TP3-Problema2-Schnorr

May 30, 2022

1 TRABALHO PRÁTICO 3 - GRUPO 14

1.1 Exercício 2 - Algoritmo Schnorr

Neste problema era pertendido que implementasse-mos o algoritmo Schnorr que tem como objetivo contruir dois inteiros $X \neq \pm Y$ que verifiquem a relação $X^2 \equiv Y^2 \mod N$, a partir de uma solução aproximada do problema BDD em reticulados. Uma vez obtidos X, Y a fatorização de Fermat obtém um fator não-trivial de N como $\text{mdc}(X+Y, N)$ ou como $\text{mdc}(X-Y, N)$. A implementação do algoritmo foi realizada com o apoio dos apontamentos e do notebook.

1.1.1 RESOLUÇÃO DO PROBLEMA

Parâmetros Os parâmetros principais utilizados para implementar o algoritmo de Schnorr são: * **N** - É o input do problema, isto é, o número que pretendemos factorizar. Este número é obtido através da multiplicação de dois números primos, sendo esses obtidos através do parâmetro *bits*. Quanto maior o valor de *bits* maior é os primos gerados. * **n** - Indica o tamanho da base e do reticulado. Quanto maior for melhor é a aproximação da base ao valor de N. Como se pretende executar o algoritmo LLL numa solução aproximada do BDD, a dimensão do reticulado não deve ir além de 300. * **m** - Este parâmetro vai determinar o número de invocações do algoritmo BDD aproximado. Por isso, m é um parâmetro que pode ser modificado dinamicamente: começa-se por um valor pequeno (da ordem da dezena) e vai-se calculando mais valores forem sendo necessários. * **Q** - Lista de todos os n primeiros primos - $\mathcal{P}_n \equiv \{q_1, q_2, \cdots, q_n\}$ - que vai ser utilizada para fatorizar valores ao longo do algoritmo. * **P** - Lista de primos que sejam maiores que os primos presentes em Q. Neste caso começamos com um primo maior que o último primo de Q, que depois será incrementado.

[188]:
```
n = 100
m = 10

bits = 8
N = random_prime(2^bits-1,lbound=2^(bits-1)) *
 ↪random_prime(2^(bits-1)-1,lbound=2^(bits-2))

Q = Primes()[:n]
P = Primes()[n] #n*m*2
```

Oráculo BDD De forma a criar o oráculo BDD que será utilizado para determinar os erros do problema foi utilizada a implementação presente no notebook do docente. O oráculo implementado que é dado pela função `BDD` vai devolver os expoentes das soluções do BDD. As expressões lambdas apresentadas vão ser utilizadas na implementação da função.

[189]:
```
pZ    = lambda z : prod([q^e for (e,q) in zip(z,Q)])

lnQ   = lambda n : QQ(log(RDF(n),2))
sqlnQ = lambda n : QQ(sqrt(log(RR(n),2)))

vq    = [lnQ(q) for q in Q]
svq   = [sqlnQ(q) for q in Q]
```

[190]:
```
def BDD(N, L):
    mQ = matrix(QQ,n,1,vq)
    mZ = matrix(QQ,1,n,[0]*n)
    mz = matrix(QQ,n,1,[0]*n)
    mI = identity_matrix(QQ,n)
    mS = diagonal_matrix(QQ,n,svq)

    mt = matrix(QQ,1,1,[-lnQ(N)])
    mT = matrix(QQ,1,n,[0]*n).augment(mt)

    um = matrix(QQ,1,1,[1])

    G_ = block_matrix(QQ,1,2,[mI , L*mQ])
    GG = block_matrix(QQ,2,2,[G_,mz,L*mT,L*um])

    GGr = GG.LLL()
    last_line = GGr[n]

    u = last_line[-1]/L
    err = RDF(last_line[-2]/L)
    z = last_line[:n]

    return z
```

Obtenção dos erros ϵ_j Após a implementação do BDD, o primeiro passo do algoritmo passa por determinar os erros ϵ_j. Esta variável para que seja utilizada nos restantes passos do algoritmo é necessário que esta gere uma fatorização *smooth*, isto é, os fatores de ϵ_j devem todos pertencer ao array Q ($\mathcal{P}_n$). Assim, o ϵ_j pode ser dado por $\epsilon_j = \prod_{i=1}^{n} q_i^{b_i}$, com $q_i \in \mathcal{P}_n$.

Para isso vamos, a cada iteração do ciclo gerar um ϵ_j smooth e caso não seja possível a iteração do ciclo repete-se com outro target, N_j, tal que $N_j \equiv p_j\, N$, com $j = 1..m$ e p_j é um primo maior do que os q_n .

Começamos por utilizar o target N_j na função *BDD* sendo o resultado dessa função utilizada pela função `u_v` para calcular os vetores u e v. Os arrays obtidos nesta função serão utilizados para determinar o produtório e assim calcular o u_j e v_j dessa iteração. De seguida determinamos o respetivo erro dado pela expressão, $\epsilon_j = |v_j\, N_j - u_j|$, com $j = 1..m$.

Teremos, agora, de verificar se o erro ϵ_j gerado tem uma fatorização *smooth*, ou seja, que cumpra com a restrição apresentada em cima. Desta forma, implementou-se a função `isSmooth` que identifica se o erro ϵ tem uma fatorização desse tipo.

Caso o erro não tenham um fatorizaçao *smooth* então vamos gerar um novo target de Nj que será obtido através da multiplicação de um novo primo p_j e o valor N determinado no ínicio. Esse valor vai ser utilizado para invocar novamente o oráculo BDD e implementar novamente o algoritmo mencionado em cima, até que haja um conjunto de m erros que cumpram com a restrição da fatorização.

[191]:
```
def u_v(z):
    u = [0]*n ; v = [0]*n
    for k in range(n):
        if z[k] >= 0:
            u[k] = z[k]
        else:
            v[k] = -z[k]
    return (u,v)
```

[192]:
```
def isSmooth(x):
    y = x
    for p in Q:
        while p.divides(y):
            y /= p
    return abs(y) == 1
```

[193]:
```
c = 3
L = N^c

mj = 0
e = []
u = []
iterator = 0

print("#### N = " + str(N) + " ######")

while mj != m:

    Nj = P*N

    print("Iteração " + str(iterator) + ": Prime - " + str(P) + "| m - " +␣
 ↪str(mj) + "| Nj - " + str(Nj))

    z = BDD(Nj, L)
```

```
    (uz,vz) = u_v(z)

    uj = pZ(uz)
    vj = pZ(vz)

    ej = abs(Nj * vj - uj)

    smooth = isSmooth(ej)

    print("-----> Erro " + str(ej) + " é smooth: " + str(smooth))
    print()

    P = next_prime(P)
    iterator = iterator + 1

    if smooth == True:

        e.append(ej)
        u.append(uj)
        mj = mj +1

print("---------------------------------------------------------------")
print("Vetor e:")
print(e)
print()
print("Vetor u:")
print(u)
```

```
#### N = 18079 ######
Iteração 0: Prime - 547| m - 0| Nj - 9889213
-----> Erro 176447 é smooth: False

Iteração 1: Prime - 557| m - 0| Nj - 10070003
-----> Erro 91858219623 é smooth: False

Iteração 2: Prime - 563| m - 0| Nj - 10178477
-----> Erro 2914741 é smooth: False

Iteração 3: Prime - 569| m - 0| Nj - 10286951
-----> Erro 31981751 é smooth: True

Iteração 4: Prime - 571| m - 1| Nj - 10323109
-----> Erro 2275607 é smooth: False

Iteração 5: Prime - 577| m - 1| Nj - 10431583
-----> Erro 2314291 é smooth: False
```

```
Iteração 6: Prime - 587| m - 1| Nj - 10612373
-----> Erro 199659269 é smooth: False

Iteração 7: Prime - 593| m - 1| Nj - 10720847
-----> Erro 187962077 é smooth: False

Iteração 8: Prime - 599| m - 1| Nj - 10829321
-----> Erro 181547593 é smooth: False

Iteração 9: Prime - 601| m - 1| Nj - 10865479
-----> Erro 7786279 é smooth: False

Iteração 10: Prime - 607| m - 1| Nj - 10973953
-----> Erro 1913701 é smooth: False

Iteração 11: Prime - 613| m - 1| Nj - 11082427
-----> Erro 3846875 é smooth: False

Iteração 12: Prime - 617| m - 1| Nj - 11154743
-----> Erro 2638180 é smooth: False

Iteração 13: Prime - 619| m - 1| Nj - 11190901
-----> Erro 132297109 é smooth: False

Iteração 14: Prime - 631| m - 1| Nj - 11407849
-----> Erro 327293 é smooth: False

Iteração 15: Prime - 641| m - 1| Nj - 11588639
-----> Erro 31634 é smooth: False

Iteração 16: Prime - 643| m - 1| Nj - 11624797
-----> Erro 1222316 é smooth: False

Iteração 17: Prime - 647| m - 1| Nj - 11697113
-----> Erro 660 é smooth: True

Iteração 18: Prime - 653| m - 2| Nj - 11805587
-----> Erro 10161 é smooth: False

Iteração 19: Prime - 659| m - 2| Nj - 11914061
-----> Erro 1764167 é smooth: False

Iteração 20: Prime - 661| m - 2| Nj - 11950219
-----> Erro 159656 é smooth: False

Iteração 21: Prime - 673| m - 2| Nj - 12167167
-----> Erro 386224 é smooth: True
```

```
Iteração 22: Prime - 677| m - 3| Nj - 12239483
-----> Erro 31721064 é smooth: False

Iteração 23: Prime - 683| m - 3| Nj - 12347957
-----> Erro 7705443 é smooth: False

Iteração 24: Prime - 691| m - 3| Nj - 12492589
-----> Erro 4140091 é smooth: True

Iteração 25: Prime - 701| m - 4| Nj - 12673379
-----> Erro 202 é smooth: True

Iteração 26: Prime - 709| m - 5| Nj - 12818011
-----> Erro 2503891 é smooth: False

Iteração 27: Prime - 719| m - 5| Nj - 12998801
-----> Erro 13290908067382 é smooth: False

Iteração 28: Prime - 727| m - 5| Nj - 13143433
-----> Erro 1790415 é smooth: False

Iteração 29: Prime - 733| m - 5| Nj - 13251907
-----> Erro 6684069 é smooth: False

Iteração 30: Prime - 739| m - 5| Nj - 13360381
-----> Erro 2612 é smooth: False

Iteração 31: Prime - 743| m - 5| Nj - 13432697
-----> Erro 224666375390 é smooth: False

Iteração 32: Prime - 751| m - 5| Nj - 13577329
-----> Erro 152 é smooth: True

Iteração 33: Prime - 757| m - 6| Nj - 13685803
-----> Erro 18281 é smooth: True

Iteração 34: Prime - 761| m - 7| Nj - 13758119
-----> Erro 2425557305 é smooth: False

Iteração 35: Prime - 769| m - 7| Nj - 13902751
-----> Erro 1645 é smooth: True

Iteração 36: Prime - 773| m - 8| Nj - 13975067
-----> Erro 61428316 é smooth: False

Iteração 37: Prime - 787| m - 8| Nj - 14228173
-----> Erro 8884 é smooth: False
```

```
Iteração 38: Prime - 797| m - 8| Nj - 14408963
-----> Erro 20332010299 é smooth: False

Iteração 39: Prime - 809| m - 8| Nj - 14625911
-----> Erro 608 é smooth: True

Iteração 40: Prime - 811| m - 9| Nj - 14662069
-----> Erro 182103 é smooth: False

Iteração 41: Prime - 821| m - 9| Nj - 14842859
-----> Erro 46242323 é smooth: False

Iteração 42: Prime - 823| m - 9| Nj - 14879017
-----> Erro 71149137 é smooth: False

Iteração 43: Prime - 827| m - 9| Nj - 14951333
-----> Erro 4982431 é smooth: False

Iteração 44: Prime - 829| m - 9| Nj - 14987491
-----> Erro 1301617 é smooth: False

Iteração 45: Prime - 839| m - 9| Nj - 15168281
-----> Erro 463 é smooth: True

--------------------------------------------------------------
Vetor e:
[31981751, 660, 386224, 4140091, 202, 152, 18281, 1645, 608, 463]

Vetor u:
[184762085908171041634, 6727879001245436333, 4605782620198789191,
41706126344554323855, 3574082889971145, 1287504803882115, 376770452942396443,
16259476989694978, 54012464324531817, 92271828878469218]
```

Obtenção dos valores $a_{j,i}$ **e** $b_{j,i}$ De forma a obter os valores $a_{j,i}$ e $b_{j,i}$ vamos ter de calcular a fatorização de todos os elementos u_j e ϵ_j. Esta fatorização vai permitir obter os expoentes dos fatores de cada elemento tal como está nas expressões $u_j = \prod_{i=1}^{n} q_i^{a_{j,i}}$ e $\epsilon_j = \prod_{i=1}^{n} q_i^{b_{j,i}}$.

Para isso, vamos utilizar a função `fact` que vai fatorizar cada elemento u_j e ϵ_j e vai armazenar o respetivo expoente de cada fator num array. Esse array é depois armazenado na matriz a e b.

[194]:
```
def fact(x):

    y = [0]*n

    l = list(factor(x))

    for elem in l:
```

```
        index = Q.index(elem[0])
        y[index] = elem[1]

    return y
```

[195]:

```
a = []
b = []
for i in range(m):
    a.append(fact(u[i]))
    b.append(fact(e[i]))

print("Vetor com os expotentes a:")
print(a)
print()
print("Vetor com os expotentes b:")
print(b)
```

```
Vetor com os expotentes a:
[[1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1,
0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1,
0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 2, 1, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2,
1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 2, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1,
1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 3, 1, 1, 0, 1,
0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1,
0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 2, 0, 0, 1, 0, 0,
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 1, 0,
1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

Vetor com os expotentes b:
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [2, 1, 1, 0, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [3, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[5, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
```

Cálculo da solução não trivial $(z_1, \cdots, z_m)$ Para calcular a solução não trivial $(z_1, \cdots, z_m)$ vamos implementar um sistema de equações modulares tais que:

$$\left\{ \begin{array}{lcll} \sum_j s_j\, z_j & \equiv & 0 \mod 2 & \\ \sum_j (a_{j,i} - b_{j,i}) z_j & \equiv & 0 \mod 2 & \text{para todo } i = 1..n \end{array} \right.$$

Tal como demos ver, estas equações demonstram que o resultado dos somatórios tem de dar um número par, desta forma é possivel reescrever o sistema segundo equações linares e assim conseguirmos utilizar mais facilmente a função *solve* do sagemath:

$$\left\{ \begin{array}{lcll} \sum_j s_j\, z_j & \equiv & 2w & \\ \sum_j (a_{j,i} - b_{j,i}) z_j & \equiv & 2w_i & \text{para todo } i = 1..n \end{array} \right.$$

Como já temos conhecimento das variáveis $a_{j,i}$ e $b_{j,i}$ podemos obter facilmente a sua subtração. As restantes variáveis serão uma solução não trivial do sistema. É de notar que a última expressão do sistema é repetida n vezes e que a expressão que envolve as variáveis s_j não conseguiu ser implementada pois, essas variáveis s_j não cumprem com as restrições $u_j \equiv (-1)^{s_j}\, \epsilon_j \mod N$, com um $s_j \in \{0, 1\}$.

[196]:
```
zj = var('z', n=m, latex_name='z')
wj = var('w', n=n, latex_name='w')


eqA = []

for i in range(n):

    A = []

    for j in range(m):

        A.append(a[j][i] - b[j][i])

    eq = 2*wj[i] == sum([ A[index] * zj[index] for index in range(m) ])

    eqA.append(eq)

variables = zj + wj


resolution = solve(eqA, variables, solution_dict=True)[0]

print("Solução não trivial:")
print(resolution)
```

```
Solução não trivial:
{z0: r63, z1: r67, z2: r61, z3: -r61 - r62 - 2*r64 - 3*r65 + r67 + 2*r70, z4:
r64, z5: r65, z6: r69, z7: r68, z8: r62, z9: r66, w0: -2*r61 - 5/2*r62 + 1/2*r63
- 1/2*r64 - 3/2*r65 + 1/2*r66 - r67 + 1/2*r68, w1: r70, w2: -1/2*r61 - 1/2*r62 -
1/2*r64 - r65 - 1/2*r68 + r70, w3: -1/2*r61 - 1/2*r62 + 1/2*r63 - r64 - r65 +
1/2*r67 - 1/2*r68 + 1/2*r69 + r70, w4: 1/2*r66 - 1/2*r67, w5: 1/2*r62 + 1/2*r63
+ 1/2*r65 + 1/2*r67 + 1/2*r68, w6: 1/2*r66 + 1/2*r68, w7: -1/2*r62 + 1/2*r64 -
1/2*r65 + 1/2*r66 + 1/2*r69, w8: 1/2*r62 + 1/2*r64 + 1/2*r65 + 1/2*r66 +
1/2*r67, w9: -1/2*r63 + 1/2*r67 + 1/2*r69, w10: 1/2*r61 + 1/2*r66 + 1/2*r67,
w11: 1/2*r64 + 1/2*r67, w12: 1/2*r61 - 1/2*r62 - r64 - 3/2*r65 + 1/2*r67 + r70,
w13: 1/2*r61 + 1/2*r64 + 1/2*r65 + 1/2*r67, w14: -1/2*r61 - 1/2*r62 + 1/2*r63 -
r64 - 3/2*r65 + 1/2*r67 - 1/2*r68 + 1/2*r69 + r70, w15: -1/2*r61 - 1/2*r62 - r64
- 3/2*r65 + 1/2*r67 + 1/2*r68 + r70, w16: 1/2*r62 + 1/2*r63 + 1/2*r66 + 1/2*r68,
w17: -1/2*r63 + 1/2*r65, w18: 0, w19: 1/2*r61 + 1/2*r68, w20: r62 + 1/2*r63 +
1/2*r69, w21: 1/2*r61 + 1/2*r64, w22: 0, w23: 1/2*r61 + 1/2*r62 + 1/2*r65 +
1/2*r67 + 1/2*r68, w24: 0, w25: 0, w26: -1/2*r61 - 1/2*r62 - 1/2*r64 - 3/2*r65 +
1/2*r67 + r70, w27: -1/2*r62 + 1/2*r63 - r64 - 3/2*r65 + 1/2*r67 + 1/2*r68 +
r70, w28: 1/2*r67, w29: 1/2*r62 + 1/2*r66 + r67 + 1/2*r69, w30: 1/2*r61 +
1/2*r63 + 1/2*r66 + 1/2*r67 + 1/2*r68, w31: -1/2*r61 - 1/2*r62 - r64 - r65 +
1/2*r67 + 1/2*r69 + r70, w32: 1/2*r62 + 1/2*r63 + 1/2*r66 + 1/2*r68 + 1/2*r69,
w33: 1/2*r62 + 1/2*r64 + 1/2*r69, w34: -1/2*r61 - 1/2*r62 + 1/2*r63 - r64 - r65
+ 1/2*r67 + r70, w35: 0, w36: 1/2*r66, w37: 0, w38: 0, w39: 0, w40: 0, w41:
-1/2*r69, w42: 0, w43: 0, w44: 0, w45: 0, w46: 0, w47: 0, w48: 0, w49: 1/2*r61 +
1/2*r62 + r64 + 3/2*r65 - 1/2*r67 - r70, w50: 0, w51: -1/2*r61, w52: 0, w53: 0,
w54: 0, w55: 0, w56: 0, w57: 0, w58: 0, w59: 0, w60: 0, w61: 0, w62: 0, w63: 0,
w64: 0, w65: 0, w66: 0, w67: 0, w68: 0, w69: 0, w70: 0, w71: 0, w72: 0, w73: 0,
w74: 0, w75: 0, w76: 0, w77: 0, w78: 0, w79: 0, w80: 0, w81: 0, w82: 0, w83: 0,
w84: 0, w85: 0, w86: 0, w87: 0, w88: 0, w89: -1/2*r66, w90: 0, w91: 0, w92: 0,
w93: 0, w94: 0, w95: 0, w96: 0, w97: 0, w98: 0, w99: 0}
```

Determinar valores para a solução não trivial Para determinarmos uma possivel solução para a solução trivial não encontrada, começamos por determinar se a solução encontrada é uma expressão ou tem uma única variável. Caso só tenha um argumento então vamos assumir o valor de 1 para essa variável. Caso a solução encontrada esteja dependente de uma expressão então vamos substituir os argumentos dessa expressão pelo o valor assumido nas restantes variávies.

Exemplo: Assumindo a solução trivial $x_0 = -r_{13} - r_{14}$, $x_1 = r_{14}$, $x_2 = r_{13}$ vamos começar por assumir o valor de 1 às variáveis x_1 e x_2 pois só estão igualadas a um argumento, ficando com $x_0 = -r_{13} - r_{14}$, $x_1 = 1$, $x_2 = 1$. A variável x_0 será transformada em $x_0 = -1 - 1 = -2$ através da substituição dos valores assumidos.

A função `getArguments` vai permitir que se consiga a arranjar um conjunto de argumentos todos diferentes que sejam igualados a 1 para depois serem usados para substituir as variavéis que estão dependentes de expressões numéricas. No final, o array *result* vai conter todas as variáveis da solução não trivial igualadas a um possivél valor numérico que será utilizado nos restantes cálculos.

[197]:
```
def getArguments(arguments):
```

```
    buildArgs = []
    listArgs = []

    for i in range(len(arguments)):

        arg = arguments[i][1]

        buildArgs.append(arg)

    for elem in buildArgs:

        if elem not in listArgs:
            listArgs.append(elem)

    return listArgs
```

[198]:
```
arguments = []

equations = []

for i in range(n+m):

    dictAux = []

    args = resolution[variables[i]].arguments()

    if len(args) == 1:

        aux = {args[0] :1}
        arguments.append((variables[i], aux))
    else:
        equations.append((variables[i], resolution[variables[i]]))

print("Novos argumentos:")
print(arguments)
print()
print("Equações:")
print(equations)
print()

result = []

for i in range(len(equations)):

    t = equations[i][1]

    listArgs = getArguments(arguments)
```

```python
    result.append((equations[i][0],t.subs(listArgs)))


for i in range(len(arguments)):

    result.append((arguments[i][0],1))

print("Resultado de todas as soluções não triviais:")
print(result)
```

```
Novos argumentos:
[(z0, {r63: 1}), (z1, {r67: 1}), (z2, {r61: 1}), (z4, {r64: 1}), (z5, {r65: 1}),
(z6, {r69: 1}), (z7, {r68: 1}), (z8, {r62: 1}), (z9, {r66: 1}), (w1, {r70: 1}),
(w28, {r67: 1}), (w36, {r66: 1}), (w41, {r69: 1}), (w51, {r61: 1}), (w89, {r66:
1})]

Equações:
[(z3, -r61 - r62 - 2*r64 - 3*r65 + r67 + 2*r70), (w0, -2*r61 - 5/2*r62 + 1/2*r63
- 1/2*r64 - 3/2*r65 + 1/2*r66 - r67 + 1/2*r68), (w2, -1/2*r61 - 1/2*r62 -
1/2*r64 - r65 - 1/2*r68 + r70), (w3, -1/2*r61 - 1/2*r62 + 1/2*r63 - r64 - r65 +
1/2*r67 - 1/2*r68 + 1/2*r69 + r70), (w4, 1/2*r66 - 1/2*r67), (w5, 1/2*r62 +
1/2*r63 + 1/2*r65 + 1/2*r67 + 1/2*r68), (w6, 1/2*r66 + 1/2*r68), (w7, -1/2*r62 +
1/2*r64 - 1/2*r65 + 1/2*r66 + 1/2*r69), (w8, 1/2*r62 + 1/2*r64 + 1/2*r65 +
1/2*r66 + 1/2*r67), (w9, -1/2*r63 + 1/2*r67 + 1/2*r69), (w10, 1/2*r61 + 1/2*r66
+ 1/2*r67), (w11, 1/2*r64 + 1/2*r67), (w12, 1/2*r61 - 1/2*r62 - r64 - 3/2*r65 +
1/2*r67 + r70), (w13, 1/2*r61 + 1/2*r64 + 1/2*r65 + 1/2*r67), (w14, -1/2*r61 -
1/2*r62 + 1/2*r63 - r64 - 3/2*r65 + 1/2*r67 - 1/2*r68 + 1/2*r69 + r70), (w15,
-1/2*r61 - 1/2*r62 - r64 - 3/2*r65 + 1/2*r67 + 1/2*r68 + r70), (w16, 1/2*r62 +
1/2*r63 + 1/2*r66 + 1/2*r68), (w17, -1/2*r63 + 1/2*r65), (w18, 0), (w19, 1/2*r61
+ 1/2*r68), (w20, r62 + 1/2*r63 + 1/2*r69), (w21, 1/2*r61 + 1/2*r64), (w22, 0),
(w23, 1/2*r61 + 1/2*r62 + 1/2*r65 + 1/2*r67 + 1/2*r68), (w24, 0), (w25, 0),
(w26, -1/2*r61 - 1/2*r62 - 1/2*r64 - 3/2*r65 + 1/2*r67 + r70), (w27, -1/2*r62 +
1/2*r63 - r64 - 3/2*r65 + 1/2*r67 + 1/2*r68 + r70), (w29, 1/2*r62 + 1/2*r66 +
r67 + 1/2*r69), (w30, 1/2*r61 + 1/2*r63 + 1/2*r66 + 1/2*r67 + 1/2*r68), (w31,
-1/2*r61 - 1/2*r62 - r64 - r65 + 1/2*r67 + 1/2*r69 + r70), (w32, 1/2*r62 +
1/2*r63 + 1/2*r66 + 1/2*r68 + 1/2*r69), (w33, 1/2*r62 + 1/2*r64 + 1/2*r69),
(w34, -1/2*r61 - 1/2*r62 + 1/2*r63 - r64 - r65 + 1/2*r67 + r70), (w35, 0), (w37,
0), (w38, 0), (w39, 0), (w40, 0), (w42, 0), (w43, 0), (w44, 0), (w45, 0), (w46,
0), (w47, 0), (w48, 0), (w49, 1/2*r61 + 1/2*r62 + r64 + 3/2*r65 - 1/2*r67 -
r70), (w50, 0), (w52, 0), (w53, 0), (w54, 0), (w55, 0), (w56, 0), (w57, 0),
(w58, 0), (w59, 0), (w60, 0), (w61, 0), (w62, 0), (w63, 0), (w64, 0), (w65, 0),
(w66, 0), (w67, 0), (w68, 0), (w69, 0), (w70, 0), (w71, 0), (w72, 0), (w73, 0),
(w74, 0), (w75, 0), (w76, 0), (w77, 0), (w78, 0), (w79, 0), (w80, 0), (w81, 0),
(w82, 0), (w83, 0), (w84, 0), (w85, 0), (w86, 0), (w87, 0), (w88, 0), (w90, 0),
(w91, 0), (w92, 0), (w93, 0), (w94, 0), (w95, 0), (w96, 0), (w97, 0), (w98, 0),
(w99, 0)]
```

```
Resultado de todas as soluções não triviais:
[(z3, -4), (w0, -6), (w2, -2), (w3, -1), (w4, 0), (w5, 5/2), (w6, 1), (w7, 1/2),
(w8, 5/2), (w9, 1/2), (w10, 3/2), (w11, 1), (w12, -1), (w13, 2), (w14, -3/2),
(w15, -3/2), (w16, 2), (w17, 0), (w18, 0), (w19, 1), (w20, 2), (w21, 1), (w22,
0), (w23, 5/2), (w24, 0), (w25, 0), (w26, -3/2), (w27, -1/2), (w29, 5/2), (w30,
5/2), (w31, -1), (w32, 5/2), (w33, 3/2), (w34, -1), (w35, 0), (w37, 0), (w38,
0), (w39, 0), (w40, 0), (w42, 0), (w43, 0), (w44, 0), (w45, 0), (w46, 0), (w47,
0), (w48, 0), (w49, 2), (w50, 0), (w52, 0), (w53, 0), (w54, 0), (w55, 0), (w56,
0), (w57, 0), (w58, 0), (w59, 0), (w60, 0), (w61, 0), (w62, 0), (w63, 0), (w64,
0), (w65, 0), (w66, 0), (w67, 0), (w68, 0), (w69, 0), (w70, 0), (w71, 0), (w72,
0), (w73, 0), (w74, 0), (w75, 0), (w76, 0), (w77, 0), (w78, 0), (w79, 0), (w80,
0), (w81, 0), (w82, 0), (w83, 0), (w84, 0), (w85, 0), (w86, 0), (w87, 0), (w88,
0), (w90, 0), (w91, 0), (w92, 0), (w93, 0), (w94, 0), (w95, 0), (w96, 0), (w97,
0), (w98, 0), (w99, 0), (z0, 1), (z1, 1), (z2, 1), (z4, 1), (z5, 1), (z6, 1),
(z7, 1), (z8, 1), (z9, 1), (w1, 1), (w28, 1), (w36, 1), (w41, 1), (w51, 1),
(w89, 1)]
```

Cálculo do c_i Sendo que $c_i \equiv \frac{\sum_j (a_{j,i}-b_{j,i})\, z_j}{2}$, com $i = 1..n$ temos que cada c_i vai corresponder aos valores de cada solução w_i encontrada. Desta forma, conseguimos obter apartir do vetor result os valores das variáveis c_i:

$$c_i = \frac{\sum_j (a_{j,i} - b_{j,i})\, z_j}{2} \equiv c_i = \frac{2w_i}{2} \equiv c_i = w_i$$

[199]:
```
z = [{}]*m
c = [0]*n

for i in range(n+m):

    aux = {}

    if result[i][0] not in zj:

        index = wj.index(result[i][0])

        c[index] = result[i][1]

    else:

        index = zj.index(result[i][0])

        aux[result[i][0]] = result[i][1]
        z[index] = aux

print("Vetor z:")
print(z)
```

```
print()
print("Vetor c:")
print(c)
```

```
Vetor z:
[{z0: 1}, {z1: 1}, {z2: 1}, {z3: -4}, {z4: 1}, {z5: 1}, {z6: 1}, {z7: 1}, {z8:
1}, {z9: 1}]

Vetor c:
[-6, 1, -2, -1, 0, 5/2, 1, 1/2, 5/2, 1/2, 3/2, 1, -1, 2, -3/2, -3/2, 2, 0, 0, 1,
2, 1, 0, 5/2, 0, 0, -3/2, -1/2, 1, 5/2, 5/2, -1, 5/2, 3/2, -1, 0, 1, 0, 0, 0, 0,
1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0]
```

Cálculo do X e Y Após obter os valores de c_i, estes serão utilizados para obter o numerador X e o denominador Y através das respetivas expressões de forma a cumprir com a relação $X^2 \equiv Y^2 \mod N$:

$$X \equiv \prod_{c_i>0} q_i^{c_i} \qquad Y \equiv \prod_{c_i<0} q_i^{-c_i}$$

Estas variáveis serão utilizadas para a fatorização de Fermat onde se obtém um fator não-trivial de N como $\text{mdc}(X+Y, N)$.

[200]:
```
X = 1
Y = 1

for i in range(n):

    if c[i] < 0:

        Y *= Q[i]^((-1) * c[i])

    if c[i] > 0:

        X *= Q[i]^(c[i])

print("X = " + str(float(X)) + " | Y = " + str(float(Y)))
print()
print("Fatorização de Fermat:")
print(gcd(float(X)+float(Y),N))
```

```
X = 2.712682300173775e+67 | Y = 1.2049464892749881e+19

Fatorização de Fermat:
1.0
```

[201]:
```
print("Variáveis satisfazem a restrição X^2 == Y^2 mod N?")
#print(float(X)^2 == mod(float(Y)^2,N))
```

```
Variáveis satisfazem a restrição X^2 == Y^2 mod N?
```

------------------------------ Em Falta --

Obtenção do triplo (u_j, ϵ_j, s_j) Desta relação conclui-se que $u_j \equiv \pm\epsilon_j \mod N$. Desta forma constrói-se m triplos de inteiros positivos "smooth" (u_j, ϵ_j, s_j), com um "sinal" $s_j \in \{0, 1\}$, que verificam $u_j \equiv (-1)^{s_j}\,\epsilon_j \mod N$

Atenção: Será que s_j ia dar sempre 0 pois os valores envolvidos na expressão serão sempre positivos?

[202]:
```
s = [-1]*m

for i in range(m):

    if u[i]%N == e[i]:
        s[i] = 0

    elif u[i]%N == (-1)*e[i]:
        s[i] = 1

print(s)
```

```
[-1, -1, -1, -1, -1, 0, -1, 0, 0, 0]
```

Cálculo da solução não trivial $(z_1, \cdots, z_m)$

[203]:
```
w = var('w')
eqS = 2*w == sum([s[i]*zj[i] for i in range(m)])
print(eqS)
```

```
2*w == -z0 - z1 - z2 - z3 - z4 - z6
```