Background

- CKKS 스킴

CKKS 는 완전한 동형 암호 스키마로 Cheon, Kim, Kim, Song 의 준말이다. CKKS 는 부동 소수점 연산에 특화되어 있으며 주로 연속적인 신호 처리나 기계 학습에 주로 사용된다. 이를 위해 CKKS 는 다항식을 사용하여 부동 소수점 숫자를 표현하고 다항식의 계수를 암호화하여 계산을 수행한다. 다항식 계수는 높은 정밀도를 유지하기 위해 여러 개의 동일한 크기의 부동 소수점 숫자로 구성된다.

CKKS 는 정밀하게 노이즈를 완리하는 BFV BGV 와 달리 복호화시 가장 가까운 거리의 값을 취하는 방식을 사용하였다.

- 연산

덧셈, 곱셈, 스칼라 곱셈, 회전 연산을 지원한다. CKKS 는 계수 스케일을 조정하여 오버플로우와 언더플로우 문제를 회피하며 이를 통해 암호화된 부동 소수점 숫자를 안정적으로 계산할 수 있다.

- CKKS 매개변수

1. 다항식의 차수 : 다항식의 차수가 높을수록 더 복잡한 계산을 수행할 수 있다. 하지만 암호화된 데이터의 크기와 계산에 소요되는 시간이 증가할 수 있다.
2. 모듈러스 목록 : 모듈러스는 다항식의 계수를 유한체에서 선택된 소수로 나눈 나머지로 표현하는데 사용된다. 모듈러스의 크기가 클수록 더 높은 정밀도를 제공하고 암호화된 데이터의 크기와 계싼에 소요되는 시간이 증가할 수 있다.
3. 스케일 파라미터 : 다항식의 계수를 실제 부동 소수점 값으로 변환하는데 사용되는 파라미터이다. 부동 소수점 값의 정밀도와 관련이 있으며 적절한 스케일 설정은 암호화된 데이터의 정밀도와 연산 결과의 정확성에 영향을 줄 수 있다.

- 하위비트에 노이즈 값

BFV, CKKS 동형암호 스킴은 암호문 하위비트에 노이즈 값이 추가되는 특성으로 인해 암호문들의 반복 연산 횟수가 누적되면 사이즈가 커진 노이즈 값이 메시지 값을 침범하게 되는 성질이 있다. 그러므로 침범하지 않는 수준까지만 반복 연산이 수행되도록 연산 횟수를 제한해야한다. 따라서 한계에 도달한 암호문의 노이즈 크기를 축소하는 부가적인 Relinearizaiton 또는 Bootstapping 연산을 수행해야한다.

Ckks_basics.cpp

암호화 과정은 다음과 같다.

1. PI * x^3 + 0.4 * x^2 + 1 에서 각 계수를 암호화한다.
2. 각 자릿수를 암호화하고 재선형화하는 작업을 수행
3. 암호화를 마친 후 각 자릿수를 더하기 위해 스케일을 정규화
4. 디코딩을 하여 결과를 확인

아래는 자세한 코드

```
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 }));
```

poly_modulus_degree : 변수의 다항식의 차수를 나타낸다. CKKS 스킴에서 사용되는 다항식의 최고차수를 의미한다.

주어진 다항식의 차수와 모듈러스 비트 크기의 목록을 기반으로 모듈러스를 생성한다. 여기선 60 40 40 60 비트의 모듈러스를 생성하여 파라미터를 설정. 이 모듈러스는 다항식의 계수를 암호화하는데 사용된다. 각 비트는 소수를 표현한다.

```
double scale = pow(2.0, 40);

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

스케일은 2 의 40 승으로 설정 스케일은 암호화된 다항식 계수를 실제 부동 소수점 값으로 변환하는 데 사용되는 파라미터이다. 암호화된 데이터의 정밀도는 2 의 40 승 크기를 가진다. 즉, 암호화된 데이터는 소수점 이전에 약 20 비트의 정밀도를 가지고 이후에도 약 10~20 비트의 정밀도를 유지한다.

```
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);
```

이 코드에서는 키를 생성하고 암호화, 평가, 복호화를 위한 객체들을 초기화하는 과정을 나타낸다.

KeyGenerator 는 context 를 기반으로 생성되며 암호화에 대한 키를 생성한다. 공개키, 비밀키, 재선형화 키, 갈루아 키를 생성한다. 재선형화키는 곱셈 연산의 결과를 다항식 차수를 줄여 노이즈 크기를 줄이고 갈루아 연산키는 다항식 요소들을 순환적으로 이동시키는 회전 연산을 수행한다.

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

CKKS 스킴에서 사용되는 CKKSEncoder 객체를 생성하고 슬롯의 개수를 확인한다. CKKSEncoder 는 부동 소수점 값을 다항식으로 인코딩하거나 다항식에서 부동 소수점 값을 디코딩하는데 사용된다. 위에서 설정한 파라미터들로 이루어진 context 를 기반으로 만들어진다.

```
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);
```

다항식을 평가하기 위한 입력벡터를 생성하는 과정이다. 벡터의 용량은 slot 사이즈만큼 정해지며 이 슬롯의 개수는 CKKSEncoder 에서 확인했다. 위의 과정을 통해 입력 벡터가 생성되고 평가할 다항식이 출력된다. 이 다항식은 PI*x^3 + 0.4*x + 1 로 정의되어 있다.

```
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);
```

평문 객체를 생성하고 부동 소수점 값을 다항식 계수로 인코딩하는 과정을 나타낸다. Encode 함수를 사용하여 PI, 0.4, 1.0 으로 인코딩하여 저장한다.

```
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);
```

입력 벡터를 인코딩하기 위한 평문 객체를 생성하고 input 을 다항식 계수로 인코딩하여 저장한다. 이 때 scale 파라미터를 사용하여 인코딩된 다항식의 스케일을 설정한다. 암호화된 값은 암호문 객체를 생성하여 저장한다. X_pain 을 암호화하여 x1_encrypted 에 저장해준다. 이러한 암호화된 값은 다음 단계에서 다항식 계산에 활용된다.

```
Ciphertext x3_encrypted;
print_line(__LINE__);
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;
```

암호화된 값에서 제곱 연산을 수행하고 그 결과를 다시 선형화 하는 과정을 나타낸다. 위에서 암호화한 값을 제곱 연산하고 x3_encrypted 에 저장한다. 여기서 재선형화는 다항식을 효율적으로 다시 암호화하는 과정으로 암호문의 크기를 줄이고 효율성을 향상시킨다. 스케일을 출력하여 정밀도를 확인한다. 여기서는 80bits 를 출력한다.

```
print_line(__LINE__);
cout << "Rescale x^2." << endl;
evaluator.rescale_to_next_inplace(x3_encrypted);
cout << "    + Scale of x^2 after rescale: " << log2(x3_encrypted.scale()) << " bits" << endl;
```

X^2 의 값을 스케일 조정한다. Evaluator 의 함수로 스케일 조정으로 암호화의 스케일을 줄이고 정밀도는 유지한다. 리스케줄이후에는 40bits 를 출력한다.
CKKS 스킴에서 암호문의 스케일을 유지하며 연산을 수행하기 위한 단계이다.

```
/* ... */
print_line(__LINE__);
cout << "Compute and rescale PI*x." << endl;
Ciphertext x1_encrypted_coeff3;
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;
```

```
Line 140 --> Compute x^2 and relinearize:
    + Scale of x^2 before rescale: 80 bits
Line 152 --> Rescale x^2.
    + Scale of x^2 after rescale: 40 bits
Line 165 --> Compute and rescale PI*x.
    + Scale of PI*x before rescale: 80 bits
    + Scale of PI*x after rescale: 40 bits
Line 180 --> Compute, relinearize, and rescale (PI*x)*x^2.
    + Scale of PI*x^3 before rescale: 80 bits
    + Scale of PI*x^3 after rescale: 40 bits
Line 192 --> Compute and rescale 0.4*x.
    + Scale of 0.4*x before rescale: 80 bits
    + Scale of 0.4*x after rescale: 40 bits
```

위에서 x^2 을 계산하였다. 그 값을 이용해 목표 계산인 PI * x^3 + 0.4 * x + 1 을 위해 PI * x^3, 0.4 * x 를 계산하고 재선형화하고 스케줄링하는 연산을 반복한다.

```
print_line(__LINE__);
cout << "Normalize scales to 2^40." << endl;
x3_encrypted.scale() = pow(2.0, 40);
x1_encrypted.scale() = pow(2.0, 40);
```

각 암호문의 스케일을 2^40 으로 정규화한다. 각 암호문들과의 연산을 수행할 수 있도록한다. 스케일 일치는 동형 암호 시스템에서 연산의 성공적인 수행을 위해 필요한 작업이다.

```
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);
```

암호화 매개변수를 가장 낮은 레벨로 정규화하는 과정이다. 각 암호문의 매개변수를 동일한 매개변수로 정규화한다. 암호문들 간에 매개변수가 일치해야 연산이 가능하다. 그러므로 최하위 레벨로 일치시킨다.

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

위에서 암호화한 객체들을 덧셈하여 원하는 값으로 나타냈다. 이것들이 가능하려면 암호화 매개변수의 일치가 연산의 성공을 보장한다.

```
Plaintext plain_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);

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

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

위의 식을 통해 다항식의 예상 결과가 저장되고 계산된 결과는 Result 에 저장된다. 각 인풋을 넣어서 나온 결과를 확인해보면 다음과 같다.

```
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, 4.5415957 ]
```

Rotation.cpp

Rotation in BFV

BFV 암호화는 다항식 모듈러스 연산과 레벨링 기법을 통해 부분 정수론을 이용하여 암호화된 데이터에 대한 계산을 가능하게 한다.

```
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));

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

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);
Encryptor encryptor(context, public_key);
Evaluator evaluator(context);
Decryptor decryptor(context, secret_key);

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;
```

결정한 파라미터를 기반으로 공개키, 비밀키, 재선형화 키를 생성하고 암호화한다. 벡터-평문 변환을 위한 배치 인코더를 초기화한다.
배치 인코더는 평문 벡터의 원소들을 암호문 벡터로 변환한다. 이 과정에서 다항식 모듈러스 연산에 필요한 암호화 매개변수와 암호화 키가 사용된다. 디코딩시에는 암호문 벡터를 평문 벡터로 변환시킨다.

```
Plaintext plain_matrix;
print_line(__LINE__);
cout << "Encode and encrypt." << endl;
batch_encoder.encode(pod_matrix, plain_matrix);
Ciphertext encrypted_matrix;
encryptor.encrypt(plain_matrix, encrypted_matrix);
cout << "    + Noise budget in fresh encryption: " << decryptor.invariant_noise_budget(encrypted_matrix) << " bits"
     << endl;
cout << endl;
```

pod_matrix 를 plain_matrix 로 인코딩한다. 여기서 batch_encoder 는 정수벡터를 평문으로 인코딩하는데 사용된다. 평문화된 값을 암호화한다.

```
GaloisKeys galois_keys;
keygen.create_galois_keys(galois_keys);
```

회전 연산에 필요한 갈루아 키를 생성한다.

```
/* ... */
print_line(__LINE__);
cout << "Rotate rows 3 steps left." << endl;
evaluator.rotate_rows_inplace(encrypted_matrix, 3, galois_keys);
Plaintext plain_result;
cout << "    + Noise budget after rotation: " << decryptor.invariant_noise_budget(encrypted_matrix) << " bits"
     << endl;
cout << "    + Decrypt and decode ...... Correct." << endl;
decryptor.decrypt(encrypted_matrix, plain_result);
batch_encoder.decode(plain_result, pod_matrix);
print_matrix(pod_matrix, row_size);

/* ... */
print_line(__LINE__);
cout << "Rotate columns." << endl;
evaluator.rotate_columns_inplace(encrypted_matrix, galois_keys);
cout << "    + Noise budget after rotation: " << decryptor.invariant_noise_budget(encrypted_matrix) << " bits"
     << endl;
cout << "    + Decrypt and decode ...... Correct." << endl;
decryptor.decrypt(encrypted_matrix, plain_result);
batch_encoder.decode(plain_result, pod_matrix);
print_matrix(pod_matrix, row_size);
```

왼쪽으로 3 단계 회전 후 노이즈 크기, 모든 열을 회전시킨 후 노이즈를 비교한다. 모두 같은 노이즈를 확인할 수 있다. 그리고 복호하면 정확히 회전연산이 수행된 것을 확인할 수 있다.

```
print_line(__LINE__);
cout << "Rotate rows 4 steps right." << endl;
evaluator.rotate_rows_inplace(encrypted_matrix, -4, galois_keys);
cout << "    + Noise budget after rotation: " << decryptor.invariant_noise_budget(encrypted_matrix) << " bits"
     << endl;
cout << "    + Decrypt and decode ...... Correct." << endl;
decryptor.decrypt(encrypted_matrix, plain_result);
batch_encoder.decode(plain_result, pod_matrix);
print_matrix(pod_matrix, row_size);
```

마지막으로 오른쪽으로 4 번 회전한 결과를 출력한다.

```
Line  84 --> Rotate rows 3 steps left.
    + Noise budget after rotation: 142 bits
    + Decrypt and decode ...... Correct.

    [  3,  0,  0,  0,  0, ...,  0,  0,  0,  1,  2 ]
    [  7,  0,  0,  0,  0, ...,  0,  0,  4,  5,  6 ]

Line  98 --> Rotate columns.
    + Noise budget after rotation: 142 bits
    + Decrypt and decode ...... Correct.

    [  7,  0,  0,  0,  0, ...,  0,  0,  4,  5,  6 ]
    [  3,  0,  0,  0,  0, ...,  0,  0,  0,  1,  2 ]

Line 111 --> Rotate rows 4 steps right.
    + Noise budget after rotation: 142 bits
    + Decrypt and decode ...... Correct.

    [  0,  4,  5,  6,  7, ...,  0,  0,  0,  0,  0 ]
    [  0,  0,  1,  2,  3, ...,  0,  0,  0,  0,  0 ]
```

Rotation in CKKS

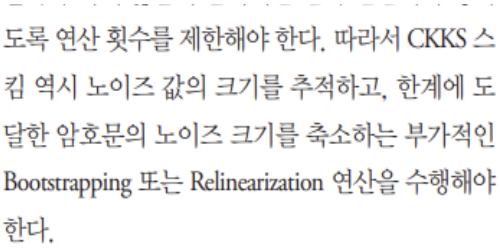
CKKS basic 예제와 기본 세팅은 같다.

```
Ciphertext rotated;
print_line(__LINE__);
cout << "Rotate 2 steps left." << endl;
evaluator.rotate_vector(encrypted, 2, galois_keys, rotated);
cout << "    + Decrypt and decode ...... Correct." << endl;
decryptor.decrypt(rotated, plain);
vector<double> result;
ckks_encoder.decode(plain, result);
print_vector(result, 3, 7);
```

회전연산은 원소들을 원형으로 이동시키는 작업을 의미한다. 위의 경우 왼쪽 방향으로 2 단계 이동시키는 것을 의미한다. SEAL 라이브러리는 암호화된 벡터를 회전할 수 있는 연산을 지원한다. 회전에는 galois_key 가 필요하다. 이후 복호화하면 평문에서 2 단계 왼쪽으로 이동한 모습을 확인할 수 있다.

왜 회전연산에서는 키가 필요할까? 회전연산은 암호화된 데이터에 대해 수행되는 연산이다. 위치를 이동시키고 데이터를 변형한다. 그러므로 암호화 체계의 보안성을 유지하기 위해 필요하다.

이해안되는 점

1. 5 번 예제에서는 Pi * x^3 의 암호문을 구할 때는 계속 재선형화를 하였는데 다른 자릿수를 구할 때는 하지 않았습니다. 왜 그런건가요?
2. 
도록 연산 횟수를 제한해야 한다. 따라서 CKKS 스킴 역시 노이즈 값의 크기를 추적하고, 한계에 도달한 암호문의 노이즈 크기를 축소하는 부가적인 Bootstrapping 또는 Relinearization 연산을 수행해야 한다.

노이즈값이 한계에 다다르면 암호문의 노이즈 크기를 축소하는 Relinearization 을 수행한다. 라고 아래의 논문 7 페이지 3 번에서는 이야기하고 있습니다. 그런데 5 번 예제에서 노이즈의 크기를 따로 확인하는 것 같지 않은데 어떻게 한계로 도달했는지 알 수 있나요?

참고문헌 : 완전동형암호 연산 가속 하드웨어 기술 동향
https://ettrends.etri.re.kr/ettrends/193/0905193001/001-012_%EB%B0%95%EC%84%B1%EC%B2%9C_193%ED%98%B8.pdf