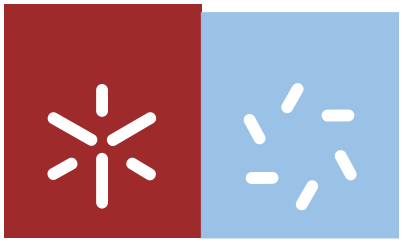
Universidade do Minho
Escola de Ciências

José Miguel Gomes Loureiro

Implementação de funções de hash e cifras de chave pública baseadas em retículos

Outubro de 2012

Implementação de funções de hash e cifras de chave pública baseadas em retículos
José Miguel Gomes Loureiro
UMinho|2012



Universidade do Minho
Escola de Ciências

José Miguel Gomes Loureiro

Implementação de funções de hash e cifras de chave pública baseadas em retículos

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho realizado sob a orientação do
Professor José Carlos Bacelar Almeida
e do
Professor José Pedro Miranda Mourão Patrício

Outubro de 2012

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA DISSERTAÇÃO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, ____/____/_____

Assinatura: _____

Agradecimentos

Em primeiro lugar gostaria de agradecer ao professor Pedro Patrício por me cultivar o interesse por uma área tão importante como a criptografia e a área de teoria de números.

Ao professor José Bacelar por me mostrar as técnicas criptográficas existentes bem como o uso de linguagens de programação para a implementação das mesmas.

Um obrigado aos professores Pedro Patrício e José Bacelar pela sua supervisão, ideias e pela constante motivação que me deram para a escrita da dissertação.

Aos meus colegas Isabel e João, por me ajudarem durante o mestrado e pelos bons momentos que tivemos juntos durante o curso. Serão dois amigos que ficarão para toda a vida.

Aos meus amigos por me apoiarem nos bons e maus momentos, por poder contar com a sua ajuda sempre que necessário.

Um agradecimento aos meus pais por me terem dado conhecimentos e valores que não vêm em livros e por sempre acreditarem em mim.

Ao meu querido avô, que faleceu muito recentemente, que sempre teve a preocupação de saber como é que iam os meus estudos e que foi um grande apoio que ficará para sempre na minha memória e no meu coração.

Aos meus familiares, pelos convívios agradáveis que tivemos, pela sua disponibilidade sempre que necessário, bem como a sua ajuda preciosa em alguns momentos da minha vida.

José Miguel
Braga, 2012

Resumo

Nesta dissertação são estudados os algoritmos e as respetivas implementações de construções criptográficas baseadas em retículos.

Os algoritmos estudados são relativos às funções de *hash*, de onde se destacam a função de *hash* de Ajtai, a função de *hash* baseada em retículos ideais e a função de *hash* `SWIFFT`, e às cifras de chave pública, de onde se destacam a cifra GGH, a cifra NTRU e a cifra LWE. Importa ainda referir o estudo que faremos do algoritmo de redução de base.

Relativamente às implementações, é usada a ferramenta Sage, disponível em

http://www.sagemath.org/,

que usa a linguagem Python.

Abstract

In this dissertation the algorithms and the respective implementations of the cryptographic constructions, based on lattices, are studied.

The algorithms studied are related to hash functions, where we highlight the Ajtai hash function, the hash function based on ideal lattices and the SWIFFT hash function and to public-key ciphers, where we highlight the GGH cipher, the NTRU cipher and the LWE cipher. We also address the base reduction algorithm.

Relatively to implementations, the Sage tool is used, available in

http://www.sagemath.org/,

which uses the Python language.

Conteúdo

1 Introdução **1**

2 Teoria dos Retículos **3**

2.1 Retículo . . . 3

2.1.1 Retículo - Exemplos . . . 5

2.2 Retículo q-ário . . . 7

2.3 Problemas dos Retículos . . . 9

2.4 Algoritmos associados à resolução dos problemas dos retículos . . . 10

2.4.1 Ortogonalização de Gram-Schmidt . . . 10

2.4.2 LLL . . . 10

2.4.3 Arredondamento de Babai . . . 10

2.5 Algoritmo de Redução de Base . . . 13

2.5.1 Implementação em Sage . . . 13

2.5.2 Utilização da função usando o Sage . . . 16

2.6 Função reticulo . . . 17

2.6.1 Implementação em Sage . . . 17

2.6.2 Utilização da função usando o Sage . . . 18

3 Funções de *Hash* **19**

3.1 Caraterização . . . 19

3.2 Aplicações das funções de *hash* . . . 19

3.3 Função de *hash* de Ajtai . . . 20

3.3.1 Construção da função de *hash* de Ajtai . . . 20

3.3.2 Algoritmo da função de *hash* de Ajtai . . . 20

3.4 Função de *hash* baseada em retículos cíclicos e retículos ideais . . . 21

3.4.1 Construção de retículos cíclicos . . . 21

3.4.2 Construção de funções de *hash* baseados em retículos ideais . . . 22

3.4.3 Algoritmo da função de *hash* baseada em retículos ideais . . . 22

3.4.4 Implementação em Sage . . . 23

3.4.5 Utilização das funções usando o Sage . . . 25

3.5 Função de *hash* `SWIFFT` . . . 26

3.5.1 Construção da função de *hash* `SWIFFT` . . . 26
3.5.2 Algoritmo da função de *hash* `SWIFFT` . . . 27
3.5.3 Implementação em Sage . . . 27
3.5.4 Utilização das funções usando o Sage . . . 30
3.6 Considerações . . . 31

4 Criptografia de Chave Pública **33**
4.1 Contextualização . . . 33
4.2 GGH . . . 34
4.2.1 Geração de Chaves . . . 34
4.2.2 Obtenção de σ . . . 34
4.2.3 Cifração . . . 34
4.2.4 Decifração . . . 35
4.2.5 Implementação em Sage . . . 35
4.2.6 Utilização das funções usando o Sage . . . 38
4.3 NTRU . . . 39
4.3.1 Retículo NTRU de Convolução Modular . . . 39
4.3.2 Definições e notações . . . 40
4.3.3 Geração de Chaves . . . 40
4.3.4 Cifração . . . 40
4.3.5 Decifração . . . 41
4.3.6 Implementação em Sage . . . 41
4.3.7 Utilização das funções usando o Sage . . . 46
4.4 LWE . . . 48
4.4.1 Definições . . . 48
4.4.2 Geração de Chaves . . . 48
4.4.3 Cifração . . . 48
4.4.4 Decifração . . . 49
4.4.5 Implementação em Sage . . . 49
4.4.6 Utilização das funções usando o Sage . . . 52
4.5 Considerações . . . 54

Conclusão **55**

Bibliografia **57**

Anexos **59**

Capítulo 1

Introdução

A criptografia tem nos dias de hoje uma grande importância, devido à necessidade urgente de comunicar de forma segura. Perante isto, é necessário o envolvimento da comunidade científica na construção de novas técnicas, bem como na análise rigorosa da segurança das mesmas.

Na criptografia moderna a noção de segurança assenta na teoria da complexidade. A prova de segurança de uma técnica criptográfica consiste tipicamente em demonstrar que comprometer os objetivos desta técnica pressupõe o uso de recursos inacessíveis. A prova é expressa na forma de redução para um problema difícil, isto é, caso alguém consiga quebrar a técnica com êxito então é capaz de resolver o problema difícil associado.

Relativamente ao surgimento de problemas difíceis ou instâncias destes, teve especial relevância a área de teoria de números computacional, onde se usaram problemas difíceis como a fatorização de números inteiros e ainda o problema do logaritmo discreto.

Avanços realizados recentemente levaram ao aparecimento de um novo paradigma computacional, designado de computação quântica, que veio fazer com que problemas difíceis como o da fatorização e o logaritmo discreto, afinal não fossem tão difíceis de resolver recorrendo a algoritmos quânticos. Mas apesar destes avanços, ainda não é possível o uso de computadores quânticos que processem valores da ordem de grandeza usados na criptografia.

Por esse motivo surgiu a necessidade de estudar novas técnicas criptográficas baseadas em problemas difíceis com a particularidade de serem imunes à computação quântica bem como aos computadores quânticos. Surgiram então aplicações nos retículos, estrutura matemática com caraterísticas importantes, que contêm problemas difíceis que possuem os requisitos descritos anteriormente.

Relativamente à dissertação, um dos objetivos é o estudo das funções de *hash* criptográficas baseadas em retículos que possuem caraterísticas, como por exemplo, serem de sentido único, podendo estas funções de *hash* serem usadas para a construção de técnicas simétricas, donde se destaca o código de autenticação da mensagem, ou MAC, ou até cifras simétricas. Outro dos objetivos é o estudo de cifras de chave pública baseadas em retículos, com a caraterística de serem funções de sentido único com segredo. Por fim, estudaram-se os algoritmos associados às funções de *hash* e às cifras de chave pública bem como as respetivas implementações. No entanto, não se procedeu à demonstração da segurança das mesmas, devido ao facto de as reduções de segurança baseadas em retículos introduzirem dificuldades consideráveis, o que tornava complicado estudar as técnicas pretendidas baseadas em retículos.

De seguida é apresentado o conteúdo desta monografia, em que é feita uma breve descrição sobre o que é abordado em cada um dos capítulos.

No capítulo 2 serão abordados os lemas, proposições, definições e corolários associados aos retículos. Aí destacar-se-ão a noção de matrizes unimodulares, a noção de retículo, retículo q-ário, bem como a noção de dual de um retículo, domínio fundamental, determinante de um retículo e a definição de dois conjuntos q-ários importantes. Destacaremos ainda os problemas associados aos retículos, como o problema do vetor mais pequeno, o problema do vetor mais perto e o problema dos vetores independentes mais pequenos e os algoritmos para os resolver, como o LLL e o arredondamento de Babai.

No capítulo 3 serão abordadas as funções de *hash* baseadas em retículos onde se destacará a função de *hash* de Ajtai, a função de *hash* baseada em retículos ideais e a função de *hash* `SWIFFT`. A primeira a surgir foi a função de *hash* de Ajtai que serviu como base para as funções de *hash* seguintes. A função de *hash* baseada em retículos ideais veio trazer uma nova versão em que se alterou a matriz usada na função de *hash* de Ajtai por uma matriz de bloco com determinadas caraterísticas. Por fim, consideraremos a função de *hash* `SWIFFT`, em que a construção da função é diferente das construções anteriores e onde se destaca o uso de matrizes de Vandermonde.

No capítulo 4 serão abordados os criptossistemas de chave pública, onde se destacarão o GGH, o NTRU e o LWE. O GGH foi o primeiro a ser apresentado, sendo este baseado no problema do vetor mais perto. O NTRU é apresentado em primeiro lugar na versão polinomial, possuindo no entanto várias versões para a sua implementação, como a matricial e a vetorial, sendo este baseado no problema do vetor mais pequeno. Finalmente, será apresentado o LWE baseado no problema aproximado do vetor mais pequeno e no problema dos vetores independentes mais pequenos.

Capítulo 2

Teoria dos Retículos

Nesta secção são apresentadas as definições, proposições, lemas e corolários associados aos retículos tendo como base [2].

2.1 Retículo

Inicia-se o estudo dos retículos com a seguinte definição:

Definição 2.1.1. *Seja $n \in \mathbb{N}$. Um retículo de dimensão n é um conjunto da forma*

$$L = L(B) = \{Bx \mid x \in \mathbb{Z}^n\} = B\mathbb{Z}^n \subseteq \mathbb{R}^n$$

para alguma matriz $B \in Gl_n(\mathbb{R})$. Dizemos que L tem dimensão n e B designa-se por base do retículo L. Se $b_1, \ldots, b_n$ são as colunas de B, então $\|B\| = max_i\|b_i\|$, para $i = 1, \ldots, n$.

Associado ao retículo estão as seguintes noções:

Definição 2.1.2. *Se $L = L(B)$ é um retículo então define-se:*

- *Discriminante de L:*
$$disc(L) = \det(B).$$
- *Domínio Fundamental:*
$$F(L) = \{t_1b_1 + ... + t_nb_n : 0 \leq t_i < 1\}.$$
- *Determinante de L:*
$$\det(L) = |\det(B)|.$$

- *Dual de L:*

$$L^* = L((B^T)^{-1}).$$

- *Determinante do dual de L:*

$$\det(L^*) = \frac{1}{\det(L)}.$$

Lema 2.1.3. *Seja L um retículo. Então* $L^* = \{y \in \mathbb{R}^n \mid \forall x \in L : \langle x, y\rangle \in \mathbb{Z}\}$.

Demonstração. Em primeiro lugar pretendemos demonstrar que $\{y \in \mathbb{R}^n \mid \forall x \in L : \langle x, y\rangle \in \mathbb{Z}\} \subseteq L^*$.

Seja $y \in \mathbb{R}^n$ tal que $\langle x, y\rangle \in \mathbb{Z}$ para todo o $x \in L = L(B)$. Seja ainda $x' = e_i \in \mathbb{Z}^n$, em que e_i corresponde ao vetor cujas entradas são 0's excepto na posição i cuja entrada é 1. Tomando $x = Bx'$ com $x' \in \mathbb{Z}^n$, tem-se que $x = b_i$ para $i = 1, \ldots, n$. Então

$$\langle x, y\rangle = \langle Bx, y\rangle = x'^T B^T y = \langle x', B^T y\rangle \in \mathbb{Z}.$$

Tomando $x'^T = e_i$ tem-se que

$$x'^T B^T y = (B^T y)_i \in \mathbb{Z}$$

para $i = 1, \ldots, n$. Para $y' = B^T y \in \mathbb{Z}^n$, tem-se que $y = (B^T)^{-1} y'$, com $y' \in \mathbb{Z}^n$. Logo $y \in L(B^T)^{-1}$.

Falta demonstrar que $L^* \subseteq \{y \in \mathbb{R}^n \mid \forall x \in L : \langle x, y\rangle \in \mathbb{Z}\}$.

Seja $y \in L((B^T)^{-1})$, isto é, $y = (B^T)^{-1} y'$, com $y' \in \mathbb{Z}^n$ e seja $x = Bx'$ com $x' \in \mathbb{Z}^n$. Tem-se então que

$$\langle x, y\rangle = \langle Bx', (B^T)^{-1} y'\rangle = (Bx')^T (B^T)^{-1} y' = x'^T (B^T)(B^T)^{-1} y' = x'^T y' \in \mathbb{Z}.$$

□

Associado ao retículo está a seguinte definição:

Definição 2.1.4. *Uma matriz* $U \in \mathbb{Z}^{n\times n}$ *diz-se uma matriz unimodular se* $\det(U) = \pm 1$.

Como exemplo de matrizes unimodulares tem-se as matrizes $A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ e $B = \begin{bmatrix} 2 & 1 \\ 3 & 1 \end{bmatrix}$, pois pela definição $A \in \mathbb{Z}^{2\times 2}$, $B \in \mathbb{Z}^{2\times 2}$ e $\det(A) = 1$ assim como $\det(B) = -1$.

Como exemplo de matrizes não unimodulares tem-se as matrizes $C = \begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix}$ e $D = \begin{bmatrix} 4 & 1 \\ 2 & 1 \end{bmatrix}$, pois pela definição $C \in \mathbb{Z}^{2\times 2}$, $D \in \mathbb{Z}^{2\times 2}$ mas $\det(C) = 2$ assim como $\det(D) = 2$.

A proposição seguinte relaciona as matrizes que definem o mesmo retículo:

Proposição 2.1.5. *Sejam $B, C \in Gl_n(\mathbb{R})$. Então $L(B) = L(C)$ se e só se existir uma matriz unimodular U tal que $C = BU$.*

Demonstração. Em primeiro lugar pretendemos demonstrar que se $L(B) = L(C)$ então existe uma matriz unimodular U tal que $C = BU$. Suponhamos então que $L(B) = L(C)$. Para $C = \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}$ temos que $c_i \in L(B)$, ou seja, $c_i = Bu_i$ com $u_i \in \mathbb{Z}^n$, para $i = 1, \ldots, n$. Tomando $C = BU$ com $U = \begin{bmatrix} u_1 & \cdots & u_n \end{bmatrix}$ tem-se que $B^{-1}C = U$, o que implica que $U \in \mathrm{Gl}_n(\mathbb{R})$. Seja $V = \begin{bmatrix} v_1 & \cdots & v_n \end{bmatrix} = U^{-1}$ então como $C = BU$ implica que

$$B = CU^{-1} = CV.$$

Logo $b_i = Cv_i$ e tem-se que $v_i \in \mathbb{Z}^n$. Logo $C = BU$ com U matriz unimodular.

Falta demonstrar que se existe uma matriz unimodular U tal que $C = BU$ então $L(B) = L(C)$.

Seja $z \in L(C)$, isto é, $z = Cy$ com $y \in \mathbb{Z}^n$. Então $z = BUy \in L(B)$ com $U \in \mathrm{Gl}_n(\mathbb{Z})$. Logo $L(C) \subseteq L(B)$. Seja agora $a \in L(B)$, isto é, $a = Bv$ com $v \in \mathbb{Z}^n$. Então

$$a = CU^{-1}v \in L(C)$$

com $U \in \mathrm{Gl}_n(\mathbb{Z})$ e portanto $L(B) \subseteq L(C)$. Em conclusão $L(B) = L(C)$. □

2.1.1 Retículo - Exemplos

Seja $B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \in \mathbb{R}^{2\times 2}$, e considere-se o retículo $L = L(B)$. Sejam $\alpha, \beta \in \mathbb{Z}$ e $x = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \in \mathbb{Z}^2$. Então tomando os elementos de L, que são da forma Bx e usando a função `reticulo` implementada em Sage, obtem-se a representação do retículo na figura 2.1.

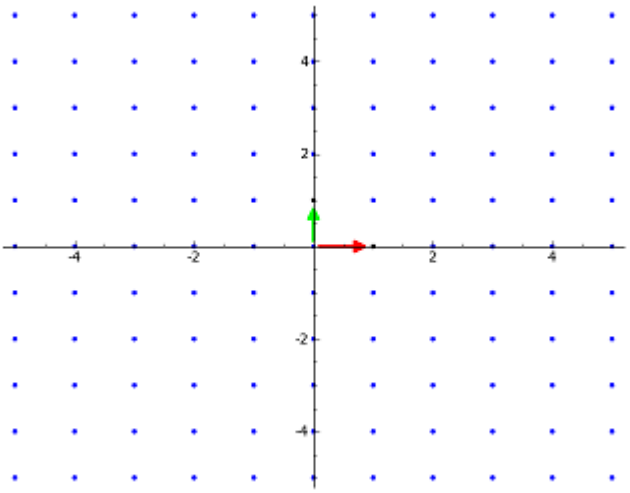


Figura 2.1: Representação de um retículo

Seja agora $B = \begin{bmatrix} 0.957640691166434 & -0.976297904768089 \\ -0.0803620469958037 & -0.543643038442881 \end{bmatrix} \in \mathbb{R}^{2\times 2}$, e considere-se o retículo $L = L(B)$. Sejam ainda $\alpha, \beta \in \mathbb{Z}$ e $x = \begin{bmatrix} \alpha \\ \beta \end{bmatrix} \in \mathbb{Z}^2$. Então tomando os elementos de L, que são da forma Bx e usando a função `reticulo` implementada em Sage, obtem-se a representação do retículo na figura 2.2.

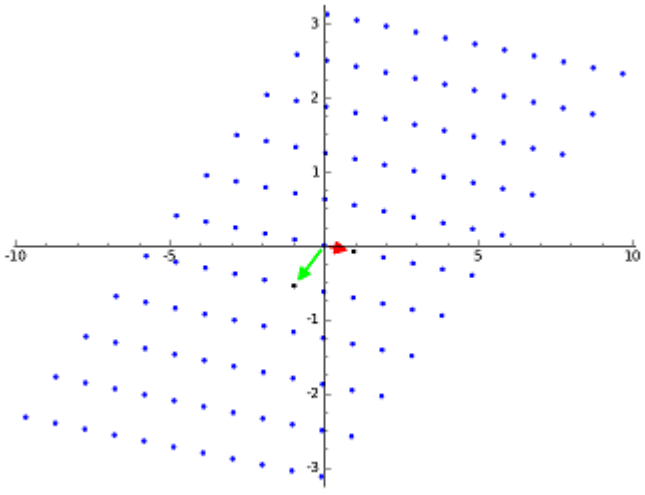


Figura 2.2: Representação de um retículo

2.2 Retículo q-ário

Definição 2.2.1. *Seja $q \in \mathbb{N}$. Um retículo L de dimensão n diz-se q-ário se $q\mathbb{Z}^n \subseteq L$.*

A proposição que se segue dá-nos uma condição suficiente para que um retículo seja q-ário à custa do seu determinante:

Proposição 2.2.2. *Se $L = L(B)$ é um retículo de inteiros, isto é, $B \in \mathbb{Z}^{n\times n}$ e se q é um múltiplo de* $\det(L)$ *então L é um retículo q-ário.*

Demonstração. Sejam $x \in \mathbb{Z}^n$ e $q = \det(L)k$, com $k \in \mathbb{Z}$ e considerando $B^{-1} = \dfrac{B^{adj}}{det(B)}$, tem-se que

$$qx = BB^{-1}qx = B\frac{B^{adj}}{det(B)}qx = BB^{adj}kx = Bx'$$

com $x' \in \mathbb{Z}^n$, isto é, $qx \in L(B)$. □

De seguida é apresentada a noção de retículo racional:

Definição 2.2.3. *Um retículo L é racional se tiver uma base $B \in \mathbb{Q}^{n\times n}$.*

Corolário 2.2.4. *Cada retículo racional é q-ário para algum $q \in \mathbb{N}$.*

Demonstração. Seja $L = L(B)$, com $B \in \mathbb{Q}^{n\times n}$. Existe $p \in \mathbb{N}$ tal que $pB = B' \in \mathbb{Z}^{n\times n}$ com $q = \mid \det(B') \mid$. Como pq é múltiplo de q, isto é, B' é pq-ário, logo tem-se que B é q-ário. □

Da noção de retículo q-ário obtem-se a seguinte definição:

Definição 2.2.5. *Sejam $n \leq m$ e $A \in \mathbb{Z}^{n\times m}$ com $car(A) = n$, ou seja, caraterística de A é máxima. Para $q \in \mathbb{N}$ e $\max A = \max_{ij} a_{ij}$, define-se:*

$$\Lambda_q(A) = \left\{y \in \mathbb{Z}^m \mid \exists s \in \mathbb{Z}^n : y \equiv A^T s \mod q\right\}$$

e

$$\Lambda_q^{\perp}(A) = \left\{y \in \mathbb{Z}^m \mid Ay \equiv 0_{n\times 1} \mod q\right\}.$$

Estes dois retículos q-ários relacionam-se da seguinte forma:

Proposição 2.2.6. *Seja $\Lambda_q(A)$ um retículo q-ário L de dimensão m. Então*

$$\Lambda_q^{\perp}(A) = qL^*.$$

Demonstração. Quer-se demonstrar que $\Lambda_q(A) = L(B)$, com $A \in \mathbb{Z}^{m\times n}$, $B \in \mathbb{Z}^{m\times m}$, $car(B) = m$ e $car(A) = n$.

Pela definição tem-se que

$$\Lambda_q(A) = \left\{ y \in \mathbb{Z}^m \mid \exists s \in \mathbb{Z}^n : y \equiv A^T s \mod q \right\}.$$

Então

$$\begin{bmatrix} A^T & qI_m \end{bmatrix} \begin{bmatrix} s \\ x \end{bmatrix} = \begin{bmatrix} A^T \\ qI_m \end{bmatrix} \begin{bmatrix} s \\ x \end{bmatrix} \equiv A^T s \mod q \equiv y \mod q,$$

com I_m a matriz identidade de ordem m. Isto é, $\Lambda_q(A) = \begin{bmatrix} A^T \\ qI_m \end{bmatrix} \mathbb{Z}^{m+n}$. Pelo algoritmo de redução de base, tem-se que $\begin{bmatrix} A^T & qI_m \end{bmatrix} \to B \in \mathbb{Z}^{m\times m}$, com $car(B) = m$. Logo

$$\Lambda_q(A) = B\mathbb{Z}^m = L(B).$$

Quer-se demonstrar que $\Lambda_q^{\perp}(A) = qL^*$.

1. Em primeiro lugar pretendemos demonstrar que $qL^* \subseteq \Lambda_q^{\perp}(A)$.

   Seja $y' \in qL^*$, o que implica que $y' = qy$ com $y \in L^*$, isto é, $\langle x, y\rangle \in \mathbb{Z}$ para todo o $x \in L(B)$. Daqui obtem-se que $Ay' = qAy$, o que implica que $Ay \equiv 0 \mod q$. Para $x = qe_i \in L(B) = \Lambda_q(A)$, em que e_i é um vector com entradas a 0 excepto na posição i cuja entrada é 1, tem-se que $x = qe_i \equiv 0 \mod q$ e $A^T e_i \equiv 0 \mod q$. Ora $\langle x, y\rangle \in \mathbb{Z}$, isto é, $\langle qe_i, y\rangle = \langle e_i, qy\rangle = (qy)_i = (\tilde{y})_i$, com $\tilde{y} \in \mathbb{Z}^m$. Seja $x = A^T e_i$, com $x \in \Lambda_q(A) = L(B)$. Logo $\langle x, y\rangle \in \mathbb{Z}$, isto é, $\langle A^T e_i, y\rangle = (A^T e_i)^T y = e_i Ay = \langle e_i, Ay\rangle = (Ay)_i$, com $Ay \in \mathbb{Z}^n$.

2. Falta demonstrar que $\Lambda_q^{\perp}(A) \subseteq qL^*$.

   Seja $\tilde{y} \in \mathbb{Z}^m$ tal que $A\tilde{y} \equiv 0 \mod q$, isto é, $A\tilde{y} = qz$, para algum $z \in \mathbb{Z}^n$. Seja então $y = q^{-1}\tilde{y}$. Pretende-se demonstrar que $\langle x, y\rangle \in \mathbb{Z}$, para todo o $x \in \Lambda_q(A) = L(B)$. Seja então $x \in \Lambda_q(A)$, isto é, $x \equiv A^T s \mod q$, com $s \in \mathbb{Z}^n$. Tomando $x = A^T s + qr$, com $r \in \mathbb{Z}^m$ tem-se que,

   $$\begin{aligned} \langle x, y\rangle &= \langle A^T s + qr, q^{-1}\tilde{y}\rangle \\ &= \langle A^T s, q^{-1}\tilde{y}\rangle + \langle qr, q^{-1}\tilde{y}\rangle \\ &= q^{-1}\langle A^T s, \tilde{y}\rangle + \langle r, \tilde{y}\rangle \\ &= q^{-1}\langle s, A\tilde{y}\rangle + \langle r, \tilde{y}\rangle \\ &= q^{-1}\langle s, qz\rangle + \langle r, \tilde{y}\rangle \\ &= \langle s, z\rangle + \langle r, \tilde{y}\rangle \in \mathbb{Z}, \end{aligned}$$

   tendo atenção que

   $$\langle A^T s, \tilde{y}\rangle = (A^T s)^T \tilde{y} = s^T A\tilde{y} = \langle s, A\tilde{y}\rangle.$$

□

Da proposição, o seguinte corolário é obtido:

Corolário 2.2.7. *Se $\Lambda_q(A)$ é um retículo então $\Lambda_q(A) = q\Lambda_q^{\perp}(A)^*$.*

Demonstração. Seja $\Lambda_q(A) = L(B)$. Então

$$q\Lambda_q^{\perp}(A)^* = q(qL((B^T)^{-1}))^* = q(L(q(B^T)^{-1}))^* = qL(((\frac{1}{q}B)^T)^{-1})^* = qL(\frac{1}{q}B) = L(B) = \Lambda_q(A).$$

□

2.3 Problemas dos Retículos

As construções criptográficas, baseadas em retículos, estão relacionadas com os problemas associados aos retículos, ou seja, é possível provar, segundo [3] e [2], que quebrar essas construções, permite definir um algoritmo para resolver uma das instâncias dos problemas associados aos retículos. Os problemas são os seguintes:

Problema do Vetor mais Pequeno - SVP

Encontrar um vetor $v \in L(B)\backslash\{0_{n\times 1}\}$ tal que:

$$\|v\| \leq \min \|\mathrm{w}\|, \text{ com } w \in L(B) \text{ e } w \neq 0_{n\times 1}.$$

Problema do Vetor mais Perto - CVP

Para $t \in \mathbb{R}^n$, encontrar um ponto do retículo $v \in L(B)$ tal que:

$$\|v - t\| \leq \min \|w - t\|, \text{ com } w \in L(B).$$

Problema dos Vetores Independentes mais Pequenos - SVIP

Encontrar $U \in \mathrm{Gl}_n(\mathbb{Z})$, tal que:

$$\|BU\| \leq \min \|BV\|, \text{ com } V \in \mathrm{Gl}_n(\mathbb{Z}).$$

2.4 Algoritmos associados à resolução dos problemas dos retículos

2.4.1 Ortogonalização de Gram-Schmidt

Para se obter uma base ortogonal a partir de uma base já existente usando um processo iterativo, recorre-se ao algoritmo de ortogonalização de Gram-Schmidt.

Por [6], seja $B = \left[\begin{array}{ccc} b_1 & \cdots & b_n \end{array} \right] \in \mathbb{R}^{n \times n}$ uma base do retículo e seja $b_1^* = b_1$. Então define-se uma nova base B^*, como sendo:

$$B^* = \left[\begin{array}{ccc} b_1^* & \cdots & b_n^* \end{array} \right],$$

em que cada $b_i^* = b_i$ - $\sum_{j=1}^{i-1} \mu_{i,j} b_j^*$, com $\mu_{i,j} = \dfrac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}$, para $i = 2, \ldots, n$.
Tem-se que B^* é a projeção ortogonal de B e $L(B^*) \neq L(B)$.

2.4.2 LLL

Este algoritmo foi proposto por Lenstra, Lenstra e Lovász e visa resolver o SVP. O algoritmo tenta encontrar uma base pequena, quase ortogonal, que se designa de base LLL-reduzida. Por [6], dado um limite δ, uma base $B \in \mathbb{R}^{n \times n}$ diz-se LLL-reduzida se verifica as seguintes condições:

1. $\forall i, j \leq n \ |\mu_{i,j}| \leq \frac{1}{2}$ (Condição de Tamanho)
2. $\forall k \leq n, \ \delta \|b_k^*\|^2 \leq \|b_{k+1}^*\|^2 + |\mu_{k+1,k}|^2 \times \|b_k^*\|^2$ (Condição de Lovász)

2.4.3 Arredondamento de Babai

O algoritmo que foi proposto por Babai visa resolver o CVP, de acordo com [6]. A ideia é considerar o vetor alvo como uma combinação linear real dos vetores da base e arredondando para fatores inteiros. Esta aproximação encontra, para cada vetor da base, o produto do vetor que está próximo do vetor alvo, o que adicionando estes produtos em conjunto, leva a que o vetor mais próximo ao vetor alvo seja encontrado. Relativamente à diferenciação de base "boa" e base "má", torna-se relevante na resolução do CVP, sendo apresentada, de seguida a ideia de resolução do CVP, bem como a influência da escolha da base, segundo [7]:

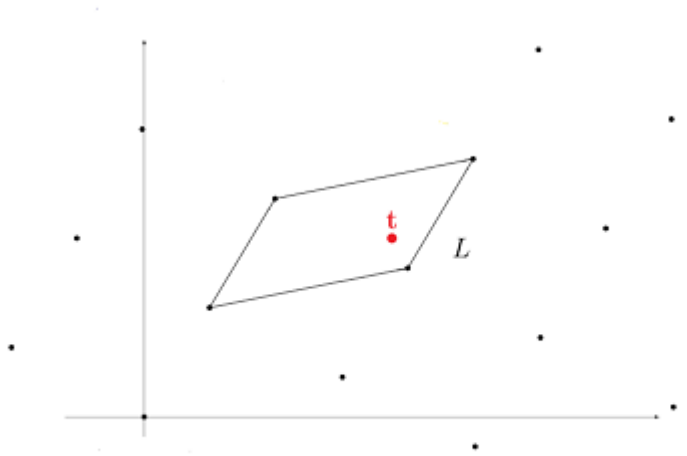


Figura 2.3: Representação do domínio fundamental

Pela figura 2.3, a ideia é usar a base do retículo para desenhar o paralelogramo, isto é, o domínio fundamental, em torno do ponto alvo t.

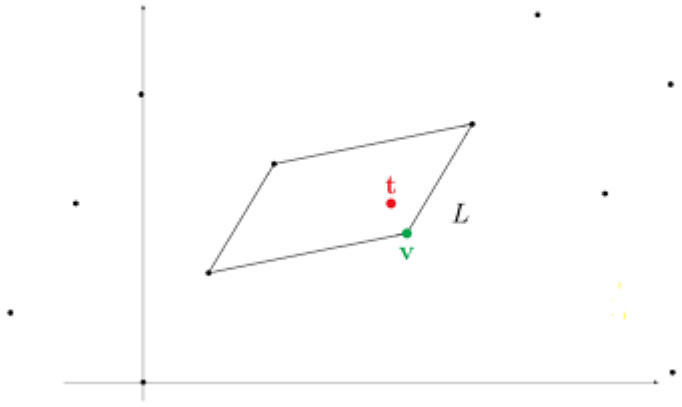


Figura 2.4: Representação do domínio fundamental

Na figura 2.4 verifica-se que o vértice v do domínio fundamental, que é o mais próximo a t, será o ponto do retículo mais próximo caso a base seja "boa", isto é, caso a base contenha vetores pequenos e que sejam razoavelmente ortogonais entre eles.

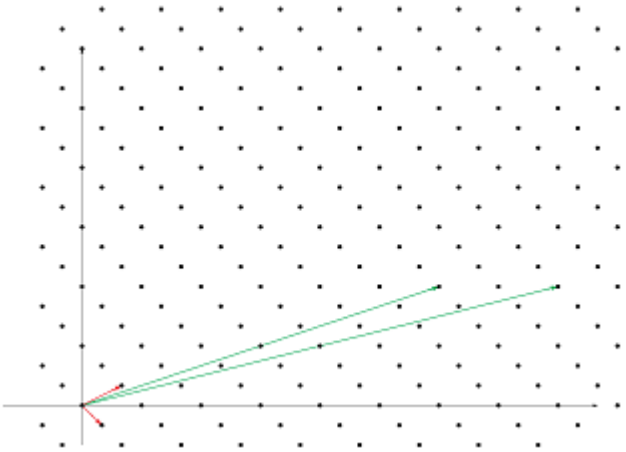


Figura 2.5: Representação de uma "má" e de uma "boa" base

Na figura 2.5 é representada uma "boa" base, a vermelho, e uma "má" base, a verde. A "má" base será usada para visualizar a resolução do CVP recorrendo à mesma.

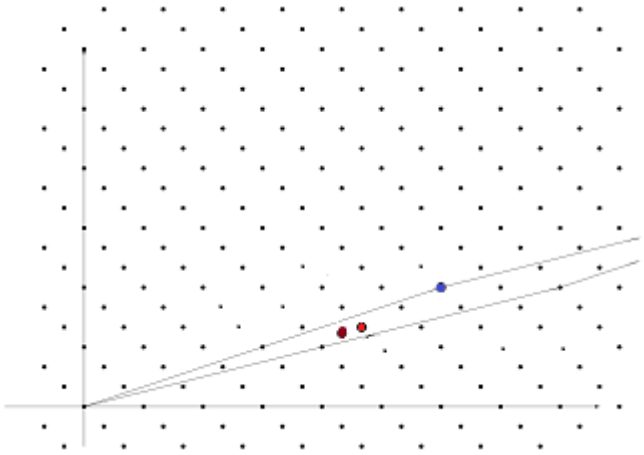


Figura 2.6: Representação da resolução do CVP usando uma "má" base

Na figura 2.6 é representada a resolução do CVP associado à "má" base representada na figura 2.5, usando a ideia descrita anteriormente para a construção do domínio fundamental, onde nesta figura é representado por linhas a preto. Pelo facto de ser uma "má" base, o vértice do domínio fundamental, assinalado a azul, é o que está mais próximo do ponto alvo, assinalado a castanho. No entanto, o ponto do retículo, assinalado a vermelho, que resolve o CVP está mais próximo do ponto alvo que o vértice do paralelogramo mais próximo.

2.5 Algoritmo de Redução de Base

Este algoritmo tem como objetivo, dada uma base do retículo, construir uma outra base que gera o mesmo retículo, ou seja, dada uma "boa" base A de um retículo, construir uma "má" base B tal que $L(A) = L(B)$. De seguida é apresentado o código associado às funções, segundo [2], que compõem a construção do algoritmo de redução de base.

2.5.1 Implementação em Sage

Função `matrizED`

Esta função recebe como argumento de entrada uma lista de inteiros *tuplo* e como saída retorna uma matriz $X \in \mathbb{Z}^{2\times(n-1)}$, que corresponde à resolução da equação diofantina da forma

$$a_1x_1 + \cdots + a_nx_n = gcd(a_1, \ldots, a_n),$$

em que para a primeira coluna corresponde à resolução da equação diofantina

$$a_1x_{1,1} + a_2x_{2,1} = gcd(a_1, a_2) = b_1$$

e para a k-ésima coluna corresponde a resolver a equação diofantina

$$b_kx_{1,k} + a_{k+1}x_{2,k} = gcd(b_{k-1}, a_{k+1})$$

para $k \geq 2$. De seguida é apresentado o código da função em Sage:

```
def matrizED(tuplo):
    if(len(tuplo)>1):
        X=matrix(ZZ,2,len(tuplo)-1);
        a=xgcd(tuplo[0],tuplo[1]);
        bi=gcd(tuplo[0],tuplo[1]);
        X[0,0]=a[1];
        X[1,0]=a[2];
        for i in range(2,len(tuplo)):
            a=xgcd(bi,tuplo[i]);
            bi=gcd(bi,tuplo[i]);
            X[0,i-1]=a[1];
            X[1,i-1]=a[2];
    else:
        X=matrix(ZZ,1,1);
        X[0,0]=1;
    return X;
```

Função `solveED`

Esta função recebe como entrada uma matriz $X \in \mathbb{Z}^{2\times(n-1)}$ e como saída retorna uma lista de inteiros l, que contém os x_i que são solução da equação diofantina

$$\sum_{i=1}^{n} a_i x_i = gcd(a_1, \ldots, a_n)$$

e são obtidos da seguinte forma:

- $x_1 = \prod_{i=1}^{n} X_{1,i}$;
- $x_b = x_{2,b-1} \prod_{i=b}^{n} X_{1,i}$, para $2 \leq b \leq n-1$;
- $x_n = X_{2,n}$;

De seguida é apresentado o código da função em Sage:

```
def solveED(X):
    l=[];
    comp=len(X.row(0));
    x1=prod(X.row(0));
    l.append(x1);
    if(X.ncols()==X.nrows()==1):
        return l;
    else:
        i=0;
        while i<comp-1:
            xi=X[1,i];
            j=i+1;
            while j<comp:
                xi=xi*X[0,j];
                j=j+1;
            l.append(xi);
            i=i+1;
        xi=X[1,comp-1];
        l.append(xi);
    return l;
```

Função `basereduction`

Esta função recebe como entrada uma matriz $A \in \mathbb{Z}^{n\times m}$, com caraterística máxima retangular ou quadrada e como saída retorna uma matriz $B \in \mathbb{Z}^{n\times n}$ triangular inferior, tal que $B^*\mathbb{Z}^n = A^*\mathbb{Z}^m$. Esta função depende da função `matrizED` e da função `solveED` para a sua construção. De seguida é apresentado o código em Sage:

```
def basereduction(A):
    B=copy(A);
    n=B.nrows();
    m=B.ncols();
    if m<n:
        print 'O número de colunas tem de ser maior ou igual ao número de linhas';
    elif rank(B)!=n:
        print 'A matriz nao tem característica máxima';
    else:
        for j in range(0,n):
            aux=[];
            for hj in range(j,m):
                aux.append(B[j,hj]);
            X=matrizED(aux);
            l=solveED(X);
            g=gcd(aux);
            i=j;
            tam=len(l);
            hh=0;
            while i<m:
                B[:,j]=l[hh]*B[:,i];
                hh=hh+1;
                i=i+1;
                while hh<tam:
                    B[:,j]=B[:,j]+l[hh]*B[:,i];
                    hh=hh+1;
                    i=i+1;
            for k in range(j+1,m):
                B[:,k]=B[:,k]-(B[j,k]*B[:,j])/g;
    if n==m:
        return B;
    elif n<m:
        return B[:,0:n];
```

2.5.2 Utilização da função usando o Sage

Para usar as construções relativas ao algoritmo de redução de base, tomam-se por exemplo, as seguintes instruções:

1. `A=matrix(ZZ,[[4,8,6],[3,2,4]])`;
2. `C=basereduction(A)`;

A primeira instrução associa a variável A à matriz $\begin{bmatrix} 4 & 8 & 6 \\ 3 & 2 & 4 \end{bmatrix}$.

A segunda instrução associa a variável C ao resultado de invocar a função `basereduction` que recebe como argumento A, isto é, C é a matriz $\begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix}$.

2.6 Função reticulo

2.6.1 Implementação em Sage

Função `reticulo`

Esta função recebe como entrada uma matriz $B \in \mathbb{Z}^{2\times 2}$ com caraterística 2 e caso os parâmetros verifiquem as condições impostas, então apresenta graficamente o retículo gerado por B. De seguida é apresentado o código da função em Sage:

```
def reticulo(B):
    if  B.ncols()==B.nrows() and B.ncols()==2 and B.nrows()==2 and rank(B)==2
    and is_Matrix(B):
        P=plot([]);
        v1=B.column(0);
        v2=B.column(1);
        for a in range(-5,6):
            for b in range(-5,6):
                if (a==0 and b==1) or (a==1 and b==0):
                    P+=plot(v1,rgbcolor=(1,0,0));
                    P+=plot(v2,rgbcolor=(0,1,0));
                    P+=point(a*v1+b*v2,rgbcolor=(0,0,0));
                else:
                    P+=point(a*v1+b*v2);
        return P;
    else:
        print 'O parâmetro B não satisfaz um dos seguintes requisitos:\n';
        print 'B é uma matriz quadrada\n';
        print 'Número de linhas deve ser igual a 2\n';
        print 'Número de colunas deve ser igual a 2\n';
        print 'A caraterística da matriz deve ser 2\n';
```

2.6.2 Utilização da função usando o Sage

Para usar a construção relativa à função `reticulo`, tomam-se por exemplo, as seguintes instruções:

1. `C=matrix(ZZ,[[1,0],[1,1]]);`
2. `reticulo(C);`

A primeira instrução associa a variável C à matriz $\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

A segunda instrução invoca a função `reticulo` que recebe como argumento a matriz C e representa graficamente o retículo gerado por C:

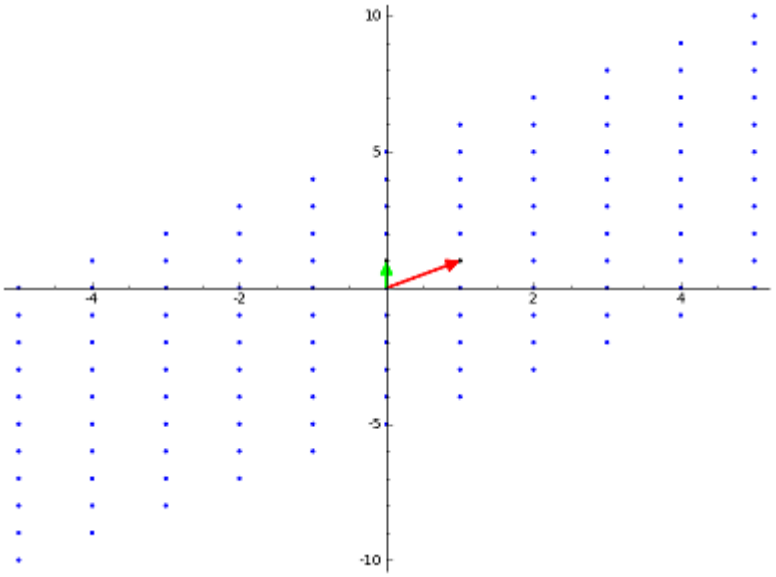


Figura 2.7: Representação do retículo gerado pela matriz C

Capítulo 3

Funções de *Hash*

3.1 Caraterização

As funções de *hash* são funções de sentido único, isto é, funções que possuem um algoritmo eficiente para o seu cálculo mas que não dispõem de um algoritmo eficiente para o cálculo da sua inversa, que recebem como argumento mensagens de um comprimento variável e produzem uma saída de comprimento fixo. Tipicamente estas funções são desenvolvidas de modo a serem resistentes a determinadas operações, dependendo do cenário onde esta função está inserida.

Seja $H : D \rightarrow R$ a função de *hash* tal que $\#R \ll \#D$. A função de *hash* deve satisfazer os seguintes requisitos:

- **(First) pre-image resistant**: Dado um valor de *hash* h, é difícil obter uma mensagem m tal que $H(m) = h$;
- **(Second) pre-image resistant**: Dada uma mensagem m_1, é difícil obter uma mensagem m_2 diferente de m_1, tal que $H(m_2) = H(m_1)$;
- **Collision resistant**: É difícil encontrar mensagens diferentes m_1 e m_2 tais que $H(m_1) = H(m_2)$;

3.2 Aplicações das funções de *hash*

As funções de *hash* tem muitas aplicações, das quais se destacam as seguintes:

- Armazenamento de palavras-chave;
- Provas de posse de informação;

- Componente de outras técnicas criptográficas.

3.3 Função de *hash* de Ajtai

Nesta secção é tomada como referência [3], para a construção da função de *hash* de Ajtai e o respetivo algoritmo.

3.3.1 Construção da função de *hash* de Ajtai

A primeira construção criptográfica baseada em retículos foi a função de *hash* apresentada por Ajtai. A função de *hash* recebe como argumentos q, m, n e $d \in \mathbb{Z}$, com a escolha de n a determinar a segurança da função de *hash*. Esta função tem a particularidade de que encontrar colisões é tão difícil como resolver o problema dos vetores independentes mais pequenos e o problema do vetor mais pequeno. A chave da função de *hash* é a matriz A escolhida uniformemente em $\mathbb{Z}_q^{n \times m}$. Então define-se $f_A : \{0, \ldots, d-1\}^m \to \mathbb{Z}_q^n$ como sendo:

$$f_A(y) = Ay \mod q.$$

3.3.2 Algoritmo da função de *hash* de Ajtai

De seguida é descrito o algoritmo da função de *hash* de Ajtai:

- **Parâmetros:** $n, m, q, d \geq 1 \in \mathbb{Z}$.
- **Chave:** matriz A escolhida uniformemente em $\mathbb{Z}_q^{n \times m}$.
- **Função de *hash*:** $f_A : \{0, \ldots, d-1\}^m \to \mathbb{Z}_q^n$ dada por $f_A(y) = Ay \mod q$.

3.4 Função de *hash* baseada em retículos cíclicos e retículos ideais

Nesta secção é tomada como referência [3], para a construção da função de *hash* baseada em retículos ideais e o respetivo algoritmo.

3.4.1 Construção de retículos cíclicos

A eficiência das funções criptográficas baseadas em retículos pode ser melhorada substituindo matrizes gerais por matrizes com uma estrutura especial. Partindo de uma lista de vetores $a^{(i)} = \begin{bmatrix} a_1^{(i)} \\ \vdots \\ a_n^{(i)} \end{bmatrix}$, vamos construir uma matriz A por blocos, isto é, A é da forma

$$A = \left[A^{(1)} | \dots | A^{(m/n)}\right],$$

onde cada bloco $A^{(i)} \in \mathbb{Z}^{n \times n}$ é da forma

$$A^{(i)} = \begin{bmatrix} a_1^{(i)} & a_n^{(i)} & \dots & a_3^{(i)} & a_2^{(i)} \\ a_2^{(i)} & a_1^{(i)} & \dots & a_4^{(i)} & a_3^{(i)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n-1}^{(i)} & a_{n-2}^{(i)} & \dots & a_1^{(i)} & a_n^{(i)} \\ a_n^{(i)} & a_{n-1}^{(i)} & \dots & a_2^{(i)} & a_1^{(i)} \end{bmatrix}.$$

Ou seja, é uma matriz cujas colunas são todas as rotações cíclicas das entradas do vetor $a^{(i)} = \begin{bmatrix} a_1^{(i)} \\ \vdots \\ a_n^{(i)} \end{bmatrix}$. Usando a notação matricial, tem-se que

$$A^{(i)} = \left[a^{(i)}, Ta^{(i)}, \dots, T^{n-1}a^{(i)}\right],$$

onde

$$T = \begin{bmatrix} 0_{1\times n-1} & 1 \\ I_{n-1\times n-1} & 0_{n-1\times 1} \end{bmatrix},$$

em que T corresponde à matriz de permutação que roda as coordenadas de $a^{(i)}$ ciclicamente.

Considerando a matriz A para base do retículo temos que o retículo designa-se de retículo cíclico, em que cada vetor da base do retículo consiste em coeficientes do vetor anterior, deslocados em uma posição.

3.4.2 Construção de funções de *hash* baseados em retículos ideais

O problema começa em tornar funções de *hash* de sentido único em funções resistentes a colisões, tendo sido resolvida independentemente por Peikert e Rosen, de acordo com [3]. A construção geral recebe como argumentos m, n, q e $d \in \mathbb{Z}$ e um vetor $f \in \mathbb{Z}^n$. A matriz A é construída como sendo uma matriz de bloco, ou seja, $A = \left[A^{(1)}|\ldots|A^{(m/n)}\right]$, em que cada $A^{(i)} \in \mathbb{Z}^{n\times n}$ é da forma $A^{(i)} = F^* a^{(i)}$, onde

$$F^* a^{(i)} = \left[a^{(i)}, Fa^{(i)}, \ldots, F^{n-1}a^{(i)}\right],$$

com

$$F = \begin{bmatrix} 0_{1\times n-1} & -f \\ I_{n-1\times n-1} & \end{bmatrix},$$

em que F corresponde à matriz de transformação das coordenadas de $a^{(i)}$. Considerando a matriz A para base do retículo temos que o retículo designa-se de retículo ideal, pois pode ser visto como o quociente entre o anel $\mathbb{Z}[x]$ e o ideal gerado por $f(x)$, isto é, $\mathbb{Z}[x]/(f(x))$, onde $f(x) = x^n + f_n x^{n-1} + ... + f_1 \in \mathbb{Z}[x]$. O vetor f, sendo arbitrário, possui as seguintes propriedades:

- Para vetores unitários u e v, isto é, vetores com norma um, $\begin{bmatrix} F^*u \end{bmatrix} v$ tem uma norma pequena.
- O polinómio $f(x)$ é irredutível sobre os inteiros, isto é, $f(x)$ não é um produto de polinómios de grau menor que f.

Daqui retira-se que para determinadas escolhas de f, as duas propriedades são satisfeitas nos seguintes casos:

- $f = (1, ..., 1) \in \mathbb{Z}^n$ em que $n + 1$ é primo.
- $f = (1, 0..., 0) \in \mathbb{Z}^n$ em que n é uma potência de dois.

3.4.3 Algoritmo da função de *hash* baseada em retículos ideais

De seguida é descrito o algoritmo da função de *hash* baseada em retículos ideais:

- **Parâmetros:** $n, m, q, d \in \mathbb{Z}$ com $n|m$ e $f \in \mathbb{Z}^n$.
- **Chave:** m/n vetores $a^{(1)}$,...,$a^{(m/n)} \in \mathbb{Z}_q^n$ escolhidos independentemente e uniformemente de forma aleatória.
- **Função de *hash*:** $f_A : \{0, \ldots, d-1\}^m \to \mathbb{Z}_q^n$ dada por $f_A(y) = \left[F^* a^{(1)}|\ldots|F^* a^{(m/n)}\right] y \mod q$.

3.4.4 Implementação em Sage

Função `matrizF`

Esta função recebe como argumentos $n \in \mathbb{N}$ e um vetor $f \in \mathbb{Z}^n$ e como saída retorna a matriz $F \in \mathbb{Z}^{n \times n}$, com $F = \begin{bmatrix} 0_{1 \times n-1} & -f \\ I_{n-1 \times n-1} & \end{bmatrix}$. De seguida é apresentado o código da função em Sage:

```
def matrizF(n,f):
    F=zero_matrix(ZZ,n);
    F[1:n,0:n-1]=identity_matrix(n-1);
    F.set_column(n-1,-1*f);
    return F;
```

Função `gerachaves`

Esta função recebe como argumentos q, m e $n \in \mathbb{N}$ e como saída retorna a lista l com m/n vetores `ai` $\in \mathbb{Z}_q^n$, gerados aleatoriamente. De seguida é apresentado o código da função em Sage:

```
def gerachaves(q,m,n):
    Zq=IntegerModRing(q);
    l=[];
    i=0;
    res=m/n;
    while i<res:
        ai=random_vector(Zq,n);
        l.append(ai);
        i=i+1;
    return l;
```

Função `matrizbloco`

Esta função recebe como argumentos $q, n \in \mathbb{N}$, uma matriz `ff` $\in \mathbb{Z}^{n\times n}$ e um vetor `ai` $\in \mathbb{Z}_q^n$ e como saída retorna a matriz de bloco $A \in \mathbb{Z}_q^{n\times n}$ da forma $A = \left[\begin{array}{cccc} ai & (ff^1) * ai & \cdots & (ff^{(n-1)}) * ai \end{array} \right]$ em que cada `ff`^{k}`*ai` corresponde a uma coluna da matriz A, para $k = 1, \ldots, n-1$. De seguida é apresentado o código da função em Sage:

```
def matrizbloco(q,n,ff,ai):
    Zq=IntegerModRing(q);
    A=zero_matrix(Zq,n);
    A[:,0]=column_matrix(ai);
    i=1;
    while i<n:
        aux=ff*A[:,i-1];
        A[:,i]=aux;
        i=i+1;
    return A;
```

Função `matrizA`

Esta função recebe como argumentos $q, m, n \in \mathbb{N}$, uma matriz `ff` $\in \mathbb{Z}^{n\times n}$ e uma lista `lista` e como saída retorna a matriz $A \in \mathbb{Z}_q^{n\times m}$ da forma $A = \left[\begin{array}{ccc} A^1 & \cdots & A^{m/n} \end{array} \right]$ em que cada $A^k = \left[\begin{array}{cccc} ak & (ff^1) * ak & \cdots & (ff^{(n-1)}) * ak \end{array} \right]$, para $k = 1, \ldots, n$ e para cada `ak` pertencente à lista. Esta função depende da função `matrizbloco` para a sua construção. De seguida é apresentado o código da função em Sage:

```
def matrizA(q,m,n,ff,lista):
    res=m/n;
    Zq=IntegerModRing(q);
    A=zero_matrix(Zq,n,m);
    j=0;
    i=0;
    while j<m:
        while i<res:
            B=matrizbloco(q,n,ff,lista[i]);
            A[:,j:j+n]=B;
            j=j+n;
            i=i+1;
    return A;
```

Função `obtemmatrizA`

Esta função recebe como argumentos $q, m, n \in \mathbb{N}$ e o vetor $f \in \mathbb{Z}^n$ e como saída retorna a matriz $A \in \mathbb{Z}_q^{n \times m}$ construída usando a função `matrizA`. Esta função depende também da função `matrizF` e da função `gerachaves`. De seguida é apresentado o código da função em Sage:

```
def obtemmatrizA(q,m,n,f):
    F=matrizF(n,f);
    key=gerachaves(q,m,n);
    A=matrizA(q,m,n,F,key);
    return A;
```

Função `funhash`

Esta função recebe como argumentos a matriz $A \in \mathbb{Z}_q^{n \times m}$ e o vetor $y \in \mathbb{Z}^m$ e como saída retorna o produto de Ay^T. De seguida é apresentado o código da função em Sage:

```
def funhash(A,y):
    return A*column_matrix(y);
```

3.4.5 Utilização das funções usando o Sage

Para usar as construções relativas à função de *hash* baseada em retículos ideais, tomam-se por exemplo, as seguintes instruções:

1. `n2=2;m2=4;q2=13;f2=zero_vector(ZZ,n2);f2[0]=1;d2=30;`
   `y2=random_vector(ZZ,m2,x=d2);`
2. `B=obtemmatrizA(q2,m2,n2,f2);`
3. `H2=funhash(B,y2);`

A primeira instrução associa a variável $n2$ a 2, $m2$ a 4, $q2$ a 13, $f2$ como sendo o vetor $\begin{bmatrix} 1 & 0 \end{bmatrix}$, $d2$ a 30 e $y2$ é um vetor gerado aleatoriamente com coordenadas no intervalo $[0, d2-1]$, neste caso considerando $y2$ como sendo o vetor $\begin{bmatrix} 19 & 12 & 4 & 26 \end{bmatrix}$.

A segunda instrução associa a variável B ao resultado de invocar a função `obtemmatrizA` que recebe como argumentos $q2, m2, n2, f2$, isto é, B é a matriz $\begin{bmatrix} 1 & 6 & 11 & 10 \\ 7 & 1 & 3 & 11 \end{bmatrix}$.

A terceira instrução associa a variável $H2$ ao resultado de invocar a função `funhash` que recebe como argumentos $B, y2$, isto é, $H2$ é o vetor $\begin{bmatrix} 5 & 1 \end{bmatrix}$.

3.5 Função de *hash* SWIFFT

Na secção anterior foi visto que se podem obter funções de *hash* eficientes substituindo uma matriz geral por uma matriz de bloco com determinadas caraterísticas. Nesta nova função de *hash*, `SWIFFT`, é usada uma matriz de Vandermonde para a sua construção em vez da matriz de transformação ou de rotação referidas na secção anterior. Nesta secção é tomada como referência [3] para a construção da função de *hash* `SWIFFT` e o respetivo algoritmo.

3.5.1 Construção da função de *hash* SWIFFT

Seja $n = 2^\alpha$, para $\alpha \in \mathbb{N}$ e considere-se q um número primo tal que $2n|(q-1)$. Seja $W \in \mathbb{Z}_q^{n \times n}$ uma matriz invertível sobre $\mathbb{Z}_q$ em que

$$W = [w^{(2i-1)(2j-1)}]_{i,j=1}^{n},$$

com $w \in \mathbb{Z}_q$, tal que a ordem de w é $2n$, ou seja, tem-se que como q é primo, $\mathbb{Z}_q \backslash \{0\}$ é um grupo multiplicativo que tem o elemento w com a ordem desejada. A função de *hash* `SWIFFT` mapeia então chaves $\tilde{a}^{(1)}, \ldots, \tilde{a}^{(m/n)}$ e entradas $y^{(1)}, \ldots, y^{(m/n)} \in \{0, ..., d-1\}^m$ em $\sum_{i=1}^{m/n} \tilde{a}^{(i)}(Wy^{(i)})$. A matriz W verifica o seguinte facto:

> $Wf_A(y) = Wf_A(y') \mod q$, em que f_A é a função descrita na subsecção referente aos retículos ideais, se e só se $f_A(y) = f_A(y') \mod q$.

3.5.2 Algoritmo da função de *hash* SWIFFT

- **Parâmetros:** $n, m, q, d \in \mathbb{Z}$ com n uma potência única de dois, q primo, $2n|(q-1)$ e $n|m$.
- **Chave:** m/n vetores $\tilde{a}^{(1)}$,...,$\tilde{a}^{(m/n)} \in \mathbb{Z}_q^n$ escolhidos independentemente e uniformemente de forma aleatória.
- **Entrada:** m/n vetores $y^{(1)}$,...,$y^{(m/n)} \in \{0, ..., d-1\}^n$.
- **Saída:** o vetor $\sum_{i=1}^{m/n} \tilde{a}^{(i)}(Wy^{(i)})$, com $(Wy^{(i)}) \in \mathbb{Z}_q^n$.

3.5.3 Implementação em Sage

Função `procura_elemento_ordem_2n`

Esta função recebe como argumentos `ordem` $\in \mathbb{N}$ e o corpo de Galois `gf` e como saída retorna w pertencente a `gf` com ordem `ordem`. De seguida é apresentado o código da função em Sage:

```
def procura_elemento_ordem_2n(ordem,gf):
    w=gf.random_element();
    if(w==gf(0)):
        w=gf(1);
    while(w.multiplicative_order()!=ordem):
        w=gf.random_element();
        if(w==gf(0)):
            w=gf(1);
    return w;
```

Função `matrizW`

Esta função recebe como argumentos $n, q \in \mathbb{N}$ e como saída retorna a matriz $W \in \mathbb{Z}_q^{n \times n}$ tal que $W = \left[\ w^{(2i-1)(2j-1)} \ \right]_{i=1,j=1}^{n,n}$. De seguida é apresentado o código da função em Sage:

```
def matrizW(n,q):
    Zq=GF(q);
    w=procura_elemento_ordem_2n(2*n,Zq);
    W=zero_matrix(Zq,n);
    for i in range(0,n):
        for j in range(0,n):
            potencia=(2*i+1)*(2*j+1);
            W[i,j]=w^potencia;
    return W;
```

Função `gerachavesSWIFFT`

Esta função recebe como argumentos $q, m, n \in \mathbb{N}$ e como saída retorna a lista l com m/n vetores `ai` $\in \mathbb{Z}_q^n$, gerados aleatoriamente. De seguida é apresentado o código da função em Sage:

```
def gerachavesSWIFFT(q,m,n):
    Zq=GF(q);
    l=[];
    i=0;
    res=m/n;
    while i<res:
        ai=random_vector(Zq,n);
        l.append(ai);
        i=i+1;
    return l;
```

Função `entrada`

Esta função recebe como argumentos $m, n, d \in \mathbb{N}$ e como saída retorna a lista l com m/n vetores `yi` $\in \mathbb{Z}^n$, gerados aleatoriamente, tal que cada componente de `yi` pertence ao intervalo $[0, d-1]$. De seguida é apresentado o código da função em Sage:

```
def entrada(m,n,d):
    l=[];
    i=0;
    res=m/n;
    while i<res:
        yi=random_vector(ZZ,n,x=d);
        l.append(yi);
        i=i+1;
    return l;
```

Função `obtemdados`

Esta função recebe como argumentos $q, m, n, d \in \mathbb{N}$ e como saída retorna a lista *dados* que contém a matriz W, as chaves *key* e um dos parâmetros da função de *hash* `SWIFFT` designado *entradas*. Esta função depende da função `matrizW`, `gerachavesSWIFFT` e `entrada`. De seguida é apresentado o código da função em Sage:

```
def obtemdados(q,m,n,d):
    res=(q-1)%(2*n);
    dados=[];
    W=matrizW(n,q);
    key=gerachavesSWIFFT(q,m,n);
    entradas=entrada(m,n,d);
    dados.append(key);
    dados.append(W);
    dados.append(entradas);
    return dados;
```

Função `funhashSWIFFT`

Esta função recebe como argumentos as chaves *chave*, a matriz *mat* e a lista de entradas *yi* e como saída retorna $\sum_{i=1}^{m/n}$ `chave[i]*(mat*yi[i]`^{T}`)`. De seguida é apresentado o código da função em Sage:

```
def funhashSWIFFT(chave,mat,yi):
    saida=0;
    tam=len(chave);
    for i in range(0,tam):
        saida=saida+chave[i]*(mat*column_matrix(yi[i]));
    return saida[0];
```

3.5.4 Utilização das funções usando o Sage

Para usar as construções relativas à função de *hash* `SWIFFT`, tomam-se por exemplo, as seguintes instruções:

1. `qq=257;mm=8;nn=4;dd=2;`
2. `Dados=obtemdados(qq,mm,nn,dd);`
3. `Saida=funhashSWIFFT(Dados[0],Dados[1],Dados[2]);`

A primeira instrução associa a variável qq a 257, mm a 8, nn a 4 e dd a 2.

A segunda instrução associa a variável *Dados* ao resultado da função `obtemdados` que recebe como argumentos qq, mm, nn, dd, isto é, *Dados* é a lista constituída pelas chaves *key*, pela matriz W e pela lista de entradas *entrada* sendo estas respetivamente
[(171, 120, 163, 9),(8, 132, 91, 5)], $\begin{bmatrix} 193 & 253 & 64 & 4 \\ 253 & 193 & 4 & 64 \\ 64 & 4 & 193 & 253 \\ 4 & 64 & 253 & 193 \end{bmatrix}$ e [(0, 0, 1, 1), (1, 0, 0, 0)].

A terceira instrução associa a variável *Saida* ao resultado da função `funhashSWIFFT` que recebe como argumentos $Dados[0], Dados[1], Dados[2]$, isto é, *Saida* é 46.

3.6 Considerações

A função de *hash* de Ajtai, segundo [3], tem o problema de ser ineficiente, devido ao tamanho da chave, mas no entanto, possui algumas vantagens pelo facto de ser resistente a colisões e de sentido único.

A função de *hash* baseada em retículos ideais, segundo [3], tem a vantagem de o armazenamento das chaves ser bastante pequeno, o produto entre a matriz e o vetor requer menos tempo recorrendo à transformada rápida de Fourier bem como ser de sentido único.

Os autores propõem para parâmetros, de modo a satisfazer as propriedades referentes ao vetor f usado na construção da função, $f = (1, 1, ..., 1)$ com $n + 1$ primo ou
$f = (1, 0, ..., 0)$ em que n é uma potência de dois.

A função de *hash* `SWIFFT`, segundo [3], pode ser vista como a versão altamente otimizada da função de *hash* baseada em retículos ideais e na prática é também altamente eficiente. Os autores propõem como parâmetros $n = 64, m = 1024, q = 257, d = 2$ e $w = 42$.

Capítulo 4

Criptografia de Chave Pública

4.1 Contextualização

Na criptografia existem dois tipos de métodos, a criptografia de chave única ou simétrica e a criptografia de chave pública ou assimétrica. Na criptografia de chave única para cada par de chaves associado à cifração e à decifração é fácil determinar uma das chaves conhecendo a outra, sendo necessário que haja um acordo de chaves entre os intervenientes, antes de se proceder à comunicação e que sejam usados canais seguros. Devido à pré-distribuição de chaves ser um processo complicado opta-se pelo uso da criptografia de chave pública. Na criptografia de chave pública é usado um par de chaves, em que uma das chaves, designada de chave pública, é usada para o processo de cifração enquanto a outra chave, designada de chave privada, é usada para o processo de decifração, mantendo-se esta secreta. A criptografia de chave pública permite tornear o problema de distribuição de chaves bem como o conhecimento de uma das chaves, não permite o conhecimento da outra.

4.2 GGH

Este criptossistema baseado em retículos foi proposto por Goldreich, Goldwasser e Halevi que se baseia no problema do vetor mais perto. A construção deste criptossistema tem como referência [1].

4.2.1 Geração de Chaves

O processo de geração de chaves recebe como argumentos $n, l \in \mathbb{N}$ e opera da seguinte forma:

1. A chave privada é uma matriz $R \in \mathbb{R}^{n \times n}$ tal que $R = R' + kI_n$, com I_n a matriz identidade de ordem n, com $R' = (r'_{ij})$ tal que $|r'_{ij}| \leq l$, para $k \approx \sqrt{n}l$.
2. A chave pública é uma matriz $B \in \mathbb{R}^{n \times n}$ tal que $B = RT^{-1}$, com $T \in \mathrm{Gl}_n(\mathbb{Z})$ e T uma matriz unimodular.

Como saída retorna a chave privada R, a matriz unimodular T e a chave pública B.

4.2.2 Obtenção de σ

O processo de obtenção de σ recebe como argumentos a chave privada R e um $\epsilon \in \mathbb{R}$ com $\epsilon > 0$ e opera da seguinte forma:

1. Calcular $\gamma = \sqrt{n}i$, com i o máximo da norma infinito das linhas de R^{-1}.

Como saída retorna σ, tal que $\sigma = \left[\frac{1}{\gamma\sqrt{8ln(2n/\epsilon)}} \right]$, em que $[a]$ representa o maior $x \in \mathbb{Z}$ tal que $x \leq a$, com a probabilidade de erro na decifração, ser limitada por ϵ.

4.2.3 Cifração

O processo de cifração recebe como argumentos a chave pública B, a mensagem $m \in \mathbb{Z}^n$ e um $\sigma \in \mathbb{N}$ e opera da seguinte forma:

1. Calcular $v = Bm$.

Como saída retorna o criptograma c, tal que $c = v + r$, com r vetor de erros e $r = (\delta_1\sigma, \ldots, \delta_n\sigma)$ com $\delta_i = \pm 1$.

4.2.4 Decifração

O processo de decifração recebe como argumentos a chave privada R, o criptograma c e a matriz unimodular T e opera da seguinte forma:

1. Calcular $z = \lceil R^{-1}c \rfloor$, em que $\lceil v \rfloor$ corresponde ao vetor em $\mathbb{Z}^n$ que é obtido por arredondamento de cada entrada de v ao inteiro mais próximo, sendo este processo designado de método de arredondamento de Babai.

Como saída retorna a mensagem original mo, tal que $mo = Tz$.

4.2.5 Implementação em Sage

Função `encontra_maior_valor_matriz`

Esta função recebe como argumento a matriz R e como saída retorna $max \in \mathbb{R}$ que corresponde à maior entrada em valor absoluto da matriz. De seguida é apresentado o código da função em Sage:

```
def encontra_maior_valor_matriz(R):
    n=R.nrows();
    m=R.ncols();
    max=abs(R[0,0]);
    for i in range(0,n):
        for j in range(0,m):
            if max<abs(R[i,j]):
                max=abs(R[i,j]);
    return max;
```

Função `obtemsigma`

Esta função recebe como argumentos $e > 0 \in \mathbb{R}$ e a chave privada $Priv$ e como saída retorna $\sigma \in \mathbb{Z}$ tal que $\sigma = \left\lfloor \frac{1}{\alpha\sqrt{8ln(2n/e)}} \right\rfloor$, em que $\alpha = \sqrt{n}$*`encontra_maior_valor_matriz`$(Priv^{-1})$, com n o número de linhas da matriz $Priv$. Esta função depende da função `encontra_maior_valor_matriz`. De seguida é apresentado o código da função em Sage:

```
def obtemsigma(e,Priv):
    n=Priv.nrows();
    alfa=sqrt(n*1.0)*encontra_maior_valor_matriz(Priv^(-1));
    aux=1/(alfa*sqrt(8*ln((2*n)/e)));
    sigma=floor(aux);
    return sigma;
```

Função `gerachavesGGH`

Esta função recebe como argumentos $n > 0, l > 0 \in \mathbb{N}$ e como saída retorna a chave pública Pub, a matriz unimodular U, a chave privada $Priv$ tal que $Priv = Ra + k * I$ e $Pub = Priv{*}U^{-1}$, com os valores de Ra em módulo menores ou iguais a l, $k \approx \sqrt{n} * l$ e I a matriz identidade de ordem n. De seguida é apresentado o código em Sage:

```
def gerachavesGGH(n,l):
    Ra=random_matrix(ZZ,n,n,x=-l,y=l+1);
    k=(sqrt(n*1.0)*l).round();
    U=random_matrix(ZZ,n,n,algorithm='unimodular');
    Priv=Ra+k*identity_matrix(n);
    while det(Priv)==0:
        Ra=random_matrix(ZZ,n,n,x=-l,y=l+1);
        Priv=Ra+k*identity_matrix(n);
    Priv=Priv*1.0;
    Pub=Priv*U^(-1);
    return (Pub,U,Priv);
```

Função `cifraGGH`

Esta função recebe como argumentos a chave pública Pub, o vetor $m \in \mathbb{Z}^n$ e um $o \in \mathbb{Z}$ e como saída retorna o criptograma $c = v + r$, onde r é um vetor cujas componentes são $+o$ ou $-o$ e $v = Pub * m$. De seguida é apresentado o código da função em Sage:

```
def cifraGGH(Pub,m,o):
    tam=len(m);
    r=zero_vector(RR,tam);
    v=Pub*column_matrix(m);
    i=0;
    while i < tam:
        num=randint(-1,1);
        if num==-1 or num==1:
            r[i]=num*o;
            i=i+1;
    c=v+column_matrix(r);
    return c;
```

Função `decifraGGH`

Esta função recebe como argumentos a chave privada $Priv$, a matriz unimodular U e o criptograma $cripto$ e como saída retorna a mensagem original mo tal que $mo = U * y$, com $y = \lceil Priv^{-1} * cripto \rfloor$. De seguida é apresentado o código da função em Sage:

```
def decifraGGH(Priv,U,cripto):
    res=Priv^(-1)*cripto;
    y=res.apply_map(lambda x: x.round());
    mo=U*y;
    return vector(mo.list());
```

4.2.6 Utilização das funções usando o Sage

Para usar a construção relativa à cifra GGH, tomam-se por exemplo, as seguintes instruções:

1. `n=4;l=20;m=random_vector(ZZ,n);`
2. `Keys=gerachavesGGH(n,l);`
3. `e=0.1;`
4. `o=obtemsigma(e,Keys[2]);`
5. `cripto=cifraGGH(Keys[0],m,o);`
6. `mo=decifraGGH(Keys[2],Keys[1],cripto);`

A primeira instrução associa a variável n a 4, l a 20, m é um vetor gerado aleatoriamente, isto é, considerando m como sendo o vetor $\begin{bmatrix} 1 & -1 & 126 & -1 \end{bmatrix}$.

A segunda instrução associa a variável $Keys$ ao resultado da função `gerachavesGGH` que recebe como argumentos n, l, isto é, $Keys$ é um tuplo que contém a chave pública Pub, a matriz unimodular U e a chave privada $Priv$ que são consideradas respetivamente como sendo $\begin{bmatrix} -10517.0 & -34948.0 & 771.0 & 145.0 \\ -2626.0 & -8745.0 & 198.0 & 51.0 \\ -11643.0 & -38651.0 & 822.0 & 152.0 \\ -9362.0 & -31097.0 & 669.0 & 133.0 \end{bmatrix}$, $\begin{bmatrix} 13 & -68 & -81 & 366 \\ -4 & 21 & 25 & -113 \\ -3 & 20 & 24 & -110 \\ -5 & 23 & 23 & -104 \end{bmatrix}$ e $\begin{bmatrix} 33.0 & 3.0 & 16.0 & 12.0 \\ -7.0 & 56.0 & 6.0 & -15.0 \\ 19.0 & -11.0 & 32.0 & -3.0 \\ 10.0 & 18.0 & 12.0 & 47.0 \end{bmatrix}$.

A terceira instrução associa a variável e a 0.1.

A quarta instrução associa a variável o ao resultado da função `obtemsigma` que recebe como argumentos e e $Keys[2]$, isto é, o é 1.

A quinta instrução associa a variável *cripto* ao resultado da função `cifraGGH` que recebe como argumentos $Keys[0], m, o$, isto é, considerando *cripto* como sendo o vetor $\begin{bmatrix} 121431.0 & 31015.0 & 130427.0 & 105895.0 \end{bmatrix}$.

A sexta instrução associa a variável *mo* ao resultado da função `decifraGGH` que recebe como argumentos $Keys[2], Keys[1], cripto$, isto é, *mo* é o vetor $\begin{bmatrix} 1 & -1 & 126 & -1 \end{bmatrix}$.

4.3 NTRU

Este criptossistema baseado em retículos foi proposto por Jeffrey Hoffstein, Jill Pipher e Joseph H. Silverman que se baseia no problema do vetor mais pequeno. A primeira versão do NTRU foi construída usando polinómios, mas mais tarde foram apresentadas novas versões recorrendo a matrizes e a vetores. A construção deste criptossistema, na versão polinomial, tem como referência [5].

4.3.1 Retículo NTRU de Convolução Modular

Um retículo NTRU de convolução modular L_h, segundo [3] e [7], associado ao vetor h e módulo q é um retículo de dimensão $2n$, da seguinte forma:

$$\begin{bmatrix} I_n & 0_n \\ T^*h & q * I_n \end{bmatrix},$$

onde I_n é a matriz identidade de ordem n, 0_n é a matriz de ordem n com entradas todas a zero,

$$T * h = \begin{bmatrix} h_0 & h_{n-1} & \dots & h_2 & h_1 \\ h_1 & h_0 & \dots & h_3 & h_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ h_{n-2} & h_{n-3} & \dots & h_0 & h_{n-1} \\ h_{n-1} & h_{n-2} & \dots & h_1 & h_0 \end{bmatrix},$$

ou seja, consiste na matriz que roda as coordenadas do vetor h (primeira coluna) ciclicamente, $q * I_n$ corresponde à matriz em que fora da diagonal as entradas são zero e na diagonal as entradas são q. Para verificar se $f(x)h(x) \equiv g(x) \mod q$, tem que se provar que $[f, g] = [f_0, \dots, f_{N-1}, g_0, \dots, g_{N-1}] \in L_h$, em que cada f_i e g_i corresponde ao coeficiente da i-ésima potência de x de f e g respetivamente, isto é, tomando $u(x) = \frac{-f(x)+g(x)}{q}$ corresponde a verificar se $[f, u]L_h^T = [f, g]$. L_h é então um retículo q-ário associado a $R = \mathbb{Z}[x]/(x^N - 1)$.

4.3.2 Definições e notações

Seja N um primo e $R = \mathbb{Z}[x]/(x^N - 1)$ o anel de polinómios de grau $\leq N - 1$, onde a multiplicação é feita módulo $x^N - 1$. Seja $f \in R$ tal que f é representado por $f(x) = f_0 + f_1 x + \cdots + f_{N-1}x^{N-1}$, em que cada $f_i \in \mathbb{Z}$. Define-se então o seguinte:

- $wt_a(f) = \#\{i \mid f_i = a\}$, para $a \in \mathbb{Z}$.
- Seja $d_1, d_2 \in \mathbb{N}$ tal que $d_1 + d_2 \leq N$. Então define-se $\mathbf{L}(d_1,d_2) = \{f \in R \mid wt_1(f) = d_1, wt_{-1}(f) = d_2, wt_0(f) = N - d_1 - d_2\}$.

Seja $p, k, d \in \mathbb{N}$ com $p > 1, 0 < k < N$ e $d \leq N - k - 1$. Define-se o espaço das mensagens da seguinte forma:

- $M = \left\{m \in R \mid m = m_0 + \cdots + m_d x^d, |m_i| \leq \frac{p-1}{2}\right\}$, com $m_i \in \mathbb{Z}$.

Na cifra NTRU serão considerados os conjuntos $L_f = \mathbf{L}(d_f,d_f\text{-}1)$, $L_g = \mathbf{L}(d_g,d_g)$, $L_r = \mathbf{L}(d_r,d_r)$ e M, com $d_f,d_g,d_r \in \mathbb{N}$.

4.3.3 Geração de Chaves

O processo de geração de chaves recebe como argumentos um primo $N, p, q, d_f, d_g \in \mathbb{N}$, tal que $p > 1$ e $q > p$, com $0 < d_f, d_g < \frac{N}{2}$. Opera da seguinte forma:

1. As chaves privadas são $f \in L_f$ e $f_p \in \mathbb{Z}_p[x]/(x^N - 1)$, com $[f]_p * f_p = 1$ em $\mathbb{Z}_p[x]/(x^N - 1)$ e em que existe f_q tal que $[f]_q * f_q = 1$ em $\mathbb{Z}_q[x]/(x^N - 1)$, com $[f]_i \in \mathbb{Z}_i[x]/(x^N - 1)$.
2. A chave pública é h tal que $h = pf_qg$, com $g \in L_g$, $f_q \in \mathbb{Z}_q[x]/(x^N - 1)$ e $h \in \mathbb{Z}_q[x]/(x^N - 1)$.

Como saída retorna a chave pública h, a chave privada f_p e a chave privada f.

4.3.4 Cifração

O processo de cifração recebe como argumentos a chave pública h, a mensagem $m \in M$, um primo N e $d_r \in \mathbb{N}$, com $0 < d_r < \frac{N}{2}$. Opera da seguinte forma:

1. Calcular $v = rh$, com $r \in L_r$ um polinómio aleatório.

Como saída retorna o criptograma $c \in \mathbb{Z}_q[x]/(x^N - 1)$, tal que $c = (v + m) \mod q$.

4.3.5 Decifração

O processo de decifração recebe como argumentos as chaves privadas f_p e f, o criptograma c, um primo N e $p, q \in \mathbb{N}$. Opera da seguinte forma:

1. Calcular $y = fc \mod q$, com os coeficientes de y no intervalo $]\frac{-q}{2}, \frac{q}{2}[$.
2. Calcular $z = [y]_p$, com $z \in \mathbb{Z}_p[x]/(x^N - 1)$.

Como saída retorna a mensagem original mo, tal que $mo = f_p z \mod p$, com os coeficientes de mo no intervalo $]\frac{-p}{2}, \frac{p}{2}[$.

4.3.6 Implementação em Sage

Função `cria_polinomio`

Esta função recebe como argumentos $N, ones, negones \in \mathbb{N}$ e como saída retorna o polinómio *pol* com grau $N - 1$ e com coeficientes inteiros entre [-1,1], com *ones* coeficientes a 1 e *negones* coeficientes a -1 sendo os restantes 0. De seguida é apresentado o código da função em Sage:

```
def gerapol(N,ones,negones):
    r=zero_vector(ZZ,N);
    uns=ones;
    nuns=negones;
    while(uns!=0 or nuns!=0):
        pos=randint(0,N-1);
        if r[pos]==0:
            if uns>0:
                r[pos]=1;
                uns=uns-1;
            elif nuns>0:
                r[pos]=-1;
                nuns=nuns-1;
    Pol.<x>=PolynomialRing(ZZ);
    pol=Pol(r.list());
    return pol;
```

Função `cria_mensagem`

Esta função recebe como argumentos `grau`, $p \in \mathbb{N}$ e como saída retorna o polinómio *pol* com grau `grau` e com coeficientes inteiros entre $]\frac{-p}{2}, \frac{p}{2}[$. De seguida é apresentado o código da função em Sage:

```
def cria_mensagem(grau,p):
    Pol.<x>=PolynomialRing(ZZ);
    pol=Pol.random_element(grau,-(p-1)/2,((p-1)/2)+1);
    return pol;
```

Função `inverso_modular`

Esta função recebe como argumentos $N \in \mathbb{N}$, o polinómio f e $p \in \mathbb{N}$ e como saída retorna o polinómio fpi tal que $f * fpi = 1$ em $\mathbb{Z}_p[x]/(x^N - 1)$. De seguida é apresentado o código da função em Sage:

```
def inverso_modular(N,f,p):
    Pol.<x>=PolynomialRing(IntegerModRing(p));
    fpi=xgcd(f,x^N-1)[1];
    return fpi;
```

Função `chavepublica`

Esta função recebe como argumentos $N \in \mathbb{N}$, o polinómio f, o polinómio g, $p \in \mathbb{N}$, $q \in \mathbb{N}$ e o inverso fqi de f em $\mathbb{Z}_q[x]/(x^N - 1)$ e como saída retorna a chave pública $p * q * fqi$. De seguida é apresentado o código da função em Sage:

```
def chavepublica(N,f,g,p,q,fqi):
    Pol.<x>=PolynomialRing(IntegerModRing(q));
    Qn=QuotientRing(Pol,x^N-1,names="x");
    return Qn(p*g*fqi);
```

Função `f_cond`

Esta função recebe como argumentos o polinómio f, o polinómio fpi, o polinómio fqi e $p, q, N \in \mathbb{N}$ e como saída retorna verdadeiro se $f * fpi = 1$ em $\mathbb{Z}_p[x]/(x^N - 1)$ e $f * fqi = 1$ em $\mathbb{Z}_q[x]/(x^N - 1)$, falso caso contrário. De seguida é apresentado o código da função em Sage:

```
def f_cond(f,fpi,fqi,p,q,N):
    PP.<z>=PolynomialRing(IntegerModRing(p));
    QQ.<y>=PolynomialRing(IntegerModRing(q));
    QPP=QuotientRing(PP,z^N-1);
    QQQ=QuotientRing(QQ,y^N-1);
    auxpf=PP(f.change_variable_name(z));
    auxqf=QQ(f.change_variable_name(y));
    pp=fpi.change_variable_name(z);
    qq=fqi.change_variable_name(y);
    if QPP(auxpf*pp)==1 and QQQ(auxqf*qq)==1:
        return True;
    return False;
```

Função `gera_f`

Esta função recebe como argumentos $N, p, q, df \in \mathbb{N}$ e como saída retorna o polinómio f tal que f contém df 1's df-1 -1's e o resto 0's e existe fpi e fqi tal que $f * fpi = 1$ em $\mathbb{Z}_p[x]/(x^N - 1)$ e $f * fqi = 1$ em $\mathbb{Z}_q[x]/(x^N - 1)$. Esta função depende da função `gerapol`. De seguida é apresentado o código da função em Sage:

```
def gera_f(N,p,q,df):
    Zp.<z>=PolynomialRing(IntegerModRing(p));
    Zq.<y>=PolynomialRing(IntegerModRing(q));
    f=gerapol(N,df,df-1);
    pp=Zp(f.change_variable_name(z));
    qq=Zq(f.change_variable_name(y));
    while(xgcd(pp,z^N-1)[0]!=1 or xgcd(qq,y^N-1)[0]!=1):
        f=gerapol(N,df,df-1);
        pp=Zp(f.change_variable_name(z));
        qq=Zq(f.change_variable_name(y));
    return f;
```

Função `gerachavesNTRU`

Esta função recebe como argumentos $N, p, q, df, dg \in \mathbb{N}$ e como saída retorna a chave pública pub, as chaves privada fpi e f tal que $f * fpi = 1$ em $\mathbb{Z}_p[x]/(x^N - 1)$. Esta função depende da função `gera_pol`, `gera_f`, `inverso_modular` e `chavepublica`. De seguida é apresentado o código da função em Sage:

```
def gerachavesNTRU(N,p,q,df,dg):
    g=gerapol(N,dg,dg);
    f=gera_f(N,p,q,df);
    fqi=inverso_modular(N,f,q);
    fpi=inverso_modular(N,f,p);
    while(f_cond(f,fpi,fqi,p,q,N)!=True):
        f=gera_f(N,p,q,df);
        fqi=inverso_modular(N,f,q);
        fpi=inverso_modular(N,f,p);
    pub=chavepublica(N,f,g,p,q,fqi);
    return (pub,fpi,f);
```

Função `cifraNTRU`

Esta função recebe como argumentos a chave pública pub, a mensagem $m \in M$ e $N, dr \in \mathbb{N}$ e como saída retorna o criptograma $r*pub+mens$, tal que r é um polinómio que contém dr -1's e 1's e o resto 0's. Esta função depende da função `gerapol`. De seguida é apresentado o código da função em Sage:

```
def cifraNTRU(pub,mens,N,dr):
    r=gerapol(N,dr,dr);
    return r*pub+mens;
```

Função `cons_pol_interval`

Esta função recebe como argumentos a lista `lista` e $q \in \mathbb{N}$ e como saída retorna o polinómio *pol* com coeficientes inteiros entre $]\frac{-q}{2}, \frac{q}{2}[$ determinados a partir da lista de coeficientes que representam o polinómio em $\mathbb{Z}_q[x]$. De seguida é apresentado o código da função em Sage:

```
def cons_pol_interval(lista,q):
    lim=floor(q/2);
    comp=len(lista);
    pol=0;
    Pol.<x>=PolynomialRing(ZZ);
    for i in range(0,comp):
        if ZZ(lista[i])>lim:
           pol=pol+(ZZ(lista[i])-q)*x^i;
        else:
            pol=pol+ZZ(lista[i])*x^i;
    return pol;
```

Função `decifraNTRU`

Esta função recebe como argumentos as chaves privadas f, fpi, o criptograma *cripto* e $q, p, N \in \mathbb{N}$ e como saída retorna a mensagem original *mo* tal que $mo = (fpi*(f*cripto \mod q)) \mod p$. Esta função depende da função `cons_pol_interval`. De seguida é apresentado o código da função em Sage:

```
def decifraNTRU(f,fpi,cripto,p,N):
    aux=f*cripto;
    aux=cons_pol_interval(aux.list(),q);
    Zp.<x>=PolynomialRing(IntegerModRing(p));
    aux=Zp(aux);
    Qp=QuotientRing(Zp,x^N-1,names="x");
    mo=Qp(fpi*aux);
    mo=cons_pol_interval(mo.list(),p);
    return mo;
```

4.3.7 Utilização das funções usando o Sage

Para usar a construção relativa à cifra NTRU, tomam-se por exemplo, as seguintes instruções:

1. `N=11;p=3;q=41;`
2. `df=randint(1,floor(N/2.0));`
3. `dg=randint(1,floor(N/2.0));`
4. `dr=randint(1,floor(N/2.0));`
5. `Chaves=gerachavesNTRU(N,p,q,df,dg);`
6. `k=randint(1,N-1);`
7. `dm=randint(0,N-k-1);`
8. `mens=cria_mensagem(dm,p);`
9. `cripto=cifraNTRU(Chaves[0],mens,N,dr);`
10. `mo=decifraNTRU(Chaves[2],Chaves[1],cripto,q,p,N);`

A primeira instrução associa a variável N a 11, p a 3 e q a 41.

A segunda, terceira e quarta instrução associam as variáveis df, dg, dr a valores gerados aleatoriamente, pertencentes a $[1, \frac{N}{2}[$, isto é, considerando df como sendo 3, dg como sendo 2 e dr como sendo 2.

A quinta instrução associa a variável $Chaves$ ao resultado da função `gerachavesNTRU` que recebe como argumentos N, p, q, df, dg, isto é, $Chaves$ é um tuplo que contém a chave pública pub, a chave privada fpi, a chave privada f que são consideradas respetivamente como sendo os polinómios $12*x^{10}+22*x^9+6*x^8+33*x^7+38*x^6+29*x^5+30*x^4+36*x^3+3*x^2+13*x+24$, $2*x^{10}+2*x^8+2*x^7+2*x^3+x^2+x$ e $x^{10}+x^7-x^4-x^2+x$.

A sexta instrução e sétima instrução associam as variáveis k, dm a valores gerados aleatoriamente, pertencentes respetivamente a $[1, N-1]$ e a $[0, N-k-1]$, isto é, considerando k como sendo 5 e dm como sendo 5.

A oitava instrução associa a variável $mens$ ao resultado da função `cria_mensagem` que recebe como argumentos dm, p, isto é, considerando $mens$ como sendo o polinómio $-x^5-x^4+x^3-x^2-x+1$.

A nona instrução associa a variável $cripto$ ao resultado da função `cifraNTRU` que recebe como argumentos $Chaves[0], mens, N, dr$, isto é, considerando $cripto$ como sendo o polinómio $21*x^{10}+31*x^9+26*x^8+13*x^7+13*x^6+13*x^5+26*x^4+38*x^3+17*x^2+36*x+10$.

A décima instrução associa a variável mo ao resultado da função `decifraNTRU` que recebe como argumentos $Chaves[2], Chaves[1], cripto, q, p, N$, isto é, mo é o polinómio $-x^5-x^4+x^3-x^2-x+1$.

4.4 LWE

Este criptossistema baseado em retículos foi proposto por Oded Regev baseado no problema aproximado do vetor mais pequeno e no problema dos vetores independentes mais pequenos. A construção deste criptossistema tem como referência [3].

4.4.1 Definições

Seja $q, t, l \in \mathbb{N}$. Seja f a função que mapeia vetores de $\mathbb{Z}_t^l$ em vetores de $\mathbb{Z}_q^l$, multiplicando cada coordenada do vetor por $\frac{q}{t}$ e arredondando ao inteiro mais próximo e seja f^{-1} a função que mapeia vetores de $\mathbb{Z}_q^l$ em vetores de $\mathbb{Z}_t^l$, multiplicando cada coordenada do vetor por $\frac{t}{q}$ e arredondando ao inteiro mais próximo.
Na cifra LWE será usada a função f para a cifração e a função f^{-1} para a decifração.

4.4.2 Geração de Chaves

O processo de geração de chaves recebe como argumentos $n, m, l, t, r, q \in \mathbb{N}$ e $\alpha > 0 \in \mathbb{R}$ e opera da seguinte forma:

1. A chave privada é uma matriz aleatória $S \in \mathbb{Z}_q^{n \times l}$.
2. As chaves públicas são uma matriz aleatória $A \in \mathbb{Z}_q^{m \times n}$ e uma matriz $P \in \mathbb{Z}_q^{m \times l}$ tal que $P = AS + E$ com $E \in \mathbb{Z}_q^{m \times l}$, uma matriz onde as entradas são escolhidas de acordo com $\overline{\Psi}_\alpha$, em que $\overline{\Psi}_\alpha$ denota a distribuição em $\mathbb{Z}_q$ obtida, por amostragem, de uma variável normal com média 0 e desvio padrão $\frac{\alpha q}{\sqrt{2\pi}}$, arredondando o resultado para o inteiro mais próximo e reduzindo módulo q.

Como saída retorna a chave privada S, a chave pública A e a chave pública P.

4.4.3 Cifração

O processo de cifração recebe como argumentos $q, t, r \in \mathbb{N}$, as chaves públicas $A \in \mathbb{Z}_q^{m \times n}$, $P \in \mathbb{Z}_q^{m \times l}$, a mensagem $v \in \mathbb{Z}_t^l$ e opera da seguinte forma:

1. Escolher vetor aleatório $a \in \{-r, \ldots, r\}^m$.
2. Calcular $u = A^T a$, com $u \in \mathbb{Z}_q^n$.
3. Calcular $c = P^T a + f(v)$, com $c \in \mathbb{Z}_q^l$.

Como saída retorna o criptograma $(u, c) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l$.

4.4.4 Decifração

O processo de decifração recebe como argumentos $q, t \in \mathbb{N}$, o criptograma $(u, c) \in \mathbb{Z}_q^n \times \mathbb{Z}_q^l$, a chave privada $S \in \mathbb{Z}_q^{n \times l}$ e opera da seguinte forma:

1. Calcular $b = c - S^T u$.

Como saída retorna a mensagem original $mo \in \mathbb{Z}_t^l$, tal que $mo = f^{-1}(b)$.

4.4.5 Implementação em Sage

Função `gera_matriz_erro`

Esta função recebe como argumentos $q, m, l \in \mathbb{N}$, $alpha > 0 \in \mathbb{R}$ e como saída retorna a matriz E pertencente a $\mathbb{Z}_q^{m \times l}$ em que E é construída de acordo com $\overline{\Psi}_{alpha}$. De seguida é apresentado o código da função em Sage:

```
def gera_matriz_erro(q,m,l,alpha):
    Zq=IntegerModRing(q);
    sigma=(alpha*q)/sqrt(2*math.pi);
    Eaux=numpy.matrix(numpy.random.normal(0,sigma,size=(m,l)));
    Eaux=Eaux.round();
    mat=MatrixSpace(Zq,m,l);
    E=mat(matrix(Eaux));
    return E;
```

Função `fun_f`

Esta função recebe como argumentos $q, t \in \mathbb{N}$, o vetor vec pertencente a $\mathbb{Z}_t^l$ e como saída retorna o vetor aux, em que cada coordenada é multiplicada por $\frac{q}{t}$, arredondada ao inteiro mais próximo e reduzida módulo q. De seguida é apresentado o código da função em Sage:

```
def fun_f(q,t,vec):
    tam=len(vec);
    Zq=IntegerModRing(q);
    aux=zero_vector(Zq,tam);
    for i in range(0,tam):
        coord=ZZ(vec[i])*(q*1.0/t);
        nat=round(coord);
        aux[i]=Zq(nat);
    return aux;
```

Função `fun_finv`

Esta função recebe como argumentos $q, t \in \mathbb{N}$, o vetor vec pertencente a $\mathbb{Z}_q^l$ e como saída retorna o vetor aux, em que cada coordenada é multiplicada por $\frac{t}{q}$, arredondada ao inteiro mais próximo e reduzida módulo t. De seguida é apresentado o código da função em Sage:

```
def fun_finv(q,t,vec):
    tam=len(vec);
    Zt=IntegerModRing(t);
    aux=zero_vector(Zt,tam);
    for i in range(0,tam):
        coord=ZZ(vec[i])*(t*1.0/q);
        nat=round(coord);
        aux[i]=Zt(nat);
    return aux;
```

Função `gerachavesLWE`

Esta função recebe como argumentos $n, m, l, t, r, q \in \mathbb{N}$, $alpha > 0 \in \mathbb{R}$ e como saída retorna as chaves públicas A, Pub, a chave privada $Priv$ tal que $Pub = A * Priv + E$, em que $E =$ `gera_matriz_erro`$(q, m, l, alpha)$. Esta função depende da função `gera_matriz_erro`. De seguida é apresentado o código em Sage:

```
def gerachavesLWE(n,m,l,t,r,q,alpha):
    Priv=random_matrix(IntegerModRing(q),n,l);
    A=random_matrix(IntegerModRing(q),m,n);
    E=gera_matriz_erro(q,m,l,alpha);
    Pub=A*Priv+E;
    return (Priv,A,Pub);
```

Função `cifraLWE`

Esta função recebe como argumentos $q, t, r \in \mathbb{N}$, a mensagem $mens$ pertencente a $\mathbb{Z}_t^l$, as chaves públicas A, Pub e retorna como saída o criptograma (u, c), tal que $u = A^T * vr$ e $c = Pub^T * vr +$ `fun_f`$(q, t, mens)$, em que vr é um vetor pertencente a $\{-r, ..., r\}^m$. Esta função depende da função `fun_f`. De seguida é apresentado o código da função em Sage:

```
def cifraLWE(q,t,r,mens,A,Pub):
    tam=A.nrows();
    vr=random_vector(ZZ,tam,x=-r,y=r+1);
    u=A.transpose()*column_matrix(vr);
    fg=fun_f(q,t,mens);
    c=Pub.transpose()*column_matrix(vr)+column_matrix(fg);
    return (u,c);
```

Função `decifraLWE`

Esta função recebe como argumentos $q, t \in \mathbb{N}$, o criptograma (u, c), a chave privada $Priv$ e como saída retorna a mensagem original mo tal que $mo =$ `fun_finv`$(c - Priv^T * u)$. Esta função depende da função `fun_finv`. De seguida é apresentado o código da função em Sage:

```
def decifraLWE(q,t,u,c,Priv):
    aux=c-Priv.transpose()*u;
    vc=vector(aux.list());
    mo=fun_finv(q,t,vc);
    return mo;
```

4.4.6 Utilização das funções usando o Sage

Para usar a construção relativa à cifra LWE, tomam-se por exemplo, as seguintes instruções:

1. `n=4;l=4;m=5;q=4093;r=4;t=2;alpha=0.0024;`
2. `vc=random_vector(IntegerModRing(t),l);`
3. `Chaves=gerachavesLWE(n,m,l,t,r,q,alpha);`
4. `cripto=cifraLWE(q,t,r,vc,Chaves[1],Chaves[2]);`
5. `mo=decifraLWE(q,t,cripto[0],cripto[1],Chaves[0]);`

A primeira instrução associa a variável n a 4, l a 4, m a 5, q a 4093, r a 4, t a 2 e $alpha$ a 0.0024.

A segunda instrução associa a variável vc como sendo um vetor gerado aleatoriamente, isto é, considerando vc como sendo o vetor $\begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$.

A terceira instrução associa a variável $Chaves$ ao resultado da função `gerachavesLWE` que recebe como argumentos $n, l, m, q, r, t, alpha$, isto é, $Chaves$ é um tuplo que contém a chave privada $Priv$, a chave pública A e a chave pública Pub que são consideradas respetivamente como sendo $\begin{bmatrix} 2968 & 3462 & 167 & 498 \\ 0 & 3902 & 3936 & 3641 \\ 2781 & 3583 & 570 & 1911 \\ 1885 & 1070 & 3318 & 3118 \end{bmatrix}$, $\begin{bmatrix} 3205 & 3299 & 3935 & 3439 \\ 1871 & 3274 & 3810 & 2395 \\ 1928 & 1939 & 2041 & 2058 \\ 2552 & 3732 & 2613 & 2722 \\ 349 & 24 & 3127 & 3088 \end{bmatrix}$ e $\begin{bmatrix} 2159 & 2734 & 222 & 2697 \\ 1843 & 585 & 3506 & 1810 \\ 2585 & 3998 & 3457 & 601 \\ 2310 & 1711 & 1893 & 3912 \\ 3589 & 2921 & 352 & 792 \end{bmatrix}$.

A quarta instrução associa a variável $cripto$ ao resultado da função `cifraLWE` que recebe como argumentos $q, t, r, vc, Chaves[1], Chaves[2]$, isto é, $cripto$ é um tuplo que contém o vetor u e o vetor c considerados respetivamente como sendo $\begin{bmatrix} 3357 & 1637 & 338 & 1975 \end{bmatrix}$ e $\begin{bmatrix} 90 & 1094 & 1204 & 2137 \end{bmatrix}$.

A quinta instrução associa a variável mo ao resultado da função `decifraLWE` que recebe como argumentos $q, t, cripto[0], cripto[1], Chaves[0]$, isto é, mo é o vetor $\begin{bmatrix} 1 & 0 & 0 & 1 \end{bmatrix}$.

4.5 Considerações

A cifra GGH, segundo [1], tem a desvantagem de ter tamanhos de chave impraticáveis e terem surgidos ataques que quebraram a cifra para determinados parâmetros propostos pelos autores da cifra.

A cifra NTRU, segundo [3] e [4], é considerado o criptossistema baseado em retículos mais eficiente e também prático de implementar. Para parâmetros de entrada considerados relativos ao NTRU, ver [3] ou [7].

O LWE, segundo [4] e [3], é considerado o criptossistema mais eficiente que admite uma prova teórica de segurança e apesar de não ser tão eficiente como o NTRU, mantém uns níveis de desempenho razoáveis que permite que o LWE seja usado na prática. Para parâmetros de entrada considerados relativos ao LWE, ver [3].

Conclusão

O principal objetivo desta tese consistia em estudar a utilidade dos retículos na criptografia. Posteriormente, fazendo uso de ferramentas informáticas, neste caso o uso da ferramenta Sage, foi feita a implementação de funções de *hash*, de criptossistemas de chave pública, do algoritmo de redução de base e uma implementação que mostrava graficamente o retículo, dado uma determinada base.

As primeiras aplicações criptográficas baseadas em retículos a surgir foram as funções de *hash*, nas quais se destaca a função de *hash* de Ajtai, a função de *hash* baseada em retículos ideais e a função de *hash* `SWIFFT`, que tem utilidade para armazenamento de palavras-chaves, provas de posse de informação e componente de outras técnicas criptográficas, sendo assim aplicações muito importantes e de uma grande utilidade na criptografia.

A função de Ajtai tem a desvantagem de ser ineficiente, devido ao tamanho da chave, mas no entanto, possui algumas vantagens pelo facto de possuir um esquema natural de assinatura e ser resistente a colisões e de sentido único.

A função de *hash* baseada em retículos ideais tem a vantagem de o armazenamento das chaves ser bastante pequeno, o produto entre a matriz e o vetor requerer menos tempo recorrendo à transformada rápida de Fourier, bem como ser de sentido único.

Relativamente à função `SWIFFT`, pode ser vista como a versão altamente otimizada da função de *hash* baseada em retículos ideais. Na prática é também altamente eficiente, pois o produto entre a matriz e o vetor, pode ser realizado de maneira eficiente, recorrendo novamente à transformada rápida de Fourier.

Nos criptossistemas baseados em retículos destaca-se principalmente o GGH, o NTRU e o LWE.

O GGH tem a desvantagem de ter tamanhos de chave impraticáveis para valores elevados do parâmetro n, pois estas são matrizes quadradas de dimensão $n \times n$.

O NTRU é considerado o criptossistema de chave pública baseado em retículos mais eficiente e também bastante prático de implementar. No entanto, a nível de segurança o GGH tenha vantagem sobre o NTRU.

O LWE é considerado o criptossistema mais eficiente que admite uma prova teórica de segurança, e apesar de não ser tão eficiente como o NTRU, mantém uns níveis de desempenho razoáveis que permite que o LWE seja usado na prática.

Relativamente às implementações em Sage, as que se tornaram mais complicadas foram o GGH e o NTRU. Uma das dificuldades no GGH foi como determinar a chave pública e a chave privada, uma vez que a chave pública dependia da chave privada. Como havia várias maneiras de determinar as chaves, optou-se pelo método que pareceu mais intuitivo e simples de perceber e implementar, que consistia para a chave privada a soma de matrizes com determinadas propriedades e para a chave pública consistia no produto de uma matriz por outra. Havia ainda no processo de cifração também a existência de várias formas de construir o criptograma, neste caso optou-se então pelo cálculo do vetor de erro que dependia de um σ, que por sua vez dependia da inversa da chave privada e ainda de uma probabilidade de a decifração falhar. O processo de decifração foi construído seguindo as instruções dadas.

No NTRU, havia inicialmente duas formas de construir a cifra, na forma matricial ou na forma polinomial, optando-se neste caso pela segunda forma. Na criação das chaves para parâmetros de entrada elevados houve o problema de a execução ser lenta, no caso em que o parâmetro N era elevado e em que o parâmetro q era uma potência única de dois. Outra dificuldade que surgiu foi no processo de decifração, como colocar os polinómios com coeficientes num determinado intervalo.

Como trabalho futuro seria interessante fazer investigação nesta área, ou projetos, pelo facto de os retículos serem resistentes à computação quântica, terem caraterísticas bastante importantes, além de permitirem construir funções de *hash* resistentes a colisões e de sentido único e esquemas criptográficos de chave pública.

Bibliografia

[1] Seong-Hung Paeng e Bae Eun Jung e Kil-Chan Ha. A lattice based public key cryptosystem using polynomial representations. In *Public Key Cryptography - PKC 2003*, 2003.

[2] Jesko Hüttenhain e Lars Wallenborn. Topics in post-quantum cryptography, 2011.

[3] Daniele Micciancio e Oded Regev. *Post-Quantum Cryptography*, chapter Lattice-based cryptography. Springer, 2009.

[4] Tore Frederiksen. A practical implementation of regev's lwe-based cryptosystem, 2011.

[5] David Joyner. *Course Notes*, chapter Python and Coding Theory. United States Naval Academy, 2009.

[6] Michael Rose. Lattice-based cryptography: a practical implementation. Master's thesis, University of Wollongong, 2011.

[7] Joseph H. Silverman. An introduction to the theory of lattices and applications to cryptography, 2006.

Anexos

Definições e notações matemáticas

Notação

- $\mathrm{Gl}_n(\mathbb{R})$ - grupo linear formado por matrizes reais invertíveis pertencentes a $\mathbb{R}^{n\times n}$, com a operação de multiplicação de matrizes.
- $\mathrm{Gl}_n(\mathbb{Z})$ - grupo linear formado por matrizes inteiras invertíveis pertencentes a $\mathbb{Z}^{n\times n}$, com a operação de multiplicação de matrizes.
- $L = L(B)$ - representa o retículo L gerado pela matriz B.

Fundamentos de Álgebra Linear

Produto Interno de vetores

Seja $u, v \in \mathbb{R}^n$. Então define-se $\langle .,. \rangle : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ como:

$$\langle u, v \rangle = u^T.v = \begin{bmatrix} u_1 & \dots & u_2 \end{bmatrix} . \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} = \sum_{i=1}^{n} u_i v_i.$$

O produto interno verifica as seguintes propriedades:

- Bilinearidade: Para $u, v, w \in \mathbb{R}^n$, $\langle u + v, w \rangle = \langle u, w \rangle + \langle v, w \rangle$ e $\langle w, u + v \rangle = \langle w, u \rangle + \langle w, v \rangle$.
- Homogeneidade: Para $u, v \in \mathbb{R}^n$ e $\lambda \in \mathbb{R}$, $\langle \lambda u, v \rangle = \lambda \langle u, v \rangle$.
- Positividade: Para $v \in \mathbb{R}^n$, $\langle v, v \rangle \geq 0$.

Vetores linearmente dependentes e independentes

Seja $a_1, \ldots, a_p \in \mathbb{R}^n$. Estes vetores dizem-se linearmente dependentes se existirem escalares $\alpha_1, \ldots, \alpha_p \in \mathbb{R}$ não todos zero, tal que:

$$\sum_{i=1}^{p} \alpha_i a_i = 0_{n\times 1}.$$

Caso contrário, são designados de vetores linearmente independentes.

Norma euclidiana de um vetor

Seja $v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix} \in \mathbb{R}^n$. Então define-se $\|.\|$: $\mathbb{R}^n \to \mathbb{R}$ como :

$$\|v\| = \sqrt{\sum_{i=1}^{n} |v_i|^2}.$$

Vetores ortogonais

Seja $u, v \in \mathbb{R}^n$. Os vetores u e v dizem-se ortogonais se $\langle u, v\rangle = 0$.

Espaço Vetorial

Seja K um corpo, onde os elementos se designam de escalares. Seja V um conjunto não vazio de vetores e onde é válido a soma de vetores e a multiplicação de vetores por um escalar. V diz-se um espaço vetorial se:

- Para $u, v \in V$, $u + v = v + u$.
- Para $u, v, w \in V$, $(u + v) + w = u + (v + w)$.
- Existe $idsoma \in V$ tal que para $v \in V$, $idsoma + v = v + idsoma = v$.
- Existe $b \in V$ tal que para $u \in V$, $u + b = b + u = idsoma$.
- Para α,$\beta \in K$, $u, v \in$ V, $\alpha(u + v) = \alpha u + \alpha v$ e $(\alpha + \beta)v = \alpha v + \beta v$.
- Para α,$\beta \in K$, $u \in V$, $(\alpha\ \beta)u = \alpha\ (\beta u)$.
- Existe $idmult \in K$ tal que para $v \in V$, $idmult * v = v * idmult = v$.

Subespaço Vetorial

Seja S um subconjunto não vazio de $\mathbb{R}^n$. S é um subespaço vetorial de $\mathbb{R}^n$ se dados $c, d \in S$ e α,$\beta \in \mathbb{R}$, $\alpha c + \beta d \in S$. Para todo o subespaço vetorial S existe um $r \in \mathbb{N}$, tal que todo o vetor de S se pode escrever como combinação linear de r vetores de S. A dimensão de S representa-se por $dim(S) = r$.

Base de um Espaço Vetorial

Uma base B de um espaço vetorial V é um conjunto de vetores linearmente independentes para o qual todo o vetor de V se pode escrever como combinação linear dos elementos de B.

Imagem de uma matriz

Seja $A \in \mathbb{R}^{n \times m}$. A imagem de uma matriz, designado de $Im(A)$, é um espaço vetorial definido da seguinte forma:

$$Im(A) = \{y \in \mathbb{R}^n \ : \ y = Ax \text{ para algum } x \in \mathbb{R}^m\},$$

ou seja, $Im(A)$ é constituída por todas as combinações lineares das colunas da matriz A.

Núcleo de uma matriz

Seja $A \in \mathbb{R}^{n \times m}$. O núcleo da matriz, designado por $N(A)$, é um espaço vetorial definido da seguinte forma:

$$N(A) = \{x \in \mathbb{R}^m \ : \ Ax = 0_{n \times 1}\}.$$

Caraterística de uma matriz

A caraterística de uma matriz $A \in \mathbb{R}^{n\times m}$, designada por $car(A)$, corresponde à dimensão da imagem de A, isto é,

$$car(A) = dim(Im(A)).$$

As seguintes propriedades são satisfeitas pela caraterística da matriz:

- $car(A) = car(A^T)$.
- $car(A) + dim(N(A)) = n$.
- $car(A) + dim(N(A^T)) = m$.

Determinante de uma matriz

Pelo teorema de Laplace, o determinante de uma matriz $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, designado por $\det(A)$, é calculado da seguinte forma:

1. Caso $n = 1$, $\det(A) = a_{11}$.
2. Caso $n > 1$, $\det(A) = \sum_{j=1}^{n}(-1)^{i+j}a_{ij}\det(M_{ij})$, com $1 \leq i \leq n$, onde M_{ij} denota a matriz de ordem $n-1$ obtida de A retirando a linha i e a coluna j.

O determinante da matriz tem as seguintes propriedades:

- $\det(A) = \det(A^T)$.
- $\det(\alpha A) = \alpha^n \det(A)$.
- $\det(AB) = \det(A)\det(B)$.
- Se B resulta de A por troca de duas colunas ou linhas, então $\det(B) = -\det(A)$.
- Se A possui duas linhas ou colunas iguais, então $\det(A) = 0$.
- O determinante de uma matriz triangular é o produto dos elementos da diagonal.

Inversa de uma matriz

Seja $A \in \mathbb{R}^{n\times n}$. A diz-se invertível se existir $X \in \mathbb{R}^{n\times n}$ tal que:

$$AX = I_n,$$

em que I_n representa a matriz identidade de ordem n. A inversa da matriz A é denotada por A^{-1}. Da definição de inversa tem-se que dadas duas matrizes A e B, é válida a igualdade $(AB)^{-1} = B^{-1}A^{-1}$. Além disso uma matriz $A \in \mathbb{R}^{n\times n}$ é invertível se e só se:

- $car(A) = car(A^T) = n$.
- $\det(A) \neq 0$.
- $N(A) = \{0_{n\times 1}\}$.
- $Im(A) = \mathbb{R}^n$.

Fundamentos de Estruturas Algébricas

Grupo

Seja G um conjunto não vazio fechado para a operação op, ou seja, $\forall a, b \in G$ a op $b \in G$, sendo op uma operação binária. Diz-se que (G, op) é um grupo se satisfizer:

1. Associatividade $\rightarrow \forall\ a, b, c \in G$, $(a\ op\ b)\ op\ c = a\ op\ (b\ op\ c)$.
2. Identidade $\rightarrow \forall\ a \in G$, $\exists\ id \in G$ tal que $id\ op\ a = a\ op\ id = a$.
3. Elemento inverso $\rightarrow \forall\ a \in G$, $\exists\ b \in G$ tal que $a\ op\ b = id = b\ op\ a$.

Se (G, op) for grupo possui ainda as seguintes propriedades:

1. Leis do cancelamento:
   - $\forall\ a, b, c \in G$, $a\ op\ b = a\ op\ c \Rightarrow b = c$.
   - $\forall\ a, b, c \in G$, $b\ op\ a = c\ op\ a \Rightarrow b = c$.
2. Ordem de um Grupo:
   - Diz-se que a ordem de G é n e denota-se por $|G| = n$, se G é um conjunto finito de n elementos.

3. Grupo Abeliano:

   - Se (G, op) é um grupo e op é uma operação comutativa.

4. Semi-Grupo e Monóide:

   - Semi-Grupo se $G \neq \emptyset$ e op é uma operação associativa. Monóide se op é associativo e G tem um elemento neutro.

Subgrupo

Seja (G, op) um grupo e H um subconjunto não vazio de G. Diz-se que H é um subgrupo de G se satisfizer:

1. $\forall\ a, b \in$ G, $a\ op\ b \in H$.
2. (H, op) é um grupo.

Anel

Diz-se que $(A,+,*)$ é um anel se satisfizer:

1. A estrutura $(A,+)$ é um grupo abeliano.
2. A operação * é associativa, ou seja, $\forall\ a, b, c \in A$, $(a * b) * c = a * (b * c)$.
3. A operação * é distributiva em relação à operação $+$, isto é,
   $\forall\ a, b, c \in A$, $a * (b + c) = a * b + a * c$ e $(b + c) * a = b * a + c * a$.

Corpo

Seja F um conjunto com pelo menos dois elementos com duas operações $+$ e *. Diz-se que F é um corpo se satisfizer:

1. $(F,+,*)$ forma um anel comutativo com identidade (multiplicação).
2. $\forall\ a \neq 0$, $\exists\ a^{-1}$ tal que $a * a^{-1} = 1_F$.