```
void example_batch_encoder(){
    print_example_banner("Example: Encoders / Batch Encoder");
```

BFV 방식

- 계산은 %(plain.mod)로 이뤄지고
- plaintext 을 다항식으로 표현하는데 다항식의 딱 한계수만 사용한다.

문제 1) 유용한 계산은 정수/실수산술이지 모듈로가 아님.

➔해결책? plain.mod 를 크게하면 거의 모든 계산이 모듈로가 아니게됨.
그러나 plain.mod 크게하면 노이즈예산 초기값↓, 소비율↑

문제 2) plaintext-다항식이 크고, 전체적으로 암호화되는데 한계수만 쓰면 낭비임.

- 여기서 시도한 방식은 , 메세지를 평문화할 때 단순 string 화가 아닌, 계산횟수도 늘리면서, plaintext-다항식의 계수 전체를 사용하도록 하는 Encoding 방식을 사용한다.

BFV Encoding 방식

- batch 화 하면 plaintext-다항식을 행렬로 표현할 수 있다.
  - **[dimension]**= 2x(N/2) where N=poly.mod.deg, **[요소 value]**= a_{ij}%T where T = plain.mod
  - 2D 행렬➔1D 로 flatten 했을 때(?) $\{a_{ij}\}$ 를 slot 이라고함 (slot count=#tot$\{a_{ij}\}$=N)
- 아주 간단한것 제외하면 거의 모든경우, batch 화하면 무조건 빨라지는 등 이점多.

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

[1] Encryption 파라미터 클래스 생성

① poly.mod.deg = 2^n=2^{13}=8192

② coeff.mod : 3 개 이상 소수의 곱, 길이상한(@poly.mod.deg=8192) =218 알아서 찾아줌.

③ plain.mod : batch 쓰려면 **plain.mod=1%2N** 조건 만족하는 소수, 선택한 비트길이(ex)20 로 찾아줌.

```
    EncryptionParameters parms(scheme_type::bfv);
    size_t poly_modulus_degree = 8192;
    parms.set_poly_modulus_degree(poly_modulus_degree);
    parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
    parms.set_plain_modulus(PlainModulus::Batching(poly_modulus_degree, 20));
```

[2] Context 생성 파라미터 정보/ 유효성/ batch 기능 enabled 되었는지 확인가능

```
    SEALContext context(parms);
    print_parameters(context);
    cout << endl;
    auto qualifiers = context.first_context_data()->qualifiers();
    cout << "Batching enabled: " << boolalpha << qualifiers.using_batching << endl;
```

```
| Encryption parameters :
        |   scheme              : BFV
        |   poly_modulus_degree: 8192
        |   coeff_modulus size   : 218 (43 + 43 + 44 + 44 + 44) bits
        |   plain_modulus       : 1032193
Batching enabled: true
```

[3] 키생성 개인/공개/재선형화 키 생성

```
    KeyGenerator keygen(context);
    SecretKey secret_key = keygen.secret_key();
    PublicKey public_key;
    keygen.create_public_key(public_key);
    RelinKeys relin_keys;
    keygen.create_relin_keys(relin_keys);
```

[4] 암호화기 생성

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

[5] Evaluator 생성

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

[6] 복호화기 생성

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

메세지(1)➔평문(1) : 메세지(1D 숫자행렬)➔ 2D plain matrix 로 batch 화(string 등으로 encoding)한다.

- batch 화는 "slot count"=N = 8192 라고 저장한다.
  - ① [pod_matrix] 메세지를 1D 행렬=1 x slont count = 1 x 8192 로 만든다.
  - ② [plain_matrix] ①을 2D=2 x (slot count/2) = 2 x 4096 의 toString(엔코딩)한 plain matrix 만듦.
  - ③ [pod_result] ②을 디코딩해서 데이터이 pod_matrix 나오는지 확인

```
    BatchEncoder batch_encoder(context);
    size_t slot_count = batch_encoder.slot_count();
    size_t row_size = slot_count / 2;
    cout << "Plaintext matrix row size: " << row_size << endl;
    vector<uint64_t> pod_matrix(slot_count, 0ULL);
    pod_matrix[0] = 0ULL;
    pod_matrix[1] = 1ULL;
    pod_matrix[2] = 2ULL;
    pod_matrix[3] = 3ULL;
    pod_matrix[row_size] = 4ULL;
    pod_matrix[row_size + 1] = 5ULL;
    pod_matrix[row_size + 2] = 6ULL;
    pod_matrix[row_size + 3] = 7ULL;
    cout << "Input plaintext matrix:" << endl;
    print_matrix(pod_matrix, row_size);
    Plaintext plain_matrix;
    print_line(__LINE__);
    cout << "Encode plaintext matrix:" << endl;
```

```
    batch_encoder.encode(pod_matrix, plain_matrix);
    vector<uint64_t> pod_result;
    cout << "      + Decode plaintext matrix ..... Correct." << endl;
    batch_encoder.decode(plain_matrix, pod_result);
    print_matrix(pod_result, row_size);
```

```
Plaintext matrix row size: 4096
Input plaintext matrix:
    [  0,   1,   2,   3,   0, ...,   0,   0,   0,   0,   0 ]
    [  4,   5,   6,   7,   0, ...,   0,   0,   0,   0,   0 ]
Line 124 --> Encode plaintext matrix:
+ Decode plaintext matrix ..... Correct.
    [  0,   1,   2,   3,   0, ...,   0,   0,   0,   0,   0 ]
    [  4,   5,   6,   7,   0, ...,   0,   0,   0,   0,   0 ]
```

평문(1)→암호(2) : Encryptor(plain matrix)≡cipher matrix, fresh 암호행렬의 노이즈예산 확인.

```
    Ciphertext encrypted_matrix;
    print_line(__LINE__);
    cout << "Encrypt plain_matrix to encrypted_matrix." << endl;
    encryptor.encrypt(plain_matrix, encrypted_matrix);
    cout << "      + Noise budget in encrypted_matrix: " <<
                    decryptor.invariant_noise_budget(encrypted_matrix) << " bits"<< endl;
```

```
Line 141 --> Encrypt plain_matrix to encrypted_matrix.
    + Noise budget in encrypted_matrix: 146 bits
```

메세지(2)→평문(2) : 메세지(1D 숫자행렬)→ 2D plain matrix 로 batch 화(string 등으로 encoding)한다.

- 앞의 암호행렬에 (+,•)연산할 두번째 행렬만드는 과정임 (dim 같음).
- c+1 에서 처럼 '1'을 plain 그대로 더하고 곱하는 function 있으니깐 따로 encrypt 안해도 됨.

```
    vector<uint64_t> pod_matrix2;
    for (size_t i = 0; i < slot_count; i++){
        pod_matrix2.push_back((i & size_t(0x1)) + 1);
    }
    Plaintext plain_matrix2;
    batch_encoder.encode(pod_matrix2, plain_matrix2);
    cout << endl;
    cout << "Second input plaintext matrix:" << endl;
    print_matrix(pod_matrix2, row_size);
```

```
Second input plaintext matrix:
    [  1,   2,   1,   2,   1, ...,   2,   1,   2,   1,   2 ]
    [  1,   2,   1,   2,   1, ...,   2,   1,   2,   1,   2 ]
```

연산 : $(X+A)^2$ where X = cipher matrix(1), A=plain matrix(2)

- X = X + A
- $X = X^2$ →곱셈뒤엔 반드시 "relinearize"
- 곱셈연산 한번 한 다음 노이즈예산 얼마나 남았는지 확인(146→114bits : Δ=-32)

```
    print_line(__LINE__);
    cout << "Sum, square, and relinearize." << endl;
    evaluator.add_plain_inplace(encrypted_matrix, plain_matrix2);
    evaluator.square_inplace(encrypted_matrix);
    evaluator.relinearize_inplace(encrypted_matrix, relin_keys);
    cout << "      + Noise budget in result: "<<decryptor.invariant_noise_budget(encrypted_matrix)<<" bits"<< endl;
```

```
Line 172 --> Sum, square, and relinearize.
    + Noise budget in result: 114 bits
```

결과 복호화 : 결과(cipher matrix)→복호(plain matrix)→디코딩(message)

```
    Plaintext plain_result;
    print_line(__LINE__);
    cout << "Decrypt and decode result." << endl;
    decryptor.decrypt(encrypted_matrix, plain_result);
    batch_encoder.decode(plain_result, pod_result);
    cout << "      + Result plaintext matrix ..... Correct." << endl;
    print_matrix(pod_result, row_size);
}
```

```
Line 187 --> Decrypt and decode result.
    + Result plaintext matrix ..... Correct.
    [  1,   9,   9, 25,   1, ...,   4,   1,   4,   1,   4 ]
    [ 25, 49, 49, 81,   1, ...,   4,   1,   4,   1,   4 ]
```

```
void example_ckks_encoder(){
    print_example_banner("Example: Encoders / CKKS Encoder");
```

- BFV Encoding-bacth (장) plaintext-다항식 전체사용, 암호문연산을 병렬화→ 훨씬 빠름.
  (단) 여전히 행렬요소값은 a_{ij}%plain.mod : 모듈러스 계산 피하려 plain.mod 크게하면 노이즈예산의 초기값↓, 소비율↑, 금세 overflow 초래한다.
- CKKS Encoding -batch (장) overflow 문제를 해결하고자 한다. (overflow 는 연산중 감지 X, 큰문제)
  (단) 단, 결과는 정확하지 않은 근사값만을 제공한다.

CKKS Encoding 방식 Cheon-Kim-Kim-Song (CKKS) : 암호화된 정수/실수/복소수에 연산을 가능케한다.

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

[1] Encryption 파라미터 클래스 생성

- BFV 와 다른점 (1) plain.mod(T)없어
  (2) coeff.mod 찾는 방식중요. (ex)"CoeffModulus::Create"로 비트길이=40 인 소수생성함.

```
    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, { 40, 40, 40, 40, 40 }));
```

[2] Context 생성

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

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

[4] 암호화기 생성

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

[5] Evaluator 생성

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

[6] 복호화기 생성

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

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

- CKKSEncoder/BatchEncoder 둘다 "다중 message→다중 plain" 의 task 수행하지만 이론이 완전 다름.
- CKKSEncoder 는 메세지(1D 실수/복소수 "벡터"를 plaintext-obj 로 엔코딩한다.
- "slot_count"=N/2 = 8192/2=4096 로 저장함 (앞선 BFV-BatchEncoder 는 slot_count=N 였음)
  ① [input] 메세지를 크기=4 인 벡터에 채운다 (정수 아니어도 실수/복소수가능)
  ② [plain] ①을 coeff.mod 비트길이 상한=5•40bits 보다는 작은 scale=**30**bit 크기의 벡터로, 엔코딩한다. 빈 부분은 0 으로 패딩함. *** scale: the scale as determining the bit-precision of the encoding; naturally it will affect the precision of the result.**
  ③ [output] ②를 디코딩해서 원본메세지 input (나머지는 0.0 으로 패딩된) 나오는지 확인

```
    CKKSEncoder encoder(context);
    size_t slot_count = encoder.slot_count();
    cout << "Number of slots: " << slot_count << endl;
    vector<double> input{ 0.0, 1.1, 2.2, 3.3 };
    cout << "Input vector: " << endl;
    print_vector(input);
    Plaintext plain;
    double scale = pow(2.0, 30);
    print_line(__LINE__);
    cout << "Encode input vector." << endl;
    encoder.encode(input, scale, plain);
    vector<double> output;
    cout << "      + Decode input vector ..... Correct." << endl;
    encoder.decode(plain, output);
    print_vector(output);
```

```
Number of slots: 4096
Input vector:   [ 0.000, 1.100, 2.200, 3.300 ]
Line 297 --> Encode input vector.
      + Decode input vector ..... Correct.
                [ -0.000, 1.100, 2.200, 3.300, ..., 0.000, -0.000, 0.000, -0.000 ]
```

평문(1)→암호(2) : Encryptor(plain)≡encrypted, fresh 암호벡터의 scale=30 확인.

- BFV 와 같은 방식으로 plain vector 를 encrypted 벡터로 암호화한다.

```
    Ciphertext encrypted;
    print_line(__LINE__);
    cout << "Encrypt input vector, square, and relinearize." << endl;
    encryptor.encrypt(plain, encrypted);
    cout<<"      + Scale in input: " << encrypted.scale() <<"("<< log2(encrypted.scale()) <<" bits)"<< endl;
```

```
Line 313 --> Encrypt input vector, square, and relinearize.
      + Scale in              input: 1.07374e+09 (30 bits)
```

연산 : X^2 where X = encrypted

- $X = X^2$ →곱셈뒤엔 반드시 "relinearize"
- 곱셈연산 한번 한 다음 scale 확인 (2^{30} (30bits)→2^{60} (60bits): Δ=+30???)

```
    evaluator.square_inplace(encrypted);
    evaluator.relinearize_inplace(encrypted, relin_keys);
   cout<<"      + Scale in squared input: " << encrypted.scale() <<"("<< log2(encrypted.scale()) <<" bits)"<< endl;
```

```
     + Scale in squared input: 1.15292e+18 (60 bits)
```

결과 복호화 : 결과(encrypted)→복호(plain)→디코딩(output)

```
    print_line(__LINE__);
    cout << "Decrypt and decode." << endl;
    decryptor.decrypt(encrypted, plain);
    encoder.decode(plain, output);
    cout << "      + Result vector ..... Correct." << endl;
print_vector(output); }
```

```
Line 333 --> Decrypt and decode.
     + Result vector ..... Correct.
    [ -0.000, 1.210, 4.840, 10.890, ..., -0.000, 0.000, 0.000, -0.000 ]
```

→output 벡터사이즈는 scale=30 → slot_count = N/2 로 더더욱 늘어났고, 빈부분은 0 으로 패딩되었는데 이는 모호한 0-padding 의 결과다.

무슨뜻인지 모르겠는 엔딩:

The CKKS scheme allows the scale to **be reduced (30→60 increased, no??????)** between encrypted computations. This is a fundamental and critical feature that makes CKKS very powerful and flexible. We will discuss it in great detail in '3_levels.cpp' and later in '4_ckks_basics.cpp'.

```
void example_encoders()
{
    print_example_banner("Example: Encoders");
    example_batch_encoder();
    example_ckks_encoder();
}
```