第二章 同态加密

学习要求：掌握同态加密的特点、定义和分类；了解同态加密的发展历史；了解典型方案的构造思想，理解同态加密的应用场景，能够运用不同类型的同态加密解决实际问题；理解自举的概念；掌握理想格的概念以及格上的两类难题；了解 BGN、Gentry 和 CKKS 方案设计的主要思想；掌握基于 Paillier 的隐私信息获取的应用示例，以及基于 SEAL 的 CKKS 的开发案例。

课时：4 课时。

建议授课进度：[2.1~~2.3]、[2.4~2.5]

2.1 基本概念

2.1.1 定义

同态加密(HE，homomorphic encryption)是一种加密算法，它可以通过对密文进行运算得到加密结果，解密后与明文运算的结果一致，如图 2.1.1 所示。

同态加密主要基于公钥密码体制构建，它允许将加密后的密文发给任意的第三方进行计算，并且在计算前不需要解密，可以在不需要密钥方参与的情况下，在密文上直接进行计算。

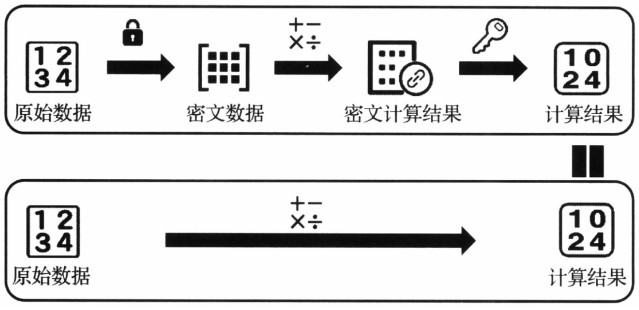


图 2.1.1 同态加密效果图

同态加密方案由 KeyGen、Encrypt、Decrypt 和 Evaluate 4 个函数构成：

- KeyGen(λ)→(pk, sk)：密钥生成函数；在给定加密参数 λ 后，生成公钥/私钥对（pk, sk）。
- Encrypt(pt, pk)→ct：加密函数；使用给定公钥 pk 将目标明文数据 pt 加密为密文 ct。
- Decrypt(sk, ct)→pt：解密函数；使用给定密钥 sk 将目标密文数据 ct 解密为明文 pt。
- Evaluate(pk, Π, ct_1, ct_2, …)→(ct'_1, ct'_2, …)：求值函数；给定公钥 pk 与准备在密文上

进行的运算函数Π，求值函数将一系列的密文输入($ct_1, ct_2, \ldots$)转化为密文输出($ct'_1, ct'_2, \ldots$)。

求值函数是同态加密方案不同于传统加密方案的部分。它的参数Π支持的运算函数种类决定了该同态加密方案支持的同态运算操作。

在给定以上 4 个函数后，同态加密方案应满足正确性、语义安全性和简短性：

[1] 正确性（Correctness）：一个同态加密系统必须要是正确的。具体来说，也就是加密之后的密文可以被成功解密，并且求值函数输出的密文也可以成功解密回原文。

[2] 语义安全（Semantic Security）：同态加密系统输出的密文必须要难以分辨。具体来说，如果有一个网络窃听者看到了所有的密文，那么这个窃听者并不能分辨出哪个密文是对应哪个原文的。

[3] 简短性（Compactness）：同态加密的求值函数输出的密文的长度需要在一个可以控制的长度范围内，确保了同态加密系统的实用性。

2.1.2 分类

根据同态加密算法所支持的同态操作种类和次数，可以将现有同态加密方案分为以下几种类型：

- 半同态加密(partial homomorphic encryption, 简称 PHE)仅支持单一类型的密文域同态运算(加或乘同态)；
- 类同态加密(somewhat homomorphic encryption, 简称 SHE)能够支持密文域有限次数的加法和乘法同态运算；
- 层级同态加密(Leveled Homomorphic Encryption，简称 LHE)能同时支持多种同态操作(加或乘同态)，并可以在安全参数中定义能够执行的操作次数上限。一般允许的操作次数越大，该同态加密方案的密文空间开销及各类操作的时间复杂度就越大；
- 全同态加密(fully homomorphic encryption, 简称 FHE)能够实现任意次密文的加、乘同态运算。

2.1.3 发展历史

类型	算法	时间	说明
半同态加密	RSA 算法	1977 年	非随机化加密，具有乘法同态性的原始算法面临选择明文攻击
	ELGamal 算法	1985 年	随机化加密，乘法同态
	Paillier 算法	1999 年	加法同态，在联邦学习中广泛应用
类同态加密	BGN 方案	2005 年	支持任意次加法和一次乘法操作的同态运算

全同态加密	第一代	Gentry 方案	2009 年	自举操作，性能差
	第二代	BGV 方案	2012 年	基于算术电路，基于模归约提升了自举性能
		BFV 方案	2012 年	基于算术电路，使用 SIMD 操作提升了自举性能
	第三代	GSW 方案	2013 年	支持任意布尔电路，基于近似特征向量
		FHEW 方案	2015 年	支持任意布尔电路，可实现快速比较
		TFHE 方案	2016 年	支持任意布尔电路，基于近似特征向量
	第四代	CKKS 方案	2017 年	可实现浮点数近似计算

（1）半同态加密

乘法同态加密：是指存在有效算法$\otimes$，使得$Enc(x) \otimes Enc(y) = Enc(x \times y)$ 或者$Dec(Enc(x) \otimes Enc(y)) = x \times y$成立，并且不泄漏$x$和$y$。

典型乘法同态加密算法是 RSA 算法和 ElGamal 算法。以 RSA 算法为例，如果$c_1 = m_1^e \ mod\ n$，$c_2 = m_2^e \ mod\ n$，那么$c_1 \times c_2 = m_1^e \ m_2^e \ mod\ n \ = (m_1 m_2)^e \ mod\ n \equiv Enc(m_1 \times m_2)$。

加法同态加密：是指存在有效算法$\oplus$，使得$Enc(x) \oplus Enc(y) = Enc(x + y)$ 或者$Dec\big(Enc(x) \oplus Enc(y)\big) = x + y$成立，并且不泄漏$x$和$y$。

典型加法同态加密算法是 Paillier 算法，详见 2.3.1 节描述。

注意：加法和乘法同态是相对明文而言所执行的操作，而非密文上执行的运算形式。

（2）类同态加密

类同态加密方案能够同时支持加法和乘法的同态操作。但由于它生成的密文随着操作次数的增加而逐渐增大，能够在密文上执行的同态操作次数是有上限的。

典型的类同态加密方案是 Boneh、Goh 和 Nissim 在 2005 年提出 Boneh-Goh-Nissim（简称 BGN）方案，它支持在密文大小不变的情况下进行任意次数的加法和一次乘法。该方案中的加法同态基于类似 Paillier 算法的思想，而一次乘法同态基于双线性映射的运算性质。虽然该方案是双同态的（同时支持加法同态和乘法同态），但只能进行一次乘法操作。

（3）Gentry 方案（第一代全同态加密方案）

在同态加密概念提出后的 30 年间，并没有真正能够支持无限制的各类同态操作的全同态加密方案问世。

2009 年，Gentry 基于所提出的类同态加密方案，提出了“自举”（bootstrapping）技术，可以将满足条件的类同态加密方案改造成全同态加密方案。其基本思想是在类同态加密算法的基础上引入自举方法来控制运算过程中的噪声增长（类同态加密算法操作次数过多会导致噪声过大而无法解密），这也是第一代全同态加密方案的主流模型。

为了避免多次运算使得噪声扩大，Gentry 方案采用了计算一次就消除一次噪声的方法，而消除噪声的方法还是使用的同态运算。但是，由于解密过程本身的运算十分复杂，运算过

程中也会产生大量噪声，因此，Gentry 方案性能极差，一次同态乘法可能需要 30 分钟。

现在的第四代全同态加密解决方案要比 Gentry 提出的方案要好得多，性能大概提高了 100 万倍，并且已经开始制定相关的标准。

（4）BGV 和 BFV 方案（第二代全同态加密方案）

第二代全同态加密方案主要包括 BGV 和 BFV，通常基于 LWE（Learning With Error，容错学习问题）和 RLWE（Ring Learning With Error，环上容错学习问题）假设，其安全性基于格困难问题。

第二代方案主要是解决自举操作带来的昂贵操作，通过引入层级同态加密等来提升性能。此外，第二代全同态加密还提出了单指令多数据 SIMD 方案通过批量处理来提高吞吐量，极大降低了均摊复杂度。SIMD 操作，简单来说，我们可以把密文切出上千个槽，把上千个明文放在这些密文槽中，这样，就可以并行处理各个槽中的数据了。在此基础上，还可以利用同构性置换各个槽中的数据，各个槽中的数据也可以相互运算。

第二代全同态加密方案的性能已经提升了很多，每个明文比特的自举时间约为 0.9 毫秒，自举一个密文能在 10 秒左右完成，具有了一定的实用性。HElib 和 SEAL 两个全同态加密开源库均支持 BGV 和 BFV 方案。

（5）TFHE 等方案（第三代全同态加密方案）

GSW、FHEW 和 TFHE 是第三代同态加密方案重要的代表作。与第二代 FHE 方案相比，自举的性能得到大幅度提升，在常见的台式机平台上速度可以达到毫秒级别；但同时因为缺少第二代 FHE 的 SIMD 特性，FHEW 只能处理若干比特（典型值为 2~7）的加法和乘法操作，也就是说同态乘法的性能较差。

（6）CKKS 等方案（第四代全同态加密方案）

CKKS（Cheon-Kim-Kim-Song）方案支持针对实数或复数的浮点数加法和乘法同态运算，但是得到的计算结果是近似值。因此，它适用于不需要精确结果的场景。

支持浮点数运算这一功能在实际中有非常重要的作用，如实现机器学习模型训练等。这个方案的性能也非常优异，大多数算法库都实现了 CKKS。

注意：有关全同态加密，有两个非常好的网络资源，大家可以关注：

- ✓ https://zhuanlan.zhihu.com/p/539145554，是一个知乎的翻译版的文档；
- ✓ 对应的视频资源为 Gentry 在 EUROCRYPT 2021 做的邀请报告的中文字幕版，网址为 https://www.bilibili.com/video/BV1rY411V7Ko/?spm_id_from=333.337.search-card.all.click，原始视频网址为 https://www.youtube.com/watch?v=487AjvFW1lk。

2.2 半同态 Paillier 方案

Paillier 加密算法是 Paillier 等人 1999 年提出的一种基于判定 n 阶剩余类难题的典型密

码学加密算法，具有加法同态性，是半同态加密方案。

2.2.1 数学基础

1. 卡迈尔克函数

在数论中，卡迈尔克函数的定义为：设$\gcd(a,n)=1$，使得$a^m \equiv 1\ (mod\ n)$成立的最小正整数 m，将 m 记作$\lambda(n)$。对于 $n=pq, p$ 和 q 都是素数，则有$\lambda(n)=lcm(p-1,q-1)$，lcm 为求最小公倍数。

在数论中，对正整数 n，欧拉函数是小于 n 的正整数中与 n 互质的数的数目。显然$\phi(1)=1$，而对于$m>1, \phi(m)$就是 $\{1,\dots,m-1\}$中与 m 互素的数的个数，比如说$p=\{$素数$\}$，则有$\phi(p)=p-1$。对于 $n=pq, p$ 和 q 都是素数，则有$\phi(n)=(p-1)(q-1)$。

下面是一张卡迈克尔函数$\lambda(n)$与欧拉函数$\phi(n)$的对比表：

n	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
$\lambda(n)$	1	1	2	2	4	2	6	**2**	6	4	10	**2**	12	6	**4**	**4**
$\phi(n)$	1	1	2	2	4	2	6	**4**	6	4	10	**4**	12	6	**8**	**8**

（1）示例

对于 8 的卡迈克尔函数是 2，即$\lambda(8)=2$，即对于任意的 a 满足$\gcd(a,8)=1$，有$a^m \equiv 1\ (mod\ 8)$，也就是说$1^2 \equiv 1\ (mod\ 8)$，$3^2 \equiv 1\ (mod\ 7)$，$5^2 \equiv 1\ (mod\ 8)$，$7^2 \equiv 1\ (mod\ 8)$。

而对于欧拉函数来说，$\phi(8)=4$，因为欧拉函数只需要满足与 8 互素的 a，有$a^4 \equiv 1\ (mod\ 8)$，不需要满足 a 是最小的。

对于 n=15，因为 n=3*5，令 p=3，q=5，$\lambda(15)=lcm(2,4)=4$，$\phi(15)=2*4=8$。

（2）卡迈尔克函数的性质

设$n=pq$，其中 p 和 q 是大素数，那么$\phi(n)=(p-1)(q-1)$，$\lambda(n)=lcm(p-1,q-1)$。为便于描述，用λ表示$\lambda(n)$。

我们有，$\left|\mathbb{Z}_{n^2}^*\right|=\phi(n^2)=n\phi(n)$，$g^\lambda=1+kn, k\in Z_n^*$。

对于任意$g\in\mathbb{Z}_{n^2}^*$，有如下性质：

$$\begin{cases} g^\lambda \equiv 1\ mod\ n \\ g^{n\lambda} \equiv 1\ mod\ n^2 \end{cases}$$

具体推导如下：

- 根据λ的定义：$\lambda=k_1(p-1)=k_2(q-1)$
- 根据费马小定理：$g^\lambda=g^{k_1(p-1)}=(g^{p-1})^{k_1}\equiv 1\ mod\ p$，同理$g^\lambda \equiv 1\ mod\ q$
- 所以：$g^\lambda \equiv 1\ mod\ pq=1\ \ mod\ n$
- 所以：$g^\lambda=1+kn, k\in Z_n^*$

- 所以：$g^{\lambda}(mod\ n^2) = 1 + kn \equiv 1\ mod\ n, k \in Z_n^*$
- 结合上式及二项式定理，可得

$$g^{n\lambda}(mod\ n^2) = (1+kn)^n(mod\ n^2) = 1 + kn \cdot n(mod\ n^2) \equiv 1$$

此外，设$1 + n \in \mathbb{Z}_{n^2}^*$，那么

$$(1+n)^2 \equiv 1 + 2n + n^2 \equiv (1+2n)\ mod\ n^2$$

$$(1+n)^3 \equiv 1 + 3n + n^3 \equiv (1+3n)\ mod\ n^2$$

$$(1+n)^v \equiv 1 + v*n + [n\text{的高次幂}] \equiv (1+v*n)\ mod\ n^2$$

2. 判定复合剩余假设

剩余类：亦称同余类，指全体整数按照对一个正整数的同余关系而分成的类。对于一个整数 m，可以把所有整数分成 m 类，每类对于 m 都同余。每一类都叫做 m 的一个剩余类。比如 5，有 5 个剩余类，对 0 同余的有{-5，0，5，...}。

复合剩余类：如果存在一个数$x \in Z_{n^2}^*$，那么符合公式 $z = x^n\ mod\ n^2$ 的数 z，称为 x 模n^2 的 n 阶剩余类。

判定复合剩余假设（decisional composite residuosity assumption , DCRA)：设 $n = pq$，p 与 q 为两个大素数，对于任意给定的整数 z，判断它是否是模n^2 的 n 阶剩余类是一个难解问题。

2.2.2 方案构造

1. 算法描述

（1）密钥生成

✧ 随机选择两个质数 p 和 q，尽可能地保证 p 和 q 的长度接近或相等（安全性高）；

✧ 计算 $n = pq$和 $\lambda = lcm(p-1, q-1)$，其中 lcm 表示最小公倍数；

✧ 随机选择 $g \in Z_{n^2}^*$，考虑计算性能优化，通常会选择$g = n + 1$；

✧ 计算$\mu = \left(L\left(g^{\lambda}\ mod\ n^2\right)\right)^{-1}\ mod\ n$，其中$L(x) = \frac{x-1}{n}$；

✧ 公钥为(n, g)；

✧ 私钥为(λ, μ)。

（2）加密算法

对于任意明文消息 $m \in Z_n$，任意选择一个随机数 $r \in Z_n^*$，计算得到密文 c：

$$c = E(m) = g^m r^n\ mod\ n^2$$

注意：密文 c 要比明文 m 长度要长。

（3）**解密算法**

对于密文$c \in Z_{n^2}^*$，计算得到明文 m：

$$m = D(c) = L\left(c^{\lambda} \bmod n^2\right) * \mu \bmod n$$

2. 正确性

依据卡迈尔克函数的性质，对于任意$g \in Z_{n^2}^*$，$n = pq$和 $\lambda = lcm(p-1, q-1)$，有

$$\begin{cases} g^{\lambda} \equiv 1 \bmod n \\ g^{n\lambda} \equiv 1 \bmod n^2 \end{cases}$$

如 2.2.1 所述，$g^{\lambda} = 1 + kn, k \in Z_n^*$，

基于上述三个性质，解密过程推导如下：

$$\begin{aligned} & D(c) = L\left(c^{\lambda} \bmod n^2\right) * \mu \bmod n \\ & = L\left((g^m r^n)^{\lambda} \bmod n^2\right) * \mu \bmod n \\ & = L\left((g^{\lambda})^m \bmod n^2\right) * \left(L\left(g^{\lambda} \bmod n^2\right)\right)^{-1} \bmod n \\ & = L((1+kn)^m \bmod n^2) * \left(L(1+kn)\right)^{-1} \bmod n \\ & = mk * k^{-1} \bmod n \\ & = m \end{aligned}$$

3. 加法同态性

对于任意明文$m_1, m_2 \in Z_n$和任意$r_1, r_2 \in Z_n^*$，对应密文$c_1 = E(m_1), c_2 = E(m_2)$，满足：

$$c_1 \cdot c_2 = g^{m_1} \cdot {r_1}^n \cdot g^{m_2} \cdot {r_2}^n \bmod n^2 = g^{m_1+m_2} \cdot (r_1 \cdot r_2)^n \bmod n^2$$

解密后得到：

$$D(c_1 \cdot c_2) = D(g^{m_1+m_2} \cdot (r_1 \cdot r_2)^n \bmod n^2) = m_1 + m_2$$

即我们得到了：$c_1 * c_2 = m_1 + m_2$，也就是，**密文乘等于明文加。**

注意：加法和乘法同态是相对明文而言所执行的操作，而非密文上执行的运算形式。

4. 标量乘同态性

标量乘法可以看作多次加法。

对于明文$m_1 \in Z_n$及其密文c_1，给定一个整数$a \in Z_n$，满足：

$$D({c_1}^a \bmod n^2) = D(g^{m_1 a} \cdot r^n \bmod n^2) = m_1 a$$

即 a 个密文相加等于 a 个明文相加。

2.2.3 应用示例

1. 典型应用

半同态加密虽然还不能同时支持加法和乘法运算，不能支持任意的计算，但是因为其与全同态相比，具有较高性能，因此，仍然具有极为广泛的应用场景，且在现实应用中起到了中重要的作用。一类典型的应用体现在隐私保护的数据聚合上。由于加法同态加密可以在密文上直接执行加和操作，不泄露明文，在到多方协作的统计场景中，可完成安全的统计求和的功能。

（1）联邦学习

在联邦学习中，不同参与方训练出的模型参数可由一个第三方进行统一聚合。使用加法 PHE，可以在明文数据不出域、且不泄露参数的情况下，完成对模型参数的更新，此方法已应用在实际应用（如 FATE），如图 2.2.1 所示。

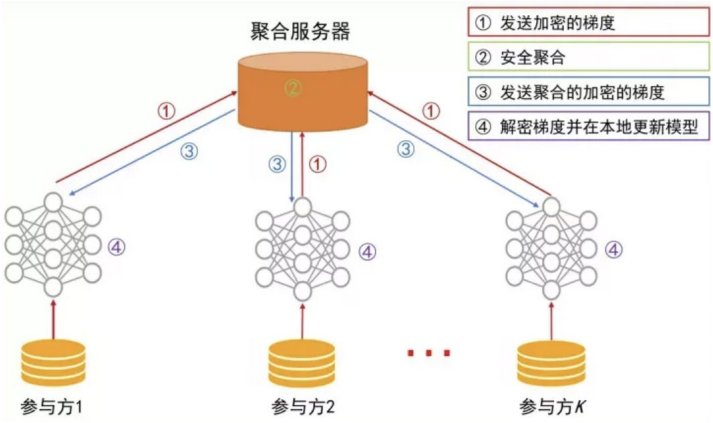


图 2.2.1 联邦学习中的安全聚合示例

（2）隐私集合求和

在在线广告投放的场景中，广告主（如商家）在广告平台（如媒体）投放在线广告，并希望计算广告点击的转化收益。然而，广告点击数据集和购买数据集分散在广告主和广告平台两方。使用加法 PHE 结合隐私集合求和（Private Intersection-Sum-with-Cardinality, PIS-C）协议可以在保护双方隐私数据前提下，计算出广告的转化率。如图 2.2.2 所示，协议中的“隐私保护求和”功能依赖于广告主将自己的交易数据用 PHE 加密发送给广告平台，使得广告平台在看不到原始数据的前提下，完成对交集中数据金额的聚合。该方案已被 Google 落地应用。

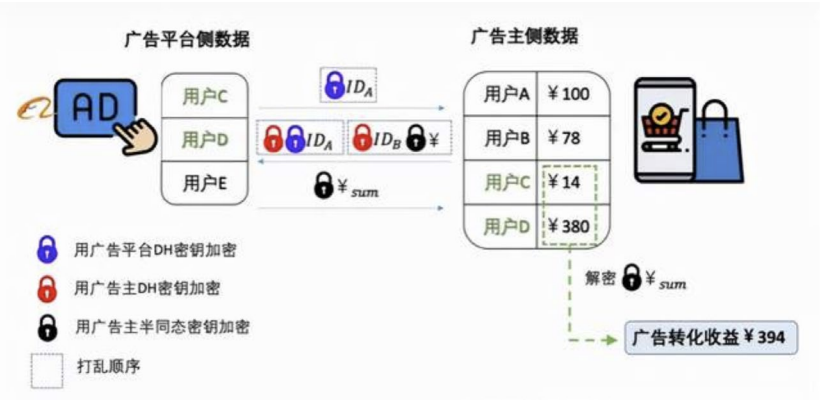


图 2.2.2 加法 PHE 在 PIS-C 中的应用

（3）数据库统计查询

在加密数据库 SQL 查询场景，在数据库不可信的情况下，可以通过部署协议和代理来保护请求者的查询隐私。其中，PHE 可以用来完成安全数据求和、均值的查询。

除了上述场景，加法 PHE 还可以用于多种“行为数据和效益数据分离”的商业场景，在应用上有着很大的想象空间。

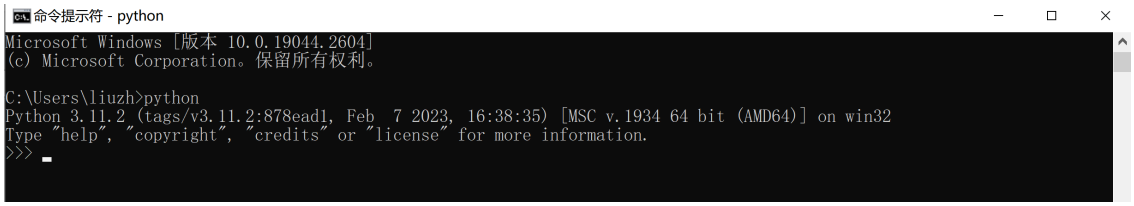
2. 实验环境安装

（1）安装 python 环境

在 Windows 下安装 python 开发环境。到官方网站 https://www.python.org/downloads/下载 windows 版本的 python 安装包。下载后双击安装即可。

提示：安装过程一定要勾选“Add python.exe to PATH”，这样会使得安装后的 python 程序路径直接加入到系统的环境变量中，在控制台可以直接使用 python 命令。如果忘记勾选，则需要通过“我的电脑”→右键“属性”→“高级系统设置”→“环境变量”的 path 中将安装的路径手动填入。

安装完毕，打开控制台，输入 python 命令，会显示：



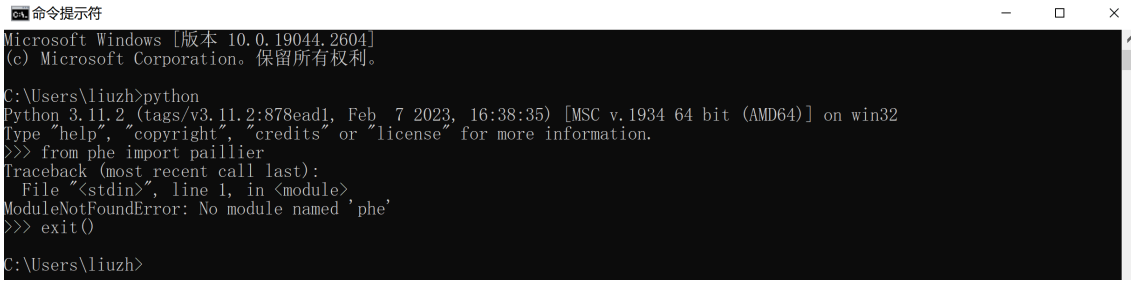
代表已经安装成功，并且进入 python 运行环境。

输入 python 程序：from phe import paillier

该命令将导入 phe 库的 paillier 功能，第一次执行会提示“ModuleNotFoundError: No module named 'phe'”。因为，默认安装 python 后，并没有安装 phe 这个库。

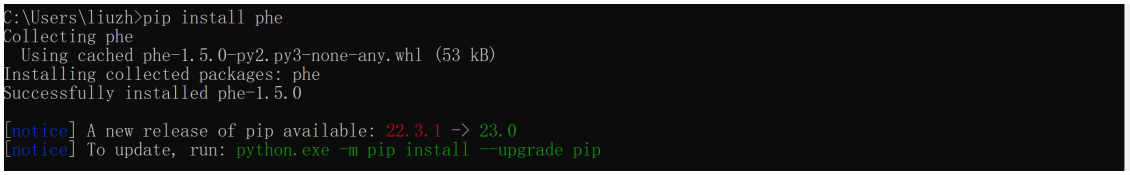
输入 python 命令：exit()

该命令可以退出当前 python 环境，切回控制台模式，如下所示：



（2）安装 phe 库

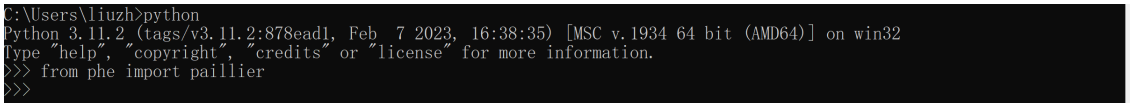
输入命令：pip install phe 完成 phe 库的安装。



Pip 是 python 的一个安装库的工具，可执行文件在 python 安装目录下可以找到。

（3）验证环境正确性

再次进入 python 环境，输入 python 代码：from phe import paillier



发现不出现错误信息，说明环境安装成功。

（4）编写 python 程序并运行

可以有三种方式调试和编写 python 程序：

- ✧ 在控制台运行 python 命令，逐行编写 python 程序并运行；
- ✧ 用文本编辑器编写完整的程序并保存为 x.py，通过控制台命令 python x.py 的方式完成整个程序的调用；
- ✧ 通过 IDLE 这个 python 的集成开发环境完成开发和调试运行。通过开始菜单，找到 IDLE 并打开，选择 File→new file 可以新建一个文件，编辑程序并保存后，选择 Run→Run Module 运行，会看到运行的结果。

3. 简单示例

Python phe 的库详见：https://python-paillier.readthedocs.io/en/develop/usage.html#usage。

实验 2.1 基于 Python 的 phe 库完成加法和标量乘法的验证。

这里给出一个集成的演示代码如下。

```
from phe import paillier # 开源库
import time # 做性能测试
```

```
###################### 设置参数
print("默认私钥大小：", paillier.DEFAULT_KEYSIZE)
#生成公私钥
public_key, private_key = paillier.generate_paillier_keypair()
# 测试需要加密的数据
message_list = [3.1415926,100,-4.6e-12]

###################### 加密操作
time_start_enc = time.time()
encrypted_message_list = [public_key.encrypt(m) for m in message_list]
time_end_enc = time.time()
print("加密耗时 s：",time_end_enc-time_start_enc)
print("加密数据（3.1415926）:",encrypted_message_list[0].ciphertext())

###################### 解密操作
time_start_dec = time.time()
decrypted_message_list = [private_key.decrypt(c) for c in encrypted_message_list]
time_end_dec = time.time()
print("解密耗时 s：",time_end_dec-time_start_dec)
print("原始数据（3.1415926）:",decrypted_message_list[0])

###################### 测试加法和乘法同态
a,b,c = encrypted_message_list # a,b,c 分别为对应密文
a_sum = a + 5 # 密文加明文，已经重载了+运算符
a_sub = a - 3 # 密文加明文的相反数，已经重载了-运算符
b_mul = b * 6 # 密文乘明文,数乘
c_div = c / -10.0 # 密文乘明文的倒数

print("a+5 密文:",a.ciphertext()) # 密文纯文本形式
print("a+5=",private_key.decrypt(a_sum))
print("a-3",private_key.decrypt(a_sub))
print("b*6=",private_key.decrypt(b_mul))
print("c/-10.0=",private_key.decrypt(c_div))

##密文加密文
```

```
print((private_key.decrypt(a)+private_key.decrypt(b))==private_key.decrypt(a+b))
#报错，不支持 a*b，即两个密文直接相乘
#print((private_key.decrypt(a)+private_key.decrypt(b))==private_key.decrypt(a*b))
```

如上述代码所示：第一，python 程序对运算符进行了承载，已经支持直接密文上的运算。第二，只支持明文的加法，不支持明文的乘法，最后一句如果将注释符去掉，将报错。

4. 隐私信息获取示例

实验 2.2 基于 Python 的 phe 库完成隐私信息获取的功能：服务器端拥有多个数值，要求客户端能基于 Paillier 实现从服务器读取一个指定的数值并正确解密，但服务器不知道所读取的是哪一个。

首先，我们要基于 Paillier 协议进行设计。

对 Paillier 的标量乘的性质进行扩展，我们知道：数值“0”的密文与任意数值的标量乘也是 0，数值“1”的密文与任意数值的标量乘将是数值本身。

基于这个特性，我们可以如下巧妙的设计：

服务器端：产生数据列表 data_list={m_1, m_2, …, m_n}

客户端：

- 设置要选择的数据位置为 pos
- 生成选择向量 select_list={0,…,1,.., 0}，其中，仅有 pos 的位置为 1
- 生成密文向量 enc_list={E(0),…, E(1),.., E(0)}
- 发送密文向量 enc_list 给服务器

服务器端：

- 将数据与对应的向量相乘后累加得到密文 c= m_1*enc_list[1]+…+ m_n*enc_list[n]
- 返回密文 c 给客户端

客户端：解密密文 c 得到想要的结果

进而，开发具体代码如下：

```
from phe import paillier #  开源库
import random #  选择随机数


###################### 设置参数
# 服务器端保存的数值
message_list = [100,200,300,400,500,600,700,800,900,1000]
length = len(message_list)
# 客户端生成公私钥
```

```
public_key, private_key = paillier.generate_paillier_keypair()
# 客户端随机选择一个要读的位置
pos = random.randint(0,length-1)
print("要读起的数值位置为：",pos)

###################### 客户端生成密文选择向量
select_list=[]
enc_list=[]
for i in range(length):
    select_list.append(   i == pos )
    enc_list.append( public_key.encrypt(select_list[i]) )

# for element in select_list:
#     print(element)
# for element in enc_list:
#     print(private_key.decrypt(element))

###################### 服务器端进行运算
c=0
for i in range(length):
    c = c + message_list[i] * enc_list[i]
print("产生密文：",c.ciphertext())

###################### 客户端进行解密
m=private_key.decrypt(c)
print("得到数值：",m)
```

2.3 类同态 BGN 方案

第一个同时支持同态加法和同态乘法的类同态加密方案由 Fellows 和 Koblitz 在 1994 年提出，其密文大小随着同态操作的次数呈指数级增长，且同态乘法的计算开销很大，无法投人实际使用。2005 年，Boneh-Goh-Nissim（BGN）加密方案提出，它支持在密文大小不变的情况下进行任意次数的加法和一次乘法。

2.3.1 数学基础

1. 群

群是一种由元素的集合和一个二元运算组成的基本代数结构。若元素集合 G 和二元运算“·”满足封闭性、结合律、单位元和逆元素四个要素，则称之为群。

• 封闭性：对于所有集合 G 中的元素 a 和 b，$a·b$ 的结果也在集合 G 中；

• 结合律：（$a·b$）$·c = a·$（$b·c$）对任意 $a,b,c \in G$ 成立；

• 单位元：集合 G 存在元素 e，满足 $e·a = a·e = a$，$\forall a \in G$；

• 逆元素：对于集合 G 中的任意一个元素 a，存在集合 G 中的另一个元素 b 使 $a·b = b·a = e$。

若一个群还满足交换律（即 $a·b = b·a$，任意 $a,b \in G$ 成立），则可进一步称其为交换群或阿贝尔群。

定义一个有限群的阶为群中元素的个数。

对于二元运算“·”，定义元素的乘方 a^2 为 $a·a$，并以此推演出元素的更高次方。

循环群。若一个群 G 的每一个元素都可以被表达成群 G 中某一个元素 g 的次方 g^m，则称 G 为循环群，记作 $G = (g) = \{g^m | m \in Z\}$，$g$ 被称为 G 的一个生成元，因为可以通过对 g 的不断自我运算来获得群中的所有元素。

注意：符号<g>与符号 (g) 相同，也经常被定义为由 g 生成的循环群。

在一个有限群中，如果对不是生成元的其他元素 a 进行这种次方运算，它最终会循环遍历一个群 G 的子集。可以证明，所有元素的这种遍历都会经过单位元 e。将满足 $a^n = e$ 的最小正整数 n 称为 a 元素的阶，生成元的阶和群的阶相等。

以整数模 6 加法群 Z_6={0，1，2，3，4，5}为例，其群的阶为 6，单位元为 0。6 个元素的阶分别是 1、6、3、2、3、6。其中 1、5 为生成元。

以元素 5 为例，经过模加运算有:

$5^1 = 5$,

$5^2 = (5+5) \pmod 6 = 4$,

$5^3 = (5+5+5) \pmod 6 = 3$,

$5^4 = (5+5+5+5) \pmod 6 = 2$,

$5^5 = (5+5+5+5+5) \pmod 6 = 1$,

$5^6 = (5+5+5+5+5+5) \pmod 6 = 0 = e$.

故称元素 5 的阶为 6，为群 G 的生成元。

很容易可以发现循环群的一个特性，即“由生成元g构建的循环群很容易满足加法同态”：$g^{r_1} \cdot g^{r_2} = g^{(r_1+r_2)}$。根据上述例子，也很容易就可以验证：对于明文 1 和 2，对应的

密文是 5^5 和 5^4，因为这里的运算符“·”就是加法，显然，$5^5 \cdot 5^4 = (5*9 = 5*6+5*3) \pmod 6 = 5^3 = 5^{(1+2)} = 3$。

2. 环

在群的基础上，还可以使用两种运算和元素集合 R 来构建环，这两种运算一般写作“+”和“·”。通常可以这么理解这两个运算：

- ✧ “+”一般表示环上的加法，其对应的单位元通常为 0，由其定义的群为加法群，群上的两个相同运算的+运算，可以记作 $a+a=2a$；
- ✧ “·”一般表示环上的乘法，其对应的单位元通常记作 e，由其定义的群为乘法群，群上的两个相同运算的·运算，可以记作 $a \cdot a = a^2$。

环可以认为是在加法交换群之上增加了乘法运算“·”，且满足如下性质：

- 封闭性：对于所有 R 中元素 a 和 b，$a \cdot b$ 的结果也在 R 中；
- 结合律：$(a \cdot b) \cdot c = a \cdot (b \cdot c)$ 对任意 $a,b,c \in R$ 成立；
- 单位元：R 存在元素 e，满足 $e \cdot a = a \cdot e = e$，$\forall a \in R$，该元素常被称作乘法单位元；
- 分配律：乘法操作可以在加法之间进行分配。即给出任意 $a,b,c \in R$，有 $a \cdot (b+c) = a \cdot b + a \cdot c$ 和 $(b+c) \cdot a = b \cdot a + c \cdot a$。

事实上，满足上述封闭性和结合律的 $(R, \cdot)$ 构成一个半群。再加上单位元，就构成一个幺半群。如果再有逆元素，就构成了一个群。

$(R, +, \cdot)$ 构成一个环（Ring），当：

- $(R, +)$ 构成交换群；
- $(R, \cdot)$ 构成幺半群；
- $(R, +, \cdot)$ 满足分配律。

在一个环 $(R, +, \cdot)$ 中，若其子集 I 与其加法构成子群 $(I, +)$，且满足 $\forall i \in I, r \in R$，$i \cdot r \in I$，则称 I 为环 R 的一个右理想（Right Ideal）。若 $\forall i \in I, r \in R$，$r \cdot i \in I$ 则称 I 为环 R 的一个左理想（Left Ideal）。若同时满足左右理想，则称 I 为环 R 上的一个理想（Ideal）。

理想对内具有乘法封闭性，对外具有乘法吸收性。

3. 双线性映射

一个双线性映射是由两个向量空间上的元素，生成第三个向量空间上一个元素之函数，并且该函数对每个参数都是线性的。理解：若 $B: V \times W \to X$ 是一个双线性映射，则 V 固定，W 可变时，W 到 X 的映射是线性的；W 固定，V 可变时，V 到 X 的映射也是线性的；也就是说，保持双线性映射中的任意一个参数固定，另一个参数对 X 的映射都是线性的。

存在一个加法循环群G_1和乘法循环群G_2，这两个群的阶都为大素数q。定义$e: G_1 \times G_1 \to$

G_2为这两个循环点群之间的一个双线性映射，且该映射满足如下三个性质：

（1）双线性：对于所有的$P, Q \in G_1$和$a, b \in Z_q^*$，有

$$e(aP, bQ) = e(P, Q)^{ab}$$

$$e((a+b)P, Q) = e(P, Q)^b \cdot e(P, Q)^b$$

注，这里的Z_q^*表示不包含 0 的整数集，Z表示整数集，q表示阶，$*$表示不包含 0 元素。

（2）非退化性：e为非平凡映射，即e不会将$G_1 \times G_1$的所有值映射到G_2的单位元。

（3）可计算性：具有有效的算法对于任何的$P, Q \in G_1$能够计算$e(P, Q)$。

满足如上三个性质的双线性映射就叫做可采纳的双线性映射。

Boneh 等给出了关于双线性映射更具体的描述，提出了与双线性相关的数学难题，并用于设计基于身份加密、基于属性的加密等密码原语。

4. 子群判定问题

子群判定问题是指：给定(n, G, G_1, e)，其中群G, G_1具有相同的阶$n = q_1 q_2$，$e: G \times G \to G_1$是一个双线性映射，给定一个元素$x \in G$，如果x的阶是q_1，则输出 1，否则输出 0。

上述问题也可以描述为：一个阶为$n = p * q$（p, q为质数）的合数阶群里，判定一个元素是否属于某个阶为p的子群的问题。

该判定问题为困难问题，BGN 方案的实现就是基于子群决策问题。

2.3.2 方案构造

BGN 能够同时支持加法和乘法的关键原因在于，它提出了一套能够构建在两个群G和G_1之间的双线性映射$e: G \times G \to G_1$的方法。BGN 提出的方法能够生成两个阶相等的乘法循环群G和G_1，并建立其双线性映射关系e，且满足当g是G的生成元时，$e(g, g)$为G_1的生成元。

在执行乘法之前，密文属于群G中的元素，可以利用群的二元操作进行密文的加法同态操作。密文的乘法同态操作通过该双线性映射函数，将密文从群G映射到G_1的元素当中。执行乘法同态操作之后，处于G_1的密文仍然能够继续使用同态加法。

（1）密钥生成

给出安全参数，选择大质数q_1, q_2并获得合数$n = q_1 q_2$，BGN 将构建两个阶为n的循环群G, G_1和双线性映射关系$e: G \times G \to G_1$。

从G中随机选取两个生成元g, u，并获得$h = u^{q_2}$。可知h为某阶为q_1的G的子群的生成元。

公钥设置为(n, G, G_1, e, g, h)，私钥设置为q_1。

（2）加密

对于消息明文m（某小于q_2的自然数），随机抽取 0 到n之间的一个整数r，生成密文：

$$c = E(m) = g^m h^r \in G$$

（3）解密

使用私钥q_1，首先计算：

$$c^{q_1} = (g^m h^r)^{q_1} = (g^m u^{q_2 r})^{q_1} = g^{mq_1} u^{q_2 q_1 r} = g^{mq_1} u^{nr} = (g^{q_1})^m,$$

然后，计算离散对数得到明文：$m = \log_{g^{q_1}}(c^{q_1})$。

2.3.3 同态性

（1）密文上的同态加法性质

由生成元g构建的循环群很容易构造加法同态，$g^{r_1} \cdot g^{r_2} = g^{(r_1+r_2)}$。

对于两个密文$c_1 = g^{m_1} \cdot h^{r_1}$和$c_2 = g^{m_2} \cdot h^{r_2}$，很明显$c_1 \cdot c_2 = g^{(m_1+m_2)} \cdot h^{(r_1+r_2)}$。

（2）密文上的同态乘法

密文上的同态乘法，则通过双线性映射函数实现，$e(u^a, v^b) = e(u, v)^{ab}$。

在密钥生成的时候，定义$g_1 = e(g, g)$和$h_1 = e(g, h)$，且将h写作$h = g^{\alpha q_2}$（因为g可生成u: $u = g^{\alpha}$），定义对c_1和c_2的同态乘法运算为：

$$e(c_1, c_2) h_1^r = e(g^{m_1} h^{r_1}, g^{m_2} h^{r_2}) h_1^r = g_1^{m_1 m_2} h_1^{\hat{r}} \in G_1。$$

式中，$\hat{r}$是前文提到的随机抽取的 0 到n之间的整数r。

由此可见，经过同态乘法之后的密文从G转移到了G_1，其解密过程在G_1上完成，即G_1的生成元$g_1 = e(g, g)$替代 g。在群G_1上依然可以进行乘法同态操作，所以 BGN 支持同态乘法运算之后的同态加法运算。

但是，因为没有下一个群可以继续映射，BGN 加密的密文只能够支持一次同态乘法运算。

2.4 全同态典型方案

2.4.1 数学基础

（1）格的定义

给定一个n维向量空间R^n，格（Lattice）是其上的一个离散加法子群。

根据线性代数知识，可以构造一组n个线性无关的向量$\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \cdots, \boldsymbol{v}_n \in R^n$。基于该组向量的整数倍的线性组合，可以生成一系列的离散点：

$$L(\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \cdots, \boldsymbol{v}_n) = \left\{ \sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i | \alpha_i \in \mathbb{Z} \right\}$$

这些元素集合和对应的加法操作$(L, +)$称为格（Lattice）。这组线性无关向量$\boldsymbol{B}$（即$\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3, \cdots, \boldsymbol{v}_n$）称为格的基，其向量个数被称为格的维度。

（2）格的示例

格基$(1,0)^T$与$(0,1)^T$可以产生二维空间的所有整数格，如图 2.4.1（左）。同时，我们发现，使用格基$(1,0)^T$与$(1,1)^T$同样可以生成二维空间的所有整数格，如图 2.4.1（右）。也就是说，一个格的基向量可以有多个，图 2.4.1（左）的基向量正交程度好一些，称之为“好基”，而图 2.4.1（右）的基向量正交程度坏一些，称之为“坏基”。

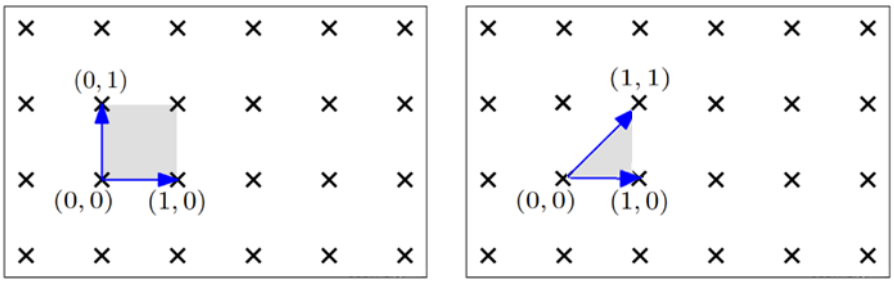


图 2.4.1 二维空间的格示例

再如，格基$(1,1)^T$与$(2,0)^T$不能产生二维空间的所有整数格。如图 2.4.2，打“×”的为可产生的格，横纵坐标相加为偶数。

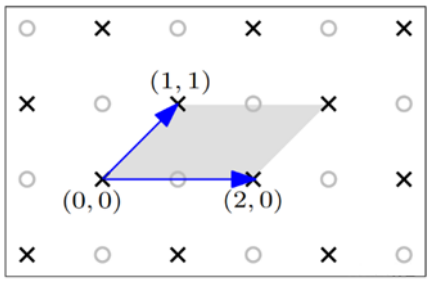


图 2.4.2 二维空间的格示例

上面都是简单的二维例子，一个格可以由无限多的维度和无限多的向量组成，所以虽然二维看起来非常简单，但是随着基向量和维度的数量增加时，问题很快会变的非常复杂。一般来说，达到足够安全性的方案，格的维度在 1000 左右。

（3）格上的难题

尽管格也由基扩展获得，它和向量空间最大的不同在于，它的系数限制为整数，从而生成一系列离散的空间向量。格上的向量的离散性质催生了一系列新的难题。

格上的主要难题是最近向量问题（Closest Vector Problem，简称 CVP)和最短向量问题（Shortest Vector Problem，简称 SVP）。

定义（最近向量问题）：给定一个格L和一个在向量空间R^n中但不在格L中的向量$\boldsymbol{w} \in R^n$，试图找到一个离$\boldsymbol{w}$最近的向量$\boldsymbol{v} \in L$，即与$\boldsymbol{w}$的欧氏距离最小的向量。

欧式距离也称欧几里得距离，以古希腊数学家欧几里得命名的距离，是最常见的距离度量，衡量的是多维空间中两个点之间的绝对距离。举例而言，在二维和三维空间中的欧式距离的就是两点之间的距离，二维的公式$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$是；三维的公式是 $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$。

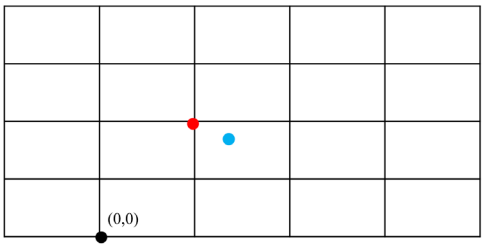


图 2.4.3 二维空间的格中的最近向量示例

如图 2.4.3 所示，给定蓝色点表示的非格上的向量，很容易找到红色点表示的格上的向量，距离蓝色点最近。但是，当基向量和维度增加时，寻找最近向量将变的很困难。

定义（最短向量问题）：对于给定的格L，找到一个非零的格向量$\boldsymbol{v}$，使得对于任意的非零向量$\boldsymbol{u} \in L$，有$\| \boldsymbol{v} \| \leq \| \boldsymbol{u} \|$。

换种说法，该问题试图找到格L上一个最短的向量$\boldsymbol{v} \in L$，其与零点的欧氏距离最小。通常使用符号$\| \boldsymbol{v} \|$来表示向量$\boldsymbol{v}$与零点的欧式距离。

同最近向量问题一样，在基向量和维度够大的情况下，寻找最短向量是一个难题。解决这些问题的难度在很大程度上取决于其对应的格的基的性质。若格的基尽可能相互正交，则存在多项式时间内解决 SVP 和 CVP 的方法。若格的基的正交程度很差，则目前解决 SVP 和 CVP 的最快算法也需要指数级的计算时间。

因此，通过将正交程度差的基作为公钥，将正交程度好的基作为私钥，将解密系统设计为解决格上的最短向量问题或者最近向量问题，就能够提供一种基于格的加密方案。以最近向量问题举例，可以将数据编码到一个安全的格点（比如图 2.4.3 的红点），加密算法会生成一个离红点近的密文（比如图 2.4.3 的蓝点），解密的时候拥有私钥的一方可以在多项式时间内找到密文（蓝点）对应的最近向量（红点），但是拥有公钥和其它公开参数的一方在多项式时间内是无法完成求解的。

基于格的密码系统具有两个优点：

- ✧ 使用线性代数操作实现加解密，具有易实现、高效率的特点。
- ✧ 安全性高。为达到k个比特的安全等级，传统基于大数分解或离散对数问题的加密系统的加解密操作需要$\mathcal{O}(k^3)$的时间复杂度，而基于格的加密系统仅需要$\mathcal{O}(k^2)$的时间复杂度。随着量子计算机的问世，大数因式分解之类的经典难题可以在多项式时间被解决，但是量子计算机尚无法在多项式时间内解决格所对应的难题。

（4）理想格

循环格。循环格是一种特殊的格，循环格最显著的优点就是能够用一个向量来表示，可以采用相关算法来加速运算，可以进一步解决基于一般格上的密码方案中密钥量大、运行效率较低的问题。对于一系列向量$\boldsymbol{v}_0 = (v_1, v_2, v_3, \cdots, v_n)^T, \boldsymbol{v}_1 = (v_n, v_1, v_2, \cdots, v_{n-1})^T, \boldsymbol{v}_2 = (v_{n-1}, v_n, v_1, \cdots, v_{n-2})^T$,…，以循环生成的$n$个向量为基生成的格被称为循环格。

理想格。理想格是对循环格概念的推广，一般格是群的子群，理想格指该格同时也是环上的理想。在多项式环$\mathbb{Z}[x]/(f(x))$上，循环格的基是通过给出一个多项式$v \in L$，然后对其连续模乘x得到$\{\boldsymbol{v}_i = \boldsymbol{v}_0 * x^i (mod\ f(x)) | i \in [0, n-1]\}$。可以证明，在多项式环上构建的循环

格即为环的理想，也称为理想格。

理想格有两个优点：

- ✧ 可以降低格表示的空间尺寸。格的表示方式需要比较大的空间，比如用一个$n \times n$矩阵来表示一组基，则需要存储n^2个元素。而理想格的表示则非常简单，对于基而言，给出一个多项式即可。
- ✧ 理想格具有理想的特性，即对内具有乘法封闭性，对外有乘法吸收性，这一特点使得理想格很容易构造全同态加密方案。

2.3.1 Gentry 方案

尽管有很多关于部分同态加密和类同态加密的方案、然而在同态加密概念提出后的 30 年间，并没有真正能够支持无限制的各类同态操作的全同态加密方案问世。直到 2009 年，斯坦福大学的博士生 Gentry 在他的论文中提出了第一个切实可行的全同态加密方案。

1. 自举操作

类同态加密方案可以支持有限次数的各类同态操作，不满足强同态。如果想要不断地进行同态操作，一种简单直接的方法是将该密文解密并且再次加密，从而能够获得一个“全新”的密文，这个过程简称为“刷新”。刷新之后的密文相当于被重置回刚刚加密的状态，从而继续支持更多的同态操作。但是这样的话，需要使用密钥对密文进行解密，这违背了同态加密在加密状态下进行持续运算的原则。

Gentry 敏锐地察觉到，如果能够设计一种加密方案，它的解密操作本身能够做成同态操作，就能够在全程不解密的情况下完成“刷新”操作，这个操作称为“自举”。

自举过程可简述如下：

- 对于给定的同态加密方案ε。在生成公私钥对(sk, pk)之后，用公钥pk加密私钥sk得到$\overline{sk}$。
- 在对密文ct进行同态加密之前，应用公钥pk再次加密得到两次加密的密文$\overline{ct}$。设解密操作为D_ε，其中ε支持同态解密，使用密文$\overline{ct}$和密钥$\overline{sk}$进行同态解密$Evaluate(pk, D_\varepsilon, \overline{ct}, \overline{sk})$。此时，更早被加密的密文已经被解密，生成密文为此轮刚加密的“全新”密文。
- 在新密文上执行一系列同态操作。

通过“自举”技术可以将原来仅支持有限次同态操作的近似同态加密方案改造成支持无限次同态操作的全同态方案。基于这一思想，Gentry 提出了基于理想格的全同态加密方案。

2. 近似同态加密方案

Gentry 的基于理想格的近似同态加密方案的具体实现如下。

（1）密钥生成

给定一个多项式环$R = \mathbb{Z}[x]/(f(x))$，R 上的一个理想 I 及其固定基B_I，通过循环格生成理想格J，满足$I + J = R$，生成两组J的基(B_J^{sk}, B_J^{pk})作为公私钥对。其中，私钥B_J^{sk}正交化程度较高，公钥B_J^{pk}正交化程度较低。另外，提供一个随机函数$Samp(B_I, x)$用于从$x + B_I$的陪集中抽样。最终，$(R, B_I, B_J^{pk}, Samp())$为公钥，$B_J^{sk}$为私钥。

注：$I + J = R$表示 I 和 J 上的元素在运算+的情况，结果在 R 上。

（2）加密

通过函数$Samp(B_I, x)$随机选择向量$\boldsymbol{r}, \boldsymbol{g}$，使用$B_J^{pk}$对明文$\boldsymbol{m} \in \{0,1\}^n$进行加密，有

$$\boldsymbol{c} = Enc(\boldsymbol{m}) = \boldsymbol{m} + \boldsymbol{r} \cdot B_I + \boldsymbol{g} \cdot B_J^{pk}$$

说明：$\boldsymbol{g} \cdot B_J^{pk}$是理想格$J$上的一个元素。

（3）解密

通过私钥B_J^{sk}解密密文$\boldsymbol{c}$，得到

$$\boldsymbol{m} = \boldsymbol{c} - B_J^{sk} \cdot \left\lfloor \left(B_J^{sk}\right)^{-1} \cdot c \right\rceil (mod\ B_I)$$

式中，⌊　⌉表示对向量各维度坐标进行四舍五入取整。

2. 正确性

在加密阶段，密文$\boldsymbol{c}$可以看作一个格J中的元素$\boldsymbol{g} \cdot B_J^{pk}$加上噪声 $\boldsymbol{m} + \boldsymbol{r} \cdot B_I$。

在解密阶段，$B_J^{sk} \cdot \left\lfloor \left(B_J^{sk}\right)^{-1} \cdot c \right\rceil$为应用取整估计法求解最近向量问题（CVP），即找到密文向量在格J中最近的向量，即$B_J^{sk} \cdot \left\lfloor \left(B_J^{sk}\right)^{-1} \cdot c \right\rceil = \boldsymbol{g} \cdot B_J^{pk}$，因此，我们有

$$\boldsymbol{m} = \boldsymbol{c} - B_J^{sk} \cdot \left\lfloor \left(B_J^{sk}\right)^{-1} \cdot c \right\rceil (mod\ B_I) = \boldsymbol{c} - \boldsymbol{g} \cdot B_J^{pk} (mod\ B_I) = \boldsymbol{m} + \boldsymbol{r} \cdot B_I\ (mod\ B_I)$$

注意：应用取整估计法解 CVP 问题要求$\boldsymbol{m} + \boldsymbol{r} \cdot B_I$足够小，才能保证其加密的格元素$\boldsymbol{g} \cdot B_J^{pk}$和解密时找到的最近的格元素是相同元素，即$B_J^{sk} \cdot \left\lfloor \left(B_J^{sk}\right)^{-1} \cdot c \right\rceil = \boldsymbol{g} \cdot B_J^{pk}$。

3. 安全性

如上述加解密过程，安全性规约到最近向量问题（CVP）的求解上，只有在格的基正交程度较高的私钥上可以获得尽可能接近的格向量，正交程度较低的基B_J^{pk}无法解密明文。

3. 同态性

在该方案中，明文和密文之间的线性关系使得同态操作易于实现，直接密文相加即可

实现同态加法：

$$\boldsymbol{c}_1 + \boldsymbol{c}_2 = \boldsymbol{m}_1 + \boldsymbol{m}_2 + (\boldsymbol{r}_1 + \boldsymbol{r}_2) \cdot B_I + (\boldsymbol{g}_1 + \boldsymbol{g}_2) \cdot B_J^{pk}$$

结果仍在密文空间中，并且只要$\boldsymbol{m}_1 + \boldsymbol{m}_2 + (\boldsymbol{r}_1 + \boldsymbol{r}_2) \cdot B_I$相对较小，即可运用上述解密方法得到明文$\boldsymbol{m}_1 + \boldsymbol{m}_2$。其同态乘法也可以直接使用密文相乘：

$$\boldsymbol{c}_1 \cdot \boldsymbol{c}_2 = \boldsymbol{e}_1\boldsymbol{e}_2 + (\boldsymbol{e}_1\boldsymbol{g}_2 + \boldsymbol{e}_2\boldsymbol{g}_1 + \boldsymbol{g}_1\boldsymbol{g}_2) \cdot B_J^{pk}$$

其中，$\boldsymbol{e}_1 = \boldsymbol{m}_1 + \boldsymbol{r}_1 \cdot B_I$，$\boldsymbol{e}_2 = \boldsymbol{m}_2 + \boldsymbol{r}_2 \cdot B_I$。该结果仍然在密文空间中，并且当$|e_1 \cdot e_2|$足够小时，可以通过上述解密方法获得$m_1 \cdot m_2$。

4. 自举

随着加法同态和乘法同态的积累，密文中的噪声项逐渐积累增大，直至无法从密文中解密明文，这个时候就得需要借助自举技术消除噪音，使其支持无限次数的加法和同态乘法。Gentry 基于上述的近似同态加密和自举技术设计了第一个全同态加密，为后面的全同态加密设计典型了方向和基础。

3. 基于模运算的全同态方案

首先，我们通过一个整数上基于模运算的全同态方案，进一步体会 Gentry 的近似同态加密方案的设计思路，进而给出一个公钥近似同态加密算法。

（1）对称近似同态加密算法

密钥生成$KeyGen(\lambda)$：密钥是一个奇数$p \in [2^{\eta-1}, 2^{\eta})$.

加密算法$Encrypt(p, m \in \{0,1\})$：密文模$p$与明文具有相同的奇偶性。密文$c = pq + 2r + m$，在其他规定的间隔内随机选择整数$q, r$，使得$2r$绝对值小于$p/2$。

解密算法$Decrypt(p, c)$：输出$(c \bmod p) \bmod 2$.

上述算法是按位执行的同态加解密，设计思想与 Gentry 的基于理想格的加解密思路相似，该方案还可以转换成如下的公钥加密算法。

怎么解释和混淆电路的关系？？

（2）公钥近似同态加密算法

γ表示公钥里面整数的位长；η是密钥的比特长度（是所有公钥整数的隐藏近似-gcd）；ρ是噪声的比特长度(即，公钥元素与私钥的最近倍数之间的距离)；τ是公钥中的整数个数。$\rho = \omega(\log \lambda)$以防止对噪声的暴力攻击；$\eta \geq \rho \cdot \Theta(\lambda \log^2 \lambda)$以支持足够深的电路的同态，以评估“压扁解密电路”；$\gamma = \omega(\eta^2 \log \lambda)$以阻止各种基于晶格的攻击潜在的近似-gcd 问题；$\tau \geq \gamma + \omega(\log \lambda)$以便使用约简中剩余的哈希引理来近似 gcd。我们还使用了二次噪声参数$\rho' = \rho + \omega(\log \lambda)$。一个方便的参数集是$\rho = \lambda, \rho' = 2\lambda, \eta = \tilde{O}(\lambda^2), \gamma = \tilde{O}(\lambda^5), \tau = \gamma + \lambda$。对于特定的$\eta$比特正奇数$p$，我们在$\gamma$位整数上使用以下分布：$\mathcal{D}_{\gamma,\rho}(p) =$

$\left\{\text{选择}q \xleftarrow{\$} \mathbb{Z} \cap \left[0, \frac{2^{\gamma}}{p}\right), r \xleftarrow{\$} \mathbb{Z} \cap (-2^{\rho}, 2^{\rho}) \text{:输出} x = pq + r\right\}$，这个分布显然是有效的抽样。

密钥生成$KeyGen(\lambda)$：私钥是一个η比特长的正奇数$p \xleftarrow{\$} (2\mathbb{Z}+1) \cap [2^{\eta-1}, 2^{\eta})$。生成$\gamma$比特整数$x_i \xleftarrow{\$} \mathcal{D}_{\gamma,\rho}(p), i = 0, \cdots, \tau$.重新标记使得$x_0$是最大的基数，且$r_p(x_0)$是偶数。公钥$pk = < x_0, x_1, \cdots, x_\tau >$。

加密算法$Encrypt(pk, m \in \{0,1\})$：选择一个随机子集$S \subseteq \{1,2,\cdots,\tau\}$，从$(-2^{\rho'}, 2^{\rho'})$随机选择一个整数$r$，输出密文

$$c \leftarrow [m + 2r + 2\sum_{i \in S} x_i]_{x_0}$$

解密算法$Decrypt(sk, c)$：$m \leftarrow (c \bmod p) \bmod 2$.

方案的安全性可以归约到近似整数最大公约数问题。随着运算次数的增加，噪声快速增长，同态乘法运算中，噪声呈指数增长。当噪声大于等于$p/2$时，加密算法将不能正确解密。我们可以“压缩解密电路”来得到一个自举方案，然后调用 Gentry 的自举定理来获得一个完全同态的公钥加密。

2.3.2 CKKS 算法

1. 容错学习

容错学习（Learning with Error, LWE）是在格的难题上构建出来的问题，可以看作解一个带噪声的线性方程组：给定随机向量$\boldsymbol{s} \in \mathbb{Z}_q^n$、随机选择线性系数矩阵$\boldsymbol{A} \in \mathbb{Z}_q^{n \times n}$和随机噪声$\boldsymbol{e} \in \mathbb{Z}_q^n$，生成矩阵线性运算结果$(\boldsymbol{A}, \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e})$。LWE 问题试图从该结果中反推$\boldsymbol{s}$的值，已经证明了 LWE 至少和格中的难题一样困难，从而能够抵抗量子计算机的攻击。

LWE 问题使得在其上构建的加密系统实现十分简单：

- 密钥生成函数：给定随机向量$\boldsymbol{s} \in \mathbb{Z}_q^n$，随机选择线性系数矩阵$\boldsymbol{A} \in \mathbb{Z}_q^{n \times n}$和随机噪声$\boldsymbol{e} \in \mathbb{Z}_q^n$，将$(-\boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e}, \boldsymbol{A})$作为公钥，$\boldsymbol{s}$作为私钥。
- 加密函数：对于需要加密的消息$\boldsymbol{m} \in \mathbb{Z}_q^n$，使用公钥加密为$(\boldsymbol{c_0}, \boldsymbol{c_1}) = (\boldsymbol{m} - \boldsymbol{A} \cdot \boldsymbol{s} + \boldsymbol{e}, \boldsymbol{A})$。
- 解密函数：使用私钥进行解密，计算$\boldsymbol{c_0} + \boldsymbol{c_1} \cdot \boldsymbol{s} = \boldsymbol{m} + \boldsymbol{e}$，当噪声$\boldsymbol{e}$足够小时，能够尽可能地恢复明文$\boldsymbol{m}$。

在 LWE 基础上，RLWE 问题将 LWE 问题扩展到环结构上。RLWE 问题在 LWE 的一维向量空间上，将原来的$\mathbb{Z}_q$环替换成n阶多项式环$\mathbb{Z}_q[X]/(X^N+1)$。于是，LWE 问题中的$\boldsymbol{s}, \boldsymbol{e}, \boldsymbol{m}$从$n$个$\mathbb{Z}_q$环中的元素被替换成一个从多项式环中选取的多项式；$n \times n$的线性系数矩阵$\boldsymbol{A}$被替换成$1 \times 1$的多项式$a \in \mathbb{Z}_q[X]/(X^N+1)$。通过转换，RLWE 的公钥大小从$\mathcal{O}(n^2)$级下降到$\mathcal{O}(n)$级，而不会减少消息$\boldsymbol{m}$携带的数据量（从$N$维向量替换成一个$n$阶多项式）。

另外，基于多项式的乘法操作可以使用离散傅立叶变换算法达到$\mathcal{O}(nlog(n))$的计算复杂度，从而比直接进行矩阵向量乘法快。

2. 方案构造

CKKS 层次同态加密方案即是基于上述 RLWE 问题实现的。具体实现如下：

- 密钥生成函数：给定安全参数，CKKS 生成私钥$s \in \mathbb{Z}_q[X]/(X^N+1)$和公钥$p = (-a \cdot s + e, a)$。式中，$a, e$皆表示多项式环中随机抽取的元素$a, e \in \mathbb{Z}_q[X]/(X^N+1)$，且$e$为较小噪声。
- 加密函数：对于给定的一个消息$m \in \mathbb{C}^{N/2}$（表示为复数向量），CKKS 首先需要对其进行编码，将其映射到多项式环中生成$r \in \mathbb{Z}[X]/(X^N+1)$。然后，CKKS 使用公钥对$r$进行如下加密：

$$(c_0, c_1) = (r, 0) + p = (r - a \cdot s + e, a)$$

- 解密函数：对密文(c_0, c_1)，CKKS 使用密钥进行如下解密：

$$\tilde{r} = c_0 + c_1 * s = r + e$$

$\tilde{r}$需要经过解码，从多项式环空间反向映射回向量空间$\mathbb{C}^{N/2}$。当噪声e足够小时，可以获得原消息的近似结果。

CKKS 支持浮点运算，为保存消息中的浮点数，在编码过程中 CKKS 设定缩放因子$\Delta > 0$，并将浮点数乘以缩放因子生成整数的多项式项，其浮点值被保存在缩放因子Δ中。

3. 再线性化和再缩放

CKKS 支持同态加法和同态乘法。给定两个密文ct_1和ct_2，其对应的同态加法如下：

$$ct_1 + ct_2 = (c_0, c_1) + (c_0', c_1') = (c_0 + c_0', c_1 + c_1')$$

对应的同态乘法操作如下：

$$ct_1 \cdot ct_2 = (c_0, c_1) \cdot (c_0', c_1') = (c_0 \cdot c_0', c_0 \cdot c_1' + c_1 \cdot c_0', c_1 \cdot c_1')$$

可见，在进行同态乘法操作后，密文的大小扩增了一半。因此，每次乘法操作后，CKKS 都需要进行再线性化（Relinearization）和再缩放（Rescaling）操作。

再线性化。再线性化技术能够将扩增的密文$(c_0 \cdot c_0', c_0 \cdot c_1' + c_1 \cdot c_0', c_1 \cdot c_1')$再次恢复为二元对$(d_0, d_1)$，从而允许进行更多的同态乘法操作。

再缩放。另外，因为在编码消息的过程中使用了缩放因子Δ，在进行同态乘法操作时，两个缩放因子皆为Δ的密文相乘，其结果的缩放因子变为Δ^2。如果连续使用同态乘法，缩放因子将会呈指数级上升。所以，每次乘法操作之后，CKKS 都会进行再缩放的操作，将密文值除以Δ以将缩放因子从Δ^2恢复到Δ。在不断再缩放除以Δ的过程中，表示密文值得可用比特每次会下降$\log(\Delta)$比特，直到最终用尽。此时，无法再继续进行同态乘法。

在 CKKS 的加密、解密、再线性化和再缩放的过程中，积累增加的噪声会影响最终解密消息的精度和准度。所以，CKKS 支持浮点运算的同时，对结果的准确性做出了一定的牺牲。CKKS 适用于允许一定误差的、基于浮点数的计算应用，比如机器学习任务。

自举。CKKS 中的再线性化和再缩放是为了保证缩放因子不变，同时降低噪音，但会造成密文模数减少，所以只能构成有限级全同态方案。CKKS 的自举操作能提高密文模数，以支持无限次数的全同态,但是自举成本很高,在满足需求的时候,甚至不需要执行自举操作，后来有一些研究针对 CKKS 方案的自举操作做了精度和效率的提升。

2.5 开发框架 SEAL

SEAL(Simple Encrypted Arithmetic Library)是微软开源的基于 C++的同态加密库，支持 CKKS 方案等多种全同态加密方案，支持基于整数的精确同态运算和基于浮点数的近似同态运算。该项目采用商业友好的 MIT 许可证在 GitHub 上（https://github.com/microsoft/SEAL）开源。SEAL 基于 C++实现，不需要其他依赖库。

2.4.1 安装部署

1. SEAL 库安装

步骤：

（1） git clone 加密库资源

在 Ubuntu 的 home 文件夹下建立文件夹 seal，进入该文件夹后，打开终端，输入命令：

```
git clone https://github.com/microsoft/SEAL
```

运行完毕，将在 seal 文件夹下自动建立 SEAL 这个新文件夹。

（2） 编译和安装

输入命令：

```
cd SEAL
cmake .
```

网络原因可能会报错，多尝试几次。该步骤成功后显示如下：

```
-- Configuring done
-- Generating done
-- Build files have been written to: /home/liuzheli/seal/SEAL
liuzheli@liuzheli-virtual-machine:~/seal/SEAL$
```

```
make
```

可能会报错，多尝试几次。该步骤成功后显示如下：

```
[100%] Linking CXX static library lib/libseal-4.0.a
[100%] Built target seal
liuzheli@liuzheli-virtual-machine:~/seal/SEAL$
```

sudo make install

最后一步成功后显示如下

```
-- Installing: /usr/local/include/SEAL-4.0/seal/util/ztools.h
liuzheli@liuzheli-virtual-machine:~/seal/SEAL$
```

安装完毕！（一定要确保最后两步成功执行）

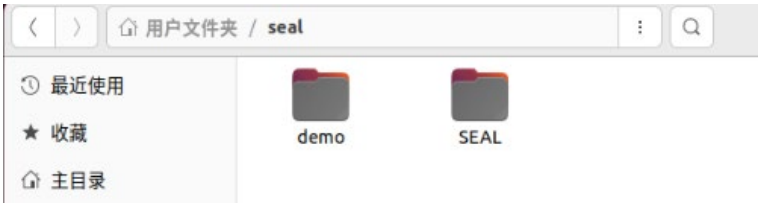
说明：cmake 是一种高级编译配置工具，它可以将多个 cpp、hpp 文件组合构建为一个大工程的语言。它能够输出各种各样的 makefile 或者 project 文件，所有操作都是通过编译 CMakeLists.txt 来完成。

通常 cmake 执行之后，再执行 make 和 make install 可以完成开源项目的编译和安装。

在 make install 步骤，相关的头文件和静态库都被安装复制到了/usr/local 文件夹下的 include 和 lib 文件夹中，以便系统里其它应用程序可以查找到并使用。

2. 简单测试程序

在 seal 文件夹下建立一个测试文件夹：demo，如下图所示：



进而，在 demo 文件夹下，使用文本编辑器建立一个 cpp 文件 test.cpp，内容如下：

```
#include "seal/seal.h"
#include <iostream>

using namespace std;
using namespace seal;

int main(){

    EncryptionParameters parms(scheme_type::bfv);
    printf("hellow world\n");
    return 0;
}
```

为了完成 test.cpp 的编译和执行，需要编写一个 CMakeLists.txt 文件，内容如下：

```
cmake_minimum_required(VERSION 3.10)
project(demo)
add_executable(test test.cpp)
add_compile_options(-std=c++17)

find_package(SEAL)
target_link_libraries(test SEAL::seal)
```

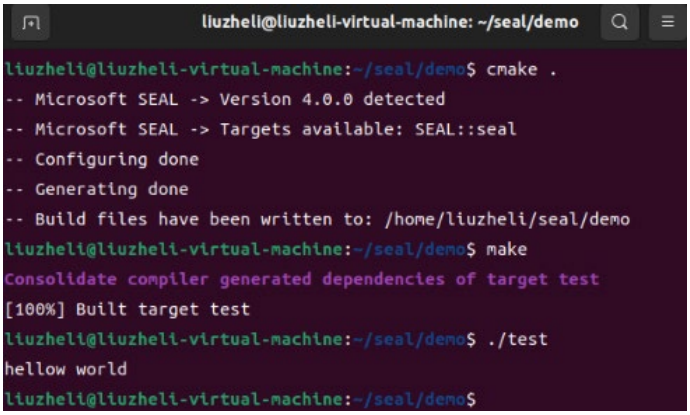
编写完毕后，打开控制台，依次运行：

cmake .

make

./test

运行结果如下：



2.4.2 应用示例

本节演示基于 CKKS 方案构建一个基于云服务器的算力协助完成客户端的某种运算。

1. 标准化构建流程

首先，CKKS 是一个公钥加密体系，具有公钥加密体系的一切特点，例如公钥加密、私钥解密等。因此，我们的代码中需要以下组件：

✧ 密钥生成器 keygenerator

✧ 加密模块 encryptor

✧ 解密模块 decryptor

其次，CKKS 是一个（level）全同态加密算法（level 表示其运算深度仍然存在限制），可以实现数据的“可算不可见”，因此我们还需要引入：

✧ 密文计算模块 evaluator

最后，加密体系都是基于某一数学困难问题构造的，CKKS 所基于的数学困难问题在一个“多项式环”上（环上的元素与实数并不相同)，因此我们需要引入：

✧ 编码器 encoder

来实现数字和环上元素的相互转换。

总结下来，整个构建过程为：

① 选择 CKKS 参数 parms

② 生成 CKKS 框架 context

③ 构建 CKKS 模块 keygenerator、encoder、encryptor、evaluator 和 decryptor

④ 使用 encoder 将数据 n 编码为明文 m

⑤ 使用 encryptor 将明文 m 加密为密文 c

⑥ 使用 evaluator 对密文 c 运算为密文 c’

⑦ 使用 decryptor 将密文 *c*’解密为明文 *m*’

⑧ 使用 encoder 将明文 *m*’解码为数据 *n*

2. 示例代码

同态加密算法最直观的应用是云计算，其基本流程为：

① 发送方利用公钥 *pk* 加密明文 *m* 为密文 *c*

② 发送方把密文 *c* 发送到服务器

③ 服务器执行密文运算，生成结果密文 *c*’

④ 服务器将结果密文 *c*’发送给接收方

⑤ 接收方利用私钥 *sk* 解密密文 *c*’为明文结果 *m*’

当发送方与接收方相同时，则该客户利用全同态加密算法完成了一次安全计算，即既利用了云计算的算力，又保障了数据的安全性，这对云计算的安全应用有重要意义。

代码如下：

```
#include "examples.h"
/*
该文件可以在 SEAL/native/example 目录下找到
*/
#include <vector>
using namespace std;
using namespace seal;
#define N 3
//本例的目的是计算 x，y，z 的乘积

int main(){
//客户端的视角：要进行计算的数据
vector<double> x, y, z;
      x = { 1.0, 2.0, 3.0 };
      y = { 2.0, 3.0, 4.0 };
      z = { 3.0, 4.0, 5.0 };

//构建参数容器 parms
EncryptionParameters parms(scheme_type::ckks);
/*CKKS 有三个重要参数：
1.poly_module_degree(多项式模数)
2.coeff_modulus（参数模数）
3.scale（规模）*/

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 }));
```

```
//选用 2^40 进行编码
double scale = pow(2.0, 40);

//用参数生成 CKKS 框架 context
SEALContext context(parms);

//构建各模块
//首先构建 keygenerator，生成公钥、私钥和重线性化密钥
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);

//构建编码器，加密模块、运算器和解密模块
//注意加密需要公钥 pk；解密需要私钥 sk；编码器需要 scale
    Encryptor encryptor(context, public_key);
    Evaluator evaluator(context);
    Decryptor decryptor(context, secret_key);

    CKKSEncoder encoder(context);
//对向量 x、y、z 进行编码
    Plaintext xp, yp, zp;
    encoder.encode(x, scale, xp);
    encoder.encode(y, scale, yp);
    encoder.encode(z, scale, zp);
//对明文 xp、yp、zp 进行加密
    Ciphertext xc, yc, zc;
    encryptor.encrypt(xp, xc);
    encryptor.encrypt(yp, yc);
    encryptor.encrypt(zp, zc);
/*对密文进行计算，要说明的原则是：
1.加法可以连续运算，但乘法不能连续运算
2.密文乘法后要进行 relinearize 操作
3.执行乘法后要进行 rescaling 操作
4.进行运算的密文必需执行过相同次数的 rescaling（位于相同 level）
*/
//基于上述原则进行运算

//至此，客户端将 pk、CKKS 参数发送给服务器，服务器开始运算
//服务器的视角：先设中间变量
    Ciphertext temp;
    Ciphertext result_c;
```

```
//计算 x*y，密文相乘，要进行 relinearize 和 rescaling 操作
        evaluator.multiply(xc,yc,temp);
        evaluator.relinearize_inplace(temp, relin_keys);
        evaluator.rescale_to_next_inplace(temp);

//在计算 x*y * z 之前，z 没有进行过 rescaling 操作，所以需要对 z 进行一次乘法和 rescaling
操作，目的是 使得 x*y 和 z 在相同的层
        Plaintext wt;
        encoder.encode(1.0, scale, wt);
//我们可以查看框架中不同数据的层级：
cout << "       + Modulus chain index for zc: "
<< context.get_context_data(zc.parms_id())->chain_index() << endl;
cout << "       + Modulus chain index for temp(x*y): "
<< context.get_context_data(temp.parms_id())->chain_index() << endl;
cout << "       + Modulus chain index for wt: "
<< context.get_context_data(wt.parms_id())->chain_index() << endl;

//执行乘法和 rescaling 操作：
        evaluator.multiply_plain_inplace(zc, wt);
        evaluator.rescale_to_next_inplace(zc);

//再次查看 zc 的层级，可以发现 zc 与 temp 层级变得相同
cout << "       + Modulus chain index for zc after zc*wt and rescaling: "
<< context.get_context_data(zc.parms_id())->chain_index() << endl;

//最后执行 temp（x*y）* zc（z*1.0）
        evaluator.multiply_inplace(temp, zc);
        evaluator.relinearize_inplace(temp,relin_keys);
        evaluator.rescale_to_next(temp, result_c);
//计算完毕，服务器把结果发回客户端

//客户端进行解密和解码
        Plaintext result_p;
        decryptor.decrypt(result_c, result_p);
//注意要解码到一个向量上
        vector<double> result;
        encoder.decode(result_p, result);
//得到结果
//正确的话将输出：{6.000，24.000，60.000，...，0.000，0.000，0.000}
        cout << "结果是：" << endl;
        print_vector(result,3,3);
return 0;
}
```

3. 参数解释

本小节对三个参数进行简单的解释。

（1）poly_modulus_degree（polynomial modulus）

该参数必须是 2 的幂，如 1024, 2048, 4096, 8192, 16384, 32768，当然再大点也没问题。

更大的 poly_modulus_degree 会增加密文的尺寸，这会让计算变慢，但也能让你执行更复杂的计算。

（2）[ciphertext] `coefficient modulus

这是一组重要参数，因为 rescaling 操作依赖于 coeff_modules。

简单来说，coeff_modules 的个数决定了你能进行 rescaling 的次数，进而决定了你能执行的乘法操作的次数。

coeff_modules 的最大位数与 poly_modules 有直接关系，列表如下：

poly_modulus_degree	max coeff_modulus bit-length
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

本文例子中的{60，40，40，60}有以下含义：

① coeff_modules 总位长 200（60+40+40+60）位

② 最多进行两次（两层）乘法操作

该系列数字的选择不是随意的，有以下要求：

① 总位长不能超过上表限制

② 最后一个参数为特殊模数，其值应该与中间模数的最大值相等

③ 中间模数与 scale 尽量相近

注意：如果将模数变大，则可以支持更多层级的乘法运算，比如 poly_modulus 为 16384 则可以支持 coeff_modules= { 60, 40, 40, 40, 40, 40, 40, 60 }，也就是 6 层的运算。

（3）Scale

Encoder 利用该参数对浮点数进行缩放，每次相乘后密文的 scale 都会翻倍，因此需要执行 rescaling 操作约减一部分，约模的大素数位长由 coeff_modules 中的参数决定。

Scale 不应太小，虽然大的 scale 会导致运算时间增加，但能确保噪声在约模的过程中被正确地舍去，同时不影响正确解密。

因此，两组推荐的参数为：

Poly_module_degree = 8196; coeff_modulus={60,40,40,60};scale = 2^40

Poly_module_degree = 8196; coeff_modulus={50,30,30,30.50};scale = 2^30

4. 注意事项

如示例代码中所述，每次进行运算前，要保证参与运算的数据位于同一“level”上。加法不需要进行 rescaling 操作，因此不会改变数据的 level。数据的 level 只能降低无法升高，所以要小心设计计算的先后顺序。

可以通过输出 p.scale()、p.parms_id()以及 context->get_context_data(p.parms_id())->chain_index()来确认即将进行操作的数据满足如下计算条件：1）用同一组参数进行加密；2）位于（chain）上的同一 level；3）scale 相同。

要想把不同 level 的数据拉到同一 level，可以利用乘法单位元 1 把层数较高的操作数拉到较低的 level（如本例），也可以通过内置函数进行直接转换。

目前，SEAL 提供了 reverse、square 等有限的计算操作，大部分复杂运算需要自己编写代码实现，在实现过程中要根据数据量把握好精度和性能的取舍。

5. 编译运行

将 seal 下的 native 下的 examples 下的 example.h 复制到 Demo 文件夹下；这个头文件定义了使用 seal 的常见头文件，并定义了一些输出函数。

定义文件 ckks_example.cpp，并将源代码复制到该文件。

更改 CMakeLists.txt 内容：

```
cmake_minimum_required(VERSION 3.10)
project(demo)
add_executable(he ckks_example.cpp)
add_compile_options(-std=c++17)

find_package(SEAL)
target_link_libraries(he SEAL::seal)
```

编写完毕后，打开控制台，依次运行：

cmake .

make

./he

运行结果如下：

```
liuzheli@liuzheli-virtual-machine:~/seal/demo$ ./he
    + Modulus chain index for zc: 2
    + Modulus chain index for temp(x*y): 1
    + Modulus chain index for wt: 2
    + Modulus chain index for zc after zc*wt and rescaling: 1
结果是:

    [ 6.000, 24.000, 60.000, ..., -0.000, 0.000, -0.000 ]
```