딩(result) vs. true result

```
    print_line(__LINE__);
    cout << "Decrypt and decode PI*x^3 + 0.4x + 1." << 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((3.14159265 * x * x + 0.4) * x + 1);
    }
    print_vector(true_result, 3, 7);

    decryptor.decrypt(encrypted_result, plain_result);
```

```
    vector<double> result;
    encoder.decode(plain_result, result);
    cout << "      + Computed result ..... Correct." << endl;
    print_vector(result, 3, 7);
}
```

```
Line 292 --> Decrypt and decode PI*x^3 + 0.4x + 1.
    + Expected result:
    [ 1.0000000, 1.0000977, 1.0001954, ..., 4.5367965, 4.5391940, 4.5415926 ]
    + Computed result ..... Correct.
    [ 1.0000000, 1.0000977, 1.0001954, ..., 4.5367995, 4.5391970, 4.5415956 ]
```