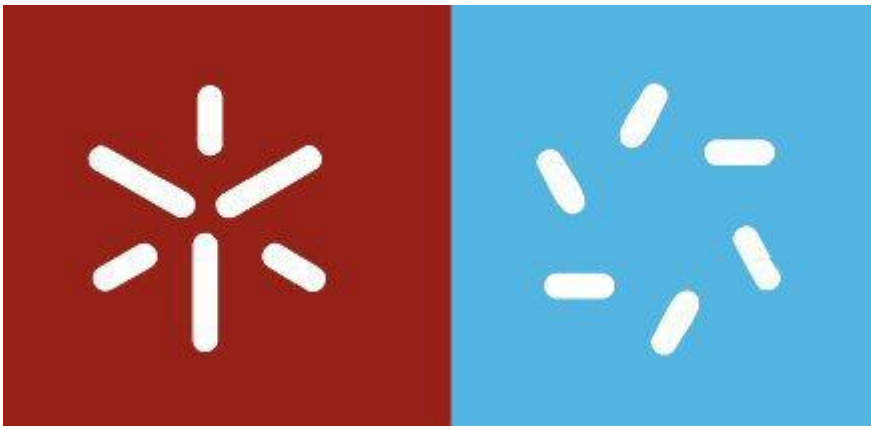
Universidade do Minho
Teoria de Códigos

Mestrado em Matemática e Computação



# Criptossistema de McEliece

Diogo Rafael Rodrigues Aires
Janeiro de 2026

# Índice

# Introdução

Com o aumento da transmissão de dados em redes públicas, foi aumentando também o risco de um terceiro conseguir interceptar estes dados.Desta forma, a segurança destas transmissões fica dependente da criptografia, a ciência de proteger mensagens contra acessos não autorizados. O processo envolve converter a mensagem num criptograma (cifração) e revertê-la ao estado original (decifração) através de uma chave privada, garantindo que apenas o destinatário correto consiga ler os dados.

Atualmente, a confiança e a privacidade nas comunicações digitais dependem de algoritmos de chave pública, nomeadamente o RSA e o ECC (Criptografia de Curva Elíptica). A segurança destes sistemas baseia-se na complexidade de problemas matemáticos, como a fatorização de números inteiros. Contudo, a computação quântica de grande escala ameaça este paradigma: o Algoritmo de Shor demonstrou a capacidade teórica de resolver estes problemas de forma eficiente, tornando os métodos tradicionais obsoletos.

Face à iminente quebra da segurança clássica, torna-se imperativo adotar a Criptografia Pós-Quântica — uma nova geração de algoritmos concebidos para resistir às capacidades de processamento tanto de computadores convencionais como quânticos.

# Criptografia

A criptografia (do grego *kryptós*, "escondido", e *gráphein*, "escrita") evoluiu de uma arte de substituição de letras para uma disciplina científica rigorosa baseada em fundamentos matemáticos.

No contexto da era digital, a criptografia deixou de ser uma ferramenta exclusivamente militar para se tornar essencial na confiança na Internet. Sem ela, atividades quotidianas como transações bancárias, compras online ou comunicações privadas seriam impossíveis de realizar com segurança através de redes públicas.

O objetivo primário da criptografia é garantir a confidencialidade: assegurar que, mesmo que uma mensagem seja interceptada por um terceiro, o seu conteúdo permaneça ilegível. Para tal, utiliza-se um processo de encriptação, onde a mensagem original é transformada num criptograma através de um algoritmo e de uma chave.

## Criptografia - Chave Privada

A criptografia de chave privada, também designada por criptografia simétrica, é um dos modelos mais antigos de proteção da informação. Neste tipo de criptografia, o emissor e o receptor partilham uma chave secreta comum, que é utilizada tanto no processo de encriptação como no de desencriptação da mensagem. A segurança do sistema depende inteiramente do facto de esta chave permanecer desconhecida para terceiros.

Num cenário de criptografia de chave privada, um atacante que intercepta a comunicação tem acesso apenas à informação cifrada, não possuindo conhecimento sobre a chave utilizada nem sobre o conteúdo da mensagem original. Desde que a chave seja suficientemente aleatória (e mantida em segredo), torna-se computacionalmente difícil recuperar a informação transmitida.

### Exemplo de Criptografia de Chave Privada - Cifra de César

A Cifra de César é um dos exemplos mais simples e históricos de criptografia de chave privada. Neste método, a chave consiste num número inteiro que representa um deslocamento fixo no alfabeto. Para cifrar uma mensagem, cada letra do texto original é substituída pela letra que se encontra um determinado número de posições à frente no alfabeto. O processo de desencriptação realiza o deslocamento inverso, permitindo recuperar a mensagem original.

### Cifra de César - Algoritmo de Encriptação e Desencriptação

Suponhamos uma mensagem em texto claro $m$ e uma chave $k$, onde $k$ representa o número de posições de deslocamento do alfabeto.

```
def cifra_cesar(mensagem, k):
    resultado = ""

    for c in mensagem:
        if c.isalpha():
            base = ord('A') if c.isupper() else ord('a')
            resultado += chr((ord(c) - base + k) % 26 + base)
        else:
            resultado += c

    return resultado


def decifra_cesar(texto_cifrado, k):
    return cifra_cesar(texto_cifrado, -k)
```

A utilização da operação módulo 26 garante que o resultado permanece dentro do alfabeto, permitindo a correta encriptação e desencriptação da mensagem.

Apesar da sua simplicidade e valor pedagógico, a Cifra de César é considerada insegura, uma vez que possui um espaço de chaves muito reduzido, podendo ser facilmente quebrada por força bruta. Ainda assim, é útil como exemplo introdutório para ilustrar os princípios básicos da criptografia simétrica.

## Exemplo de Criptografia de Chave Privada - AES

O Advanced Encryption Standard (AES) é um dos algoritmos de criptografia de chave privada mais utilizados na atualidade. Trata-se de um criptossistema simétrico que utiliza chaves secretas de 128, 192 ou 256 bits para cifrar e decifrar dados. O AES é amplamente adotado em diversas aplicações práticas, incluindo comunicações seguras, redes sem fios e proteção de dados armazenados.

O processo de encriptação no AES baseia-se numa sequência de transformações matemáticas, como substituições, permutações e operações de mistura, que são repetidas ao longo de várias rondas. Estas operações tornam o texto cifrado altamente resistente a ataques criptográficos conhecidos. A desencriptação aplica-se às operações inversas utilizando a mesma chave secreta.

AES - Algoritmo de Encriptação e Desencriptação

Suponhamos uma mensagem de texto $m$ e uma chave $key$ de 128 bits

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

# Chave de 16 bytes (128 bits)
key = b'ChaveSecretaAES!'
cipher = AES.new(key, AES.MODE_ECB)
mensagem = m

# Encriptação
mensagem_cifrada = cipher.encrypt(pad(mensagem, 16))

# Desencriptação
mensagem_decifrada = unpad(cipher.decrypt(mensagem_cifrada), 16)
```

A segurança do AES depende da confidencialidade da chave partilhada e da robustez do algoritmo, sendo atualmente considerado seguro e eficiente. No entanto, tal como acontece em qualquer sistema de criptografia simétrica, subsiste o desafio da distribuição segura das chaves entre as entidades comunicantes

## Criptografia de Chave Pública

A criptografia de chave pública, ou criptografia assimétrica, introduz uma abordagem inovadora ao utilizar duas chaves distintas para processar a informação, ao contrário dos métodos tradicionais que dependem de uma única chave partilhada. Neste sistema, cada utilizador gera um par de chaves matematicamente ligadas: uma chave pública, que pode ser distribuída livremente, e uma chave privada, que deve permanecer sob controlo exclusivo do seu proprietário.

O funcionamento baseia-se num princípio de exclusividade: a chave pública é utilizada para cifrar o conteúdo, mas, uma vez transformado, esse conteúdo só pode ser recuperado através da chave privada correspondente. Esta dinâmica elimina o problema logístico da distribuição segura de chaves, permitindo que qualquer emissor envie dados protegidos sem a necessidade de um canal prévio para troca de segredos.

A segurança destes algoritmos não reside no segredo do método de cifração, mas sim na complexidade matemática da relação entre as chaves. O sistema é desenhado para que a derivação da chave privada a partir da pública seja um problema computacionalmente "difícil" — ou seja, exigiria recursos e tempo impraticáveis para os padrões atuais.

# Exemplo de Criptografia de Chave Pública - RSA

O RSA é um dos sistemas de criptografia de chave pública mais conhecidos e utilizados no mundo. Ele permite que duas partes se comuniquem de forma segura sem precisar partilhar uma chave secreta previamente. A segurança do RSA baseia-se na dificuldade de fatorizar números grandes em primos, o que torna praticamente impossível descobrir a chave privada a partir da chave pública.

No RSA, cada utilizador possui uma chave pública, que pode ser partilhada com qualquer pessoa, e uma chave privada, que é mantida em segredo. Mensagens cifradas com a chave pública só podem ser decifradas com a chave privada correspondente, garantindo a confidencialidade da comunicação.

## RSA - Raciocínio e Algoritmo

Geração das Chaves:

1. Escolher dois números primos grandes $p$ e $q$
2. Calcular $n$ (parte da chave pública):

$$n = p \times q$$

3. Calcular $\phi(n)$:

$$\phi(n) = (p - 1)(q - 1)$$

4. Escolher $e$ tal que $1 < e < \phi(n)$ e coprimo de $\phi(n)$:

$$gcd(e, \phi(n)) = 1$$

5. Calcular $d$ (chave privada) tal que:

$$d \cdot e \equiv 1 \pmod{\phi(n)}$$

Para cifrar então uma mensagem $M$

$$C = M^e \bmod n$$

onde $C$ é a mensagem cifrada.

Para decifrar a mensagem $C$

$$M = C^d \bmod n$$

onde $M$ é então a mensagem decifrada.

import random

```
# 1. Função para encontrar o Máximo Divisor Comum (MDC)
def mdc(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# 2. Função para encontrar o inverso modular (Algoritmo de Euclides)
def inverso_modular(e, phi):
    d = 0
    x1, x2 = 0, 1
    y1, y2 = 1, 0
    temp_phi = phi
    while e > 0:
        temp1 = temp_phi // e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2

        x = x2 - temp1 * x1
        y = y2 - temp1 * y1

        x2, x1 = x1, x
        y2, y1 = y1, y

    if temp_phi == 1:
        return y2 % phi

# 3. Geração de Chaves
def gerar_chaves(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)

    # Escolher e tal que 1 < e < phi e mdc(e, phi) == 1
    e = random.randrange(1, phi)
    while mdc(e, phi) != 1:
        e = random.randrange(1, phi)

    # Calcular a chave privada d
    d = inverso_modular(e, phi)

    # Retorna (Chave Pública, Chave Privada)
    return ((e, n), (d, n))
```

```
# 4. Funções de Encriptação e Desencriptação
def cifrar(pk, texto_limpo):
    e, n = pk
    cifrado = [pow(ord(char), e, n) for char in texto_limpo]
    return cifrado

def decifrar(sk, texto cifrado):
    d, n = sk
    decifrado = [chr(pow(char, d, n)) for char in texto_cifrado]
    return "".join(decifrado)
```

# O Criptossistema de McEliece

Embora sistemas como o RSA tenham dominado a segurança digital nas últimas décadas, o advento da computação quântica forçou a comunidade científica a procurar alternativas. O problema reside no facto de a segurança do RSA ou do ECC (Curvas Elípticas) assentar em problemas matemáticos que, embora difíceis para computadores clássicos, são vulneráveis a algoritmos quânticos como o de Shor.

É neste contexto de vulnerabilidade que o Criptossistema de McEliece se destaca. Proposto em 1978 por Robert McEliece, este sistema não utiliza a fatoração de números primos como barreira de segurança. Em vez disso, introduz um conceito radicalmente diferente: a utilização de Códigos Corretores de Erros.

## Base de Segurança: Problema NP-Difícil

A segurança do criptossistema de McEliece não se baseia em fatoração de números grandes, como acontece no RSA, mas sim na complexidade da Teoria de Códigos. Em termos práticos, decifrar uma mensagem sem a chave privada equivale a resolver problemas considerados NP-difíceis, ou seja, problemas que não se resolvem eficientemente com algoritmos conhecidos.

O ponto forte do McEliece foi provado por Berlekamp, McEliece e van Tilborg: decodificar um código linear genérico é praticamente impossível com os algoritmos atuais.

### Problema da Decodificação Geral (Nearest Codeword Problem)

Suponhamos que temos um código linear definido por uma matriz geradora $\hat{G}$ um vetor recebido $y$ que contém alguns erros. O objetivo é descobrir a palavra de código original $x \in C$ que está mais próxima de $y$, usando a distância de Hamming como referência.

A dificuldade surge porque, para um código aleatório, a única forma de encontrar xxx é testar todas as possibilidades — o que cresce exponencialmente à medida que a matriz aumenta.

### Problema do Peso das Palavras de Código (Small Weight Codeword Problem)

Outro problema importante é encontrar vetores dentro de um código que tenham poucos elementos não nulos, ou seja, peso de Hamming baixo.

Se fosse possível identificar palavras de código de peso reduzido na chave pública, seria possível reconstruir a estrutura do código de Goppa ou até descobrir o vetor de erro $e$ usado na encriptação.

## Geração de Chaves - Matriz

O ponto forte do criptossistema de McEliece está na capacidade de transformar a estrutura altamente organizada de um Código de Goppa numa matriz que parece aleatória.

### Chave Privada

#### Matriz Geradora ($G$)

- É uma matriz $k \times n$ que define um código de Goppa binário, capaz de corrigir até $t$ erros.
- Permite a decodificação rápida usando um algoritmo específico $D$.

#### Matriz Scrambler($S$)

- Uma matriz invertível $k \times k$ escolhida aleatoriamente.
- Serve para combinar as linhas de $G$ de forma linear, escondendo a forma original das linhas sem alterar o código.

#### Matriz de Permutação ($P$)

- Uma matriz $n \times n$ que possui exatamente um “1” por linha e coluna.
- A sua função é “emaranhar” as colunas da matriz resultante, tornando impossível identificar as propriedades originais do código de Goppa.

### Chave Pública

A chave pública é a matriz que o recetor disponibiliza e é obtida pela operação:

$$\hat{G} = S \times G \times P$$

Embora $\hat{G}$ gere essencialmente o mesmo código que $G$, a sua estrutura interna está completamente oculta:

- $S$ esconde quais eram as linhas originais.
- $P$ esconde a ordem original das colunas.

Para qualquer observador que não conheça $S$ e $P$, $\hat{G}$ parece simplesmente uma matriz aleatória.

### Viabilidade do Sistema

Segundo McEliece, o número de combinações possíveis para $S$ e $P$ é astronómico. Para matrizes de tamanho típico, há tantas matrizes invertíveis e de permutação que qualquer tentativa de "desmontar" $S \times G \times P$ por força bruta é computacionalmente inviável.

Por outras palavras, o segredo permanece seguro graças à complexidade combinatória — o disfarce protege o código original, tornando praticamente impossível para um atacante recuperar $G$ sem a chave privada.

## Distinção entre Codificação e Encriptação

A codificação e a encriptação têm objetivos distintos. A codificação procura garantir a fiabilidade da transmissão, introduzindo redundância para permitir a deteção e correção de erros causados pelo ruído do canal. Já a encriptação tem como finalidade a confidencialidade, tornando a informação incompreensível para quem não possui a chave adequada.

O criptossistema de McEliece cruza estas duas áreas ao recorrer a códigos corretores de erros com um propósito criptográfico. Neste esquema, a encriptação passa por introduzir deliberadamente erros na mensagem, erros esses que apenas o destinatário legítimo consegue remover, graças ao conhecimento da estrutura específica do código utilizado.

A segurança do sistema assenta na grande dificuldade de decodificar um código aparentemente genérico quando a sua estrutura interna é desconhecida. Um problema clássico da teoria da codificação transforma-se, assim, num elemento central da criptografia pós-quântica. Deste modo, o McEliece subverte a lógica habitual: o erro deixa de ser um fenómeno indesejado a combater e passa a desempenhar o papel de mecanismo de proteção contra acessos não autorizados.

.

# O Processo de Encriptação: Codificação e Ruído

No sistema de McEliece, encriptar uma mensagem consiste em transformar a mensagem original $m$ num criptograma $y$ usando a combinação da teoria de códigos com técnicas de segurança criptográfica.

## Codificação com a chave pública

A mensagem $m$, representada como um vetor binário de comprimento $\mathrm{k}$, é multiplicada pela matriz geradora pública $\hat{G}$ ( $k \times n$ ). O resultado $m\hat{G}$ é uma palavra de código válida dentro do código "disfarçado".

## Introdução intencional de erros

Para impedir que alguém recupere $m$ apenas resolvendo sistemas lineares simples, o emissor adiciona um vetor de erro aleatório $e$. Este vetor tem comprimento $n$ e um peso de Hamming exatamente igual a $t$, que é o número máximo de erros que o código de Goppa consegue corrigir.

O criptograma final enviado pelo canal público é:

$$y = m\hat{G} + e$$

Assim, $y$ parece ser uma palavra de código normal, mas contém exatamente $\mathrm{t}$ erros, tornando impossível para quem não conhece a estrutura de $\hat{G}$ decifrar a mensagem.

# Processo de Desencriptação: Reversão e Correção

Decifrar uma mensagem no McEliece é basicamente remover as camadas de proteção usando a chave privada ($S,G,P,D$).

## Remoção da Permutação e Ajuste do Ruído

Quando o receptor recebe $y$, o primeiro passo é multiplicar o criptograma pela inversa da matriz de permutação $P^{-1}$:

$$y' = yP^{-1} = (mSGP + e)P^{-1} = mSG + eP^{-1}$$

Como $P$ é apenas uma permutação das colunas, o vetor de erros é $e'=e\,P^{-1}$ mantém o mesmo peso $t$, mas agora os erros estão nas posições corretas para o código de Goppa. Ou seja, o vetor $y'$ agora "encaixa" na estrutura do código privado G.

### Aplicação do Algoritmo de Decodificação D

Nesta fase, o receptor utiliza o algoritmo de decodificação eficiente $D$ (normalmente o Algoritmo de Patterson para códigos de Goppa). Este algoritmo consegue identificar e remover o vetor de erro $e'$, recuperando a palavra de código limpa:

$$D(y') = D(mSG + e') = mSG$$

### Recuperação da Mensagem Original

Depois de eliminar o ruído, o receptor obtém o vetor $m'=mS$. Como $S$ é invertível, o passo final é multiplicar pelo inverso $S^{-1}$ para recuperar a mensagem original:

$$m = m'S^{-1} = (mS)S^{-1}$$

## Resumo do Criptossistema de McEliece

O Criptossistema de McEliece é um algoritmo de chave pública baseado na Teoria de Códigos Corretores de Erros. Ao contrário de sistemas como o RSA, que dependem da dificuldade de fatorar números primos, o McEliece baseia a sua segurança na dificuldade de decodificar um código linear genérico ao qual foi adicionado ruído intencional — um problema classificado como NP-difícil.

O funcionamento do sistema pode ser resumido em três pilares:

1. A Chave : O utilizador escolhe um código estruturado que sabe decifrar eficientemente (geralmente um Código de Goppa). Esta estrutura é "escondida" através da multiplicação por duas matrizes secretas (uma de baralhamento e outra de permutação), gerando uma chave pública que parece um código aleatório e sem estrutura.
2. Encriptação: Para enviar uma mensagem, o emissor multiplica o texto pela chave pública e adiciona deliberadamente um vetor de erro (ruído). Este erro "suja" a mensagem de tal forma que apenas quem possui a "receita" original do código consegue limpá-la.
3. Desencriptação: O receptor legítimo usa a sua chave privada para desfazer o disfarce e aplicar o seu algoritmo de correção de erros. Ele remove o ruído intencional e recupera a mensagem original de forma rápida e eficiente.

# Vantagens e Desvantagens do Criptossistema de McEliece

Apesar de possuir propriedades de segurança excepcionais, as suas exigências técnicas impuseram barreiras à sua adoção em larga escala.

A maior vantagem atual é a sua imunidade aos algoritmos quânticos conhecidos. Enquanto o Algoritmo de Shor torna o RSA e o ECC obsoletos, o McEliece permanece seguro, pois o problema de decodificação de códigos lineares não sofre uma aceleração exponencial em computadores quânticos.

Ao contrário de muitos novos algoritmos que surgem para a era pós-quântica, o McEliece tem quase 50 anos de criptoanálise. Durante décadas, foi alvo de inúmeras tentativas de ataque, e a sua base matemática (Códigos de Goppa) permanece íntegra, o que lhe confere um nível de confiança inigualável.

Os processos de encriptar (multiplicação de matrizes) e desencriptar (algoritmos de correção de erros como o de Patterson) são extremamente rápidos e exigem pouco esforço computacional, superando muitas vezes o RSA em termos de velocidade de processamento.

Enquanto uma chave RSA segura pode ter cerca de 2048 a 4096 bits, uma chave pública de McEliece pode atingir centenas de kilobytes ou até megabytes. Isto torna o sistema difícil de implementar em protocolos que exigem trocas rápidas de chaves, como o aperto de mão (*handshake*) do TLS/HTTPS.

O texto cifrado é consideravelmente maior do que a mensagem original, devido à estrutura de redundância e ruído do código, o que aumenta o consumo de largura de banda na transmissão de dados.

Devido ao tamanho das chaves, o McEliece exige uma quantidade significativa de memória RAM para processar as matrizes, o que pode ser um impedimento para dispositivos IoT ou *smartcards*.

# Bibliografia

Christopher Roering,
Coding Theory-Based Cryptography: McEliece Cryptosystems in Sage, College of Saint Benedict/Saint John's University, Honors Theses, 1963-2015, 17, 2013,
http://digitalcommons.csbsju.edu/honors_theses/17

A new cryptosystem based on the McEliece Pedro de Melo Branco Instituto Superior T´ecnico, Lisboa, Portugal October 2017
https://fenix.tecnico.ulisboa.pt/downloadFile/1970719973967112/ExtendedAbstract.pdf

# Implementação do Criptossistema de McEliece

(Implementação retirada da Tese de Christopher Roering)

```
class GoppaCode:
    def __init__(self, n, m, g):
        t = g.degree()
        F2 = GF(2)
        F_2m = g.base_ring()
        Z = F_2m.gen()
        PR_F_2m = g.parent()
        X = PR_F_2m.gen()

        codelocators = []
        for i in range(2^m-1):
            codelocators.append(Z^(i+1))
        codelocators.append(F_2m(0))

        h = PR_F_2m(1)
        for a_i in codelocators:
            h = h*(X-a_i)

        gamma = []
        for a_i in codelocators:
            gamma.append((h*((X-a_i).inverse_mod(g))).mod(g))

        H_check_poly = matrix(F_2m, t, n)
        for i in range(n):
            coeffs = list(gamma[i])
            for j in range(t):
                if j < len(coeffs):
                    H_check_poly[j,i] = coeffs[j]
                else:
                    H_check_poly[j,i] = F_2m(0)

        H_Goppa = matrix(F2, m*H_check_poly.nrows(),
H_check_poly.ncols())
        for i in range(H_check_poly.nrows()):
            for j in range(H_check_poly.ncols()):
                be = bin(eval(H_check_poly[i,j].int_repr()))[2:]
                be = '0'*(m-len(be)) + be
                be = list(be)
                H_Goppa[m*i:m*(i+1), j] = vector(map(int, be))
```

```
        G_Goppa = H_Goppa.right_kernel().basis_matrix()

        SyndromeCalculator = matrix(PR_F_2m, 1, len(codelocators))
        for i in range(len(codelocators)):
            SyndromeCalculator[0,i] = (X -
codelocators[i]).inverse_mod(g)

        self._n = n
        self._m = m
        self._g = g
        self._t = t
        self._codelocators = codelocators
        self._SyndromeCalculator = SyndromeCalculator
        self._H_Goppa = H_Goppa
        self._G_Goppa = G_Goppa

    def Encode(self, message):
        return (message*self._G_Goppa)

    def _split(self, p):
        Phi = p.parent()
        p0 = Phi([sqrt(c) for c in p.list()[0::2]])
        p1 = Phi([sqrt(c) for c in p.list()[1::2]])
        return (p0, p1)

    def _g_inverse(self, p):
        (d,u,v) = xgcd(p, self.goppa_polynomial())
        return u.mod(self.goppa_polynomial())

    def _norm(self, a, b):
        X = self.goppa_polynomial().parent().gen()
        return 2^((a^2 + X*b^2).degree())

def _lattice_basis_reduce(self, s):
        g = self.goppa_polynomial()
        t = g.degree()
        a = []; b = []
        a.append(0); b.append(0)
        (q,r) = g.quo_rem(s)
        (a[0], b[0]) = simplify((g - q*s, 0 - q))

        if self._norm(a[0], b[0]) > 2^t:
            a.append(0); b.append(0)
            (q,r) = s.quo_rem(a[0])
            (a[1], b[1]) = (r, 1 - q*b[0])
        else:
```

```
            return (a[0], b[0])

        i = 1
        while self._norm(a[i], b[i]) > 2^t:
            a.append(0); b.append(0)
            (q,r) = a[i-1].quo_rem(a[i])
            (a[i+1], b[i+1]) = (r, b[i-1] - q*b[i])
            i += 1

        return (a[i], b[i])

    def Decode(self, word_):
        g = self._g
        word = copy(word_)
        X = g.parent().gen()

        synd = self._SyndromeCalculator * word.transpose()
        syndrome_poly = 0
        for i in range(synd.nrows()):
            syndrome_poly += synd[i,0]*X^i

        (g0,g1) = self._split(g)
        sqrt_X = g0 * self._g_inverse(g1)
        (T0,T1) = self._split(self._g_inverse(syndrome_poly) - X)
        R = (T0 + sqrt_X*T1).mod(g)

        (alpha, beta) = self._lattice_basis_reduce(R)
        sigma = (alpha*alpha) + (beta*beta)*X

        for i in range(len(self._codelocators)):
            if sigma(self._codelocators[i]) == 0:
                word[0,i] += 1

        return word


def generator_matrix(self):
        return self._G_Goppa

    def goppa_polynomial(self):
        return self._g

    def parity_check_matrix(self):
        return self._H_Goppa

class McElieceCryptosystem:
```

```
    def __init__(self, n, m, g):
        goppa_code = GoppaCode(n, m, g)
        assert goppa_code.generator_matrix().nrows() <> 0
        k = goppa_code.generator_matrix().nrows()

        S = matrix(GF(2), k, [random() < 0.5 for _ in range(k^2)])
        while rank(S) < k:
            S[floor(k*random()), floor(k*random())] += 1

        rng = range(n)
        P = matrix(GF(2), n)
        for i in range(n):
            p = floor(len(rng)*random())
            P[i, rng[p]] = 1
            rng = rng[:p] + rng[p+1:]

        self._m_GoppaCode = goppa_code
        self._g = g
        self._t = g.degree()
        self._S = S
        self._P = P
        self._PublicKey = S * goppa_code.generator_matrix() * P

    def _GetRowVectorWeight(self, n):
        weight = 0
        for i in range(n.ncols()):
            if n[0,i] == 1:
                weight += 1
        return weight

    def Encrypt(self, message):
        assert message.ncols() == self.public_key().nrows()
        err_vec = matrix(1,
self.goppa_code().generator_matrix().ncols())
        while self._GetRowVectorWeight(err_vec) < self.max_num_errors():
            err_vec[0, randint(1,
self.goppa_code().generator_matrix().ncols()-1)] = 1
        code_word = message * self.public_key()
        return copy(code_word + err_vec)

    def Decrypt(self, received_word):
        assert received_word.ncols() == self.public_key().ncols()
        message = received_word * ~(self._P)
        message = self.goppa_code().Decode(message)
        message = (self._S *
self.goppa_code().generator_matrix()).solve_left(message)
```

```
        return copy(message)

    def public_key(self):
        return copy(self._PublicKey)

    def goppa_code(self):
        return copy(self._m_GoppaCode)

    def max_num_errors(self):
        return copy(self._t)

def _GetRandomPermutationMatrix(n):
    rng = range(n)
    P = matrix(GF(2), n)
    for i in range(n):
        p = floor(len(rng)*random())
        P[i, rng[p]] = 1
        rng = rng[:p] + rng[p+1:]
    return copy(P)

def _GetColumnVectorWeight(n):
    weight = 0
    for i in range(n.nrows()):
        if n[i,0] == 1:
            weight += 1
    return weight

def SternsAlgorithm(H, w, p, l):
    H_Stern = copy(H)
    codeword_found = false

    while not codeword_found:
        n_k = H_Stern.nrows()
        k = H_Stern.ncols() - n_k
        I_n = identity_matrix(n_k)
        singular = true

        while singular:
            P_Stern = _GetRandomPermutationMatrix(H_Stern.ncols())
            H_Prime = H_Stern * P_Stern
            H_Prime.echelonize()
            if H_Prime.submatrix(0,0,n_k,n_k) == I_n:
                singular = false

        Z = set()
        while len(Z) < n_k - l:
```

```
            Z.add(randint(0, n_k-1))
        Z = list(Z)
        Z.sort()

        H_Prime_l = copy(H_Prime)
        H_Prime_l = H_Prime_l.delete_rows(Z)

        X_Indices = []
        Y_Indices = []
        for i in range(k):
            if randint(0,1) == 0:
                X_Indices.append(i+n_k)
            else:
                Y_Indices.append(i+n_k)

        Subsets_of_X_Indices = Subsets(X_Indices, p)
        Subsets_of_Y_Indices = Subsets(Y_Indices, p)

        pi_A = []
        pi_B = []

        for i in range(Subsets_of_X_Indices.cardinality()):
            column_sum = 0
            for j in range(p):
                column_sum += H_Prime_l.submatrix(0,
Subsets_of_X_Indices[i][j], H_Prime_l.nrows(), 1)
            pi_A.append(column_sum)

        for i in range(Subsets_of_Y_Indices.cardinality()):
            column_sum = 0
            for j in range(p):
                column_sum += H_Prime_l.submatrix(0,
Subsets_of_Y_Indices[i][j], H_Prime_l.nrows(), 1)
            pi_B.append(column_sum)

        for i in range(len(pi_A)):
            for j in range(len(pi_B)):
                if pi_A[i] == pi_B[j]:
                    sum = 0
                    for k in Subsets_of_X_Indices[i]:
                        sum += H_Prime.submatrix(0, k, H_Prime.nrows(),
1)
                    for k in Subsets_of_Y_Indices[j]:
                        sum += H_Prime.submatrix(0, k, H_Prime.nrows(),
1)
                    if _GetColumnVectorWeight(sum) == (w - 2*p):
```

```
                        codeword_found = true
                        weight_w_codeword = matrix(GF(2),
H_Stern.ncols(), 1)
                        for index in range(n_k):
                            if sum[index,0] == 1:
                                weight_w_codeword[index,0] = 1
                        for k in Subsets_of_X_Indices[i]:
                            weight_w_codeword[k,0] = 1
                        for k in Subsets_of_Y_Indices[j]:
                            weight_w_codeword[k,0] = 1
                        weight_w_codeword =
weight_w_codeword.transpose() * ~(P_Stern)
                        return copy(weight_w_codeword)
```