Chapter 1

Introdução à criptografia completamente homomórfica com implementação em Sage

Hilder V. L. Pereira (imec-COSIC, KU Leuven) e Eduardo Morais (Disigma)

Abstract

When new cryptographic schemes are proposed in the literature, their implementation can't be based on previous code, and a common strategy on these situations is to first implement such constructions using mathematical programming languages like Sage*. By doing that we require less effort to obtain a proof of concept implementation, which later can be used as a base for production-level development in a different programming language. This way we can also postpone dealing with some abstract mathematical concepts, which are provided in* Sage*, making it a good language for prototyping cryptographic schemes. In this course we are going to demonstrate how fully homomorphic encryption can be constructed using* Sage*. We will provide working code for the main building blocks, therefore the student can experiment with different parameters, or construct new applications.*

Resumo

Quando novos esquemas criptográficos são propostos na literatura, suas implementações não podem ser baseadas em código existente, e uma estratégia nestas situações é primeiramente implementar tais construções usando linguagens de programação como Sage*. Com isso exigimos menos esforço para obter uma prova de conceito, que pode depois servir de base para a implementação que será colocada em produção, e que pode ser desenvolvida em outra linguagem. Desta forma podemos postergar a solução de detalhes matemáticos abstratos, que estão disponíveis em* Sage*, o que a torna uma boa linguagem para prototipagem de esquemas criptográficos. Neste curso nós demonstraremos como a encriptação completamente homomórfica pode ser construída usando* Sage*. Apresentaremos código-fonte funcional para os principais componentes, logo o estudante pode experimentar com diferentes parâmetros, ou pode construir novas aplicações.*

1.1. Introdução

Com a evolução da internet e a popularização do acesso a conexões de alta velocidade, cada vez mais, aplicações que antes eram executadas localmente têm sido delegadas para terceiros. Por exemplo, em vez de buscar, baixar e armazenar vídeos e músicas para reproduzi-los posteriormente, utilizam-se sites e programas com catálogos online e reprodução por *stream*, como Netflix e Spotify; em vez de armazenar dados localmente e criar rotinas de *backup* por conta própria, utilizam-se serviços de armazenamento como Dropbox e Microsoft OneDrive; provedores de e-mail como Gmail substituíram em grande parte programas que gerenciavam e-mails localmente, como o Outlook e o Mozilla Thunderbird.

Esses serviços remotos são conhecidos vulgarmente como nuvem e são atrativos, principalmente, por conta da praticidade, já que é necessário menos conhecimento técnico e também uma infraestrutura mais simples para usar um serviço em nuvem do que para implementá-lo e gerenciá-lo localmente. Por outro lado, geralmente abre-se mão da privacidade, já que muitos dados dos usuários são enviados aos servidores. Em particular, o fato de uma funcionalidade depender de dados sensíveis pode inviabilizar sua utilização. Um exemplo clássico é o de um hospital que tem uma base de dados com informações sobre exames feitos em pacientes e de um servidor na nuvem que tem um modelo de aprendizagem de máquina, como uma rede neural, capaz de gerar um diagnóstico. O hospital gostaria então de enviar os dados de pacientes à nuvem para obter um diagnóstico que poderia auxiliar seus médicos. No entanto, os dados dos pacientes são sigilosos e não podem ser compartilhados com terceiros. Para proteger a privacidade dos dados, seria possível cifrá-los e enviar ao servidor apenas os criptogramas, porém, com esquemas de criptografia tradicionais, o servidor não seria capaz de executar sua rede neural usando apenas dados cifrados como entrada.

Esse é o problema que criptografia homomórfica resolve: com ela, é possível fazer computação sobre dados cifrados como se estivéssemos computando sobre os dados originais em claro. Além da entrada, o resultado da computação também é cifrado, logo, apenas o usuário que cifrou os dados é capaz de decifrar o resultado. Em um cenário ideal, seria possível, por exemplo, cifrar um texto, enviar o criptograma correspondente ao Google, receber o resultado da busca cifrado e então decifrá-lo. Ou seja, seria possível fazer uma busca na internet sem revelar o que se está buscando nem o que foi encontrado.

Mas, como acontece quase sempre, adicionar uma nova funcionalidade também adiciona custo computacional. Nesse caso, a nova funcionalidade é manter a privacidade dos dados e o custo computacional adicional é enorme tanto em termos de memória quanto de processamento, pois os criptogramas são geralmente centenas ou milhares de vezes maiores que os dados originais e cada operação homomórfica é ordens de magnitude mais lenta que a operação correspondente em texto claro. A despeito desse custo adicional, criptografia homomórfica já é razoavelmente prática para algumas aplicações e há muita pesquisa voltada a usá-la para implementar protocolos que processem dados preservando a privacidade, principalmente envolvendo algoritmos de aprendizagem de máquina, como classificadores. Por exemplo, [BMMP18] projetou uma rede neural homomórfica que recebe imagens de dígitos escritos a mão, cifradas, e devolve um criptograma que cifra um valor entre 0 e 9 correspondendo ao dígito contido na imagem. Essa rede neural

tem uma acurácia de 96% e leva apenas 1,6 segundos para classificar uma imagem. Apesar de a base de dados ser relativamente simples, esses resultados já são impressionantes, tendo em vista a tarefa sendo executada: a rede neural classifica uma imagem sem ter acesso à ela, mas apenas aos criptogramas, que não revelam qualquer informação sobre as mensagens que eles cifram.

Outras linhas de pesquisa atuais relacionadas à criptografia homomórfica focam em melhorar a eficiência, na análise dos problemas criptográficos utilizados, na criptoanálise dos esquemas existentes e em descobrir outras primitivas criptográficas mais avançadas que podem ser construídas a partir da criptografia homomórfica. Em suma, esse é um tópico em voga, que tem gerado muito interesse da comunidade acadêmica, com diversas publicações nas mais importantes conferências de criptografia, e também da indústria, com grandes empresas investindo nessa tecnologia[1]. Inclusive, recentemente iniciou-se o esforço para padronizar cifras homomórficas existentes e definir critérios mínimos de segurança [CCD+17], uma interface comum para facilitar compatibilidade e modularidade [BDH+17], e formas de utilização adequadas a diversos cenários práticos [ACC+17]. O processo de padronização deve durar aproximadamente 5 anos e trazer maturidade para que esta tecnologia seja amplamente adotada.

1.1.1. Objetivos do curso

O objetivo deste curso é desmistificar a criptografia completamente homomórfica e mostrar que os conceitos principais desse tipo de primitiva criptográfica, mesmo os utilizados nos artigos científicos mais recentes da área, estão ao alcance até mesmo de alunos de graduação.

Utilizaremos uma abordagem prática em que os aspectos teóricos serão apresentados, explicados e então implementados em Sage [Sag20], que é uma linguagem simples, com sintaxe praticamente idêntica à da linguagem Python, mas que tem uma coleção de bibliotecas para as mais diversas áreas da matemática. Poderemos, portanto, dar ênfase às ideias principais por trás dos esquemas criptográficos e nos preocupar pouco com os detalhes técnicos de implementação, todavia, obtendo códigos funcionais conforme avançamos nos tópicos.

Espera-se que, ao final do curso, os alunos possam: analisar e entender esquemas de criptografia completamente homomórfica; ter familiaridade com os dois principais problemas criptográficos utilizados para construir cifras completamente homomórficas; implementar esses esquemas, incluindo o *bootstrapping*; usar esses esquemas em aplicações como computação segura na nuvem.

1.1.2. O que é criptografia (completamente) homomórfica

Dizemos que uma cifra é homomórfica para uma operação $\star$ se dados dois criptogramas $\mathsf{Enc}(m_1)$ e $\mathsf{Enc}(m_2)$, é possível gerar um novo criptograma $\mathsf{Enc}(m_1 \star m_2)$ sem usar a chave secreta. Por exemplo, considerando a cifra assimétrica RSA na sua versão mais básica,

[1]Para citar apenas alguns exemplos: Google (https://github.com/google/fully-homomorphic-encryption), IBM (https://youtu.be/JyK1BnmXQwU), Intel (https://arxiv.org/abs/2103.16400) e Microsoft (https://www.microsoft.com/en-us/research/project/homomorphic-encryption/).

temos $\mathsf{Enc}(m_i) \equiv m_i^e \pmod{n}$ e n é uma informação pública. Então, é fácil ver que

$$\mathsf{Enc}(m_1) \cdot \mathsf{Enc}(m_2) \equiv m_1^e \cdot m_2^e \equiv (m_1 \cdot m_2)^e \pmod{n}.$$

Ou seja, ao multiplicar dois criptogramas, obtém-se o *produto* das mensagens, encriptado sob a mesma chave pública e. Por isso, dizemos que o RSA é homomórfico com relação à multiplicação. Como é possível multiplicar vários criptogramas, pode-se gerar $\mathsf{Enc}(m_1 \cdot ... \cdot m_\ell)$ para qualquer $\ell \in \mathbb{N}$, no entanto, não é possível gerar $\mathsf{Enc}(m_1 + m_2)$, ou seja, o RSA não é homomórfico para a adição. Já outros esquemas criptográficos, como o Paillier [Pai99], são homomórficos para a adição, mas não para a multiplicação, o que significa que é possível gerar criptogramas como $\mathsf{Enc}(m_1 + ... + m_\ell)$ para qualquer $\ell \in \mathbb{N}$.

Vemos que com essas cifras, o conjunto de funções que podem ser avaliadas homomorficamente é bastante limitado: mesmo funções simples como uma média ponderada usam pelo menos duas operações diferentes, isto é, requerem ao menos que se possa calcular $\mathsf{Enc}(m_1 \cdot m_2 + m_3 \cdot m_4)$. desde a criação do RSA, em 1977, construir uma cifra que permitisse avaliar homomorficamente qualquer função computável se mostrou uma tarefa difícil. Apenas em 2005, foi proposta uma cifra homomórfica tanto para a adição quanto para a multiplicação [BGN05], no entanto, apenas um produto era possível, ou seja, as funções que podem ser calculadas homomorficamente ainda eram bem restritas. Foi apenas em 2009, cerca de 30 anos depois da publicação do RSA, que a primeira cifra *completamente* homomórfica foi proposta [Gen09]. Basicamente, com essa cifra, é possível encriptar bits e aplicar portas lógicas como *AND*, *OR*, e *NOT*, denotados respectivamente pelas operações Booleanas $\wedge$, $\vee$ e $\neg$. Ou seja, a partir de $\mathsf{Enc}(m_i)$ e $\mathsf{Enc}(m_j)$, pode-se gerar $\mathsf{Enc}(m_i \wedge m_j)$, $\mathsf{Enc}(m_i \vee m_j)$ e $\mathsf{Enc}(\neg m_i)$. Como qualquer circuito pode ser expresso em termos dessas três portas lógicas, pode-se avaliar qualquer circuito homomorficamente. Finalmente, como qualquer função computável pode ser expressa por um circuito, é possível calcular $\mathsf{Enc}(f(m_1, ..., m_\ell))$ para qualquer função computável f. Por isso, dizemos que essa cifra é completamente homomórfica[2], enquanto RSA, Paillier, e outras cifras que suportam apenas um tipo de operação são chamadas de cifras *parcialmente* homomórfica.

Grosso modo, as cifras completamente homomórficas cifram uma mensagem m da seguinte forma:

- combinam a chave secreta sk com algum valor aletório, obtendo $a \cdot \mathsf{sk}$;
- adicionam um ruído r, que é apenas um valor aletório pequeno, obtendo $a \cdot \mathsf{sk} + r$;
- finalmente, adicionam a mensagem, então o criptograma é da forma $a \cdot \mathsf{sk} + r + m$.

Para decifrar, primeiro usa-se a chave secreta para remover o termo $a \cdot \mathsf{sk}$, então aplica-se alguma técnica de correção de erros elementar para remover o ruído r e recuperar m. No entanto, é importante notar que o deciframento só funciona se o ruído for relativamente pequeno e esse é o fator que dificulta a criação de uma cifra completamente homomórfica, pois cada operação aumenta o ruído. Por exemplo, para calcular $m_1 \wedge m_2 \vee m_3$

[2]O termo original, em inglês, é *fully homomorphic encryption*.

homomorficamente, começa-se com $c_i := \mathsf{Enc}(m_i)$, $i = 1,2,3$, todos com pouco ruído, então calcula-se $c_4 := \mathsf{Enc}(m_1 \wedge m_2)$, que tem mais ruído que os criptogramas originais, e finalmente aplica-se um *OR* homomórfico em c_4 e c_3, obtendo $c_5 := \mathsf{Enc}(m_1 \wedge m_2 \vee m_3)$ com ruído ainda maior que o de c_4. Como há um limite para a quantidade de ruído que um criptograma pode surportar, fica claro que não se pode calcular qualquer função homomorficamente. Para resolver esse problema e transformar uma cifra que suporta uma quantidade limitada de operações em uma cifra completamente homomórfica, usa-se um procedimento chamado *bootstrapping*. Seu funcionamento é explicado em detalhe na seção 1.3.3, mas, por ora, saiba que ele serve para reduzir o ruído contido em um criptograma e que ele é a parte mais complexa e computacionalmente cara de toda cifra completamente homomórfica. Então, para avaliar uma função f homomorficamente, primeiro representa-se f em termos de operações homomórficas disponíveis, e.g, portas lógicas *AND*, *OR* e *NOT*, então prossegue-se aplicando essas operações até que o ruído atinga um limiar, e então executa-se o *bootstrapping* antes de continuar aplicando as próximas operações. Esse processo é repetido até que f seja inteiramente calculada.

1.1.3. Brevíssima introdução à linguagem de programação Sage

Sage[3] ou SageMath é uma linguagem de programação livre e código aberto bastante versátil: contém elementos dos paradigmas procedural, funcional e orientado a objetos, trabalha com cálculo simbólico e numérico, e oferece funcionalidades relacionadas com diversas áreas da matemática e criptografia. Além disso, é uma linguagem simples, com sintaxe praticamente igual a do Python. A principal exceção é que `x^y` significa "x elevado a y" em Sage e "x XOR y" em Python.

Sage trabalha nativamente tem diversos tipos de dados avançados, como números complexos, polinômios, vetores e matrizes. Além disso, em geral há conversão automática entre os tipos, como ilustrado no programa a seguir:

```
A = Matrix(ZZ, 2, 2, [[1, 2], [3, 4]]) # matriz integral 2x2
print(A.det()) # determinante de A: 1*4 - 2*3 = -2
u = vector(QQ, [0, 1/4]) # QQ = conjunto dos racionais
v = A*u
print(v) # imprime (1/2, 1)
```

É possível trabalhar diversas estruturas algébricas, como grupos, anéis e corpos. Também pode-se facilmente definir quocientes. Por exemplo, o anel $\mathbb{Z}/q\mathbb{Z}$, formado pelo quociente de $\mathbb{Z}$ e $\langle q \rangle$ pode ser obtido com o comando `ZZ.quotient(q)`. Dado um anel A, pode-se criar o anel de polinômios $A[X]$ usando `P.<x> = A['x']`. Novamente, é possível criar um quociente, como $A[X]/\langle X^N+1 \rangle$, o anel de polinômios módulo X^N+1 e com coeficientes em A, usando o comando `P.quotient(x^N + 1)`. Ademais, geralmente é possível obter um elemento aleatório de um conjunto qualquer, digamos, P, usando o comando `P.random_element()`. O código a seguir constrói o anel $R = \mathbb{Z}_q[X]/\langle X^N - 1 \rangle$ e imprime um vetor com n elementos aleatórios de R.

```
# -*- coding: utf-8 -*-
```

[3] Sigla em inglês para *system for algebra and geometry experimentation* (sistema algébrico e geométrico de experimentações).


```
N, n = 4, 10
q = next_prime(16) # primeiro primo maior que 16
Zq = ZZ.quotient(q) # inteiros mod 17
P.<x> = Zq['x'] # anel de polinômios mod q
f = x^N - 1 # definindo um polinômio
R = P.quotient(f) # quociente entre P e <f>
u = (R^n).random_element() # vetor aleatório
print(u) # imprime os n elementos
```

1.2. O problema do divisor comum aproximado (ACD)

Nesta seção, vamos estudar o problema computacional conhecido como ACD, do inglês *Approximate Common Divisor problem*[4]. Ele é o problema subjacente de uma família de cifras homomórficas conhecida como *FHE over the integers* (criptografia completamente homomórfica sobre os inteiros). Vamos implementar em Sage as distribuições relacionadas a esse problema e um algoritmo para resolvê-lo. Posteriormente, na Seção 1.3, vamos implementar uma cifra homomórfica baseada no ACD.

1.2.1. Definição do problema

Imagine que lhe foram dados vários múltiplos de um número primo p desconhecido, i.e., vários inteiros da forma $x_i := pq_i$ com q_i aleatório, e que sua tarefa é descobrir o valor de p. Como proceder? Claramente, esse problema pode ser resolvido com o cálculo de $\mathsf{mdc}(x_1, ..., x_\ell)$, i.e., dados vários múltiplos de p, calcule o máximo divisor comum e ele será provavelmente p. No entanto, se em vez de múltiplos de p, lhe forem dados "múltiplos aproximados", valores próximos dos múltiplos, da forma $x_i := pq_i + r_i$, onde r_i é um número aleatório relativamente pequeno em comparação com p. Como proceder agora? Esse problema simples é o ACD e todos os algoritmos para resolvê-lo levam tempo exponencial, mesmo algoritmos quânticos, por isso, esquemas criptográficos baseados no ACD são chamados de *pós-quânticos*.

Antes precisamos definir a distribuição de probabilidade subjacente, conforme segue.

Definição 1.2.1. *Sejam ρ, η e γ inteiros usados para parametrizar o problema e tais que $\gamma > \eta > \rho > 0$. Seja p um número primo que satisfaz $2^{\eta-1} \leq p \leq 2^\eta$. A distribuição de probabilidade $\mathscr{D}_{\gamma,\rho}(p)$, cujo suporte é $[\![0, 2^\gamma[\![$, é definida como*

$$\mathscr{D}_{\gamma,\rho}(p) := \{\textit{Amostre } q \leftarrow [\![0, 2^\gamma / p\,[\![\;\; e \;\; r \leftarrow]\!] - 2^\rho, 2^\rho [\![\,, \textit{ devolva } x := pq + r\}.$$

Agora conseguimos definir o problema ACD.

Definição 1.2.2 (ACD)**.** *O problema do divisor comum aproximado, com parâmetros ρ, η e γ, ou simplesmente $(\rho, \eta, \gamma) - \mathsf{ACD}$, é o problema de descobrir p dada uma amostra arbitrariamente grande da distribuição $\mathscr{D}_{\gamma,\rho}(p)$.*

A versão decisional do problema, chamada $(\rho, \eta, \gamma) - \mathsf{ACD}$ decisional, é o problema de distinguir entre a distribuição $\mathscr{D}_{\gamma,\rho}(p)$ e a distribuição uniforme $\mathscr{U}([\![0, 2^\gamma[\![)$.

[4]Também conhecido como AGCD *problem*, do inglês *Approximate Greatest Common Divisor problem*.

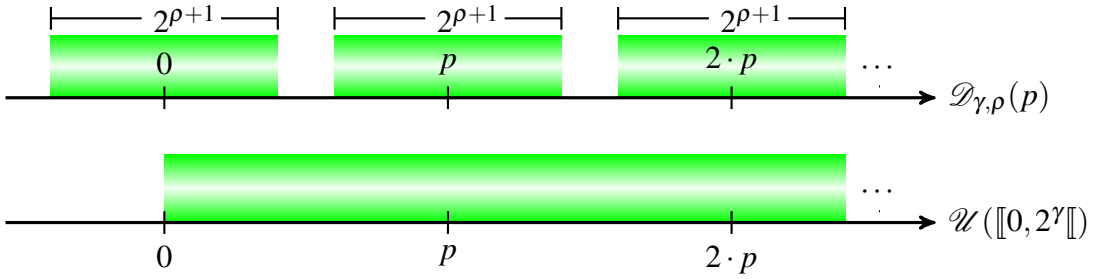

Figure 1.1. As duas distribuições que devem ser distinguidas na versão decisional do problema ACD**. As áreas verde representam os valores (inteiros) que podem ser amostrados de ambas as distribuições.**

Note que se x é uniformemente distribuído em $[\![0,2^{\gamma}[\![$, então usando a divisão Euclideana para escrever $x = pq + r$, temos que a distribuição de r é próxima da uniforme em $[\![0, p-1[\![$, em outras palavras, r pode ser igual a qualquer valor entre 0 e $p-1$. No entanto, todos os valores de $pq+r$ com $2^{\rho} \leq r < p \approx 2^{\eta}$ têm probabilidade zero em $\mathscr{D}_{\gamma,\rho}(p)$. Portanto, quanto maior for $\eta - \rho$, maior será a distância estatística entre $\mathscr{D}_{\gamma,\rho}(p)$ e $\mathscr{U}([\![0,2^{\gamma}[\![)$, logo, mais fácil o problema se torna. Isso é ilustrado na Figura 1.1.

Implementar a distribuição $\mathscr{D}_{\gamma,\rho}(p)$ em Sage é simples: basta usar o comando `ZZ.random_element`, como ilustrado no código a seguir:

```
def sample_r(rho):
    return ZZ.random_element(-2^rho,2^rho)
def sample_q(gamma, eta):
    return ZZ.random_element(0, 2^(gamma - eta))
def sample_acd(gamma, p, rho):
    eta = ceil(log(p, 2))
    r = sample_r(rho)
    q = sample_q(gamma, eta)
    return p*q + r
```

1.2.2. Criptoanálise do problema ACD

Os parâmetros γ, η e ρ não são variáveis livres que podem ser escolhidos livremente, pois a segurança dos esquemas criptográficos baseados no ACD depende desses parâmetros. Em geral, eles são escolhidos de tal forma que todos os algoritmos conhecidos para resolver o ACD levem tempo exponencial em λ, um parâmetro de segurança, i.e., se um algoritmo $\mathscr{A}$ resolve o ACD em tempo $T(\gamma,\eta,\rho)$, então $T(\gamma,\eta,\rho) = \Omega(2^{\lambda})$. Com isso, "quebrar a segurança" de um esquema baseado no ACD requer ao menos 2^{λ} operações e dizemos que o esquema oferece λ bits de segurança.

As principais duas famílias de algoritmos usados para resolver o ACD são os ataques por mdc (máximo divisor comum) e os ataques de reticulados ortogonais. Nesta seção, vamos estudar a estratégia geral dos ataques por mdc, implementar uma versão simples desse algoritmo e deduzir quais restrições ele impõe aos parâmetros.

A principal ideia por trás desse tipo de ataque é tentar remover o termo r_i de

$x_i := pq_i + r_i$ para obter múltiplos de p, então prosseguir calculando mdc deles para obter múltiplos cada vez menores. Note que se $a = p\prod_{i=1}^{n} q_i$ e $b = p\prod_{i=1}^{m} s_i$, então $\mathsf{mdc}(a,b) = p\prod_{t_i \in T} t_i$ sendo $T = \{q_1,...,q_n\} \cap \{s_1,...,s_m\}$. Então, quando o múltiplo obtido tem apenas η bits, é porque na verdade ele já é igual a p.

Para obter um múltiplo a partir de $x_i := pq_i + r_i$, basicamente, fazemos uma busca exaustiva no elemento r_i calculando o seguinte produto: $m = \prod_{r=-2^\rho}^{2^\rho}(x_i - r)$. Note que algum fator é igual a pq_i, logo $m \in p\mathbb{Z}$, como esperado.

Uma implementação em Sage do algoritmo é apresentada a seguir. Ele recebe uma lista $x_0,...,x_\ell \leftarrow \mathscr{D}_{\gamma,\rho}(p)$, da qual obtém os múltiplos de p. Para reduzir o custo de calcular o mdc, os fatores primos pequenos são removidos manualmente já no começo. Como todos os $\Theta(2^\rho)$ valores possíveis de r são calculados, o tempo de execução é exponencial em ρ, por isso, sugerimos que você o execute usando valores pequenos para os parâmetros, como $(\rho,\eta,\gamma) = (10,20,30)$ e uma lista com poucos elementos, e.g., $\ell = 25$. Existem várias formas de melhorar esse algoritmo, por exemplo, acelerando o cálculo de $\prod_{r=-2^\rho}^{2^\rho}(x_i - r)$ [CN12] ou tornando o algoritmo probabilístico [CNT12]. Assim, a complexidade temporal se torna $O(\mathsf{poly}(\gamma,\rho)\cdot 2^\rho)$, então, para garantir que o custo seja ao menos 2^ρ operações, basta usar $\rho = \Omega(\lambda)$.

```
# -*- coding: utf-8 -*-
# Este código usa a função sample_agcd definida anteriormente

def remove_fatores_pequenos(a, num_fact=1000):
    q = 2
    for _ in range(num_fact):
        while q.divides(a):
            a /= q
        q = next_prime(q)
    return ZZ(a)

def ataque_por_mdc(gamma, eta, rho, list_samples):
    x0 = list_samples[0]
    mult_p = prod([x0 - r for r in range(-2^rho, 2^rho)])
    mult_p = remove_fatores_pequenos(mult_p) # mult. de p

    for i in range(1, len(list_samples)):
        xi = list_samples[i]
        mi = prod([xi - r for r in range(-2^rho, 2^rho)])
        mult_p = gcd(mult_p, mi) # máximo divisor comum
        print("bitlen mult_p = %d" % mult_p.nbits())
        if eta >= log(mult_p, 2) >= eta-1:
            break

    return mult_p
```

Os ataques por reticulados [CS15,GGM16] têm complexidade $2^{\Omega(\gamma/(\eta-\rho)^2)}$, então basta usar $\gamma = \Omega(\lambda\cdot(\eta-\rho)^2)$ para que o custo seja de ao menos 2^λ operações. Portanto,

considerando todos os ataques, temos $\rho = \Omega(\lambda)$ e $\gamma = \Omega(\lambda \cdot (\eta - \lambda)^2)$, sendo η o único parâmetro livre. Obviamente, ele deve ser maior que λ, senão uma busca exaustiva poderia encontrar p em menos de 2^λ passos, mas quão maior que λ? Isso é definido pelas propriedades dos esquemas criptográficos baseados no ACD, como veremos na Seção 1.3.

1.3. DGHV: Uma cifra homomórfica simples baseada no problema ACD

1.3.1. Definição da cifra e de suas propriedades

Primeiramente vamos apresentar um esquema simétrico. Depois disso descreveremos os detalhes de implementação diretamente com trechos da implementação em Sage. Finalmente, mostraremos como o esquema simétrico pode ser transformado em um esquema assimétrico.

Antes de mais nada precisamos definir uma notação para representar a *redução modular centralizada*, conforme segue.

Definição 1.3.1. *Definimos pela notação* $[x]_n$ *o único inteiro* $-n/2 \leq x' < n/2$ *tal que* $x' \equiv x \pmod{n}$.

Essa notação também pode ser aplicada a polinômios, reduzindo cada coeficiente, e a vetores e matrizes, reduzindo cada entrada. A seguir, mostramos o um script Sage que implementa essas operações e que será nomeado utils.sage, para que possa ser importado em outros programas que serão apresentados nas seções seguintes:

```
def sym_mod(a, n):
    a = ZZ(a) % n
    if 2*a > n:
        return a - n
    return a

def sym_mod_poly(poly, q):
    return Zx([sym_mod(ZZ(ai), q) for ai in poly.list()])

def sym_mod_vec(vec, q):
    return [sym_mod_poly(vi, q) for vi in vec]
```

Algoritmo 1.1. utils.sage

Agora podemos descrever a cifra DGHV em detalhes.

Definição 1.3.2. *A seguir as funções do criptossistema* DGHV *são definidas.*

- $\mathsf{KeyGen}(1^\lambda)$*: recebe o parâmetro de segurança* λ *e gera os valores* γ, η, ρ *como descrito na seção anterior. A chave secreta* sk *é escolhida como sendo um primo* p *de* η *bits.*

- $\mathsf{Enc}(\mathsf{sk}, m)$*: gera um criptograma* c *que cifra* m *sob a chave* $\mathsf{sk} = p$*. Para isso, amostra-se* $c' \leftarrow \mathscr{D}_{\gamma,\rho}(p)$ *até que* $[c']_p$ *seja par e devolve-se* $c = c' + m$*. Note que* c *é da forma* $pq + 2r + m$*.*

- $\mathsf{Dec}(\mathsf{sk}, c)$: *recupera a mensagem m cifrada por c. Para isso, calcula-se* $m = [c]_p \pmod 2$.

Agora vamos mostrar o código Sage que implementa o esquema DGHV. Antes disso, alguns comentários são importantes. Note que a definição 1.3.2 não determina qual é o espaço de texto claro, ou seja, não especifica o domínio da mensagem m. Para que o crescimento do ruído seja minimizado, devemos escolher um espaço de texto claro de tamanho mínimo. Em geral, podemos considerar que $m \in \mathbb{Z}_t$, com t pequeno. Aqui, utilizamos $t = 2$, de modo que $m \in \{0,1\}$. Além disso, é possível perceber uma diferença em relação a distribuição $\mathscr{D}_{\gamma,\rho}(p)$, pois ao cifrar uma mensagem temos que o ruído r é multiplicado por t. Isto é necessário para que seja possível decifrar a mensagem posteriormente utilizando uma redução modular por t, eliminando o ruído.

Outro comentário importante é sobre a variável x_0, calculada como sendo um múltiplo exato de p. Para que seja computacionalmente difícil calcular p dado x_0, é necessário que a fatoração de x_0 seja difícil o suficiente, o que pode ser obtido escolhendo η e γ suficientemente grande. Nós utilizaremos x_0 para limitar o tamanho do texto cifrado, já que o espaço de texto cifrado é dado por $\mathbb{Z}_{x_0}$.

```
load("utils.sage")
load("distribution_acd.sage")

class DGHV:
    def __init__(self, gamma, eta, rho, t = 2, p = 1):
        assert(gamma > eta)
        assert(eta > rho)
        if 1 == p:
            p = random_prime(2^eta, lbound=2^(eta - 1))
        else:
            # if p is given, it must have eta bits
            assert(eta-1 <= p.nbits() <= eta)

        self.gamma = gamma
        self.eta = eta
        self.rho = rho
        self.t = t
        self.p = p
        self.x0 = p * sample_q(gamma, eta) #+ self.sample_r()
        self.Zp = ZZ.quotient(p)
        self.Zx0 = ZZ.quotient(self.x0)

    def enc(self, m):
        q = sample_q(self.gamma, self.eta)
        r = sample_r(self.rho)
        c = self.p*q + self.t * r + m
        c %= self.x0
        return c
```

```
    def dec(self, c):
        noisy_msg = sym_mod(c, self.p) # == t * r + msg
        return noisy_msg % self.t
```

Agora podemos examinar como são realizadas as operações homomórficas. Como o x espaço de texto encriptado é $\mathbb{Z}_0$, então as operações homomórficas são realizadas neste anel, e portanto são de fácil implementação em Sage.

```
    def not_gate(self, c):
        return (1 - c) % self.x0

    def add(self, c1, c2):
        return (c1 + c2) % self.x0

    def mult(self, c1, c2):
        return c1 * c2 %self.x0
```

1.3.2. Análise da evolução do ruído devida às operações homomórficas

Como a multiplicação homomórfica aumenta o ruído significativamente, enquanto a adição praticamente não muda o ruído, a quantidade de produtos efetuados para gerar um criptograma é comumente usada como estimativa do ruído. Nosso objetivo agora é implementar uma comparação homomórfica de inteiros com ℓ bits e verificar como o ruído aumenta conforme as operações são efetuadas.

Primeiro, note que dados bits x e y, a função $c(x,y) = (x+y)+1 \bmod 2$ é igual a 1 se os dois bits são iguais e a 0 se são diferentes – $c(x,y)$ é equivalente a $\neg(x \text{ xor } y)$. Portanto, dados inteiros a e b com n bits, podemos testar se eles são iguais comparando os bits correspondentes e multiplicando o resultado, i.e., $\text{comp}(a,b) = \prod_{i=1}^{n} c(a_i,b_i)$, onde a_i representa o i-ésimo bit de a (e analogamente para b_i). O código a seguir implementa essa comparação. Para executá-lo, é preciso escolher uma quantidade de bits L e inicializar um objeto do tipo DGHV com parâmetros como $(\gamma,\eta,\rho) = (\lambda^2,(L+1)\cdot\lambda,\lambda)$, então invocar a função passando o objeto e n=L.

```
def comparacao_homomorfica(dghv, n=3):
    m0 = ZZ.random_element(0, 2^n)
    m1 = ZZ.random_element(0, 2^n)
    bits0 = m0.digits(base=2, padto=n) # lista com n bits
    bits1 = m1.digits(base=2, padto=n) # lista com n bits
    c0 = [dghv.enc(bi) for bi in bits0] # cifra cada bit
    c1 = [dghv.enc(bi) for bi in bits1] # cifra cada bit

    # compara homomorficamente
    c, m = 1, 1
    for i in range(n):
        cmp_i = dghv.add(c0[i], c1[i]) # enc(0) <==> c0[i] == c1[i]
        cmp_i = dghv.not_gate(cmp_i) # enc(1) <==> c0[i] == c1[i]
```

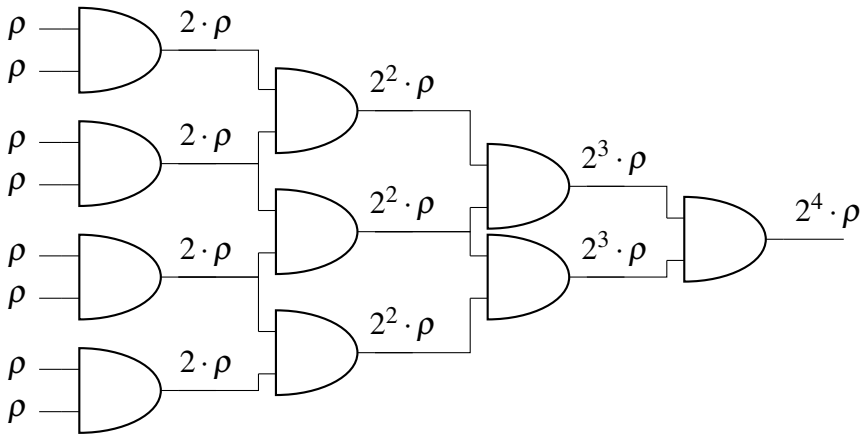

Figure 1.2. Circuito composto apenas de portas lógicas e (multiplicações). Os valores $k\rho$ indicam o logaritmo dos ruídos. A profundidade do circuito é 4, os criptogramas da entrada têm ruído 2^{ρ} e a magnitude do ruído da saída é $2^{2^4 \cdot \rho}$.

```
   c = dghv.mult(c, cmp_i) # c *= cmp_i
# decifra e verifica
res = dghv.dec(c)
assert((m0 == m1) == res)
```

Algoritmo 1.2. Comparação de inteiros homomórfica.

Para estimar o ruído durante a execução, a cada criptograma é associado um nível. Criptogramas que não passaram por nenhuma operação homomórfica são considerados como estando no "nível 1" e o ruído tem magnitude próxima de 2^{ρ}. Ao multiplicar dois criptogramas cujos níveis são n_1 e n_2 e ruídos são aproximadamente $2^{n_1\rho}$ e $2^{n_2\rho}$, obtém-se um criptograma no nível n_1+n_2 com ruído próximo de $2^{(n_1+n_2)\rho}$. Portanto, a comparação homomórfica que implementamos gera um criptograma com ruído aproximadamente $2^{n\rho}$. Mas note que no pior caso a profundidade multiplicativa do circuito corresponde ao logaritmo do nível do criptograma, como ilustrado na Figura 1.2.

Como o deciframento só funciona enquanto o ruído for menor do que $p/2 \approx 2^{\eta}$, a profundidade L dos circuitos que podem ser executados homomorficamente é limitada por $2^{2^L\rho} < 2^{\eta}$, ou seja, $L = O(\log(\eta/\rho))$. Essa é uma enorme limitação do esquema DGHV, pois (sem *bootstrapping*) ele não permite mesmo que circuitos "rasos", com profundidade λ, sejam avaliados homomorficamente, já que para isso seria necessário usar $\eta = \Omega(2^L \cdot \rho) = \Omega(2^{\lambda} \cdot \lambda)$, ou seja, exponencial em λ, logo, como $\gamma > \eta$ e todos os criptogramas tem γ bits, mesmo cifrar uma mensagem levaria tempo exponencial. Por isso, é preciso se limitar a $L = O(\log \lambda)$, para que η e γ sejam apenas polinomiais em λ.

1.3.3. *Boostrapping*: transformando uma cifra homomórfica em uma cifra completamente homomórfica

Como vimos anteriormente, há uma quantidade limitada de operações homomórficas que são permitidas, pois quando o ruído ultrapassa um determinado limiar, então não é mais possível decifrar as mensagens. Sendo assim, gostaríamos de ter um mecanismo que reduzisse o ruído para seu nível inicial, isto é, aquele que é gerado ao cifrar a mensagem,

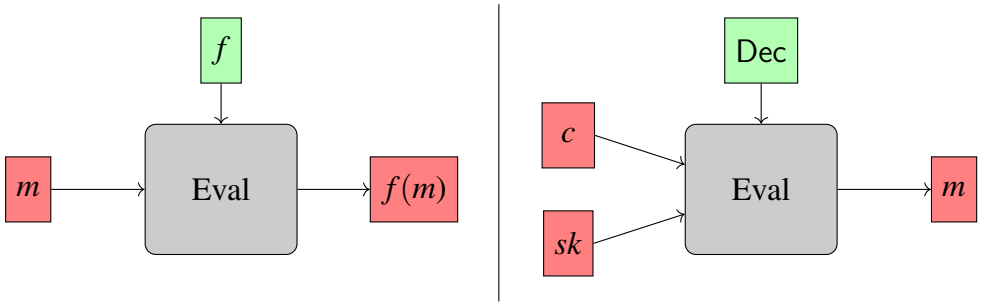

Figure 1.3. Como a avaliação homomórfica funciona em geral e como ela é usada durante o *bootstrapping*. Caixas vermelhas representam valores cifrados e verdes representam valores públicos.

ou seja, um ruído com ρ bits. Intuitivamente, gostaríamos de renovar o texto cifrado sempre que o ruído estivesse próximo de ultrapassar o limiar. Uma forma indesejada de resolver tal problema é simplemente decifrando e cifrando novamente, já que decifrar remove completamente o ruído e cifrar novamente gera um novo criptograma com pouco ruído. No entanto, decifrar exige conhecimento da chave privada. A principal observação de Gentry em [Gen09] foi que podem-se usar as operações homomórficas da cifra para avaliar sua própria função de decifração. Como avaliar homomomorficamente f em um texto cifrado $\mathsf{Enc}(m)$ gera $\mathsf{Enc}(f(m))$, é possível avaliar $f = \mathsf{Dec}$ em $\mathsf{Enc}(c)$ e $\mathsf{Enc}(sk)$ para obter $\mathsf{Enc}(f(c,\mathsf{sk})) = \mathsf{Enc}(\mathsf{Dec}_{\mathsf{sk}}(c)) = \mathsf{Enc}(m)$, ou seja, uma nova encriptação de m. Todo o ruído de c é removido pela decifração e o ruído contido no novo criptograma consiste apenas no ruído acumulado pelas operações homomórficas necessárias para computar o circuito Dec. Esse operação, chamada de *bootstrapping*, é ilustrada na figura 1.3.3. Note que a chave secreta nunca é revelada, pois apenas $\mathsf{Enc}(\mathsf{sk})$ é usado. No entanto, isso introduz uma nova hipótese de segurança, a saber, a segurança circular: é preciso supor que é seguro cifrar sk sob a própria chave sk. É importante notar, contudo, que essa hipótese é vista como fraca, já que se acredita que a maioria das cifras é circularmente segura.

1.3.4. Aplicando o *bootstrapping* à cifra DGHV

O esquema descrito na Definição 1.3.2, porém, não é capaz de avaliar homomomorficamente seu próprio circuito de deciframento, em resumo porque reduções modulares são caras. Deste modo, mostraremos modificações sobre a proposta apresentada na seção anterior que tornam possíveis a implementação do *bootstrapping*. Como precisamos ser capazes de avaliar o circuito de deciframento homomorficamente, temos que, informalmente, facilitar o processo de deciframento, sem que isso seja um problema de segurança, pois esta facilidade não pode estar disponível para adversários. Porém, o algoritmo de deciframento descrito anteriormente necessita de operações cujo circuito é muito caro em termos de profundidade multiplicativa, pois precisa calcular reduções modulares. Sendo assim, veremos nesta seção como é possível melhorar este aspecto. Inicialmente, consideremos a seguinte adaptação, que permite construir um esquema assimétrico. Note que esta etapa não é estritamente necessária, mas será apresentada a título de completude.

Definição 1.3.3. - $\mathsf{KeyGen}(\lambda)$*: obtenha o inteiro aleatório p com η bits. Para* $0 \leq$

$i \leq \tau$, calcule $x_i = \mathscr{D}_{\gamma,\rho}(p)$. Renomeie os índices de modo que x_0 corresponda ao maior elemento. Repita até que x_0 seja ímpar e $x_0 \pmod p$ seja par. A chave pública é dada por $\mathsf{pk} = (x_0, \ldots, x_\tau)$ *e a chave privada é dada por* $\mathsf{sk} = p$.

- $\mathsf{Enc}(m)$*: escolha aleatoriamente o conjunto $S \subset \{0, 1, \ldots, \tau\}$ e o inteiro aleatório r no intervalo $(-2^{\rho'}, 2^{\rho'})$ e compute $c = [m + 2r + \sum_{i \in S} x_i]_{x_0}$.*

- $\mathsf{Dec}(c)$*: retorne $m = [c]_p \pmod 2$.*

- $\mathsf{Eval}(C, c_1, \ldots, c_t)$*: dado o circuito C e t textos cifrados, execute as operações de C módulo x_0 sobre os textos cifrados, dados por números de precisão arbitrária, e retorne o resultado.*

Agora podemos mostrar mais uma adaptação, que faz com que o algoritmo de deciframento seja eficiente o suficiente para permitir o *bootstrapping*. Em particular, note que o deciframento precisa de um nível de multiplicações por z_i e após isso pode ser concluído apenas por somas e subtrações.

Definição 1.3.4. *Esta construção usa 3 novos parêmetros: $\kappa = \gamma\eta/\rho'$, $\theta = \lambda$ e $\Theta = \omega(\kappa \log \lambda)$, ou seja, eles são todos polinomiais em relação ao parâmetro de segurança λ.*

- $\mathsf{KeyGen}(\lambda)$*: compute* sk *e* pk *como na definição 1.3.3. Compute $x_p = \lfloor 2^\kappa / p \rceil$, escolha aleatoriamente o vetor $s = \langle s_1, \ldots, s_\Theta \rangle$, com Θ bits e* peso de Hamming θ*. O conjunto S é definido por*

$$S = \{i \mid s_i = 1\}.$$

Escolha aleatoriamente os inteiros u_i, onde $1 \leq i \leq \Theta$, com no máximo κ bits, tal que $\sum_{i \in S} u_i = x_p \pmod{2^{\kappa+1}}$. Compute $y_i = u_i / 2^\kappa$, tal que cada y_i é um inteiro positivo menor ou igual a 2, com κ bits de precisão após a parte fracionária. Com isso temos que $[\sum_{i \in S} y_i]_2 = (1/p) - \Delta_p$, para $\Delta_p < 2^{-\kappa}$.

A chave privada é dada pelo vetor $(s_1, \ldots, s_\kappa)$ e a chave pública é dada por pk *e o vetor $(y_1, \ldots, y_\Theta)$.*

- $\mathsf{Enc}(m)$*: compute c como no esquema original. para $1 \leq i \leq \Theta$, compute $z_i = [c y_i]_2$, mantendo apenas $\lceil \log \theta \rceil + 3$ de precisão para cada z_i. Returne c e o vetor $(z_1, \ldots, z_\Theta)$.*

- $\mathsf{Dec}(c)$*: returne $m' = [c - \lfloor \sum_{i \in S} s_i z_i \rceil]_2$.*

- $\mathsf{Eval}(C, c_1, \ldots, c_t)$*: Somas e multiplicações são calculadas pelas operações usuais sobre os racionais.*

Os detalhes deste processo podem ser encontrados no minicurso [DM12] de 2012. Aqui, o objetivo maior é adquirir a intuição de como o *bootstrapping* funciona. Em resumo, o que há por trás desta modificação é que parte do trabalho de deciframento foi transferida para o algoritmo de encriptação, que passou a ser responsável por encontrar

valores z_i peculiares, tais que o deciframento possa ser realizado basicamente pela soma destes elementos. Porém, para que tal esquema ainda seja seguro, tais elementos devem estar escondidos em meio a um conjunto maior, de modo que um adversário não seja capaz de advinhar quais elementos deste conjunto devem um não ser usados no deciframento. Este é um problema clássico em computação, e é conhecido como SSP (*subset sum problem*). Os parâmetros deste esquema modificado devem então proteger contra adversários que tentem resolver o problema SSP subjacente, além dos problemas já mencionados de força bruta no ruído e o ACD. Em resumo, para obter segurança, precisamos escolher θ suficientemente grande para que o problema ofereça grau de segurança λ. Também é preciso ter Θ pertencente a $\omega(\kappa \log \lambda)$, onde κ é o tamanho em bits dos números que formam a chave pública.

1.3.5. Outras direções

Existem alguns trabalhos que podemos apontar para aqueles que tiverem mais interesse no assunto. Por exemplo:

- é possível operar sobre vetores de textos claros, onde a operação de soma e multiplicação é realizada coordenada a coordenada, oferecendo portanto paralelismo ao sistema. Para isso, podemos usar o teorema Chinês dos restos, como neste artigo [CKLY15]. Resumidamente, o espaço de texto claro ao invés de $\mathbb{Z}_t$ passa a ser $\mathbb{Z}_{t_1} \times \ldots \mathbb{Z}_{t_\ell}$, com t_i primos entre si. Em particular, o esquema pode ser atraente quando ℓ é grande. O texto claro também pode ser interpretado como um único inteiro módulo $\prod_{i=0}^{\ell}(t_i)$;

- um resultado relevante é o trabalho que mostra como implementar o *bootstrapping* em menos de um segundo [Per21a], tornando possível implementar FHE sobre os números inteiros de modo mais eficiente;

- neste outro trabalho importante da literatura [CS15] temos estabelecida uma relação entre o problema ACD e o problema LWE, que será o foco da próxima seção.

1.4. Os problemas criptográficos LWE e RLWE

O problema LWE (aprendizagem com ruídos, ou aprendizagem com erros, do inglês, *Learning With Errors*) [Reg09] se tornou uma ferramenta fundamental para a criptografia moderna, pois é um problema considerado difícil mesmo para computadores quânticos, e é versátil o suficiente para que diversas primitivas criptográficas possam ser baseadas nele. Por exemplo, no processo de padronização de primitivas pós-quânticas organizado pelo NIST[5], há cifras de chave pública, esquemas de encapsulamento de chave e de assinatura digital baseados no LWE (ou variantes do LWE).

Talvez sua característica mais singular seja a garantia de que mesmo instâncias aleatórias do problema são difíceis. Isso não é verdade para outros problemas criptográficos usados frequentemente. Por exemplo, considere a fatoração de inteiros da forma $n = pq$ usada no RSA. É sabido que não se pode escolher p e q como dois números

[5]https://csrc.nist.gov/Projects/Post-Quantum-Cryptography

primos quaisquer, pois é muito mais fácil fatorar n quando seus dois fatores satisfazem certas propriedades (e.g., se $|p-q| < \sqrt{n}$, então o simples método de fatoração de Fermat é suficiente para recuperar p e q).

O problema LWE consiste no seguinte: considere um vetor secreto $\mathbf{s} \in \mathbb{Z}^n$ e m equações lineares $b_i = \mathbf{a}_i \cdot \mathbf{s} \in \mathbb{Z}_q$, onde $\mathbf{s}$ é o mesmo em todas as equações. Convenientemente, dadas $m = n$ equações, podemos construir uma matriz $\mathbf{A} \in \mathbb{Z}_q^{n \times n}$ em que cada linha corresponde a um vetor $\mathbf{a}_i$. Da mesma forma, podemos construir um vetor $\mathbf{b} = (b_1, ..., b_n)^T$. então, como sabemos que $\mathbf{b} = \mathbf{A} \cdot \mathbf{s} \pmod q$, basta calcular $\mathbf{A}^{-1} \cdot \mathbf{b} \bmod q$ para encontrar $\mathbf{s}$. No entanto, se em vez de equações, soubéssemos apenas que $b_i \approx \mathbf{a}_i \cdot \mathbf{s} \pmod q$, ou seja, que $b_i = \mathbf{a}_i \cdot \mathbf{s} + e_i \pmod q$, onde e_i é algum inteiro pequeno, como poderíamos encontrar $\mathbf{s}$? Adicionar esses pequenos "erros", ou ruídos, às equações faz com que seja extremamente difícil descobrir o valor de $\mathbf{s}$. Mais formalmente, os termos e_i são ruídos Gaussianos discretos, ou seja, seguem uma distribuição análoga à normal, mas sobre $\mathbb{Z}$ em vez de $\mathbb{R}$. Denotamos essa Gaussiana discreta por χ_σ, ou apenas χ quando σ estiver claro no contexto, e cada inteiro x é amostrado de χ com probabilidade proporcional a $e^{-x^2/(2\sigma^2)}$.

Definição 1.4.1. *Considere $n, q \in \mathbb{N}^*$ e $\sigma \in \mathbb{R}$ tal que $\sigma > 0$. Seja $\mathbf{s} \in \mathbb{Z}_q^n$ um vetor fixo. A distribuição $\mathscr{A}_{\mathbf{s},\sigma}$ é definida como*

- *amostre $\mathbf{a}$ uniformemente de $\mathbb{Z}_q^n$ e amostre e seguindo a distribuição χ_σ;*
- *calcule $b := \mathbf{a} \cdot \mathbf{s} + e \bmod q$;*
- *devolva $(\mathbf{a}, b)$.*

Definição 1.4.2 (LWE)**.** *O problema* LWE *com parâmetros n, q e σ, ou simplesmente $(n, q, \sigma) -$* LWE*, é o problema de encontrar $\mathbf{s}$ dada uma amostra arbitrariamente grande da distribuição $\mathscr{A}_{\mathbf{s},q,\sigma}$.*

O problema LWE *de decisão consiste em distinguir entre as distribuições $\mathscr{U}(\mathbb{Z}_q^n \times \mathbb{Z}_q)$ e $\mathscr{A}_{\mathbf{s},q,\sigma}$.*

É sabido que as duas versões desse problema são computacionalmente equivalentes, i.e., se existe um algoritmo A que resolve qualquer uma delas em tempo polinomial, então é possível usar A como uma sub-rotina de um algoritmo A' que resolve a outra versão do problema também em tempo polinomial [Reg09].

Assim como a distribuição $\mathscr{D}_{\gamma,\rho}(p)$ associada com o problema ACD, implementar $\mathscr{A}_{\mathbf{s},q,\sigma}$ em Sage é simples:

```
from sage.stats.distributions.discrete_gaussian_integer \
        import DiscreteGaussianDistributionIntegerSampler \
        as DiscreteGaussian

class LWEDistribution:
    def __init__(self, s, q, sigma=3.2):
        self.n = len(s)
        self.s = s
```

```
        self.sigma = sigma
        self.D = DiscreteGaussian(sigma)
        self.Zq = ZZ.quotient(q)

    def random_noise(self):
        return self.D()

    def random_a(self):
        a = [self.Zq.random_element() for _ in range(self.n)]
        a = vector(self.Zq, a) # converte lista em vetor
        return a

    def sample(self):
        n, s = self.n, self.s
        a = self.random_a() # vetor em Zq^n
        e = self.random_noise()
        b = a*s + e
        return a, b
```

1.4.1. Criptanálise do problema LWE

Os parâmetros n, q, σ e também a distribuição de $\mathbf{s}$ são escolhidos de tal forma que todos os algoritmos que resolvem o LWE levem tempo exponencial no parâmetro de segurança λ. Na verdade, tratar q e σ separadamente é irrelevante para as análises de segurança, pois a dificuldade de resolver o LWE depende de $\alpha := q/\sigma$, i.e., da razão entre q e o tamanho dos ruídos. Para entender intuitivamente isso, note que se $(\mathbf{a}, b := \mathbf{as} + e \pmod q)$ é uma amostra LWE, então, $(k \cdot \mathbf{a}, k \cdot b := k \cdot \mathbf{as} + k \cdot e \pmod{k \cdot q})$ também o é, mas com respeito ao módulo kq. Obviamente, esse novo LWE definido sobre $\mathbb{Z}_{kq}$ não pode ser mais fácil que o original, pois se houvesse um algoritmo eficiente para resolver esse $(n, kq, k\sigma)$-LWE, poderíamos transformar o (n, q, σ)-LWE multiplicando por k e usar esse algoritmo eficiente para recuperar $\mathbf{s}$. Mas note que $kq/(ke) = q/e$, ou seja, podemos aumentar livremente o ruído e o módulo, mantendo a mesma razão ente eles, sem aumentar a dificuldade do problema. Um argumento parecido (mas mais delicado) mostra que também é possível dividir ambos o ruído e o módulo sem alterar a dificuldade do LWE.

Por isso, em vez de trabalhar com a fração q/σ para derivar os parâmetros, é comum simplesmente definir σ como um valor pequeno, como 3.2, e ajustar q para que a fração tenha o valor desejado. Assim, para simplificar a análise, também usaremos essa metodologia.

Como a distribuição normal é concentrada perto da média[6], o valor dos ruídos adicionados nas amostras do LWE são pequenos: com probabilidade praticamente igual a um, cada e_i pertence ao intervalo $[\![-6\sigma, 6\sigma]\!]$, ou seja, há 13σ valores possíveis. Portanto, assim como no caso do problema ACD, é possível tentar eliminar o ruído e obter equações exatas, então resolvê-las para achar $\mathbf{s}$. Para isso, seriam necessárias n amostras de $\mathscr{A}_{\mathbf{s},q,\sigma}$,

[6]Valores que estão mais do que três desvios-padrão de distância da média já se encontram nas caldas da normal e têm probabilidade muito baixa de serem amostrados.

para termos $\mathbf{b} = \mathbf{As} + \mathbf{e} \pmod{q}$ sendo $\mathbf{A}$ uma matriz quadrada, $n \times n$, como descrito anteriormente. Nesse caso, calcularíamos $\mathbf{A}^{-1}$ e então, para cada $\mathbf{e}_i$ possível, definiríamos $\mathbf{b}_i = \mathbf{b} - \mathbf{e}_i$ e $\mathbf{s}_i = \mathbf{A}^{-1}\mathbf{b}_i \bmod q$. Note que quando $\mathbf{e}_i = \mathbf{e}$, o valor correto de $\mathbf{s}$ é recuperado. Como há $(13\sigma)^n$ vetores $\mathbf{e}_i$ possíveis, basta tomar $n = \Omega(\lambda)$ para inviabilizar esse ataque, fazendo sua complexidade temporal ser $2^{\Omega(\lambda)}$.

Considerando ainda que σ é um valor fixo e pequeno, resta saber como escolher q: as restrições sobre q são obtidas considerando ataques por reticulados. Para um n fixo, esses ataques se tornam mais eficientes a medida que q cresce. Basicamente, para um nível de segurança λ, temos a restrição $\log q \in O(n\log(\lambda)/\lambda)$.

A escolha do vetor secreto $\mathbf{s}$ também influencia a segurança do LWE, mas seu impacto é bem pequeno. A forma padrão de usar o LWE é escolhendo $\mathbf{s}$ uniformemente em $\mathbb{Z}_q^n$, mas é possível amostrar cada coordenada s_i seguindo a mesma distribuição que o ruído, i.e., χ, ou até como um bit aleatório, i.e., $s_i \leftarrow \mathscr{U}(\{0,1\})$.

Para obter valores concretos em vez de assintóticos, o estimador *online* apresentado em [APS15] é frequentemente usado.

1.4.2. LWE sobre anéis polinomiais

Em uma amostra do LWE, $(\mathbf{a}, b := \mathbf{as} + e)$, o vetor $\mathbf{a}$ é uniformemente distribuido em $\mathbb{Z}_q^n$. Assumindo que o problema LWE é difícil, obtemos que b também é uniforme, ou mais precisamente, b é computacionalmente indistinguível de um valor uniforme em $\mathbb{Z}_q$. Então, dada uma mensagem m, a soma $b + m \bmod q$ serve como uma cifra de m, pois b age como *one-time pad*. No entanto, para decifrar b, não basta conhecer a chave secreta $\mathbf{s}$, o vetor $\mathbf{a}$ também é necessário, logo, ele deve ser incluído no criptograma. Isso significa que um bit (ou um inteiro pequeno) m gera um criptograma $\mathbf{c} \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, o que representa uma expansão enorme, de $(n+1)\log q$ bits. Se cada criptograma tivesse o mesmo tamanho, mas pudesse encriptar n bits em vez de um, a expansão seria reduzida para $((n+1)\log q)/n = O(\log q)$.

Essa é a principal motivação do problema RLWE (LWE sobre anéis, do inglês *Ring Learning With Errors*). Ele é definido fixando um anel $R = \mathbb{Z}[X]/\langle f(X)\rangle$, onde f é um polinômio de grau n, e substituindo os vetores $\mathbf{a}$ e $\mathbf{s}$ por polinômios $a(X), s(X) \in R$. Note que $R = \{g(X) \in \mathbb{Z}[X] : \mathsf{grau}(g) < n\}$ e todas as operações em R são feitas módulo f. Assim, $a \cdot s \bmod f$ é um polinômio em vez de um inteiro, logo, tem mais espaço para guardar informação, mais especificamente, podemos encriptar um polinômio $m(X) = \sum_{i=0}^{n-1} m_i \cdot X^i$. Para isso, o ruído também é definido como um elemento de R e b é calculado como $(a \cdot s + e \bmod f) \bmod q$. Agora, um criptograma tem $2n\log q$ bits, mas cifra n bits, logo, a expansão é de $(2n\log q)/n = O(\log q)$, como desejado.

Por razões de segurança, escolhemos N tal que $n = \varphi(N)$, onde $\varphi(\cdot)$ é a função phi de Euler[7], e o polinômio f é definido como o N-ésimo polinômio ciclotômico, denotado por $\Phi_N(X)$ e definido como $\Phi_N(X) = \prod_{k \in S_N}(X - \exp(2i\pi k/N))$, onde $S_N := \{x \in \mathbb{N} : x \leq N \text{ e } \mathsf{mdc}(x,N) = 1\}$ e i é a unidade imaginária, i.e., $i^2 = -1$. É fácil ver que o grau de $\Phi_N(X)$ é $|S_N| = \varphi(N)$. Ademais, apesar de ser definido como um polinômio com raízes

[7] $\varphi(N)$ é definida como a cardinalidade do conjunto $\{x \in \mathbb{N} : x \leq N \text{ e } \mathsf{mdc}(x,N) = 1\}$, ou seja, a quantidade de coprimos menores que ou iguais a N.

complexas, todos os seus coeficientes são inteiros e ele é irredutível em $\mathbb{Z}[x]$. Em Sage, há um comando específico para obter o N-ésimo polinômio ciclotômico:

```
print(cyclotomic_polynomial(4)) # x^2 + 1
print(cyclotomic_polynomial(5)) # x^4 + x^3 + x^2 + x + 1
print(cyclotomic_polynomial(6)) # x^2 - x + 1
print(cyclotomic_polynomial(16))# x^8 + 1
print(cyclotomic_polynomial(20))# x^8 - x^6 + x^4 - x^2 + 1
```

Como ilustrado acima, em geral, é difícil prever os coeficientes de $\Phi_N(X)$, mas quando N é uma potência de 2, temos $n := \varphi(N) = N/2$ e $\Phi_N(X)$ é simplesmente $X^n + 1$. Além disso, essa escolha de N facilita a implementação dos esquemas e os torna mais eficientes, já que a reduzir um polinômio módulo $X^n + 1$ requer apenas substituir X^n por -1. Por exemplo, $X^{2n} + 3X^{n+2} + 4 \equiv (-1)^2 + 3 \cdot (-1) \cdot X^2 + 4 \equiv -3X^2 + 5 \pmod{X^n + 1}$. Ademais, para multiplicar polinômios módulo $X^n + 1$, pode-se usar algoritmos de transformadas rápidas de Fourier mais simples do que para o caso geral, módulo $\Phi_N(X)$. Por isso, é comum considerar apenas potências de dois ao se construir cifras baseadas no RLWE e também faremos isso ao definirmos o esquema GSW na seção 1.5.2.

Além do anel R, também é preciso definir $R_q := R/qR = \mathbb{Z}_q[X]/\langle \Phi_N(X) \rangle$, i.e., anel de polinômios com grau menor que $n := \varphi(N)$ e coeficientes em $\mathbb{Z}_q$. Todas as operações nesse anel são feitas módulo $\Phi_N(X)$ e módulo q. Considerando o que foi discutido até então, o problema RLWE é definido como a seguir:

Definição 1.4.3 (Distribuição adjacente do RLWE)**.** *Considere $N, q \in \mathbb{N}^*$ e $\sigma \in \mathbb{R}$ tal que $\sigma > 0$. Sejam $n := \varphi(N)$, $R := \mathbb{Z}[X]/\langle \Phi_N(X) \rangle$ e $R_q := R/qR$. Finalmente, seja $s \in R_q$ um polinômio fixo. A distribuição $\mathscr{R}_{s,n,q,\sigma}$ é definida como*

- *amostre a uniformemente de R_q;*
- *para $0 \leq i < n$, amostre $e_i \leftarrow \chi_\sigma$ e defina $e = \sum_{i=0}^{n-1} e_i X^i \in R$;*
- *calcule $b := a \cdot s + e$ em R_q;*
- *devolva $(a, b) \in R_q^2$.*

Definição 1.4.4 (RLWE)**.** *O problema* RLWE *com parâmetros N, q e σ, ou simplesmente $(N, q, \sigma) - \mathsf{LWE}$, é o problema de encontrar s dada uma amostra arbitrariamente grande da distribuição $\mathscr{R}_{s,n,q,\sigma}$. O problema* RLWE *decisional é o problema de distinguir entre as distribuições $\mathscr{U}(R_q \times R_q)$ e $\mathscr{R}_{s,n,q,\sigma}$.*

O código a seguir mostra como implementar a distribuição $\mathscr{R}_{s,n,q,\sigma}$ em Sage. É notável a semelhança com o código que implementa $\mathscr{A}_{\mathbf{s},q,\sigma}$.

```
from sage.stats.distributions.discrete_gaussian_integer \
        import DiscreteGaussianDistributionIntegerSampler \
        as DiscreteGaussian

Zx.<x> = ZZ['x']
```

```
class RLWEDistribution:
    def __init__(self, s, N, q, sigma=3.2):
        self.n = N
        self.f = x^N + 1 # assume N = 2^k
        self.s = s
        self.sigma = sigma
        self.D = DiscreteGaussian(sigma)
        self.Zqx = ZZ.quotient(q)['x']
        self.Rq = self.Zqx.quotient(self.f)


    def random_noise(self):
        # polinômio com grau <= n-1 e coeficientes gaussianos
        return Zx([self.D() for _ in range(self.n)])

    def random_a(self):
        return self.Rq.random_element()

    def sample(self):
        s = self.s
        a = self.random_a() # coeficientes em Zq
        e = self.random_noise()
        b = (a*s + e) # mod f e q calculado automaticamente
        return [a, b]
```

1.5. Diminuindo a taxa de crescimento do ruído

A velocidade com que o ruído cresce ao se efetuar operações homomórficas é um dos fatores mais importantes para determinar a eficiência da cifra, pois se o ruído acumulado por cada operação é grande, são necessários parâmetros grandes, o que implica em criptogramas longos e operações mais lentas. Além disso, a classe de circuitos que podem ser avaliados homomorficamente sem *bootstrapping* também diminui conforme a taxa de crescimento do ruído aumenta. Por exemplo, na Seção 1.3.2, vimos que o ruído do DGHV cresce de forma exponencial, portanto, para avaliar um circuito de profundidade L, é preciso escolher parâmetros maiores que 2^L, e isso dificulta inclusive a execução do *bootstrapping*. Por isso, diversos artigos apresentaram ideias para diminuir o crescimento do ruído [BGV12, FV12, GSW13, CS15].

Enquanto as adições homomórficas praticamente não aumentam o ruído, as multiplicações têm um efeito desastroso. Ao se analisar o porquê disso, observa-se que ao multiplicarmos dois criptogramas, os ruídos são multiplicados por valores cuja magnitude é enorme. Por exemplo, no DGHV, $c_1 \cdot c_2$ resulta em $c_1 \cdot 2r_2$ e, finalmente, em $4r_1 \cdot r_2$. Então, uma forma de reduzir o crescimento do ruído é fazendo com que os ruídos sejam multiplicados apenas por valores pequenos. Mas como fazer isso? A ideia é a seguinte: em vez de multiplicar c_1 e c_2 diretamente, primeiro aplica-se uma decomposição a um dos

operandos, então calcula-se o produto entre essa decomposição e o outro criptograma, assim, em vez de $\mathsf{err}(c_1) \cdot \mathsf{err}(c_2)$, como no DGHV, obtém-se $\mathsf{decomp}(c_1) \cdot \mathsf{err}(c_2)$. Como a decomposição é constituída de valores pequenos, o ruído não aumenta muito.

Nas próximas seções, essa técnica será explicada em detalhes e ela será usada para construir a cifra conhecida como GSW.

1.5.1. Decomposição dos criptogramas

Considere o módulo q usado no (R)LWE. Fixe uma base B, defina $\ell := \lceil \log_B q \rceil$ e $\mathbf{g} = (B^0, B^1, ..., B^{\ell-1})$. Represente $a \in \mathbb{Z}_q$ por um inteiro entre $-q/2$ e $q/2$ e defina $g^{-1}(a)$ como o vetor $(a_0, ..., a_{\ell-1}) \in [\![-B, B]\!]^\ell$ representando a decomposição de $|a|$ na base B, mas multiplicada pelo sinal de a e com o bit ou a palavra menos significa à esquerda. Por exemplo, se $q = 2^7$ e $B = 4$, então $\ell = 4$. Portanto, $g^{-1}(5) = (1,1,0,0)$, $g^{-1}(-24) = (0,-2,-1,0)$ e $g^{-1}(-64) = (0,0,0,-1)$. O código a seguir implementa g^{-1}:

```
def inv_g_ZZ(a, B, q):
    a = sym_mod(ZZ(a), q) # definida em utils.sage
    l = ceil(log(q, B))
    return vector(a.digits(base=B, padto=l))
```

Note que multiplicar $g^{-1}(a)$ por $\mathbf{g}$ tem o efeito de combinar as palavras com as respectivas potências, o que resulta novamente em a, i.e., $g^{-1}(a) \cdot \mathbf{g} = \sum_{i=0}^{\ell-1} a_i B^i = a$. Por isso a notação g^{-1} é comumente usada.

Para decompor um polinômio $a(X)$, basta decompor cada coeficiente a_i, multiplicar por X^i, obtendo um vetor $(a_{i,0}X^i, ..., a_{i,\ell-1}X^i)$, e somar todos os vetores. O resultado é um vetor com ℓ polinômios cujos coeficientes pertencem a $[\![-B, B]\!]$. Como B é em geral pequeno, a norma do vetor resultante também o é. Mais explicitamente, abusamos da notação e definimos $g^{-1}(a) := \sum_{i=0}^{N-1} g^{-1}(a_i)X^i$ para $a \in R$, onde $g^{-1}(a_i)$ é a decomposição de inteiros. Note que $g^{-1}(a) \cdot \mathbf{g} = \sum_{i=0}^{N-1} g^{-1}(a_i) \cdot \mathbf{g} \cdot X^i = \sum_{i=0}^{N-1} a_i \cdot X^i = a$, logo, assim como para os inteiros, ao multiplicar a decomposição por $\mathbf{g}$, obtemos novamente a.

```
def inv_g_poly(a, B, q, n):
    l = ceil(log(q, B))
    result = vector(Zx, [0] * l)
    pow_x = 1
    for i in range(n):
        result += pow_x * inv_g_ZZ(a[i], B, q)
        pow_x *= x
    return result
```

Finalmente, estendemos essa decomposição para um vetor de polinômios simplesmente aplicando g^{-1} a cada coordenada do vetor e concatenando as decomposições, i.e., fixando-se uma dimensão d, para qualquer $\mathbf{c} \in R_q^d$, definimos

$$G^{-1}(\mathbf{c}) := (g^{-1}(c_0), ..., g^{-1}(c_{d-1})) \in R_q^{d\ell}.$$

Como anteriormente, queremos obter novamente $\mathbf{c}$ a partir de $G^{-1}(\mathbf{c})$. Como agora temos um vetor de dimensão $d\ell$, não podemos multiplicar por $\mathbf{g}$, cuja dimensão é ℓ.

Então, definimos a matriz $\mathbf{G} := \mathbf{I}_d \otimes \mathbf{g}^T \in \mathbb{Z}^{d\ell \times d}$, i.e., o produto tensorial entre a identidade $d \times d$ e o vetor-coluna $\mathbf{g}$ transposto. Note que esse produto é calculado substituindo cada entrade $a_{i,j}$ de $\mathbf{I}_d$ pelo vetor $a_{i,j} \cdot \mathbf{g}^T$, assim, temos que

$$\mathbf{G} = \begin{pmatrix} \mathbf{g} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{g} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{g} \end{pmatrix}.$$

Por exemplo, se $d = 2$, $q = 2^3$ e $B = 2$, então $\ell = 3$ e

$$\mathbf{G} = \begin{pmatrix} 1 & 0 \\ 2 & 0 \\ 4 & 0 \\ 0 & 1 \\ 0 & 2 \\ 0 & 4 \end{pmatrix}.$$

Observe o que ocorre ao multiplicar $G^{-1}(\mathbf{c})$ por $\mathbf{G}$: considerando a primeira coluna de $\mathbf{G}$, as ℓ primeiras componentes de $G^{-1}(\mathbf{c})$ são multiplicadas por $\mathbf{g}$ e as outras $(d-1)\ell$ componentes são multiplicadas por zero, logo, o resultado é $g^{-1}(c_0)\mathbf{g} = c_0$. Ao multiplicar $G^{-1}(\mathbf{c})$ pela segunda coluna de $\mathbf{G}$, obtém-se $g^{-1}(c_1)\mathbf{g} = c_1$, e assim sucessivamente. Ou seja, $G^{-1}(\mathbf{c})\mathbf{G} = (g^{-1}(c_0)\mathbf{g}, ..., g^{-1}(c_{\ell-1})\mathbf{g}) = \mathbf{c}$, como desejado.

Essa matriz $\mathbf{G}$ é conhecida como *gadget matrix* e a ideia principal do esquema homomórfico GSW é incluí-la nos criptogramas para que seja possível aplicar G^{-1} durante a multiplicação homomórfica, reduzindo assim o ruído inserido por cada produto.

1.5.2. GSW: uma cifra cuja evolução do ruído é apenas linear

Nesta seção, a cifra GSW [GSW13] será apresentado, mas usando basicamente o formato sugerido [DM15], portanto, baseado no problema RLWE em vez do LWE. Para evitar confusão com os parâmetros do problema LWE usado na cifra que será introduzida na seção 1.7.1, a chave secreta será denotada por z em vez de s e usaremos N em vez de n, ou seja, doravante, definimos $R := \mathbb{Z}[X]/\langle X^N + 1\rangle$, onde N é uma potência de 2.

O GSW pode ser vista como um compromisso entre a memória e o crescimento do ruído. Mais especificamente, cada criptograma é definido com $O(\log q)$ amostras RLWE em vez de apenas uma, portanto, usa-se mais memória, no entanto, o crescimento do ruído devido a cada produto homomórfico é quase linear, ou seja, ao multiplicar criptogramas c_0 e c_1, obtém-se um novo criptograma c tal que $\mathsf{erro}(c) = O(\mathsf{erro}(c_0) + \mathsf{erro}(c_1))$. Isso é possível pois, durante a multiplicação homomórfica, os ruídos dos criptogramas não são multiplicados entre si, ao contrário de outros esquemas, como o DGHV, onde o ruído do criptograma resultante tem um termo $r_1 \cdot r_2$.

As funções de geração de chave, ciframento e deciframento do GSW são mostradas em detalhe a seguir:

- $\mathsf{GSW.ParamGen}(1^\lambda)$: Escolha um valor real $\sigma > 0$ e inteiros N, B e ℓ, sendo N uma potência de dois. Defina $q := B^\ell$. Os parâmetros N, q e σ devem guarantir λ bits de segurança considerando o problema RLWE. Devolva $\mathsf{params} := (N, q, \sigma, B, \ell)$.

- $\mathsf{GSW.KeyGen}(\mathsf{params})$: Defina $z(x) = \sum_{i=0}^{N-1} z_i X^i$ com cada z_i amostrado uniformemente de $\{-1, 0, 1\}$. Devolva $\mathsf{sk} := z$.

- $\mathsf{GSW.Enc}(\mathsf{sk}, m, \mathsf{params})$: Para cifrar um polinômio $m \in R$ cujos coeficientes pertencem a $\mathbb{Z}_B$, amostre $(a_i, b_i) \leftarrow \mathscr{R}_{s,N,q,\sigma}$ para $0 \leq i < \ell$ e defina $\mathbf{C}' \in R_q^{2\ell \times 2}$ com cada linha i igual a (a_i, b_i). Note que $\mathbf{C}'$ é da forma $[\mathbf{a}, \mathbf{b}]$ com $\mathbf{b} = \mathbf{a} \cdot z + \mathbf{e} \pmod q$. Finalmente, devolva $\mathbf{C} = \mathbf{C}' + m \cdot \mathbf{G} \bmod q$.

- $\mathsf{GSW.Dec}(\mathsf{sk}, \mathbf{C}, \mathsf{params})$: Seja $(a, b) \in R_q^2$ a última linha de $\mathbf{C}$. Calcule $u := b - a \cdot \mathsf{sk} \in R_q$, interprete u como um polinômio em $\mathbb{Z}[X]$ e devolva $\lfloor B \cdot u / q \rceil \bmod B$.

Como a matriz $\mathbf{G}$ tem potências de B nas primeiras ℓ entradas da primeira coluna, e elas são multiplicadas por m e somadas às amostras RLWE, os criptogramas têm este formato distinto, em que as primeiras ℓ linhas guardam a mensagem na primeira coluna, ou seja, no "termo a" do RLWE, em vez de ser no "termo b":

$$\mathbf{C} = \begin{pmatrix} a_0 + B^0 \cdot m & a_0 \cdot z + e_0 \\ \vdots & \vdots \\ a_{\ell-1} + B^{\ell-1} \cdot m & a_{\ell-1} \cdot z + e_{\ell-1} \\ a_\ell & a_\ell \cdot z + e_\ell + B^0 \cdot m \\ \vdots & \vdots \\ a_{2\ell-1} & a_{2\ell-1} \cdot z + e_{2\ell-1} + B^{\ell-1} \cdot m \end{pmatrix} \in R_q^{2\ell \times 2}.$$

No deciframento, fica claro como os criptogramas têm muita informação redundante, pois apenas a linha $2\ell - 1$ é necessária para recuperar a mensagem. Essa linha pode ser escrita como $(a, b) \in R_q^2$ com $b = a \cdot z + e + B^{\ell-1} \cdot m$. Mas como $q = B^\ell$, temos $B^{\ell-1} = q/B$ e $u := b - a \cdot \mathsf{sk} = e + m \cdot q/B \pmod q$. Portanto, sobre os inteiros, existe um polinômio v tal que $u = e + m \cdot q/B + v \cdot q \in \mathbb{Z}[x]$. Em seguida, calcula-se

$$w := \left\lfloor \frac{B \cdot u}{q} \right\rceil = \left\lfloor \frac{B \cdot e}{q} + m + vB \right\rceil = \left\lfloor \frac{B \cdot e}{q} \right\rceil + m + vB,$$

onde a última igualdade vale porque todos os coeficientes de m e v são inteiros.

Este é o ponto que define o critério de corretude do deciframento. A saber, como o valor devolvido é $w \bmod B$, é preciso que $\lfloor B \cdot e/q \rceil$ seja zero para que o resultado seja exatamente $m \bmod B$. Mas para isso, é necessário que todos os coeficientes dessa fração sejam menores que meio, i.e., $\|B \cdot e/q\|_\infty < 1/2$, o que é equivalente a $\|e\|_\infty < q/(2B)$. Em outras palavras, se $\|e\|_\infty < q/(2B)$, então $\left\lfloor \frac{B \cdot e}{q} \right\rceil = 0$ e $\left\lfloor \frac{B \cdot u}{q} \right\rceil = m \pmod B$, logo, o deciframento devolve corretamente $m \bmod B$.

Portanto, para garantir a corretude do deciframento, é preciso garantir que o ruído não seja maior que essa fração de q. Perceba a semelhança com o esquema DGHV, apresentado na Seção 1.3, em que o ruído não pode ser maior que uma fração de p.

Antes de apresentar a implementação do GSW, precisamos incluir as seguintes funções no arquivo utils.sage definido na seção 1.3.1:

```
def round_poly(h):
  return Zx([round(hi) for hi in h.list()])

def infinity_norm(poly):
  if 0 == poly:
    return 0
  return vector(ZZ, poly.list()).norm(Infinity)

def infinity_norm_vec(vec):
  return max([infinity_norm(vi) for vi in vec])
```

Assim, o código em Sage que implementa esses procedimentos do GSW é apresentado a seguir. O arquivo *decompositions.sage* contém as funções mostradas na seção 1.5.1.

```
load("utils.sage")
load("decompositions.sage")
load("distribution_rlwe.sage")

class GSW:
  def __init__(self, n, q, sigma, B = 2):
    self.n, self.sigma, self.B = n, sigma, B
    f = x^n + 1
    self.l = ceil(log(q, B))
    self.q = B^self.l
    g = Matrix(ZZ, self.l, 1, [B^i for i in range(self.l)])
    I = Matrix.identity(2)
    self.G = I.tensor_product(g) # gadget matrix

    self.Rq = ( ZZ.quotient(q) )['x'].quotient(f)
    self.sk = self.keygen()
    self.dist_rlwe = RLWEDistribution(self.sk, n, q, sigma)

  def keygen(self):
    sk = 0
    while 0 == sk:
      sk = Zx([ZZ.random_element(-1, 2) for _ in range(self.n)])
    return sk

  def enc(self, m):
    Rq, l, sk = self.Rq, self.l, self.sk
    C = Matrix(Rq, 2*l, 2)
    for i in range(2*l):
      C[i] = self.dist_rlwe.sample()
    C += m * self.G
```

```
    return C

  def dec(self, C):
    B, l, q, sk = self.B, self.l, self.q, self.sk
    a = C[2*l - 1, 0]
    b = C[2*l - 1, 1] # == a*sk + e + (q/B) * m
    noisy_m = Zx((b - a*sk).lift())
    rounded_m = round_poly(B * noisy_m / q)
    return sym_mod_poly(rounded_m, B)
```

Algoritmo 1.3. Procedimentos básicos da cifra GSW

As operações homomórficas do GSW são definidas da seguinte forma:

- GSW.Add$(\mathbf{C}_0, \mathbf{C}_1)$: simplesmente adicione as entradas correspondentes, i.e., devolva $\mathbf{C}_{add} := \mathbf{C}_0 + \mathbf{C}_1 \in R_q^{2\ell \times 2}$.

- GSW.Mult$(\mathbf{C}_0, \mathbf{C}_1)$: decomponha $\mathbf{C}_0$ em base B e multiplique por $\mathbf{C}_1$ fazendo operações em R_q, i.e., devolva $\mathbf{C}_{mult} := G^{-1}(\mathbf{C}_0) \cdot \mathbf{C}_1 \in R_q^{2\ell \times 2}$.

O código em Sage que implementa as operações homomórficas do GSW é apresentado a seguir e deve ser inserido na definição da classe GSW apresentada no Algoritmo 1.3.

```
    def add(self, C0, C1):
        C_add = C0 + C1
        return C_add

    def inv_g_row_ciphertex(self, c):
        B, l, n = self.B, self.l, self.n
        a, b = c[0], c[1]
        res = vector(self.Rq, [0] * 2 * l)
        res[0 : l] = inv_g_poly(a, B, l, n)
        res[l : 2*l] = inv_g_poly(b, B, l, n)
        return res

    def mult(self, C0, C1):
        result = Matrix(self.Rq, 2*self.l, 2)
        for i in range(2*self.l):
            decomp = self.inv_g_row_ciphertex(C0[i])
            prod_in_Rq = decomp * C1
            result[i] = prod_in_Rq
        return result
```

Analisar a adição homomórfica é trivial. Considerando que os operandos são da

forma $\mathbf{C}_i = [\mathbf{a}_i, \mathbf{b}_i] + m_i\mathbf{G}$, com $\mathbf{b}_i = \mathbf{a}_i \cdot z + \mathbf{e}_i \in R_q^{2\ell}$, temos

$$\begin{aligned}\mathbf{C}_{add} &= [\mathbf{a}_0 + \mathbf{a}_1,\ \mathbf{b}_0 + \mathbf{b}_1] + (m_0 + m_1)\mathbf{G} \\ &= [\underbrace{\mathbf{a}_0 + \mathbf{a}_1}_{\mathbf{a}_{add}},\ (\mathbf{a}_0 + \mathbf{a}_1) \cdot z + \underbrace{\mathbf{e}_0 + \mathbf{e}_1}_{\mathbf{e}_{add}}] + (m_0 + m_1)\mathbf{G}.\end{aligned}$$

Logo, vemos que $\mathbf{C}_{add} = [\mathbf{a}_{add},\ \mathbf{a}_{add} \cdot z + \mathbf{e}_{add}] + (m_0 + m_1)\mathbf{G}$, ou seja, um criptograma válido que cifra a soma das mensagens. Além disso, o ruído cresce apenas aditivamente, i.e., $\mathsf{erro}(\mathbf{C}_{add}) \leq \mathsf{erro}(\mathbf{C}_0) + \mathsf{erro}(\mathbf{C}_1)$.

Analisar a multiplicação homomórfica é ligeiramente mais complicado. Primeiro, note que cada operando $\mathbf{C}_i$ é uma matriz $2\ell \times 2$ e que a decomposição expande cada linha com 2 elementos transformando-as em linhas com 2ℓ elementos, ou seja, $G^{-1}(\mathbf{C}_0)$ é uma matriz $2\ell \times 2\ell$, portanto, o produto $G^{-1}(\mathbf{C}_0) \cdot \mathbf{C}_1$ está bem definido e resulta em uma matriz $2\ell \times 2$, ou seja, com as dimensões de um criptograma válido.

Agora, note que sobre R_q, temos

$$\mathbf{C}_{mult} = G^{-1}(\mathbf{C}_0) \cdot ([\mathbf{a}_1,\ \mathbf{b}_1] + m_1\mathbf{G}) = [G^{-1}(\mathbf{C}_0) \cdot \mathbf{a}_1,\ G^{-1}(\mathbf{C}_0) \cdot \mathbf{b}_1] + m_1 G^{-1}(\mathbf{C}_0) \cdot \mathbf{G}.$$

Definindo $\mathbf{a}' := G^{-1}(\mathbf{C}_0) \cdot \mathbf{a}_1$ e notando que $\mathbf{b}_1 = \mathbf{a}_1 \cdot s + \mathbf{e}_1$, vemos que

$$\mathbf{C}_{mult} = [\mathbf{a}',\ \mathbf{a}' \cdot z + G^{-1}(\mathbf{C}_0) \cdot \mathbf{e}_1] + m_1 G^{-1}(\mathbf{C}_0) \cdot \mathbf{G}.$$

Além disso, como o produto entre a decomposição de $\mathbf{C}_0$ e $\mathbf{G}$ resulta novamente em $\mathbf{C}_0$, temos $m_1 G^{-1}(\mathbf{C}_0) \cdot \mathbf{G} = m_1([\mathbf{a}_0,\ \mathbf{b}_0] + m_0\mathbf{G}) = [m_1\mathbf{a}_0,\ m_1\mathbf{a}_0 \cdot z + m_1\mathbf{e}_0] + m_1 m_0 \mathbf{G}$, portanto,

$$\mathbf{C}_{mult} = [\mathbf{a}' + m_1\mathbf{a}_0,\ (\mathbf{a}' + m_1\mathbf{a}_0) \cdot z + G^{-1}(\mathbf{C}_0) \cdot \mathbf{e}_1 + m_1\mathbf{e}_0] + m_1 m_0 \cdot \mathbf{G}. \tag{1}$$

Assim, vemos que $\mathbf{C}_{mult} = [\mathbf{a}_{mult},\ \mathbf{a}_{mult} \cdot z + \mathbf{e}_{mult}] + m_0 m_1 \mathbf{G}$, onde $\mathbf{a}_{mult} := \mathbf{a}' + m_1\mathbf{a}_0$ e $\mathbf{e}_{mult} := G^{-1}(\mathbf{C}_0) \cdot \mathbf{e}_1 + m_1\mathbf{e}_0$, ou seja, é um criptograma válido e realmente cifra o produto das mensagens.

Agora fica claro o porquê de o ruído crescer apenas de forma quase aditiva, pois o produto $\mathbf{e}_0 \cdot \mathbf{e}_1$ não aparece em $\mathbf{e}_{mult}$. Para analisar como o ruído cresce, ou seja, quão grande é a norma de $\mathbf{e}_{mult}$ em comparação às normas de $\mathbf{e}_0$ e $\mathbf{e}_1$, primeiro precisamos do seguinte fato básico – sua prova é facilmente derivada usando o produto do vetor de coeficientes pela matriz anticirculante, ambos definidos na seção 1.7.6.1. Denotando a norma infinita por $\|\cdot\|$, temos:

Lema 1.5.1. *Sejam f e g elementos de $R := \mathbb{Z}[X]/\langle X^N + 1\rangle$. Então, $\|f \cdot g\| \leq N\|f\|\|g\|$.*

Usando esse lema, é fácil ver que se $\mathbf{A} \in R^{d \times \ell}$, $\mathbf{v} \in R^\ell$ e $\mathbf{u} := \mathbf{A} \cdot \mathbf{v}$, então $\|\mathbf{u}\| \leq N\ell\|\mathbf{A}\| \cdot \|\mathbf{v}\|$, pois $\|\mathbf{u}\| = \max\{\|u_0\|, ..., \|u_{d-1}\|\}$ e cada coordenada u_i satisfaz

$$\|u_i\| = \left\| \sum_{j=0}^{\ell-1} a_{i,j} v_j \right\| \leq \sum_{j=0}^{\ell-1} \left\| a_{i,j} v_j \right\| \leq \sum_{j=0}^{\ell-1} N \left\| a_{i,j} \right\| \left\| v_j \right\| \leq N\ell \|\mathbf{A}\| \cdot \|\mathbf{v}\|.$$

Assim, podemos ver que

$$\begin{aligned}\|\mathbf{e}_{mult}\| &\leq \left\|G^{-1}(\mathbf{C}_0)\cdot\mathbf{e}_1\right\| + \|m_1\mathbf{e}_0\| && \text{(desigualdade triangular)}\\ &\leq 2N\ell\left\|G^{-1}(\mathbf{C}_0)\right\|\|\mathbf{e}_1\| + \|m_1\mathbf{e}_0\| &&\\ &\leq 2NB\ell\|\mathbf{e}_1\| + \|m_1\mathbf{e}_0\| && \text{(pois } \left\|G^{-1}(\mathbf{C}_0)\right\| \leq B).\end{aligned}$$

Geralmente, os coeficientes de m_1 são pequenos, por exemplo, menores que B, e B é uma constante pequena. Nesses casos, temos $\|m_1\mathbf{e}_0\| \leq NB\|\mathbf{e}_0\|$. Mas em vários cenários, como no *bootstrapping* apresentado na Seção 1.7.2, as mensagens cifradas pertencem à $\{-1,0,1\}$ ou são monômios da forma $\pm 1\cdot X^k$. Nesses casos, temos $\|m_1\mathbf{e}_0\| = \|\mathbf{e}_0\|$ e o crescimento do ruído é ainda menor.

Uma observação importante é que m_0 não aparece no ruído de $\mathbf{C}_{mult}$, portanto, ele não depende da mensagem cifrada pelo operando a esquerda, assim, se for sabido que a mensagem de um dos operandos é possivelmente maior que a do outro, pode-se sempre utilizá-lo como o termo $\mathbf{C}_0$. Essa assimetria é importante durante o *bootstrapping*, pois em um dado momento, é preciso multiplicar homomorficamente a mensagem $\lfloor q/8 \rceil$, que é extremamente grande (em relação a q).

Note também que é importante colocar a esquerda o criptograma que tem o maior ruído, ou seja, que já passou por mais operações homomórficas, pois assim seu ruído é multiplicado apenas pela mensagem do outro criptograma, que normalmente, é pequena. Por exemplo, imagine que $\mathbf{C}$ é um criptograma "fresco", i.e., devolvido por $\mathsf{GSW.Enc}$ e que não passou por qualquer operação homomórfica. Considere que $\mathbf{C}$ cifra $m = 1$ com ruído $\mathbf{e}$. Então, temos $\mathsf{erro}(\mathbf{C}) = \|\mathbf{e}\| \approx \sigma$. Além disso, considere um criptograma $\mathbf{C}'$ obtido após uma multiplicação homomórfica envolvendo dois criptogramas frescos e suponha que $\mathbf{C}'$ cifra $m' = 1$ com ruído $\mathbf{e}'$. Como visto acima, temos $\mathsf{erro}(\mathbf{C}') = \|\mathbf{e}'\| \approx 2NB\ell\sigma$. Então, a magnitude do ruído de $\mathsf{GSW.Mult}(\mathbf{C}', \mathbf{C})$ é

$$\left\|G^{-1}(\mathbf{C}')\mathbf{e} + m\mathbf{e}'\right\| \leq 2NB\ell\sigma + \left\|\mathbf{e}'\right\| \approx 4NB\ell\sigma,$$

no entanto, o ruído de $\mathsf{GSW.Mult}(\mathbf{C}, \mathbf{C}')$ é

$$\left\|G^{-1}(\mathbf{C})\mathbf{e}' + m'\mathbf{e}\right\| \leq 2NB\ell\left\|\mathbf{e}'\right\| + \|\mathbf{e}\| \approx (2NB\ell)^2 + \sigma.$$

É fácil também analisar a evolução do ruído ao se efetuar uma sequência de produtos do tipo $\prod_{i=0}^{k} m_i$. Nosso primeiro objetivo é estudar o formato do ruído.

Lema 1.5.2. *Seja $k \geq 1$ um inteiro. Para $0 \leq i \leq k$, seja $\mathbf{C}_i \in R_q^{2\ell\times 2}$ um criptograma que cifra uma mensagem $m_i \in R_B$ com ruído $\mathbf{e}_i$. Defina $\mathbf{C}'_0 := \mathbf{C}_0$ e, para $1 \leq i \leq k-1$, defina $\mathbf{C}'_{i+1} := \mathsf{GSW.Mult}(\mathbf{C}'_i, \mathbf{C}_{i+1})$. Note que $\mathbf{C}'_1$ cifra $m_0 \cdot m_1$, $\mathbf{C}'_2$ cifra $m_0 \cdot m_1 \cdot m_2$, e assim sucessivamente. Então, o ruído de $\mathbf{C}'_k$ é*

$$\mathbf{e}'_k = \mathbf{e}_0 \prod_{i=1}^{k} m_i + \sum_{i=0}^{k-1} G^{-1}(\mathbf{C}'_i)\mathbf{e}_{i+1}\left(\prod_{j=i+2}^{k} m_j\right). \tag{2}$$

Prova. Podemos provar por indução. O caso base, $k = 1$, é trivial, pois como mostrado na Equação 1, temos $\mathbf{e}'_1 = G^{-1}(\mathbf{C}'_0)\cdot\mathbf{e}_1 + m_1\mathbf{e}'_0$, que é idêntica à Equação 2.

Agora, suponha que a Equação 2 vale para um valor arbitrário $k-1$.

$$\mathbf{e}'_{k-1} = \mathbf{e}_0 \prod_{i=1}^{k-1} m_i + \sum_{i=0}^{k-2} G^{-1}(\mathbf{C}'_i)\mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k-1} m_j \right).$$

Como $\mathbf{e}'_k$ é o ruído de $\mathsf{GSW.Mult}(\mathbf{C}'_{k-1}, \mathbf{C}_k)$, temos que

$$\begin{aligned}
\mathbf{e}'_k &= G^{-1}(\mathbf{C}'_{k-1})\mathbf{e}_k + m_k \mathbf{e}'_{k-1} && \text{(Equação 1)} \\
&= G^{-1}(\mathbf{C}'_{k-1})\mathbf{e}_k + m_k \left(\mathbf{e}_0 \prod_{i=1}^{k-1} m_i + \sum_{i=0}^{k-2} G^{-1}(\mathbf{C}'_i)\mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k-1} m_j \right) \right) && \text{(Hipótese indutiva)} \\
&= G^{-1}(\mathbf{C}'_{k-1})\mathbf{e}_k + \mathbf{e}_0 \prod_{i=1}^{k} m_i + \sum_{i=0}^{k-2} G^{-1}(\mathbf{C}'_i)\mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k} m_j \right) \\
&= \mathbf{e}_0 \prod_{i=1}^{k} m_i + \sum_{i=0}^{k-1} G^{-1}(\mathbf{C}'_i)\mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k} m_j \right)
\end{aligned}$$

onde a última igualdade vale pois $\prod_{j=k-1+2}^{k} m_j = 1$, logo o termo $G^{-1}(\mathbf{C}'_{k-1})\mathbf{e}_k$ a esquerda pode ser incluído como o $(k-1)$-ésimo termo somatório. □

Usando esse lema, pode-se finalmente mostrar como o ruído cresce com uma sequência de produtos:

Lema 1.5.3. *Usando a mesma notação do Lema 1.5.2, considere $\mathbf{C}'_k$, que cifra $\prod_{i=0}^{k} m_k$ com ruído $\mathbf{e}'_k$. Então,*

$$\left\|\mathbf{e}'_k\right\| \leq \left\| \mathbf{e}_0 \prod_{i=1}^{k} m_i \right\| + \sum_{i=0}^{k-1} 2NB\ell \left\| \mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k} m_j \right) \right\|. \tag{3}$$

Prova. Usando o Lema 1.5.2, temos

$$\begin{aligned}
\left\|\mathbf{e}'_k\right\| &\leq \left\| \mathbf{e}_0 \prod_{i=1}^{k} m_i \right\| + \sum_{i=0}^{k-1} \left\| G^{-1}(\mathbf{C}'_i)\mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k} m_j \right) \right\| && \text{(Desigualdade triangular)} \\
&\leq \left\| \mathbf{e}_0 \prod_{i=1}^{k} m_i \right\| + \sum_{i=0}^{k-1} 2N\ell \left\| G^{-1}(\mathbf{C}'_i) \right\| \cdot \left\| \mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k} m_j \right) \right\| \\
&\leq \left\| \mathbf{e}_0 \prod_{i=1}^{k} m_i \right\| + \sum_{i=0}^{k-1} 2NB\ell \left\| \mathbf{e}_{i+1} \left(\prod_{j=i+2}^{k} m_j \right) \right\|
\end{aligned}$$

□

Finalmente, considere $S := \{-1, 0, 1\} \cup \{-X^1, ..., -X^{N-1}\} \cup \{X^1, ..., X^{N-1}\}$. Se todas as mensagens m_i pertencerem a S (como ocorre no *bootstrapping*), então todos os produtos de mensagens também pertencem a S, pois serão sempre iguais a zero ou a $\pm 1 \cdot X^k$ para algum k, logo, ao serem reduzidas módulo $X^N + 1$, obtém-se $\pm 1 \cdot X^{k \bmod N}$.

Além disso, para qualquer $e \in R$ e $m \in S$, o produto $m \cdot e$ é zero ou um polinômio com os mesmos coeficientes que e mas possivelmente em outra ordem e multiplicados por -1. Por exemplo, se $n = 4$, $e = 2X^3 - X$ e $m = X^2$, então, $e \cdot m = 2X^5 - X^2 = -X^2 - 2X \pmod{X^N + 1}$. Portanto, $\|e\| = \|e \cdot m\|$. Com isso, tem-se o seguinte corolário:

Corolário 1.6. *Considere a mesma notação do Lema 1.5.2 e assuma que $m_i = 0$ ou $m = \pm 1 \cdot X^k$ para algum $0 \leq k < n$. Além disso, suponha que o ruído $\mathbf{e}_i$ de cada criptograma é limitado por um valor β. Então, $\mathbf{C}'_k$ cifra $\prod_{i=0}^{k} m_k$ com ruído $\mathbf{e}'_k$ e*

$$\left\|\mathbf{e}'_k\right\| \leq \beta + 2kNB\ell\beta. \tag{4}$$

Perceba a diferença em relação ao DGHV: se todos os criptogramas tiverem ruído menor que um valor β, uma multiplicação homomórfica com o DGHV aumenta o ruído para $O(\beta^2)$, a segunda multiplicação o aumenta para $O(\beta^3)$, e assim sucessivamente. Então, avaliar $\prod_{i=0}^{k} m_i$ homomorficamente gera um criptograma com ruído $O(\beta^k)$, ou seja, o crescimento é exponencial em k, enquanto o ruído com o GSW aumenta apenas linearmente em k, i.e., de β para $O(k\beta)$.

1.6.1. Comparando as comparações

Nesta seção, implementaremos a comparação homomórfica de números inteiros, discutiremos o crescimento do ruído e compararemos com a Seção 1.3.2, que usa o DGHV para implementar essa comparação homomórfica.

A única operação homomórfica usada no Algoritmo 1.2 que não está implementada na classe GSW é a negação, mas ela pode ser implementada usando a mesma lógica: basta somar o valor 1. No entanto, no caso do GSW, uma encriptação do 1 trivial e sem ruído não é apenas o inteiro 1, mas sim $1 \cdot \mathbf{G}$. Logo, o seguinte código deve ser incluído na definição da classe GSW:

```
    def not_gate(self, C):
        return C + self.G
```

Note que essa porta lógica funciona módulo 2, portanto, para utilizá-la, é preciso instanciar o GSW com o parâmetro $B = 2$.

Para acompanhar o crescimento do ruído na prática, definiremos também uma função que devolve o logaritmo da norma do ruído. Como um criptograma do GSW é definido como $\mathbf{C} = [\mathbf{a}\ ,\ \mathbf{b}] + m \cdot \mathbf{G}$ e o ruído está presente no termo $\mathbf{b}$, i.e., $\mathbf{b} = \mathbf{a} \cdot z + \mathbf{e}$, primeiro subtraímos $m \cdot \mathbf{G}$ para recuperar $\mathbf{a}$ e $\mathbf{b}$, então calculamos $\log(\left\|[\mathbf{b} - \mathbf{a} \cdot z]_q\right\|)$, como no código a seguir, que também deve ser incluído na definição da classe GSW:

```
  def get_noise(self, C, msg):
    l, sk = self.l, self.sk
    C -= msg * self.G
    a = vector(C[:, 0])
    b = vector(C[:, 1]) # == a*sk + e (mod q)
    e = b - a*sk
    e = sym_mod_vec(e, self.q) # definida em utils.sage
    norm_e = infinity_norm_vec(e) # definida em utils.sage
```

```
    if norm_e == 0: # log de zero não está definido
      return -1

    return log(norm_e, 2).n()
```

Uma vez definida a porta *not* e essa função que devolve o ruído, é trivial converter o Algoritmo 1.2: basta substituir o objeto do tipo DGHV por um do tipo GSW, inicializar o criptograma c com a matriz $1 \cdot G$ em vez de simplesmente 1, i.e., `c = gsw.G`, e remover a redução módulo 2 ao calcular m, ou seja, `m *= bits0[i] + bits1[i] + 1`.

Para fins de teste, podemos negligenciar a segurança e escolher um valor de N pequeno, por exemplo, para comparar inteiros com L bits, podemos instanciar um objeto do tipo GSW como `gsw = GSW(n=8, q=3*L, sigma=3.2, B=2)`. Execute o algoritmo algumas vezes e observe como o ruído evolui durante a avaliação homomórfica da comparação. Veja se o ruído do criptograma produzido no fim da comparação tem ruído próximo de $\log q$ – lembre-se que o ruído não pode passar o limiar $q/(2B) = q/4$, senão a decifração não funcionará corretamente. Note que o algoritmo primeiro cria o criptograma `cmp_i` adicionando dois criptogramas frescos e avaliando a porta lógica not. A adição aumenta o ruído para aproximadamente 2σ e a porta lógica não altera o ruído, portanto, o ruído de `cmp_i` é pequeno. No entanto, o criptograma `c` é o resultado de várias multiplicações homomórficas, logo, seu ruído é grande. Por isso, como discutido anteriormente, deve-se usar a assimetria do GSW e colocar `c` a esquerda na multiplicação, i.e., deve-se usar `c = gsw.mult(c, cmp_i)`. Para verificar na prática como essa assimetria é importante, troque essa linha por `c = gsw.mult(cmp_i, c)` e veja como o ruído cresce muito mais rapidamente e faz com que a decifração deixe de funcionar.

Observe que cada criptograma `cmp_i` cifra uma mensagem da forma $m_i = a_i + b_i + 1$, com a_i e b_i binários, logo, m_i pode ser no máximo 3, o que é um valor pequeno. No entanto, como essas mensagens são multiplicadas, o valor $\prod_{i=0}^{k} m_i$ pode ser tão grande quanto 3^k no pior caso, i.e., quando $a_i = b_i = 1$ para $0 \leq i \leq k$. Mas como foi provado no Lema 1.5.3, o ruído final depende do valor da mensagem, portanto, no pior caso, ele será dominado por 3^k e acabará sendo exponencial no número de bits comparados homomorficamente. Mais especificamente, usando o Lema 1.5.3, vemos que o ruído final pode ser aproximadamente $3^k\sigma(1+4kN\log q)$. Como ele deve ser menor que $q/4$ para garantir a corretude da decifração, aplicando o logaritmo, temos a seguinte desigualdade

$$\log(q/4) \geq k\log(3) + \log(\sigma) + \log(1+4kN\log q).$$

Como σ é uma constante pequena e $N = \Theta(\lambda)$, ignorando o fator $\log\log q$, obtemos

$$\log q = \Theta(k + \log(k\lambda)).$$

Finalmente, como cada criptograma é uma matriz de dimensões $2\log q \times 2$, cujas entradas são polinômios de grau menor que n e coeficientes com $\log q$ bits, vemos que o tamanho de cada criptograma necessário para comparar homomorficamente inteiros com k bits é

$$\Theta(N\log^2 q) = \Theta(\lambda\log^2 q) = \Theta(\lambda(k+\log(k\lambda))^2) = \Theta(\lambda k^2 + \lambda k\log(k\lambda)).$$

Ou seja, cada criptograma tem tamanho essencialmente linear no parâmetro de segurança λ e quadrático no número de bits k.

Já no caso do DGHV, o ruído inicial de cada criptograma é $2^{\rho} \approx 2^{\lambda}$, o ruído final é $2^{k\rho} \approx 2^{k\lambda}$ e ele precisa ser menor que $p \approx 2^{\eta}$, o que nos dá $\eta = \Theta(k\lambda)$. O número de bits de cada criptograma é

$$\gamma = \Theta(\lambda(\rho - \eta)^2) = \Theta(\lambda(k\lambda)^2) = \Theta(\lambda^3 k^2),$$

ou seja, para comparar homomorficamente inteiros de k bits, no pior caso, com o DGHV, é preciso manipular criptogramas com tamanho quadrático em k e cúbico no parâmetro de segurança! Obviamente, $\Theta(\lambda^3 k^2)$ é muito pior que $\Theta(\lambda k^2)$.

Evitando que a mensagem cresça

Note que se a mensagem não crescesse a ponto de ser exponencial em k, mais especificamente, 3^k, o ruído final produzido pelo GSW seria muito menor, podendo ser apenas logarítmico em k. Esse é um problema comum com o GSW: em geral, é preciso evitar que as mensagens cresçam muito. Uma abordagem possível seria expressar todas as operações que podem aumentar o tamanho da mensagem em termos de portas lógicas do tipo NAND (AND negado), pois elas garantem que a mensagem será sempre binária, já que $\mathsf{NAND}(a,b) := 1 - a \cdot b \in \{0,1\}$. Lembre-se de que o NAND é uma porta lógica universal, portanto, todo circuito pode ser expresso apenas por ela.

Por exemplo, no Algoritmo 1.2, a operação $a + b + 1$ na verdade corresponde a um XNOR (XOR negado), e é ela que permite que a mensagem cifrada cresça de 1 para 3, então, pode-se substituí-la por uma sequência de portas NAND usando o fato de que

$$\mathsf{XNOR}(a,b) = \mathsf{NAND}(\mathsf{NAND}(a,b), \mathsf{NAND}(\mathsf{NAND}(a,a), \mathsf{NAND}(b,b))).$$

A tradução do NAND para a versão homomórfica é literal, i.e., multiplica-se os dois criptogramas e subtrai-se o resultado do valor 1, que corresponde a $1 \cdot \mathbf{G}$:

$$\mathsf{GSW.Nand}(\mathbf{C}_0, \mathbf{C}_1) := \mathbf{G} - \mathsf{GSW.Mult}(\mathbf{C}_0, \mathbf{C}_1).$$

1.6.2. O *bootstrapping* do esquema GSW

Como apresentado até então, a cifra GSW não é completamente homomórfica, pois ela só pode avaliar circuitos de profundidade limitada. No entanto, como demonstrado no Lema 1.5.3, o ruído cresce lentamente com o GSW, principalmente se as mensagens forem binárias e se mantiverem pequenas com a avaliação homomórfica, como quando o circuito é expresso usando a porta NAND.

Portanto, ao contrário da cifra DGHV, é possível avaliar circuitos com qualquer profundidade L, não apenas $\log \lambda$. Logo, dado L, a profundidade desejada, pode-se escolher os parâmetros q, N, etc, de tal forma que seja possível avaliar todos os circuitos de profundidade L. Por isso, o esquema GSW que obtemos até então é chamado de *cifra completamente homomórfica por nível*[8]. Note a diferença: em uma cifra completamente homomórfica, é possível escolher um conjunto de parâmetros para o qual todos os circuitos podem ser computados homomorficamente, ou seja, pode-se instanciar o esquema

[8]Do inglês *levelled homomorphic encryption scheme.*

e depois avaliar qualquer função sobre os dados cifrados. Já com uma cifra homomórfica por nível, é preciso primeiro decidir as funções que serão calculadas, depois instanciar a cifra usando parâmetros que permitam avaliar tais funções homomorficamente.

Para transformar o GSW em uma cifra completamente homomórfica no sentido estrito (não por nível), como discutido seção 1.3.3, basta que ele seja capaz de avaliar seu próprio circuito de decifração e, de fato, ele o é: basta identificar a profundidade L desse circuito e escolher parâmetros que permitam avaliar circuitos de profundidade $L+k$, para alguma constante pequena k que permita que outras operações úteis sejam feitas entre os *bootstrappings*. Note que não é necessário publicar informação auxiliar sobre a chave secreta para simplificar o circuito de decifração, como foi feito para o DGHV na seção 1.3.4. Portanto, o *bootstrapping* é mais simples, direto e não usa hipóteses de segurança adicionais além da segurança circular.

No entanto, apesar de ser possível e ser muito mais simples do que no caso do DGHV, expressar a função de decifração como um grande circuito binário e avaliar cada porta lógica homomorficamente faz com que o *bootstrapping* seja extremamente lento. Assim, nas seções seguintes, em vez de aplicar o *bootstrapping* ao GSW, o utilizaremos para aplicar o *bootstrapping* a um esquema mais simples, obtendo assim o esquema mais rápido e eficiente que existe atualmente.

1.7. *Bootstrapping in vitro*

Nesta seção, mostraremos como o *bootstrapping* pode ser avaliado eficientemente. Para isso, primeiro definiremos uma cifra homomórfica simples, chamada cifra de base, que pode avaliar apenas uma porta NAND, e então utilizaremos o GSW para decifrar homomorficamente os criptogramas dessa cifra de base.

1.7.1. A cifra homomórfica mais simples possível

Nesta seção, seguiremos a abordagem de [DM15] e utilizaremos o problema LWE para construir uma cifra homomórfica por nível extremamente simples: Ela cifra um bit no nível 1, então pode-se avaliar homomorficamente uma única porta NAND, gerando um criptograma no nível 0, que precisa então ser recifrado com o *bootstrapping* para voltar ao nível 1 e reduzir o ruído. Essa cifra é definida como a seguir. Para simplificar a exposição, assumiremos que o módulo q é um múltiplo de 8, mas pode ser facilmente eliminada:

- $\mathsf{HE.ParamGen}(1^\lambda)$: Escolha os parâmetros n, q e σ do problema LWE de tal forma que se obtenha λ bits de segurança e que $q \in 8\mathbb{Z}$. Devolva $\mathsf{params} := (n, q, \sigma)$.

- $\mathsf{HE.KeyGen}(\mathsf{params})$: Amostre $\mathbf{s}$ uniformemente de $\{0,1\}^n$. Devolva $\mathsf{sk} := \mathbf{s}$.

- $\mathsf{HE.Enc}(\mathsf{sk}, m)$: Para cifrar um bit m em um criptograma no nível 1, amostre $(\mathbf{a}, b') \leftarrow \mathscr{A}_{\mathbf{s},q,\sigma}$, defina $b := b' + (q/4)m \bmod q$ e devolva $c = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$.

- $\mathsf{HE.Dec}(\mathsf{sk}, c)$: Para decifrar um criptograma $\mathbf{c} = (\mathbf{a}, b)$ do nível 1, calcule $c' = b - \mathbf{a} \cdot \mathbf{s} \bmod q$ e devolva $\left[\left\lfloor \frac{4c'}{q} \right\rceil\right]_2$.

Traduzir esses procedimentos para a linguagem Sage é trivial, vide o código a seguir:

```
load("distribution_lwe.sage")

class LWEScheme:
  def __init__(self, n, q, sigma=3.2):
    self.n, self.q = n, q
    self.keygen()
    self.dist_lwe = LWEDistribution(self.sk, q, sigma)

  def keygen(self):
    n = self.n
    bin_list = [ZZ.random_element(0, 2) for _ in range(n)]
    self.sk = vector(ZZ, bin_list)

  def enc(self, m):
    a, _b = self.dist_lwe.sample()
    b = _b + round(self.q / 4) * m
    return [a, b]

  def dec(self, c):
    q, s = self.q, self.sk
    noisy_m = ZZ(c[1] - c[0]*s) # == e + (q / 4)*m + u*q
    return round(4 * noisy_m / q) % 2
```

Algoritmo 1.4. Procedimentos básicos da cifra de base

A segurança desse esquema segue diretamente da versão decisional do problema LWE, pois ela garante que a amostra LWE $(\mathbf{a}, b')$ é indistinguível de um vetor uniformemente aleatório em $\mathbb{Z}_q^{n+1}$, portanto, $(\mathbf{a}, b') + (\mathbf{0}, \mu) \bmod q$ continua sendo indistinguível de um vetor uniforme para qualquer $\mu \in \mathbb{Z}_q$, em particular, para $\mu = (q/4) \cdot m$, como usado na encriptação.

Para verificar a corretude da decifração, note que $c' := b - \mathbf{as} = e + (q/4)m + uq$ para algum $u \in \mathbb{Z}$. Logo,

$$\left\lfloor \frac{4c'}{q} \right\rceil = \left\lfloor \frac{4e}{q} + m + 4q \right\rceil = \left\lfloor \frac{4e}{q} \right\rceil + m + 4q$$

e precisamos que o termo $\lfloor 4e/q \rceil$ seja zero, o que ocorre se $4|e|/q < 1/2$, ou seja, se $|e| < q/8$. Em outras palavras, se $|e| < q/8$, então $[\lfloor 4c'/q \rceil]_2 = [m + 4q]_2 = m$ e a mensagem correta é devolvida.

Apesar da decifração exigir apenas que o ruído seja menor que $q/8$, para que a porta lógica NAND apresentada a seguir funcione corretamente, é necessário que $|e| < q/16$, por isso, será sempre exigido que os criptogramas desse esquema respeitem essa inequação. Para enfatizar esse fato, temos a definição a seguir:

Definição 1.7.1. *Sejam* $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$, $\mathbf{s} \in \{0,1\}^n$ *e* $m \in \{0,1\}$. *Dizemos que* $\mathbf{c}$ *é uma* encriptação válida *de m sob a chave secreta* $\mathbf{s}$ *se existe* $e \in \mathbb{Z}$ *tal que* $b = \mathbf{a} \cdot \mathbf{s} + e + (q/4)m$ *e* $|e| < q/16$.

Assim, é preciso garantir que tanto os criptogramas devolvidos pela função Enc quanto o s obtidos após o bootstrapping sejam *válidos*, pois isso garantirá que qualquer função possa ser avaliada homomorficamente, já que, dado um criptograma válido, será possível aplicar a porta NAND e em seguida o *bootstrapping*, que produzirá novamente um criptograma válido.

O NAND homomórfico é obtido com simples adições e subtrações vetoriais, i.e., coordenada a coordenada, como mostrado a seguir:

- $\mathsf{HE.Nand}(\mathbf{c}_0, \mathbf{c}_1)$: Sejam $\mathbf{c}_0$ e $\mathbf{c}_1$ dois criptogramas no nível 1. Devolva

$$\mathbf{c} := (\mathbf{0}, 5q/8) - \mathbf{c}_0 - \mathbf{c}_1 \in \mathbb{Z}_q^{n+1}.$$

Para entender como essa porta lógica homomórfica funciona, note que o criptograma que ela produz é da forma $(\mathbf{a}, b)$ com $\mathbf{a} = -\mathbf{a}_0 - \mathbf{a}_1$ e

$$b = 5q/8 - b_0 - b_1 = \mathbf{a} \cdot \mathbf{s} - e_0 - e_1 + 5q/8 - (q/4)(m_0 + m_1),$$

ou seja, b já contém o termo $\mathbf{a}$ multiplicado pela chave secreta, como esperado. O que não está claro é que ele cifra realmente $\mathsf{NAND}(m_0, m_1) = 1 - m_0 \cdot m_1$. Para ver que esse é realmente o caso, lembre-se de que ambos m_0 e m_1 são binários, logo $m_i^2 = m_i$, portanto, $(m_0 - m_1)^2 = m_0 - 2m_0m_1 + m_1$. Então,

$$(q/4)(m_0 + m_1) = (q/4)((m_0 - m_1)^2 + 2m_0m_1) = (q/4)(m_0 - m_1)^2 + (q/2)m_0m_1.$$

Note agora que, como $5q/8 = q/8 + q/2$, temos

$$5q/8 - (q/4)(m_0 + m_1) = q/8 - (q/4)(m_0 - m_1)^2 + (q/2)(1 - m_0m_1).$$

Portanto, o termo b do criptograma devolvido por $\mathsf{HE.Nand}$ é da forma

$$b = \mathbf{a} \cdot \mathbf{s} + e + (q/2)(1 - m_0 \cdot m_1),$$

como esperado. Seu ruído é então $e := -e_0 - e_1 + q/8 - (q/4)(m_0 - m_1)^2$. Note que a mensagem é exatamente $\mathsf{NAND}(m_0, m_1)$ e que ela é multiplicada por $q/2$ em vez de $q/4$, portanto, temos um criptograma no nível zero.

Para analisar o tamanho desse ruído, perceba que $(m_0 - m_1)^2 \in \{0,1\}$, pois ambas as mensagens são binárias, logo, $q/8 - (q/4)(m_0 - m_1)^2 \in \{q/8, -q/8\}$, portanto, podemos reescrever e como $-e_0 - e_1 \pm q/8$. Como $|e_i| < q/16$ por hipótese, temos

$$|e| < q/16 + q/16 + q/8 = q/4.$$

Resumindo, se ambos $\mathbf{c}_0$ e $\mathbf{c}_1$ forem criptogramas válidos, então $\mathsf{HE.Nand}(\mathbf{c}_0, \mathbf{c}_1)$ devolve um criptograma que cifra $\mathsf{NAND}(m_0, m_1)$ com ruído menor que $q/4$ e no *nível*

zero. Note que esse ruído é muito próximo de q, por isso, não é possível aplicar mais nenhuma operação homomórfica. Para prosseguir avaliar as próximas portas lógicas do circuito, precisamos executar o *bootstrapping*, como veremos nas próximas seções.

A seguir, apresenta-se o código em Sage que implementa o NAND homomórfico. Ele deve ser incluído na definição da classe LWEScheme apresentada no Algoritmo 1.4.

```
def nand(self, c0, c1):
  a0, b0 = c0
  a1, b1 = c1
  a = -a0 - a1
  b = round(5 * self.q / 8) - b0 - b1
  return [a, b]
```

1.7.2. Usando GSW para implementar o *bootstrapping*

Após o NAND homomórfico, obtém-se um criptograma $\mathbf{c} := (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ no qual $b = \mathbf{a}\cdot\mathbf{s} + e + (q/2)m$ e o ruído e satisfaz $|e| < q/4$. O objetivo do *bootstrapping* é decifrar $\mathbf{c}$ homomorficamente e obter um criptograma válido, segundo a definição 1.7.1. Então, o primeiro passo é definir a função de deciframento que será avaliada pela cifra GSW, por isso, considere a seguinte função:

$\mathsf{Dec}_{\mathbf{s}}(\mathbf{a}, b)$: calcule $c' := [q/4 + b - \mathbf{a}\cdot\mathbf{s}]_q$. Devolva 0 se $c' < q/2$ ou 1 caso contrário.

Note que essa função devolve corretamente m, pois $c' = q/4 + e + (q/2)m$, e, como $-q/4 < e < q/4$, vemos que $0 < q/4 + e < q/2$. Portanto, se $m = 0$, então $c' = q/4 + e < q/2$, mas se $m = 1$, então $c' = q/4 + e + q/2$ e vemos que $q/2 < c' < q$.

Então, o primeiro passo do *bootstrapping* é calcular o produto escalar $\mathbf{a}\cdot\mathbf{s}$ e para fazê-lo, temos criptogramas GSW cifrando cada coordenada s_i de $\mathbf{s}$. Esse conjunto de criptogramas é chamado de *chave do bootstrapping* e denotada por bk. Assim, utilizando bk, poderíamos facilmente usar as operações homomórficas do GSW para calcular $\mathsf{GSW.Enc}(\mathbf{a}\cdot\mathbf{s})$ e então $\mathsf{GSW.Enc}(q/4 + b - \mathbf{a}\cdot\mathbf{s})$, no entanto, o segundo passo envolveria avaliar homomorficamente um circuito que compara com $q/2$ e isso seria muito caro.

A principal observação do bootstrapping apresentado em [DM15] é que a comparação $c' < q/2$ é trivial se, em vez de $\mathsf{GSW.Enc}(c')$, calcularmos $\mathsf{GSW.Enc}(X^{c'})$, ou seja, uma encriptação de X elevado a $q/4 + b - \mathbf{a}\cdot\mathbf{s}$ mod q. Note que para isso, basta ser capaz de usar bk e a_i para calcular $\mathsf{GSW.Enc}(X^{-a_i\cdot s_i})$, pois uma vez que esses criptogramas são gerados, pode-se multiplicá-los homomorficamente para gerar

$$\prod_{i=0}^{n-1} \mathsf{GSW.Enc}(X^{-a_i\cdot s_i}) = \mathsf{GSW.Enc}(X^{-\sum_{i=0}^{n-1} a_i\cdot s_i}) = \mathsf{GSW.Enc}(X^{-\mathbf{a}\cdot\mathbf{s}}),$$

e finalmente, basta multiplicar por $X^{q/4+b}$ para obter o criptograma desejado.

Em [DM15], a técnica apresentada para gerar cada $\mathsf{GSW.Enc}(X^{-a_i\cdot s_i})$ envolve $\Theta(\log q)$ produtos homomórficos, portanto, o primeiro passo do *bootstrapping* requer $\Theta(n \log q)$ multiplicações do GSW. Mas os autores de [CGGI16] mostraram que é possível obter $\mathsf{GSW.Enc}(X^{-a_i\cdot s_i})$ com apenas uma multiplicação se supormos que a chave

secreta $\mathbf{s}$ é binária, portanto, reduziram o custo do *bootstrapping* para $\Theta(n)$ multiplicações homomórficas. Além disso, eles introduziram o "produto externo" homomórfico, que é mais rápido que o produto original do GSW por um fator de $\Theta(\log q)$, portanto, obtiveram um *bootstrapping* cerca de $\log^2(q)$ vezes mais rápido que o de [DM15].

Uma vez calculado $\mathsf{GSW.Enc}(X^{q/4+b-\mathbf{a}\cdot\mathbf{s}}) = \mathsf{GSW.Enc}(X^{e'+(q/2)m})$, o segundo passo consiste em transformar esse criptograma em uma encriptação de $\mathsf{Enc}(m)$, o que é obtido por meio de uma transformação relativamente simples, que será explicada na seção 1.7.6. Essa transformação recebe um criptograma cifrado com o problema RLWE e devolve um novo criptograma baseado no problema LWE, como requerido pelo esquema de base, no entanto, ela não gera um criptograma cifrado sob a chave $\mathbf{s}$ do esquema de base, mas sim sob a chave z da cifra GSW, ou, mais especificamente, sob uma chave $\mathbf{z} := (z_0, ..., z_{N-1}) \in \mathbb{Z}^N$, ou seja, uma chave definida como o vetor de coeficientes de z. Para finalmente obter um criptograma válido (segunda a definição 1.7.1), o último passo é aplicar um procedimento chamado *substituição de chaves*[9], com o qual é possível substituir a chave $\mathbf{z}$ pela chave $\mathbf{s}$.

Essas três etapas do *bootstrapping* são ilustradas na figura 1.4. A primeira etapa recebe um criptograma com ruído próximo do limite aceitável pela decifração e a chave de *bootstrapping* bk, que tem pouco ruído. Cada operação homomórfica efetuada durante o *bootstrapping* acumula ruído sobre o ruído já presente em bk, então a quantidade de ruído contida no criptograma recifrado, i.e., devolvido pelo *bootstrapping*, é igual ao ruído inicial de bk mais o ruído adicionado pelas operações. Para que o *bootstrapping* funcione, é preciso que esse ruído final seja menor do que o ruído contido no criptograma de entrada. Mais especificamente, é preciso garantir que o ruído do criptograma devolvido pelo *bootstrapping* seja menor que $q/16$, pois esse é o limite para a magnitude do ruído de um criptograma válido, segundo a definição 1.7.1.

1.7.3. Produto externo

Em [CGGI16], foi introduzida uma multiplicação homomórfica entre um criptograma usual sob o problema RLWE, ou seja, um vetor $(a,b) \in R^2$, e um criptograma do GSW, ou seja, uma matriz $\mathbf{C} \in R^{2\ell\times 2}$. Ela é efetuada decompondo a amostra RLWE e multiplicando por $\mathbf{C}$, ou seja, multiplicando um vetor por uma matriz, o que gera novamente uma amostra RLWE, ou seja, um vetor. Esse produto foi chamado de *produto externo*. A principal observação é que as linhas de um criptograma do GSW são amostras RLWE, então a multiplicação homomórfica do GSW já é composta por uma série de produtos externos. Além disso, como um criptograma GSW é redundante, isto é, ele armazena mais informação do que o necessário para decifrar a mensagem, já que apenas uma linha é usada para a deciframento, então efetuar uma série de produtos externos e obter apenas uma amostra RLWE no fim, é suficiente para implementar o *bootstrapping*, já que um dos últimos passos do *bootstrapping* de [DM15] consistia em extrair uma linha do criptograma GSW.

Primeiramente, precisamos definir um criptograma RLWE:

Definição 1.7.2. *Dizemos que* $\mathbf{c} := (a,b) \in R_q^2$ *é um* criptograma RLWE *que cifra uma mensagem* $m \in \{0,1\}$ *com ruído* $e \in R$ *e sob uma chave* $z \in R$ *se* $b = a\cdot z + e + (q/8)m \in R_q$.

[9] Do inglês *key switching*.

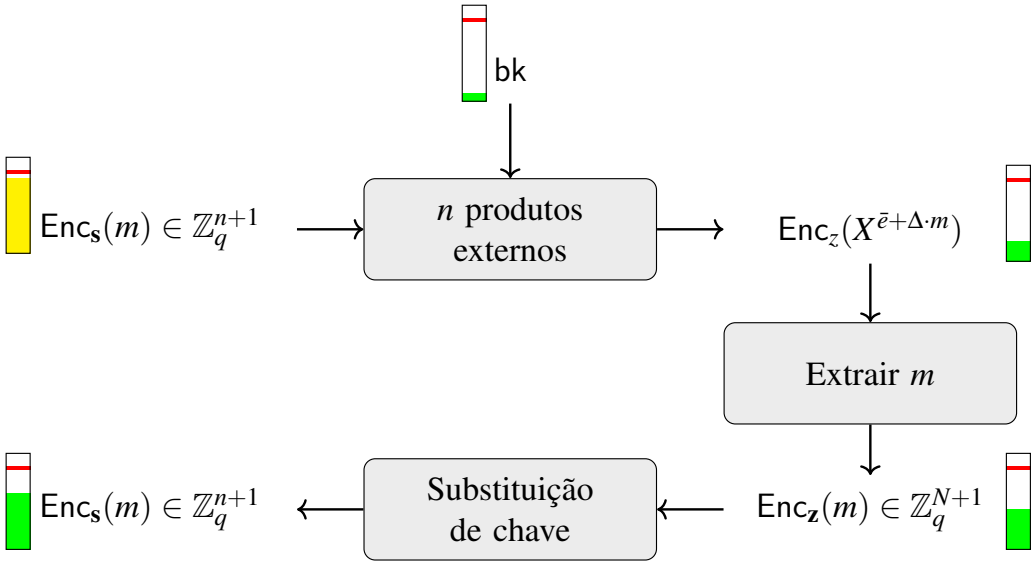

Figure 1.4. Passo a passo da recifração do esquema de base com o GSW. Os retângulos representam o ruído contido nos criptogramas e a linha vermelha representa o limiar que o ruído pode atingir antes de a decifração falhar.

Em vez multiplicar a mensagem por $q/8$, a definição poderia usar qualquer outro valor Δ maior do que a magnitude do ruído para que o deciframento funcionasse, no entanto, o valor $q/8$ é escolhido aqui porque o criptograma obtido no fim da sequência de n produtos externos deve ser transformado em um criptograma válido do esquema de base e, como veremos na seção 1.7.6, esse valor facilita a transformação.

Finalmente, o produto externo é definido como segue:

Definição 1.7.3. *O produto externo entre um criptograma* RLWE $\mathbf{c} \in R_q^2$ *e um criptograma* GSW $\mathbf{C} \in R^{2\ell\times 2}$ *é um criptograma* RLWE $\mathbf{c}'$ *calculado como*

$$\mathbf{c}' := G^{-1}(\mathbf{c}) \cdot \mathbf{C} \in R_q^2.$$

Note que a multiplicação homomórfica do GSW corresponde a 2ℓ produtos externos, onde $\ell = O(\log q)$, logo, cada produto externo é $O(\log q)$ vezes mais barato que uma multiplicação do GSW. Além disso, a corretude e o crescimento do ruído são iguais aos dos produtos do GSW. Como anteriormente, temos $G^{-1}(\mathbf{c}) \cdot \mathbf{G} = \mathbf{c}$, logo, considerando que $\mathbf{c}$ cifra m_0 e $\mathbf{C}$ cifra m_1, o produto externo satisfaz

$$\begin{aligned}
\mathbf{c}' &= G^{-1}(\mathbf{c}) \cdot ([\mathbf{a},\ \mathbf{b}] + m_1\mathbf{G}) \\
&= [G^{-1}(\mathbf{c}) \cdot \mathbf{a},\ G^{-1}(\mathbf{c}) \cdot \mathbf{b}] + m_1 G^{-1}(\mathbf{c}) \cdot \mathbf{G} \\
&= [G^{-1}(\mathbf{c}) \cdot \mathbf{a},\ G^{-1}(\mathbf{c}) \cdot \mathbf{b}] + m_1 \cdot [a, b] \\
&= [a',\ G^{-1}(\mathbf{c}) \cdot \mathbf{b} + m_1 \cdot b] && (a' := G^{-1}(\mathbf{c}) \cdot \mathbf{a} + m_1 \cdot a) \\
&= [a',\ a'z + \underbrace{G^{-1}(\mathbf{c}) \cdot \mathbf{e} + m_1 \cdot e}_{e'} + (q/8)m_0 \cdot m_1]
\end{aligned}$$

Note que o ruído e' é essencialmente o mesmo que o ruído gerado por uma multiplicação do GSW, como mostrado na equação 1, portanto, uma sequência de n produtos

externos aumenta o ruído assim como mostrado no lema 1.5.3 e no corolário 1.6.

O código a seguir implementa o produto externo e deve ser incluído na definição da classe GSW, apresentada no código 1.3.

```
  def extern_prod(self, c, C):
    decomp = self.inv_g_row_ciphertex(c)
    return decomp * C
```

1.7.4. Geração da chave de *bootstrapping*

A chave de *bootstrapping* é um conjunto de criptogramas GSW, em que cada um cifra uma entrada s_i da chave secreta $\mathbf{s}$ do esquema de base. Então, definimos

$$\mathsf{bk} := \{\mathsf{bk}_i := \mathsf{GSW.Enc}(s_i) : 0 \leq i < n\}.$$

Com a classe GSW definida no código 1.3, a geração de bk pode ser feita simplesmente como `bk = [gsw.enc(s[i]) for i in range(n)]`, mas para facilitar a implementação do *bootstrapping*, agruparemos a geração de bk e outras funções relacionadas na classe `Bootstrapper` definida a seguir:

```
# -*- coding: utf-8 -*-
load("lwe_base_scheme.sage")
load("GSW.sage")
load("utils.sage")

class Bootstrapper:
  # gsw: cifra usada como acumulador
  # lwe_base_scheme: gera criptogramas a serem recifrados
  def __init__(self, _gsw, lwe_scheme):
    assert(_gsw.q == lwe_scheme.q)
    self.gsw = _gsw
    self.base_scheme = lwe_scheme
    self.one = self.gsw.G
    self.boot_key_gen()
    self.key_swt_gen() # a ser definido

  def boot_key_gen(self):
    s = self.base_scheme.sk # vetor binário de dimensão n
    n = len(s)
    self.bk = [self.gsw.enc(s[i]) for i in range(n)]
```

Algoritmo 1.5. Definição da classe *Bootstrapper*.

1.7.5. Calculando $\mathsf{Enc}(X^{\bar{e}+\Delta\cdot m})$

Como explicado anteriormente, a primeira etapa do *bootstrapping* recebe um criptograma do esquema de base, $\mathbf{c} = (\mathbf{a}, b) \in \mathbb{Z}_q^n$ e usa a chave bk para calcular um criptograma RLWE de $X^{b-\mathbf{a}\cdot\mathbf{s}}$. No entanto, é preciso que o cálculo feito no expoente seja realizado módulo

q, mas a ordem de X em R é $2N$, que não necessariamente é igual a q. Isto é, como as operações em R são calculadas módulo X^N+1, então $X^N=-1$ em R e $X^{2N}=(X^N)^2=1$. Sendo assim, todas as operações realizadas no expoente de X são feitas módulo $2N$. Por exemplo, se multiplicarmos X^{N+2} com X^{N+5} em R, obteremos $X^{2N+7}=X^{2N}\cdot X^7=X^7$.

Poderíamos restringir a escolha de parâmetros para garantir que q fosse igual a $2N$, no entanto, isso tornaria o *bootstrapping* menos flexível. Em vez disso, antes de começarmos a operar com $\mathbf{c}$, primeiro o multiplicamos por $2N/q$, o que muda o módulo de q para $2N$, ou seja, definimos

$$\mathbf{c}' := (\mathbf{a}', b') := \left(\left\lfloor \frac{2N\mathbf{a}}{q} \right\rceil, \left\lfloor \frac{2Nb}{q} \right\rceil \right).$$

Note que existe $u \in \mathbb{Z}$ tal que $b = \mathbf{a}\cdot\mathbf{s} + e + (q/2)m + u\cdot q \in \mathbb{Z}$, logo,

$$2Nb/q = (2N\mathbf{a}/q)\cdot\mathbf{s} + 2Ne/q + Nm + u\cdot 2N.$$

Além disso, como para todo $r \in \mathbb{R}$ existe um $|\varepsilon| \leq 1/2$ tal que $\lfloor \alpha \rceil = \alpha + \varepsilon$, temos que

$$\lfloor 2Nb/q \rceil = \lfloor 2N\mathbf{a}/q \rceil \cdot \mathbf{s} + \underbrace{2Ne/q + \varepsilon\cdot\mathbf{s} + \varepsilon'}_{e'} + Nm + u\cdot 2N.$$

Ou seja, se o erro e originalmente em $\mathbf{c}$ era menor do que $q/4$, então agora o erro e' em $\mathbf{c}'$ é praticamente igual a $2Ne/q$, logo, menor que $(2N/q)\cdot(q/4) = N/2$ (com grande probabilidade). Assim, $\mathbf{c}' = (\mathbf{a}', b')$ cifra m módulo $2N$. Além disso, se $\mathbf{c}$ é decifrável módulo q, então $\mathbf{c}'$ também o é, mas sobre $\mathbb{Z}_{2N}$, exatamente como necessário para o *bootstrapping*.

O código a seguir implementa essa operação e deve ser incluído na classe *Bootstrapper* definida no algoritmo 1.5.

```
# Recebe um criptograma LWE (a, b) em Z_q^(n+1) e devolve
# outro criptograma LWE que cifra a mesma mensagem, mas
# é definido módulo 2*N
def mod_switch_to_2N(self, a, b):
  q, N = self.base_scheme.q, self.gsw.n
  _a = vector([round(ZZ(ai) * 2 * N / q) for ai in a])
  _b = round(ZZ(b) * 2*N / q)
  return _a, _b
```

Agora, dado $\mathbf{c}' = (\mathbf{a}', b')$ e bk, nosso primeiro objetivo é obter uma encriptação RLWE de $X^{-a'_i\cdot s_i}$, para $0 \leq i < n$. Para isso, note que como s_i é binário, então substituindo $s_i = 0$ e $s_i = 1$ dos dois lados da seguinte igualdade, vemos que ela é válida:

$$X^{-a'_i\cdot s_i} = 1 + (X^{-a'_i} - 1)\cdot s_i.$$

Em [CGGI16], a operação que produz uma encriptação de $X^{-a'_i\cdot s_i}$ a partir de a'_i e de uma encriptação GSW de s_i foi chamada de cmux. Para calculá-la homomorficamente, substitui-se o 1 por $\mathbf{G}$ e s_i por bk_i na expressão anterior, logo, temos o seguinte:

$$\mathsf{cmux}(a'_i, \mathsf{bk}_i) := \mathbf{G} + (X^{-a'_i} - 1)\cdot \mathsf{bk}_i \in R_q^{2\ell\times\ell}. \tag{5}$$

Note que como bk_i é da forma $[\mathbf{a}, \mathbf{b}] + s_i\mathbf{G}$, temos

$$\begin{aligned}\mathsf{cmux}(a'_i, \mathsf{bk}_i) &= \mathbf{G} + (X^{-a'_i} - 1) \cdot [\mathbf{a}, \mathbf{b}] + (X^{-a'_i} - 1)s_i\mathbf{G} \\ &= \mathbf{G} + [\mathbf{a}', \mathbf{b}'] + (X^{-a'_i} - 1)s_i\mathbf{G} \\ &= [\mathbf{a}', \mathbf{b}'] + (1 + (X^{-a'_i} - 1)s_i)\mathbf{G} \\ &= [\mathbf{a}', \mathbf{b}'] + X^{-a'_i \cdot s_i} \cdot \mathbf{G}.\end{aligned}$$

Ou seja, $\mathsf{cmux}(a'_i, \mathsf{bk}_i)$ é uma encriptação GSW de $X^{-a'_i \cdot s_i}$, como desejado. Além disso, se $\mathbf{e}$ é o ruído contido em bk_i, então o ruído de $\mathsf{cmux}(a'_i, \mathsf{bk}_i)$ é igual a $\mathbf{e}' := (1 + (X^{-a'_i} - 1) \cdot \mathbf{e}$. Assim sendo, vemos que $\|\mathbf{e}'\| \leq \|\mathbf{e}\| + \left\|X^{-a'_i} \cdot \mathbf{e}\right\| + \|-\mathbf{e}\| = 3\|\mathbf{e}\|$. Portanto, o cmux praticamente não aumenta o ruído.

Para completar o primeiro passo do *bootstrapping*, usamos o produto interno para multiplicar $X^{N/2+b'}$ por todas as encriptações de $X^{-a'_i \cdot s_i}$. Primeiramente, note que $(0, (q/8) \cdot X^{N/2+b'}) \in R_q^2$ é um criptograma RLWE trivial, pois satisfaz a Definição 1.7.2 com ambos a e e iguais a zero. Assim, podemos usar o produto externo para multiplicar $(0, (q/8) \cdot X^{N/2+b'})$ por $\mathsf{cmux}(a'_0, \mathsf{bk}_0)$ e obter uma encriptação RLWE de $X^{N/2+b'-a'_0 \cdot s_0}$, que pode então ser multiplicada por $\mathsf{cmux}(a'_1, \mathsf{bk}_1)$ com outro produto externo, o que gera uma encriptação de $X^{N/2+b'-a'_0 \cdot s_0 - a'_1 \cdot s_1}$, e assim sucessivamente. Ou seja, a primeira etapa do *bootstrapping* consiste em calcular

$$(0, (q/8) \cdot X^{N/2+b'}) \prod_{i=0}^{n} \mathsf{cmux}(a'_i, \mathsf{bk}_i),$$

onde cada multiplicação é efetuada com um produto externo.

É fácil ver que os expoentes são adicionados módulo $2N$. Portanto, como $b' = \mathbf{a}' \cdot \mathbf{s} + e' + Nm \bmod 2N$, o resultado obtido é um criptograma RLWE da forma $(a, b) \in R_q^2$ com $b = a \cdot z + e + (q/8) \cdot X^y$ e

$$y = N/2 + b' - \mathbf{a}' \cdot \mathbf{s} = \underbrace{N/2 + e'}_{\bar{e}} + Nm \pmod{2N}$$

Lembre-se que $-N/2 < e' < N/2$, logo, tomando $\bar{e} := N/2 + e'$, temos $0 < \bar{e} < N$. Essa inequação é fundamental para a transformação usada no segundo passo do *bootstrapping*.

Em suma, para implementar essa primeira etapa do *bootstrapping*, podemos adicionar o seguinte código à classe *Bootstrapper*:

```
  #  acc: criptograma RLWE cifrando alguma mensagem X^z
  #  a_i: um inteiro módulo 2*N
  #  bk_i: um criptograma GSW cifrando um bit s_i.
  #  Devolve um criptograma RLWE cifrando X^(z - a_i * s_i)
  def cmux_gate(self, acc, a_i, bk_i):
    N = self.gsw.n
    C = self.one + (x^(2*N-a_i) - 1) * bk_i
```

```
    acc = self.gsw.extern_prod(acc, C)
    return acc

  #   Recebe c = (a, b) \in Z_{2N}^{n+1}, i.e., um criptograma LWE
  # definido módulo 2*N e que cifra uma mensagem m sob a chave s.
  #   Devolve um criptograma RLWE cifrando X^(e_bar + N*m)
  # sob a chave z do esquema GSW.
  def linear_part_dec(self, c):
    Q, N, RQ = self.gsw.q, self.gsw.n, self.gsw.Rq
    a, b = c
    acc = [RQ(0), round(Q/8) * RQ(x)^(N // 2 + b)] # a = e = 0
    for i in range(len(a)):
      acc = self.cmux_gate(acc, a[i], self.bk[i])
    return acc
```

1.7.6. Transformando $X^{\bar{e}+N\cdot m}$ em m

Após a sequência de n produtos externos, obtém-se um criptograma RLWE c da forma $c=(a,b)\in R_q^2$ com $b=a\cdot z+e+(q/8)\cdot X^{\bar{e}+N\cdot m}\in R_q$, com $0<\bar{e}<N$. O segundo passo do *bootstrapping* consiste em transformar c em um criptograma $\mathbf{c}$ que cifra m e é baseado no problema LWE em vez do RLWE. Essa transformação consiste basicamente de um produto escalar entre o vetor de coeficientes de b e o vetor $\mathbf{u}=(-1,...,-1)\in\mathbb{Z}^N$. Para entender como ela funciona, primeiro é necessário entender como adições e multiplicações em R podem calculadas usando operações entre vetores e matrizes.

1.7.6.1. Matrizes anticirculantes e vetores de coeficientes

Seja $g=\sum_{i=0}^{N-1} g_i\cdot X^i$ um elemento do anel R, definimos o vetor de coeficientes de g como $\phi(g)=(g_0,...,g_{N-1})\in\mathbb{Z}^N$. É fácil ver que se $g,h\in R$, então $\phi(g+h)=\phi(g)+\phi(h)$, logo, os vetores de coeficientes permitem que as adições em R sejam representadas por somas de vetores. No entanto, as multiplicações não são tão simples e exigem também o que chamamos de matrizes anticirculantes: Para todo $g\in R$, definimos $\Phi(g)$ como a matriz $N\times N$ tal que cada linha $i=0,...,N-1$ é igual a $\phi(x^i\cdot g \bmod x^N+1)$. Note que como $x^N=-1$ em R, então $x\cdot g=g_0\cdot X+...+g_{N-2}\cdot X^{N-1}-g_{N-1}\in R$, portanto, $\phi(x\cdot g)=(-g_{N-1},g_0,g_1,...,g_{N-2})$, ou seja, é uma rotação cíclica de $\phi(g)$, mas com o último coeficiente multiplicado por -1. Em geral, $\phi(x^i\cdot g)$ multiplica as últimas i posições de $\phi(g)$ por -1 e rotaciona todas as entradas em i posições. Logo, temos que a matriz anticirculante de g é igual a

$$\Phi(g)=\begin{pmatrix} g_0 & g_1 & g_2 & \cdots & g_{N-1} \\ -g_{N-1} & g_0 & g_1 & \cdots & g_{N-2} \\ -g_{N-2} & -g_{N-1} & g_0 & \cdots & g_{N-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -g_1 & -g_2 & -g_3 & \cdots & g_0 \end{pmatrix}\in\mathbb{Z}^{n\times n}.$$

Agora, é fácil representar o produto de polinômios módulo X^N+1 usando de

ϕ e Φ. Primeiro, note que $\phi(X^i)$ é um vetor com um único 1 na i-ésima posição, i.e., $\phi(X^i) = (0,0,...,0,1,0,...,0)$. Então, $\phi(X^i)\Phi(h)$ é igual a i-ésima linha de $\Phi(h)$, que por definição, é $\phi(X^i h)$, ou seja, $\phi(X^i)\cdot\Phi(h) = \mathrm{lin}_i(\Phi(h)) = \phi(X^i h)$. Com isso, pode-se provar o seguinte lema:

Lema 1.7.1. *Para todo $g,h \in R$, temos que $\phi(gh) = \phi(g)\Phi(h)$*

Proof. Como ϕ é aditivo, é fácil ver que $\phi(g) = \sum_{i=0}^{N-1}\phi(g_i X^i) = \sum_{i=0}^{N-1} g_i\phi(X^i)$. Assim,

$$\phi(g)\Phi(h) = \sum_{i=0}^{N-1} g_i\phi(X^i)\Phi(h) = \sum_{i=0}^{N-1} g_i\phi(X^i h) = \sum_{i=0}^{N-1}\phi(g_i X^i h) = \phi\left(\sum_{i=0}^{N-1} g_i X^i h\right) = \phi(gh)$$

□

Essas funções podem ser implementadas em Sage como a seguir:

```
# Devolve vetor que representa g mod X^N + 1.
def poly_to_vec(g, N):
  f = x^N+1
  v = (g % f).coefficients(sparse=False)
  return vector(v + [0]*(N - len(v)))

# Dado a \in R, devolve matriz A tal que para todo h \in R
#     poly_to_vec(h) * A = poly_to_vec(h*a).
def poly_to_mat(a, N):
  A = Matrix(ZZ, [poly_to_vec(a*x^i, N) for i in range(N)])
  return A
```

1.7.6.2. A transformação RLWE $\rightarrow$ LWE

Agora, usando vetores de coeficientes e matrizes anticirculantes, podemos finalmente mostrar como efetuar o segundo passo do *bootstrapping*. Primeiro note que usando o lema 1.7.1, é fácil provar o seguinte:

Corolário 1.8. *Seja $b = a\cdot z + e + (q/8)\cdot X^j \in R_q$, então, $\phi(b) = \phi(z)\cdot\Phi(a) + \phi(e) + (q/8)\cdot\phi(X^j)$.*

A principal observação agora é que o criptograma obtido após os produtos externos cifra $X^{\bar{e}+Nm}$ com $0 < \bar{e} < N$. Portanto, temos

$$m = 0 \implies X^{\bar{e}+Nm} = X^{\bar{e}}$$

e

$$m = 1 \implies X^{\bar{e}+Nm} = -1\cdot X^{\bar{e}}.$$

Isso significa que $\phi(X^{\bar{e}+Nm})$ é um vetor da forma $(0,..,0,1,0,...,0)$ se $m = 0$ e da forma $(0,..,0,-1,0,...,0)$ se $m = 1$. Mas então, se definirmos $\mathbf{u} = (1,1,...,1) \in \mathbb{Z}^N$, temos

$$m = 0 \implies \phi(X^{\bar{e}+Nm})\cdot\mathbf{u} = -1 = 2\cdot m - 1$$

e

$$m = 1 \implies \phi(X^{\bar{e}+Nm}) \cdot \mathbf{u} = 1 = 2 \cdot m - 1.$$

Ou seja, como $m \in \{0,1\}$, a equação $\phi(X^{\bar{e}+Nm}) \cdot \mathbf{u} = 2 \cdot m - 1$ é sempre válida.

Assim, dado o criptograma (a,b) com $b = a \cdot z + e + (q/8)X^{\bar{e}+Nm} \in R_q$, temos o seguinte sobre $\mathbb{Z}_q$:

$$\begin{aligned}
\phi(b) \cdot \mathbf{u} &= \phi(z) \cdot \Phi(a) \cdot \mathbf{u} + \mathbf{e} \cdot \mathbf{u} + (q/8)\phi(X^{\bar{e}+Nm}) \cdot \mathbf{u} && \text{(corolário 1.8)} \\
&= \mathbf{a}' \cdot \phi(z) + \mathbf{e} \cdot \mathbf{u} + (q/8)\phi(X^{\bar{e}+Nm}) \cdot \mathbf{u} && (\mathbf{a}' := \Phi(a) \cdot \mathbf{u}) \\
&= \mathbf{a}' \cdot \mathbf{z} + \mathbf{e} \cdot \mathbf{u} + (q/8)\phi(X^{\bar{e}+Nm}) \cdot \mathbf{u} && (\mathbf{z} := \phi(z)) \\
&= \mathbf{a}' \cdot \mathbf{z} + \mathbf{e} \cdot \mathbf{u} + (q/8) \cdot (2m-1).
\end{aligned}$$

Lembre-se que em vez de $2m-1$, queremos obter uma encriptação de m. Além disso, em vez de $q/8$, a mensagem precisa ser multiplicada por $q/4$ para ser um criptograma válido segundo a definição 1.7.1. Então, o que resta fazer é simplesmente adicionar $q/8$ à $\phi(b) \cdot \mathbf{u}$. Assim, obtém-se:

$$\phi(b) \cdot \mathbf{u} + q/8 = \mathbf{a}' \cdot \mathbf{z} + \mathbf{e} \cdot \mathbf{u} + (q/4) \cdot m.$$

Em suma, dado o criptograma (a,b) com $b = a \cdot z + e + (q/8)X^{\bar{e}+Nm} \in R_q$, o segundo passo do *bootstrapping* devolve $(\mathbf{a}', b') \in \mathbb{Z}_q^{N+1}$ onde

$$\mathbf{a}' := \Phi(a) \cdot \mathbf{u} \in \mathbb{Z}_q^N \quad \text{e} \quad b' := \phi(b) \cdot \mathbf{u} + q/8 \in \mathbb{Z}_q.$$

E vemos que $b' = \mathbf{a} \cdot \mathbf{z} + e' + (q/4)m \in \mathbb{Z}_q$ com $e' := \mathbf{e} \cdot \mathbf{u}$. Em outras palavras, $(\mathbf{a}', b')$ cifra m sob a chave $\mathbf{z}$.

Essa etapa do *bootstrapping* pode ser implementada da seguinte forma em Sage, considerando que este código é adicionado à definição da classe *Bootstrapper*. Note que ele usa as funções definidas no código 1.7.6.1, que deve então ser importado.

```
  def convert_rlwe_lwe(self, acc):
    N, Q = self.gsw.n, self.gsw.q
    a, b = Zx(acc[0].lift()), Zx(acc[1].lift())
    u = vector([-1] * N)

    lwe_a = (poly_to_mat(a, N) * u) % Q
    lwe_b = poly_to_vec(b, N) * u
    lwe_b = (lwe_b + round(Q/8)) % Q

    return [lwe_a, lwe_b] # in Z_Q^(N+1)
```

1.8.1. Substituição de chaves

Com as duas primeiras etapas do *bootstrapping*, já se pode transformar um criptograma $\mathbf{c}$ do esquema de base, no nível zero e com muito ruído, em um criptograma $\mathbf{c}'$ também baseado no problema LWE, no nível um e com pouco ruído. No entanto, $\mathbf{c}$ é cifrado sob a

chave secreta $\mathbf{s}$, mas $\mathbf{c}'$ usa a chave $\mathbf{z} := \phi(z)$, onde z é a chave secreta cifra GSW. Além disso, $\mathbf{c}$ é definido usando o problema LWE com dimensão n, enquanto $\mathbf{c}'$ tem dimensão N. Portanto, não é possível aplicar a porta NAND homomórfica a $\mathbf{c}'$ e criptogramas do esquema de base.

Uma forma de resolver esse problema é escolher $N = n$ e $\mathbf{s} = \phi(z)$. Porém, essa abordagem restringe muito a escolha de parâmetros e a eficiência do *bootstrapping*, pois, em geral, N precisa ser um valor grande para garantir a segurança (e.g., os artigos [DM15, CGGI16, Per21a] usam $N = 1024$), enquanto n pode ser escolhido com um valor muito menor, digamos, cerca de 600. Como a quantidade de produtos externos e de criptogramas em bk são lineares em n, é desejável escolher $\mathbf{s}$ e z de forma independente.

Então, assumindo que $\mathbf{s} \neq \phi(z)$, é preciso de alguma forma transformar $\mathbf{c}'$ em uma encriptação da mesma mensagem, mas sob a chave $\mathbf{s}$ e sem aumentar muito o ruído. Isso é feito com um procedimento chamado *substituição de chaves*, que funciona da seguinte forma: a chave de *bootstrapping* cifra $\mathbf{s}$ sob z. Então, de certa forma, fechamos esse círculo criando uma chave ksk que cifra $\mathbf{z}$ sob $\mathbf{s}$. Obviamente, a geração de ksk deve ser feita de forma privada, no entanto, a chave ksk é pública. Então, usamos ksk para "substituir" a chave de $\mathbf{c}'$ por $\mathbf{s}$.

Para reduzir o crescimento do ruído durante a substituição de chaves, usamos algo parecido com a decomposição usada no GSW, então, ksk cifra a chave $\mathbf{z}$ multiplicada por potências de dois. Logo, para $0 \leq i < N$ e $0 \leq j < \lceil \log_2(q) \rceil$, definimos

$$\mathsf{ksk}_{i,j} := (\mathbf{a}_{i,j}, b_{i,j} := \mathbf{a}_{i,j} \cdot \mathbf{s} + e_{i,j} + 2^j \cdot z_i) \in \mathbb{Z}_q^{n+1}.$$

Note que cada $\mathsf{ksk}_{i,j}$ cifra $2^j \cdot z_i$ com o problema LWE usando a dimensão n do esquema de base. Finalmente, definimos $\mathsf{ksk} := \{\mathsf{ksk}_{i,j} : 0 \leq i < N \text{ e } 0 \leq j < \lceil \log_2(q) \rceil\}$.

O código que implementa a geração da chave de substituição de chaves é apresentado a seguir e também deve ser incluído na definição da classe *Bootstrapper*:

```
def key_swt_gen(self):
  N, Q = self.gsw.n, self.gsw.q
  l = ceil(log(Q, 2))
  z = poly_to_vec(self.gsw.sk, N)

  # K_{i, j} = enc(z_i * 2^j)
  K = [0] * N
  for i in range(N):
    zi = z[i]
    enc_zi = [0] * l
    for j in range(l):
      c = self.base_scheme.enc(0)
      c[1] = (c[1] + 2^j * zi) % Q
      enc_zi[j] = c
    K[i] = enc_zi
  self.ksk = K
```

Dado um criptograma $(\mathbf{a}, b := \mathbf{a}\cdot\mathbf{z} + e + (q/4)m) \in \mathbb{Z}_q^{N+1}$, a ideia principal da substituição de chaves é que ao subtrair $\mathbf{a}\cdot\mathbf{z} + \hat{\mathbf{a}}\cdot\mathbf{s}$ de b, obtém-se um inteiro da forma $\hat{b} = -\hat{\mathbf{a}}\cdot\mathbf{s} + e + (q/4)m$, logo, $(-\hat{\mathbf{a}}, \hat{b}) \in \mathbb{Z}_q^{n+1}$ é um criptograma válido sob a chave $\mathbf{s}$ em vez de $\mathbf{z}$.

Como $\mathbf{a}\cdot\mathbf{z} = \sum_{i=0}^{N-1} a_i \cdot z_i$, usamos ksk para subtrair cada $a_i \cdot z_i$ de b da seguinte forma: Decompomos a_i em base 2 e obtemos $(a_{i,0}, ..., a_{i,L-1}) \in \{0,1\}^L$, onde $L := \lceil \log q \rceil$. Então, note que como $\mathsf{ksk}_{i,j} = (\mathbf{a}_{i,j}, b_{i,j} := \mathbf{a}_{i,j}\cdot\mathbf{s} + e_{i,j} + 2^j \cdot z_i) \in \mathbb{Z}_q^{n+1}$, se multiplicarmos $a_{i,j}$ por $\mathsf{ksk}_{i,j}$, obtemos uma encriptação de $a_{i,j}\cdot 2^j \cdot z_i$ sob a chave $\mathbf{s}$. Então, somando esses criptogramas, obtemos uma encriptação de $\sum_{j=0}^{L-1} a_{i,j}\cdot 2^j \cdot z_i = a_i \cdot z_i$, como desejado. Finalmente, basta subtrair cada encriptação de $a_i \cdot z_i$ de b.

Ou seja, a substituição de chaves consiste em calcular o seguinte criptograma:

$$(\hat{\mathbf{a}}, \hat{b}) := (\mathbf{0}, b) - \sum_{i=0}^{N-1}\sum_{j=0}^{L-1} a_{i,j}\cdot \mathsf{ksk}_{i,j}.$$

Note que se e é o ruído de $(\mathbf{a}, b)$ e se cada $\mathsf{ksk}_{i,j}$ tem ruído próximo de β, então o ruído de $(\hat{\mathbf{a}}, \hat{b})$ é $O(\|e\| + N\cdot L\cdot\beta)$.

O procedimento de substituição de chaves é implementado em Sage com o seguinte método da classe *Bootstrapper*:

```
#   Recebe um criptograma LWE (a, b) que cifra m sob a
# chave z do GSW e devolve uma encriptação de m sob a
# chave s do esquema de base.
def switch_key(self, c):
  a, b = c
  q, n = self.base_scheme.q, self.base_scheme.n
  L = ceil(log(q, 2))
  K = self.ksk
  out_a = vector([0]*n)
  out_b = 0
  for i in range(len(a)):
    v = vector(ZZ(a[i]).digits(base=2, padto=L)) # decompõe ai
    for j in range(L):
      if 1 == v[j]:
        out_a += K[i][j][0] # adiciona "termo a" de ksk
        out_b += K[i][j][1] # adiciona "termo b" de ksk

  # Agora, out_b = out_a * s + e + a * z
  # então, b - out_b = -out_a*s - e + e_b + (q / 4) * m
  out_b = (b - out_b) % q
  out_a = (-out_a ) % q
  return [out_a, out_b]
```

1.8.2. *Bootstrapping*: o procedimento completo

Para implementar o *bootstrapping*, agora basta agrupar como na figura 1.4 os três procedimentos implementados nas seções anteriores. Isso é feito com a seguinte função, que deve também ser incluída na definição da classe *Bootstrapper*:

```
def refresh(self, c):
    a, b = self.mod_switch_to_2N(c[0], c[1])
    acc = self.linear_part_dec([a, b]) # enc_z(X^(e_bar + N*m))

    c = self.convert_rlwe_lwe(acc) # enc_z(m) in Z_q^(N+1)

    c = self.switch_key(c) # enc_s(m) in Z_q^(n+1)

    return c
```

Como especificado na definição 1.7.1, para que o criptograma devolvido pelo *bootstrapping* seja válido, é preciso que seu ruído seja menor que $q/16$. Então, agora analisaremos o ruído inserido pelas operações feitas durante o *bootstrapping* e mostraremos como podemos escolher os parâmetros para que o ruído final do criptograma recifrado seja aceitável.

Como ilustrado na figura 1.4, o *bootstrapping* recebe como entrada a chave bk, que consiste em um conjunto de criptogramas com pouquíssimo ruído, então, cada uma das três etapas acumula ruído sobre bk.

Como os criptogramas de bk têm seus ruídos gerados com uma distribuição gaussiana discreta cujo desvio-padrão é σ, podemos dizer que o ruído de bk tem magnitude $O(\sigma)$. Assim, o ruído aumenta durante o *bootstrapping* da seguinte forma:

- A primeira etapa consiste de n produtos externos envolvendo criptogramas gerados pelo cmux. Lembre-se que o cmux no máximo multiplica o ruído por 3, então o ruído de cada criptograma envolvido nos produtos externos ainda é $O(\sigma)$. Logo, conforme mostrado no corolário 1.6, que também é válido para produtos externos, o ruído final dessa etapa é $O(n\sigma N\log(q))$.
- Na segunda etapa, o ruído é multiplicado pelo vetor $\mathbf{u} = (-1,...,-1) \in \mathbb{Z}^N$, então, sua magnitude pode ser limitada por $\sum_{i=1}^{N} O(n\sigma N\log(q)) = O(n\sigma N^2\log(q))$.
- Como discutido na seção 1.8.1, a terceira e última etapa adiciona ao ruído um termo igual a $O(N\beta\log q)$, onde β é um limitante superior para o ruído da chave ksk. Em geral, $\beta = O(1)$, portanto, o ruído final do criptograma recifrado é

$$O(n\sigma N^2\log(q) + N\log q) = O(n\sigma N^2\log(q)).$$

É necessário que o ruído final seja menor que $q/16$, portanto, basta escolher

$$q = \Theta(N^3\log N)$$

se assumirmos $N > n$ e $\sigma = O(1)$.

Essa análise do crescimento do ruído pode ser vista como uma análise de pior caso, pois ela foi derivada limitando somas do tipo $e_1 + ... + e_k$ por $k \cdot \max(e_1, ..., e_k)$. Mas foi observado em [ASP14, DM15] que, como os ruídos seguem uma distribuição gaussiana, o valor esperado para somas dessa forma é na verdade $\sqrt{k} \cdot \max(e_1, ..., e_k)$. Com isso, é possível fazer uma análise de caso médio da seguinte forma:

- A magnitude esperada para o ruído obtido na primeira etapa é $O(\sigma \cdot \sqrt{nN\log(q)})$.
- Na segunda etapa, espera-se que a magnitude do ruído seja $\sum_{i=1}^{N} O(\sigma \cdot \sqrt{nN\log(q)}) = O(\sigma N \cdot \sqrt{n\log(q)})$.
- A terceira etapa então adiciona o termo $O(\beta \cdot \sqrt{N\log q})$, onde $\beta = O(1)$ é um limitante para o ruído de ksk. Então, com grande probabilidade, o ruído no final do *bootstrapping* é limitado por

$$O(\sigma N \cdot \sqrt{n\log(q)} + \sqrt{N\log q}) = O(\sigma N \cdot \sqrt{n\log(q)}).$$

Assim, novamente tomando $N > n$ e $\sigma = O(1)$, reduz-se q para $\Theta(N^{1.5}\log N)$.

Para testar o *bootstrapping*, o seguinte código em Sage pode ser usado:

```
def test_bootstrapp(k=5):
    n, N, sigma, = 2^3, 2^5, 3.2
    B, l = 2, 17
    q = Q = B^l # assume q = Q
    lwe_scheme = LWEScheme(n, q, sigma)
    gsw = GSW(N, Q, sigma, B)
    print("%s\n%s" % (lwe_scheme, gsw))

    boot = Bootstrapper(gsw, lwe_scheme)

    m = random_bit()
    c = lwe_scheme.enc(m)
    for i in range(1, k+1):
        mi = random_bit()
        m = (1 - m * mi)
        ci = lwe_scheme.enc(mi)
        cnand = lwe_scheme.nand(c, ci)

        c = boot.refresh(cnand)

        dec_m = lwe_scheme.dec(c, level=1)
        assert(m == dec_m) # verifica se c é válido

        noise = lwe_scheme.get_noise(c, m, level=1)
        print("Refreshed noise: %f" % noise)
```

Note que os valores de n, N e q usados nesse código são muito pequenos para garantir que as cifras sejam seguras, mas já são suficientes para garantir a corretude do *bootstrapping*. Para um nível de segurança de 128 bits, parâmetros típicos são $n \approx 600$, $N \approx 1024$ e $q \approx 2^{32}$ [DM15, CGGI16]. Obviamente, a implementação apresentada neste curso tem como finalidade ser clara e fácil de entender, o que faz com que ela seja ineficiente. Assim, usar tais parâmetros com essa implementação faz com que o *bootstrapping* seja lento. Mas as implementações apresentadas em [DM15] e em [CGGI16] usam a linguagem *C++* e bibliotecas altamente optimizadas para multiplicar polinômios, logo, conseguem executar o *bootstrapping* com nível de segurança $\lambda = 128$ em cerca de 0.1 segundo em um computador comercial comum, em uma única *thread*.

Se considerarmos que os cálculos homomórficos serão normalmente executados em servidores "na nuvem", que são muito mais potentes que computadores usais, e também que esses procedimentos são altamente paralelizáveis, essa estratégia de computação homomórfica feita com a avaliação de uma porta lógica seguida de um *bootstrapping* extremamente rápido se mostra muito promissora.

1.9. Conclusão

Neste minicurso foi apresentado código Sage que implementa criptografia completamente homomórfica. Isto permite ao estudante experimentar diferentes parâmetros e desenvolver novas aplicações. O uso da linguagem Sage torna a compreensão mais fácil, enquanto ao mesmo tempo torna possível a prototipagem rápida das funções necessárias. Componentes que possivelmente tenham sido omitidos no texto por falta de espaço podem ser encontrados no repositório [Per21b].

References

[ACC+17] David Archer, Lily Chen, Jung Hee Cheon, Ran Gilad-Bachrach, Roger A. Hallman, Zhicong Huang, Xiaoqian Jiang, Ranjit Kumaresan, Bradley A. Malin, Heidi Sofia, Yongsoo Song, and Shuang Wang. Applications of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3), 2015.

[ASP14] Jacob Alperin-Sheriff and Chris Peikert. Faster bootstrapping with polynomial error. In *Advances in Cryptology – CRYPTO 2014*, pages 297–314, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[BDH+17] Michael Brenner, Wei Dai, Shai Halevi, Kyoohyung Han, Amir Jalali, Miran Kim, Kim Laine, Alex Malozemoff, Pascal Paillier, Yuriy Polyakov, Kurt Rohloff, Erkay Savaş, and Berk Sunar. A Standard API for RLWE-based Homomorphic Encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[BGN05] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF formulas on ciphertexts. In *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption Without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.

[BMMP18] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Advances in Cryptology - CRYPTO 2018*, volume 10993 of *Lecture Notes in Computer Science*, pages 483–512. Springer, 2018.

[CCD+17] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[CKLY15] Jung Hee Cheon, Jinsu Kim, Moon Sung Lee, and Aaram Yun. Crt-based fully homomorphic encryption over the integers. *Inf. Sci.*, 310(C):149–162, July 2015.

[CN12] Yuanmi Chen and Phong Nguyen. Faster Algorithms for Approximate Common Divisors: Breaking Fully Homomorphic Encryption Challenges over the Integers. In *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*. Springer, 2012.

[CNT12] Jean-Sébastien Coron, David Naccache, and Mehdi Tibouchi. Public key compression and modulus switching for fully homomorphic encryption over the integers. In *Advances in Cryptology – EUROCRYPT 2012*, pages 446–464, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CS15] Jung Hee Cheon and Damien Stehlé. Fully Homomophic Encryption over the Integers Revisited. In *EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Sofia, Bulgaria, April 2015.

[DM12] R. Dahab and E. M. Morais. Encriptação homomórfica. In A. L. (Org.) Santos, A. (Org.) Santin, C. (Org.) Maziero, and P. A. S. Gonçalves, editors, *Minicursos do XII Simpósio em Segurança da Informação e de Sistemas Computacionais. 12ed.*, pages 1–195, 2012.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `https://eprint.iacr.org/2012/144`.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[GGM16] Steven D. Galbraith, Shishay W. Gebregiyorgis, and Sean Murphy. Algorithms for the approximate common divisor problem. *LMS Journal of Computation and Mathematics*, 19(A), 2016.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[Per21a] Hilder Vitor Lima Pereira. Bootstrapping fully homomorphic encryption over the integers in less than one second. In *Public-Key Cryptography – PKC 2021*, pages 331–359, Cham, 2021. Springer International Publishing.

[Per21b] Hilder Vitor Lima Pereira. SAGE scripts for DGHV and GSW. github, 2021. `https://github.com/hilder-vitor/FHE_for_SBSeg21`.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), September 2009.

[Sag20] Sage Developers. *SageMath, the Sage Mathematics Software System*, 2020. `https://www.sagemath.org`.

