De la quantique en cryptographie

Élie Besnard, Malo Leroy,
Yun Marcola–da-Cunha Macedo

26 juin 2022

Table des matières

1 Ordinateur quantique

1.1 Qubits

```python
from random import choices
from calcul import Matrice, un, zero, int_log2, int_vers_strbin, bin_vers_int, int_vers_bin


class EtatPropre:
    def __init__(self, nom):
        # nom est un int ou un str a l'entree (il est converti en str)
        self.nom =str(nom)

    def __str__(self):
        return '|' +self.nom +'  '

    def __and__(self, autre):
        assert isinstance(autre, EtatPropre)
        return EtatPropre(self.nom +autre.nom)

    def __eq__(self, autre):
        # On ne teste pas l'egalite des noms
```

```
        return isinstance(autre, EtatPropre) and autre.valeur ==self.valeur

    # Change le nom d'un etat propre
    def renomme(self, nom):
        self.nom =str(nom)


# L'equivalent d'un Qubit, mais avec dim etats propres differents
class Qudit:
    # Un Qudit possede dim etats possibles
    def __init__(self, mat: Matrice):
        assert mat.q ==1
        self.dim =mat.p
        nom = lambda i: int_vers_strbin(i, taille =int_log2(self.dim)-1)
        self.base =[EtatPropre(nom(i)) for i in range(self.dim)] # vecteurs propres
        self.matrice =mat

    @staticmethod
    def zero(dim: int):
        return Qudit(Matrice.colonne(*([un] +[zero] *(dim-1))))

    def __eq__(self, autre):
        return self.matrice ==autre.matrice

    def __getitem__(self, bras):
        if isinstance(bras, int):
            return self.matrice[bras]
        assert all([(i ==0 or i ==1) for i in bras])
        return self.matrice[bin_vers_int(*bras)]

    def __setitem__(self, bras, valeur):
        self |= bras, valeur

    # Notation plus commode pour les circuits
    def __rshift__(self, autre):
        return autre *self

    # Stockage d'une valeur dans une composante
    # La condition de normalisation peut ne plus tre verifiee apres
    def __ior__(self, autre):
        bra, val =autre
        if isinstance(bra, int):
            self.matrice[bra] =val
        else:
            assert isinstance(bra, Bra)
            self |= (bin_vers_int(*bra.composante), val)
        return self

    def __mul__(self, bra):
        assert isinstance(bra, Bra)
        return self.matrice *bra.matrice

    def __matmul__(self, autre):
        return Qudit(self.matrice @ autre.matrice)

    def mesure(self):
        probs =[float(abs(self[i])*abs(self[i])) for i in range(self.dim)]
        choix =choices(list(range(self.dim)), weights =probs)[0]
```

```
        for i in range(self.dim):
            self |= (i, zero)
        self |= (choix, un)
        return self

    def __neg__(self):
        return Qudit(Matrice([
            [- self[i]] for i in range(self.dim)
        ]))

    def __str__(self):
        return ' + '.join([
            str(self[i]) +str(self.base[i])
            for i in range(self.dim)
            if self[i] !=zero
        ])

    def __repr__(self):
        return str(self)


class Qubit(Qudit):
    def __init__(self, c0 =None, c1 =None):
        super().__init__(Matrice.colonne(un, zero))
        if c0 is not None and c1 is not None:
            assert abs(c0) *abs(c0) +abs(c1) *abs(c1) ==un
            self.matrice[0] =c0
            self.matrice[1] =c1

    @staticmethod
    def propre(n: int):
        assert n ==0 or n ==1
        if n ==0:
            return Qubit.zero()
        return Qubit.un()

    @staticmethod
    def zero():
        return Qubit()

    @staticmethod
    def un():
        return Qubit(zero, un)

    def _puissance_rapide(self, n :int):
        if self ==ket(0):
            return Qudit.zero(2 **n)
        return Qudit(Matrice.colonne(*([zero] *(2**n-1) +[un])))

    def __pow__(self, n: int):
        if self ==ket(0) or self ==ket(1):
            return self._puissance_rapide(n)
        if n ==1:
            return self
        a = self **(n//2)
        if n % 2 ==1:
            return self @ (a @ a)
        return a @ a
```

```
def ket(*arg, taille =None):
    assert taille is None or isinstance(taille, int)
    d = []
    for i in arg:
        d += int_vers_bin(int(i))
    assert all([i ==0 or i ==1 for i in d])
    if taille is not None:
        d = (taille -len(d)) *[0] +d
    q = Qubit.propre(d[0])
    for i in d[1:]:
        q = q @ Qubit.propre(i)
    return q


class Bra:
    def __init__(self, *composante):
        self.composante =()
        for i in composante:
            self.composante +=tuple(int_vers_bin(int(i)))
        self.matrice =Matrice.zeros(1, 2 **len(self.composante))
        self.matrice[bin_vers_int(*self.composante)] =un

    def __eq__(self, autre):
        return (isinstance(autre, Bra)
            and self.composante ==autre.composante)

    def __or__(self, autre):
        if isinstance(autre, Qudit):
            return autre[self.composante]
        # Il s'agit d'une assignation
        return self, autre

    def __matmul__(self, autre):
        assert isinstance(autre, Bra)
        return Bra(*(self.composante +autre.composante))

    def __pow__(self, n):
        n = int(n)
        if n ==1:
            return self
        a = self **(n//2)
        if n % 2:
            return self @ (a @ a)
        return (a @ a)

    def __str__(self):
        if isinstance(self.composante, int):
            return '  ' + str(self.composante) +'|'
        return '  ' + ''.join([str(i) for i in self.composante]) +'|'


def bra(*composante):
    return Bra(*composante)
```

1.2 Portes et oracles

```python
from calcul import un, i, sqrt, Matrice, expi, Expi, pi
from qubit import Qudit, bra, ket

class Porte:
    # Une ‘Porte‘ s’utilise comme une matrice, en multipliant
    def __init__(self, matrice):
        assert isinstance(matrice, Matrice)
        assert matrice.p ==matrice.q
        assert (matrice.p ==1) or matrice.p % 2 ==0
        self.matrice =matrice
        self.taille =matrice.p //2 # 0 si c’est la porte neutre

    @staticmethod
    def neutre():
        return _neutre

    def __eq__(self, autre):
        return isinstance(autre, Porte) and self.matrice ==autre.matrice

    def __mul__(self, autre):
        if self ==Porte.neutre():
            return autre
        if autre ==Porte.neutre():
            return self
        if isinstance(autre, Qudit):
            return Qudit(self.matrice *autre.matrice)
        elif isinstance(autre, Porte):
            return Porte(self.matrice *autre.matrice)
        raise TypeError(f’{autre} n\’est ni une porte chanable ni un qudit’)

    def __rshift__(self, autre):
        return autre *self

    def dague(self):
        return Porte(self.matrice.transposee().conjuguee())

    def __matmul__(self, autre):
        assert isinstance(autre, Porte)
        return Porte(self.matrice @ autre.matrice)

    def __pow__(self, n: int):
        if self ==H and n ==7:
            return __import__(’hadamarapide’).H7
        return Porte(self.matrice **n)

    def __neg__(self):
        return Porte(- self.matrice)

    def __str__(self):
        return str(self.matrice)

    def __repr__(self):
        return str(self)


_neutre =Porte(Matrice.identite(1))
```

```
I = Identite =Porte(Matrice([[1, 0], [0, 1]]))

H = Hadamard =Porte(sqrt(un /2) *Matrice([[1, 1], [1, -1]]))

X = PauliX =Porte(Matrice([[0, 1], [1, 0]]))

Y = PauliY =Porte(Matrice([[0, -i], [i, 0]]))

iY = iPauliY =Porte(Matrice([[0, 1], [-1, 0]]))

Z = PauliZ =Porte(Matrice([[1, 0], [0, -1]]))

R = lambda phi: Porte(Matrice([[1, 0], [0, expi(phi)]]))

PhaseCond =lambda n: Porte((ket(0) **n) *(bra(0) **n) *2 -(Matrice.identite(2**n)))

S = SWAP =Porte(Matrice([
    [1, 0, 0, 0],
    [0, 0, 1, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1]
]))

cX = CNOT =Porte(Matrice([
    [1, 0, 0, 0],
    [0, 1, 0, 0],
    [0, 0, 0, 1],
    [0, 0, 1, 0]
]))

def QFT(N: int):
    omega =Expi(pi *2 / N)
    t = [[None] *N for i in range(N)]
    for k in range(N):
        for j in range(N):
            t[k][j] =(omega **(k * j)).sous()
    return Porte(sqrt(un /N) *Matrice(t))
```

```
from qubit import bra, ket, Qudit, Qubit
from calcul import zero, un, F2, int_vers_bin, sqrt, int_log2, Matrice, F2Uplet, bin_vers_int

class Oracle:
    @staticmethod
    def phase(f):
        return OracleDePhase(f)

    @staticmethod
    def somme(f, *, m =1):
        return OracleDeSomme(f, m =m)

    @staticmethod
    def brut(f):
        return OracleBrut(f)


class OracleBrut:
    def __init__(self, f):
```

```
        self.f =f

    def __mul__(self, qudit):
        assert isinstance(qudit, Qudit)
        return ket(self.f(qudit))


class OracleDePhase:
    # f est une fonction de (F2)^n dans F2
    def __init__(self, f):
        self.f =f

    def __mul__(self, qudit):
        n = int_log2(qudit.dim) -1
        r = Qudit.zero(qudit.dim)
        for i in range(qudit.dim):
            r |= bra(i) | (bra(i) | qudit) *(-un) **self.f(
                *[F2(i) for i in int_vers_bin(i, taille=n)])
        return r


class OracleDeSomme:
    # f est une fonction de (F2)^n dans (F2)^m
    # par defaut, la taille de y est 1 (f va de (F2)^n dans F2)
    def __init__(self, f, *, m =1):
        self.f =f
        self.m =m

    def _trouve_i0(self, psi):
        for i in range(psi.dim):
            if bra(i) | psi !=zero:
                return i

    def _coord_y(self, c_non_nul, c_autre):
        a = (c_autre /c_non_nul)
        return sqrt((a*a +un).inverse())

    def _trouve_alpha_beta(self, psi):
        i0 = self._trouve_i0(psi)
        i1 = i0+1 if i0 % 2 ==0 else i0-1
        c = self._coord_y(psi[i0], psi[i1])
        if i0 % 2 ==0:
            alpha =c
            beta =sqrt(un -alpha*alpha)
        else:
            beta =c
            alpha =sqrt(un -beta*beta)
        return alpha, beta

    def _trouve_coord_x(self, alpha, beta, psi):
        n = psi.dim //2
        if alpha !=zero:
            a_inv =alpha.inverse()
            return [a_inv *psi[2*i] for i in range(n)]
        b_inv =beta.inverse()
        return [b_inv *psi[2*i+1] for i in range(n)]

    def _trouve_x_y(self, psi):
```

```
        x, y =psi, None
        for i in range(self.m):
            alpha, beta =self._trouve_alpha_beta(x)
            coord_x =self._trouve_coord_x(alpha, beta, x)
            x = Qudit(Matrice.colonne(coord_x))
            if y is None:
                y = Qubit(alpha, beta)
            else:
                y = Qubit(alpha, beta) @ y
        return x, y

    def __mul__(self, qudit):
        psi = Qudit(
            Matrice([[abs(qudit[i])] for i in range(qudit.dim)]))
        n = int_log2(psi.dim //2) -self.m
        x, y =self._trouve_x_y(psi)
        res = Qudit.zero(psi.dim)
        res[0] =zero
        for i in range(2**n):
            fx = self.f(*[F2(k) for k in int_vers_bin(i, taille=n)])
            if isinstance(fx, F2): fx =(fx,)
            u = F2Uplet(*fx)
            for j in range(2**self.m):
                v = F2Uplet(*int_vers_bin(j, taille=self.m))
                c = tuple(int_vers_bin(i, taille=n)) +tuple(u +v)
                res[bin_vers_int(*c)] =x[i] *y[j]
        return res
```

1.3 Fonctions utiles

```
from portes import Porte, H, I
from qubit import bra, ket
from calcul import un, zero, int_vers_bin, int_log2, Matrice

# Teste si une liste d'entiers correspond a un qubit
# (ne fonctionne evidemment que pour les etats propres)
def sequence_egale(sequence_attendue, qubit_obtenu):
    val = bra(*sequence_attendue)
    resultat =val | qubit_obtenu
    return (2 **len(sequence_attendue) ==qubit_obtenu.dim
            and resultat ==un)

def etat_de_base(n_principal, n_auxiliaire, val_auxiliaire =0):
    return (ket(0) **n_principal) @ (ket(val_auxiliaire) **n_auxiliaire)

# ne fonctionne que pour les etats propres
def ket_vers_liste(q):
    n = q.dim
    for i in range(n):
        if bra(i) | q !=zero:
            return int_vers_bin(i, taille=int_log2(n)-1)

# Fonctionne comme range, la fin est exclue.
# Si `fin` est negatif on part de la fin.
def H_option(total, *, debut, fin):
    assert isinstance(total, int) and isinstance(fin, int)
```

```
    assert isinstance(debut, int) and debut >=0
    if fin <0:
        fin = total +fin
    A = (I ** debut)
    B = (H ** (fin - debut))
    return kron_id(A @ B, total -fin)

# Cree des etats propres et les fait tous passer dans une porte de Hadamard.
def qubits_intriques(n, valeur =0):
    assert isinstance(n, int) and isinstance(valeur, int)
    return (ket(valeur)**n) >>(H**n)

# On fait le calcul `m @ I(n)`, avec I(n) l'identite de taille n,
# et m une matrice quelconque.
# Les analyses montrent que c'est en moyenne 30 fois plus rapide.
def kron_id_mat(m, n):
    r = Matrice.zeros(m.p *n, m.q *n)
    for i in range(m.p):
        for j in range(m.q):
            for k in range(n):
                r[i*n +k, j*n +k] = m[i, j]
    return r

# On calcule la *porte* `P @ (I ** 2)`, avec I l'identite de taille 2.
def kron_id(P, n):
    return Porte(kron_id_mat(P.matrice, 2**n))
```

2 Algorithmes

2.1 Deutsch-Jozsa et Bernstein-Vazirani

```
from calcul import un, zero
from fonctions_utiles import qubits_intriques
from portes import H
from oracle import Oracle
from qubit import bra

def dj(f, n):
    q = qubits_intriques(n)
    U = Oracle.phase(f)
    C = q >> U >> (H**n)
    return C

def est_constante(f, n):
    q = dj(f, n)
    test =bra(*([0]*n)) | q
    return (test ==un or test ==-un)

def point(x_list, a_list):
    n = len(x_list)
    d = zero
    for i in range(n):
        d += (x_list[i]*a_list[i])
    return d
```

```
def bv(a):
    return dj(lambda *args: point(args, a), len(a))
```

2.2 Grover

```
from math import asin, pi, sqrt
import matplotlib.pyplot as plt

from calcul import F2, int_log2, int_vers_bin
from fonctions_utiles import etat_de_base, H_option
from portes import H, I, PhaseCond
from oracle import Oracle

def main():
    ef = grover(*indicatrice(0, 1, 0, 1, 1, 0))
    print('L\'etat de sortie est :', ef)
    print('Une solution est', solution(ef))
    affiche_amplitudes(ef)

def grover(f, n, M =1):
    theta0 =asin(sqrt(M /(2 **n)))
    rep = int(pi /(4 * theta0))
    H_op =H_option(n, debut=0, fin=-1)
    Uf = Oracle.phase(f)
    e = etat_de_base(n-1, 1, 1) >>(H**n)
    B = H_op >>(PhaseCond(n-1) @ I) >>H_op
    for i in range(rep):
        e = e >> B >> Uf
    return e

def solution(ef):
    s, fm =None, None
    for i in range(ef.dim):
        f = float(ef[i].abs_carre())
        if s is None or fm <f:
            s, fm =i, f
    return tuple(int_vers_bin(s, taille=int_log2(ef.dim)-1))

def affiche_amplitudes(ef):
    l = [float(ef[i].abs_carre()) for i in range(ef.dim)]
    x = list(range(ef.dim))
    plt.bar(x, l)
    plt.xlabel('etat propre')
    plt.ylabel('Probabilite (module au carre de l\'amplitude)')
    plt.show()

def indicatrice(*solution):
    def f(*valeurs):
        if valeurs ==tuple([F2(i) for i in solution]):
            return F2(1)
        return F2(0)
    return f, len(solution)


if __name__ =='__main__':
    main()
```

2.3 Shor

```
from random import randint
from time import time
import matplotlib.pyplot as plt

from calcul import zero, pgcd, bin_vers_int, int_vers_bin, Naturel
from portes import QFT
from oracle import Oracle
from fonctions_utiles import etat_de_base, H_option, kron_id

m = 4

def main():
    N = int(input('Entrez un nombre : '))
    t0 = time()
    f = facteurs(N)
    print('-' *20)
    print(f"{N} = {'  '.join([str(i) for i in f])}")
    print(f'(calcul effectue en {(time() - t0):.2f} s)')

def facteurs(N):
    n, p = deux_facteurs_shor(N)
    if n ==1:
        print(f'| {N} est premier')
        return [N]
    return facteurs(n) +facteurs(p)

def deux_facteurs_shor(N):
    if N ==2:
        return 1, 2
    print('Decomposition en deux facteurs de', N)
    deja_vus =[]
    while True:
        a = randint(2, N-1)
        if a in deja_vus:
            continue
        else:
            deja_vus.append(a)
        d = pgcd(a, N)
        if d !=1:
            print(f'Coup de bol ! {d} divise le NPA {a} et {N}')
            n, p = d, N //d
            return min(n, p), max(n, p)
        print(f'Recherche de periode [NPA={a}, N={N}] ...')
        r = periode(a, N)
        print(f'\tLa periode obtenue avec le NPA {a} est {r}')
        if r % 2 ==0 and a**(r//2) % N !=N-1:
            n = pgcd(a **(r // 2) +1, N)
            p = pgcd(a **(r // 2) -1, N)
            if n *p == N:
                print(f'C\'est une periode valide, {N} = {n}  {p}')
                return min(n, p), max(n, p)
        print('Periode invalide, on recommence')

def periode(a, N):
    ef = recherche_periode(a, N)
    demander_affichage(ef)
```

```
    for i in range(ef.dim):
        x = ef[i].abs_carre()
        if x !=zero:
            inv = x.inverse()
            if not inv.appartient(Naturel):
                raise ValueError(f'la periode {inv}, de type {type(inv)} n\'est pas un naturel')
            return int(inv)
    raise ValueError('etat final nul')

def cree_f(a, N):
    def f(*bits):
        x = bin_vers_int(*bits)
        return tuple(int_vers_bin((a **x) % N, taille=m))
    return f

def recherche_periode(a, N):
    U = Oracle.somme(cree_f(a, N), m =m)
    print('\tCreation de l\'etat initial ...')
    e0 = etat_de_base(m, m , 0)
    print('\tIntrication des etats ...')
    e1 = e0 >> H_option(2*m, debut=0, fin=m)
    print('\tPassage dans l\'oracle ...')
    e2 = e1 >> U
    print('\tCreation du circuit QFT ...')
    Q = QFT(2**m).dague()
    P = kron_id(Q, m)
    print('\tPassage dans le circuit QFT ...')
    e3 = e2 >> P
    return e3

def demander_affichage(ef):
    if input('\tAfficher l\'etat final ? [y/n] ') =='y':
        print('\tL\'etat final est :', ef)
    if input('\tAfficher les amplitudes ? [y/n] ') =='y':
        affiche_amplitudes(ef)

def affiche_amplitudes(ef):
    l = [float(ef[i].abs_carre()) for i in range(ef.dim)]
    x = list(range(ef.dim))
    plt.bar(x, l)
    plt.xlabel('etat propre')
    plt.ylabel('Amplitude (module au carre)')
    plt.show()

if __name__ =='__main__':
    main()
```

2.4 Parité

```
from portes import R, X
from oracle import Oracle
from qubit import bra, ket


def f(q):
    return (bra(1) | q).arg() % 2
```

```
Uf = Oracle.brut(f)

def etat_sortie(n):
    return ket(0) >>X >>R(n) >>Uf

def est_pair(n):
    return etat_sortie(n) ==ket(0)

def main():
    n = int(input('Entrez un entier : '))
    S = etat_sortie(n)

    print('L\'etat de sortie est ' +str(S) +'.')
    parite ='pair' if S ==ket(0) else 'impair'
    print(f'Donc {n} est {parite} !')

if __name__ =='__main__':
    main()
```

```
from portes import cX
from qubit import ket
from calcul import int_vers_bin
from fonctions_utiles import sequence_egale


def est_pair(n):
    psi =ket(int_vers_bin(n)[-1])
    s = (psi @ ket(1)) >>cX
    return sequence_egale([0, 1], s)


def affiche_parite(n):
    if est_pair(n):
        print("Le nombre pris en entree est pair.")
    else:
        print("Le nombre pris en entree est impair.")

if __name__ =='__main__':
    n = int(input("Entrez un nombre entier : "))
    affiche_parite(n)
```

3 Polarisation

```
import random
import matplotlib.animation as animation
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle
import numpy as np

fig, ax =plt.subplots()
plt.axis('square')

N_item =5 #nombre de photons a animer
```

```
photons ={}
for i in range(N_item):
    photons[i], =ax.plot([], [], ls ='none', marker ='o', color ='purple')

plt.xlim(-10,10)
plt.ylim(-10.5,10.5)

N = 300 #resulution de l'echantillonage de (Ox)
disp =20 #coeff d'ecartement
base =np.linspace(-10, 10, N)
x_pol_1 =-2
x_pol_2 =4
X = {}
for i in range(N_item):
    X[i] =np.concatenate((np.full(disp*i, -10), np.linspace(-10, 10, N), np.full(N, 10)), axis =None)
color =['blue', 'green']

plt.plot([x_pol_1,x_pol_1], [10.5,-10.5], color ='black')
plt.plot([x_pol_2,x_pol_2], [10.5,-10.5], color ='black')

ax.set_xticklabels([])
ax.set_yticklabels([])

plt.tick_params(axis ='x', length =0)
plt.tick_params(axis ='y', length =0)

key =['0'] *N_item

def list_to_str(l):
    ret =""
    for e in l:
        ret =ret +e
    return ret

text ={}
text[0] =ax.text(4, 4, list_to_str(key), fontsize =15, fontweight ='bold', color ='black')
ax.add_patch(Rectangle((8, -1), 2, 2))

def ret_tuple(dico):
    return tuple(dico[c] for c in dico)

def animate(i):
    for c in photons:
        photons[c].set_data(X[c][i], 0)

        if abs(photons[c].get_xdata() +10) <=10**-1:
                photons[c].set_color('purple')

        if abs(photons[c].get_xdata() -x_pol_1) <=10**-1:
                photons[c].set_color(random.choice(color))

        if abs(photons[c].get_xdata() -x_pol_2) <=10**-1 and photons[c].get_color() =='green':
            photons[c].set_alpha(0.0)

        if abs(photons[c].get_xdata() -8) <=10**-1 and photons[c].get_alpha() !=0.0:
            key[c] ='1'
            text[0].set_text(list_to_str(key))
```

```
    return ret_tuple(photons) +tuple(text[c] for c in text)

ani = animation.FuncAnimation(fig, animate, frames=range(2*N), blit =True, interval =5, repeat =False
                                                        )
plt.show()
```