Application de l'algèbre des courbes elliptiques à la cryptographie

Sauvat Émile

Table des matières

1 Introduction

Apparu pour la première fois en 1936 sous le nom de "machine de Turing", l'ordinateur a depuis beaucoup évolué et occupe aujourd'hui une place majeure dans la société, place qui tend a être toujours plus importante, particulièrement dans les métropole avec la connexion quasi-systématique des appareils (transports, bâtiments, commerces ...) au réseau informatique. Cet avènement du numérique a pour conséquence le développement de la cryptographie moderne, afin d'assurer la sécurité des échanges privés tels que les paiements en ligne dans un contexte ou toute les formes de communication sont publiques.

La cryptographie est la science du codage de message. Si l'on en retrouve des traces dès l'antiquité, où l'on tatouait des messages sur le crâne préalablement rasé des esclaves, elle doit aujourd'hui faire face à la puissance de calcul toujours plus grande des ordinateurs et s'appuie sur des concepts mathématiques d'algèbre très particuliers pour rester efficace.

Comme solution au besoin de bien coder des messages et pouvoir identifier un individu avec sûreté en ligne, trois chercheurs du MIT - Adi Shamir, Léonard Adleman et Ronald Rivest - mettent au point un algorithme de chiffrement dit asymétrique, c'est-à-dire avec une clé publique permettant à n'importe qui de chiffrer simplement un message, et une clé privée, en pratique très dure à retrouver, permettant à une unique personne de déchiffrer un message codé. C'est le premier du genre. Fonctionnant sur le principe du calcul dans un anneau $\mathbb{Z}/n\mathbb{Z}$ où $n \in \mathbb{N}$, cet algorithme appelé "RSA" est majoritairement utilisé depuis sa création en 1977 car d'une efficacité remarquable. Il commence cependant a être mis à mal par les attaques de plus en plus puissante dont il fait face, comme par exemple celle imaginée par Wiener (sujet de mon camarade), utilisant le développement en fraction continues des éléments de la clé publique pour trouver celle privée.

J'ai ainsi étudié un autre cryptosystème asymétrique, plus efficace que le RSA, mais encore peu utilisé car imparfait, utilisant des résultats d'algèbre sur les courbes elliptiques tracées dans des corps finis.

2 Algèbre des courbes elliptiques

2.1 Définition d'une courbe elliptique

Après les premières semaines de recherche qui constituèrent la première phase de ce travail, j'avais assez bien appréhendé l'objet mathématique qu'est une courbe elliptique, à savoir un courbe projective plane. Cet aspect, bien qu'hors du programme de CPGE, ce qui a limité nos calculs au début, est nécessaire pour appréhender un point important de la courbe : le point à l'infini, qui peut être assimilé au seul point de la courbe dont la troisième coordonnée est nulle. La courbe est donc définie sur le plan projectif d'un corps, dont la définition est la suivante.

Plan projectif Soit $\mathbb{K}$ un corps , on appelle plan projectif de $\mathbb{K}$ l'ensemble

$$\mathbb{P}(\mathbb{K}) = \frac{\mathbb{K}^3 \setminus \{(0,0,0)\}}{\sim} \tag{1}$$

Avec $\sim$ la relation de proportionnalité, c'est-à-dire $\forall \left((x_1,y_1,z_1),(x_2,y_2,z_2)\right) \in \left(\mathbb{K}^3\right)^2$,

$$(x_1,y_1,z_1) \sim (x_2,y_2,z_2) \;\Leftrightarrow\; \exists\, \lambda \in \mathbb{K} :\; (x_2,y_2,z_2) = \lambda(x_1,y_1,z_1)$$

On note $[x,y,z],\ (x,y,z) \in \mathbb{K}^3$ les éléments du plan projectif

Cet ensemble s'identifie par isomorphisme à l'ensemble des droites vectorielles de l'espace $\mathbb{K}^3$ ce qui nous ramène à un contexte beaucoup plus abordable et permettant un bonne visualisation des objets étudiés.

NB : On se place désormais dans un corps algébriquement clos, ie où tout les polynômes sont scindés. Ceci est nécessaire pour un résultat postérieur.

Courbe elliptique Une courbe elliptique sur un corps $\mathbb{K}$, que l'on notera $E(\mathbb{K})$ est un ensemble de points $[x,y,z]$ de $\mathbb{P}(\mathbb{K})$ qui satisfont une équation de la forme

$$z*y^2 = x^3 + a*z^2*x + b*z^3\ ,\quad 4a^3 + 27b^2 \neq 0 \tag{2}$$

,

Comme on peut le voir sur cette visualisation, une courbe elliptique peut être séparée en 2 parties : celle dans le plan $z = 0$ et celle qui intersecte le plan $z = 1$ (ou qui n'est pas strictement incluse dans le plan $z = 0$).

Théorème 1

Soit $\mathbb{K}$ un corps algébriquement clos et $E(\mathbb{K})$ une courbe elliptique sur ce corps. On a alors

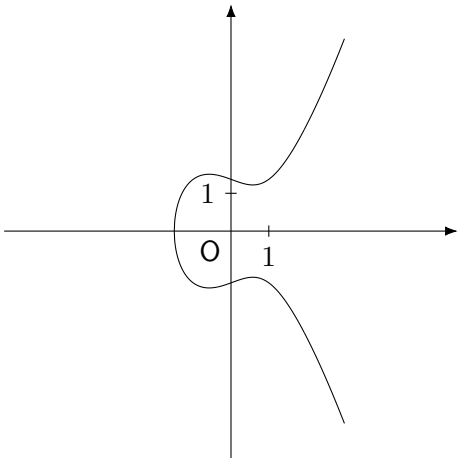
$$E(\mathbb{K}) \cap \left\{[x,y,0] \in \mathbb{P}(\mathbb{K})\right\} = \{O\}$$

où $O = [0,1,0]$ est dit point à l'infini

On note alors $E(\mathbb{K}) = U \cup \{O\}$ où U est la partie de la courbe qui intersecte le plan $z = 1$. Pour simplifier les calculs, on utilise le fait que la partie du plan projectif de $\mathbb{K}$ dont la coordonnée z est non nulle est isomorphe à $\mathbb{K}^2$ (via $[x, y, z] \mapsto (\frac{x}{z}, \frac{y}{z})$). On a alors

$$U = \left\{(x, y) \in \mathbb{K} \mid y^2 = x^3 + ax + b\right\} \tag{3}$$

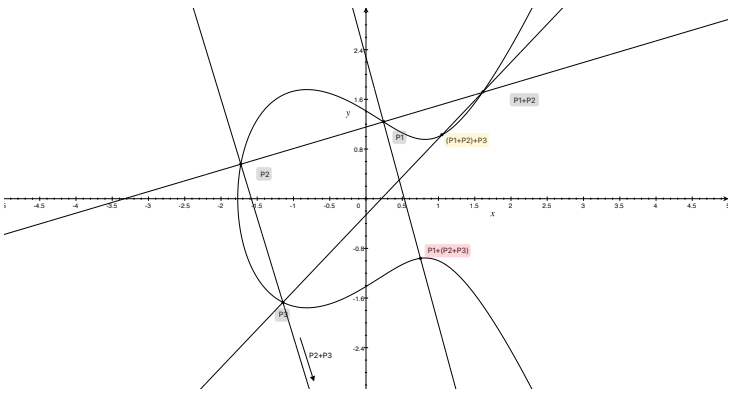
On obtient alors la représentation suivante de la partie affine sur $\mathbb{R}$ (EC1) :



2.2 Loi d'addition et structure de groupe

Ces courbes précédemment définies furent longtemps utilisées dans le calcul d'intégrales elliptiques (d'où elles tirent leur nom), qui sont utilisées pour calculer des aires d'ellipses par exemple, ou encore qui permirent à Andrew Willes de démontrer le "dernier théorème de Fermat", ce qui lui valu plusieurs médailles et distinctions au terme de 7 ans de recherche.

Cependant, en 1985, Neal Koblitz [1] et Victor S. Miller [3] proposent, indépendamment, une nouvelle approche, algébrique, de ces courbes elliptiques. En effet, on peut montrer que toute droite coupant la courbe en deux points distincts la coupe en un troisième, et que toute tangente à un point d'ordonnée non nulle coupe la courbe en un second point. On peut ainsi intuiter une loi de composition interne sur la courbe commutative et de neutre O le point à l'infini. On l'appelle loi des cordes-tangentes et on la note $*$. Elle se visualise très bien géométriquement avec des courbe tracées sur $\mathbb{R}$:



Cette loi peut être formalisée à l'aide des formules suivantes :
Soit $(P, Q) \in E(\mathbb{K})$, on note $P = (x, y)$ et $Q = (x', y')$
• Si $P = O$, alors $P * Q = -Q$ et on a de même symétriquement si $Q = O$

• Si P et Q sont symétriquement opposés, alors $P * Q = O$
• Si $P = Q$ on pose $m = \frac{3x^2+a}{2y}$ et $p = \frac{-x^3+ax+2b}{2y}$; on a alors

$$P * Q = \left(m^2 - 2x \ , \ m(m^2 - 2x) + p\right) \tag{4}$$

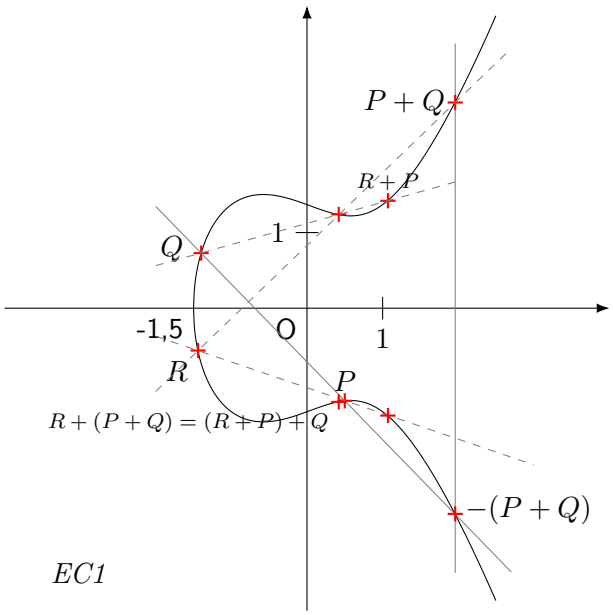
• Si $P \neq Q$ on pose $m = \frac{y-y'}{x-x'}$ et $p = \frac{xy'-x'y}{x-x'}$; on a alors

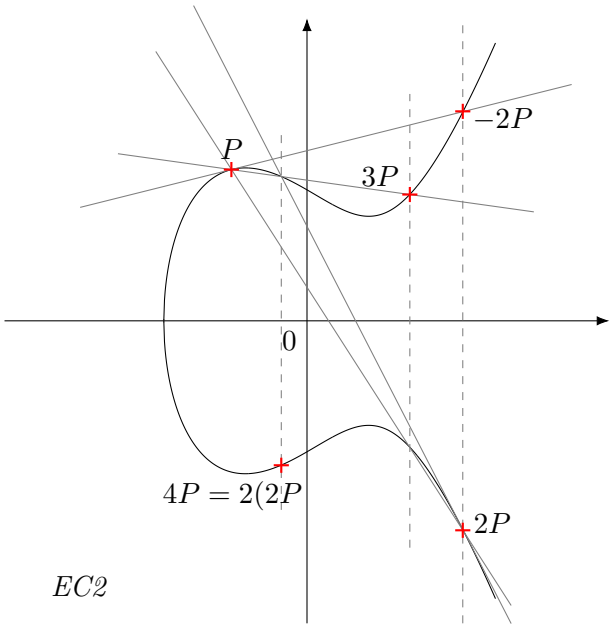
$$P * Q = \left(m^2 - x - x' \ , \ m(m^2 - x - x') + p\right) \tag{5}$$

Cette formalisation permet de définir la loi sur n'importe quel corps algébriquement clos, ce dernier caractère permettant d'assurer l'existence du point obtenu. On a malheureusement une loi qui n'est pas associative, seule caractéristique d'une loi de groupe qui lui manque. On défini alors une loi additive dérivée de la loi des cordes-tangentes comme suit :

Loi de groupe Soit E une courbe elliptique sur un corps $\mathbb{K}$, on muni l'ensemble $E(\mathbb{K})$ d'une loi notée $+$ qui à deux points P et Q de la courbe associe le symétrique par rapport à l'axe des abscisses de $P * Q$, on obtient ainsi :

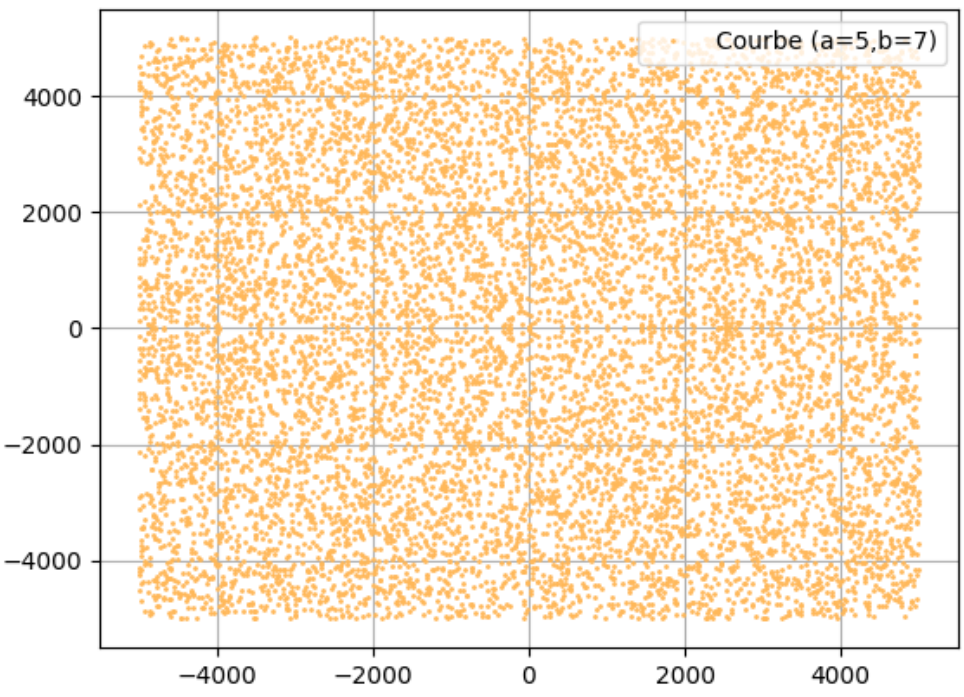
$$+ \left(\begin{array}{ccc} E(\mathbb{K}) \times E(\mathbb{K}) & \to & E(\mathbb{K}) \\ (P, Q) & \mapsto & P + Q \ = \ (P * Q) * O \end{array} \right) \tag{6}$$





2.3 Cas d'un corps fini

On se place désormais dans $\mathbb{K}$ un corps fini algébriquement clos. En pratique, on se limitera ici aux corps isomorphes à $\mathbb{F}_p$ ou p est un nombre premier (avec du temps supplémentaire, nous aurions souhaité traiter des exemple sur des extensions de type $\mathbb{F}_q$ où $q = p^n, n \in \mathbb{N}$).
On notera q l'ordre de $\mathbb{K}$. Dans ce contexte, les courbes elliptiques sont également finies (d'ordre naturellement inférieur à $q^2 + 1$) et bien que l'on perde le caractère géométrique des lois d'addition, elles restent vérifiées avec leurs définitions par le calcul. (ici EC3 sur $\mathbb{F}_{10007}$)



On a le théorème de structure suivant sur le groupe $\big(E(\mathbb{K}),+\big)$:

Théorème 2

Soit $\mathbb{K}$ un corps fini de cardinal q et E une courbe elliptique
Alors $E(\mathbb{K}) \simeq (\mathbb{Z}/n_1\mathbb{Z}) \times (\mathbb{Z}/n_2\mathbb{Z})$
Avec $n_1|n_2$ et $n_1|(q-1)$

Ce théorème est admis

3 Application au cryptosystème d'ElGamal

Pour répondre au besoin croissant d'algorithmes de cryptage, Taher ElGamal publie en 1984 dans [4] un modèle de cryptosystème asymétrique fondé sur le problème du logarithme discret dans un groupe fini. Ce problème, analogue au logarithme népérien sur $\mathbb{R}$ revient à, étant donnée deux élément x et y d'un groupe fini, de trouver (en supposant qu'il existe) un entier $n \in \mathbb{N}$ tel que $y = x^n$. Ce problème se décline très bien dans le groupe d'une courbe elliptique, avec la loi d'addition. Il est aujourd'hui irrésoluble en temps raisonnable sur certains groupes, et en particulier sur certaines courbes elliptiques, ce qui sert de base à plusieurs algorithmes de cybersécurité (par exemple Ed25519 avec l'algorithme ECDSA, algorithme de signature numérique non étudié). Nous avons donc conçu un algorithme de cryptage utilisant le système d'ElGamal sur un groupe elliptique.

Principe de l'algorithme On appellera traditionnellement Alice et Bob les utilisateurs du système (Alice possédant la clé privée). Le principe est le suivant :

• Alice choisit un groupe G cyclique et un générateur g de ce groupe, puis elle choisit aléatoirement un entier naturel a plus petit que l'ordre de G, c'est la clé privée, elle publie alors la clé publique suivante

$$C = (G, g, h) \; avec \; h = g^a \tag{7}$$

• Bob s'il veut envoyer un message $m \in G$ (se traduisant dans le langage souhaité à l'aide d'une fonction de hachage) choisit un entier naturel k plus petit que $|G|$ et envoie le couple

$$(g^k, m.h^k) \tag{8}$$

• Alice, pour retrouver m, multiplie le deuxième terme par l'inverse du premier, mis à la puissance a et retrouve ainsi m car $(g^k)^a = (g^a)^k = h^k$

Implémentation avec les courbes elliptiques Nous avons implémenté [1]des algorithmes de calcul sur ces courbes elliptique, mais avons rapidement été limités par l'impossibilité, avec nos connaissance et le temps imparti, de s'affranchir de la nécessité de calculer l'intégralité de la courbe, ce qui nous a restraint à des calculs sur des corps d'ordre environ 1 000 000 (en pratique on se place dans des corps d'ordre supérieur à 10^{100}

1. Le code python est originalement créé par moi-même, mais la structure des courbes et points est largement inspirée de [6]

4 Bibilographie

[1] Neal Koblitz : Elliptic Curve Cryptosystem :
Mathematics of computation n°48 p.203-209
[2] Alain Kraus : Cours de C cryptographie - Chapitre VII - Courbes elliptiques
[3] Victor S. Miller : Use of Elliptic Curves in Cryptography
Advance in cryptology p. 217-226
[4] Taher ElGamal : A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms *: Advances in cryptology p.10-18*
[5] Wikipédia : Cryptographie sur les courbes elliptiques
[6] Github - DockBlack89 : Courbe elliptique

5 Annexes

- EC1 : $y^2 = x^3 - x + 1.875$
- EC2 : $y^2 = x^3 - 2x + 3$

ECC tools

```
### Classes point et courbe ###


class Point:

    ''' Classe des point de la courbe sauf le point infini '''

    def __init__(self,x,y):
        self.x = x
        self.y = y

    def isInf(self):
        return False

    def co(self):       #Coordonnées du point
        if self.isInf():
            return None
        else:
            return (self.x,self.y)


class PointInf:

    ''' Classe point à l'infini '''

    def __init__(self):
        Point.__init__(self,None,None)

    def isInf(self):
        return True

    def co(self):       #Coordonnées du point
        if self.isInf():
            return None
        else:
            return (self.x,self.y)
```

```python
class Courbe:

    ''' Classe de la courbe '''

    def __init__(self,a,b,p):

        ''' Vérification des données '''
        assert a>0
        assert b>0
        assert p>2

        self.a = a
        self.b = b
        self.prem = p #Courbe sur Z/pZ

    ''' Fonction vérification d'appartenance '''
    def isOn(self,p):
        if p.isInf():
            return True
        else:
            return ( p.y**2 % self.prem ) == ((p.x**3+self.a*p.x+self.b) %self.prem)

    def egal(self,P,Q):
        if P.isInf():
            return Q.isInf()
        elif Q.isInf():
            return P.isInf()
        else:
            (px,py),(qx,qy) = P.co(),Q.co()
            return px %self.prem==qx %self.prem and py %self.prem==qy %self.prem



def copy(p):
    if p.isInf():
        return PointInf()
    else:
        return Point(p.x,p.y)


### Nombre centré ###



def mod(x,p):
    ''' Renvoie le représentant de la classe de x
    dans Z/pZ centré sur 0 '''
    y=x%p
    p2=p//2
    if y<=p2:
        return y
    else:
        return y-p
```

```
### Exponentiation modulaire rapide ###


def fme(x,y,n):
    res = 1
    while y>0 :
        if ((y&1)!=0) :
            res = (res * x) % n
        y= y >> 1
        x = x*x %n
    return mod(res,n)


### Tri fusion pour des couples ###


def fusion_simple(l1,l2):
    res=[]
    i1,i2 = 0,0
    t1,t2 = len(l1),len(l2)
    while i1<t1 and i2<t2:
        if l1[i1]<l2[i2] :
            res.append(l1[i1])
            i1 += 1
        else:
            res.append(l2[i2])
            i2 += 1
    if i1<t1:
        res = res + l1[i1::]
    else:
        res = res + l2[i2::]
    return res


def ajoute(l,x):
    if l==[] or x[1] != l[-1][1]:
        l.append(x)
    else:
        l[-1] = (fusion_simple(l[-1][0],x[0]),x[1])


def partition(l):
    n=len(l)
    return (l[:n//2],l[n//2::])
```

```
def fusion(l1,l2):
    res=[]
    i1,i2 = 0,0
    t1,t2 = len(l1),len(l2)
    while i1<t1 and i2<t2:
        if l1[i1][1]<l2[i2][1] :
            ajoute(res,l1[i1])
            i1 += 1
        else:
            ajoute(res,l2[i2])
            i2 += 1
    while i1<t1:
        ajoute(res,l1[i1])
        i1 += 1
    while i2<t2:
        ajoute(res,l2[i2])
        i2 += 1
    return res



def tri_fus(liste):
    if len(liste) <= 1:
        return liste
    else:
        g,d = partition(liste)
        return fusion(tri_fus(g),tri_fus(d))
```

ECC courbe

```
from ECC_tools import *



def XY_E(C):

    p=C.prem  #p premier
    p2=p//2

    ''' Création de la liste (y,y^2) '''
    Y = [([0],0)]
    for i in range(1,p2+1):
        Y.append(([-i,i],fme(i,2,p)))

    ''' Creation de la liste (x,x^3+ax+b) '''
    X=[]
    for i in range(-p2,p2+1):
        a=(fme(i,3,p)+C.a*i+C.b)%p
        if a>p2:
            a=(a-p)
        X.append(([i],a))

    ''' Renvoie des listes triées selon le deuxième élément
    du couple et rassemblée par liste des x dont le second
    élément est identique '''
    return (tri_fus(X),tri_fus(Y))
```

```python
def ajoute_permut(liste,l1,l2):

    a,b = len(l1),len(l2)
    for i in range(a):
        for j in range(b):
            liste.append((l1[i],l2[j]))



def EC(C):

    ''' Création de la courbe elliptique '''
    X,Y = XY_E(C)
    supp = [] #Liste des couples (x,y) ATTENTION ce n'est pas la courbe
    ix,iy = 0,0
    tx,ty = len(X),len(Y)

    while ix<tx and iy<ty: # Parcours des listes triées linéaire
        if X[ix][1]==Y[iy][1]:
            ''' On ajoute tout les couples (x,y) tels que y^2=x^3+ax+b '''
            ajoute_permut(supp,X[ix][0],Y[iy][0])
            ix += 1
            iy += 1
        elif X[ix][1]<Y[iy][1]:
            ix += 1
        else:
            iy += 1

    res = [PointInf()] # Initialisation de la courbe
    #file = open("courbe.txt","w")
    for e in supp:
        res.append(Point(e[0],e[1]))
    #    file.write(str(e) + "\n")
    #file.close()
    return res

def liste(nom):
    lignes = [PointInf()]
    with open(nom,"r",encoding="utf-8") as f:
        for ligne in f:
            ligne = ligne.rstrip()
            a,b = tuple(map(int, ligne[1:-1].split(',')))
            lignes.append(Point(a,b))
        return lignes
```

ECC calcul

```
from ECC_tools import *

### Calcul de l'inverse ###


def inv_mod(x,p):
    if x==0:
        return None
    else:
        return fme(x,p-2,p)


### Somme dans E(K) ###


def somme(P,Q,C):

    ''' Vérification des données '''
    assert C.isOn(P)
    assert C.isOn(Q)

    if P.isInf():   #Cas triviaux
        return Q
    elif Q.isInf():
        return P

    else:           #Cas généraux

        n = C.prem
        ((xp,yp),(xq,yq)) = (P.co(),Q.co())

        if xp != xq:
            d=inv_mod(xp-xq,n)
            l = mod((yp-yq)*d,n)
            nu = mod((xp*yq-xq*yp)*d,n)
            x1 = mod(l**2-xp-xq,n)
            y1 = mod(-l*x1-nu,n)
            return Point(x1,y1)

        elif yp != yq:
            return PointInf()

        else:

            if yp==0:
                return PointInf()

            else:
                d = inv_mod(2*yp,n)
                l = (3*xp**2+C.a)*d
                nu = (-(xp**3)+C.a*xp+2*C.b)*d
                x1 = mod(l**2 - 2*xp,n)
                y1 = mod(-l*x1-nu,n)
                return Point(x1,y1)
```

```
### Multiplication dans E(K) ###


def mult(n,P,C):

    assert C.isOn(P)

    ''' Calcul de nP dans C par exponentiation rapide '''
    Q = copy(P)
    R = PointInf()

    while n>0:
        if ((n&1)==1):
            R = somme(R,Q,C)
        n = n>>1
        Q=somme(Q,Q,C)

    return R


### Inverse dans E(K) ###


def inverse(P,C):

    assert C.isOn(P)

    (x,y) = P.co()
    return Point(x,-y)
```

ECC générateur

```
from ECC_tools import *
from ECC_courbe import *
from ECC_calcul import *


def diviseurs(n:int) -> list:
    res = []
    for i in range(1,n+1):
        if n==(n//i)*i:
            res.append(i)
    return res

def ordre(P,Q,liste_d,C):
    if Q.isInf():
        for e in liste_d:
            if mult(e,P,C).isInf():
                return e
    else:
        for e in liste_d:
            if C.egal(mult(e,P,C),Q):
                return e
        return -1
```

```
def generateur(E,C):
    p=len(E)
    R=PointInf()
    j=1
    liste_d = diviseurs(p)
    x=liste_d[len(liste_d)//2]
    for P in E:
        o = ordre(P,R,liste_d,C)*j
        if o==p:
            return P
        elif o>x:
            R=P
            j=o
            liste_d=diviseurs(o)
            x=liste_d[len(liste_d)//2]


def generateur2(E,C):
    p=len(E)
    liste_d = diviseurs(p)[:-1]
    for P in E:
        flag = True
        for e in liste_d:
            if mult(e,P,C).isInf():
                flag = False[
                break
        if flag:
            return P

def generateur3(E,C):
    p=len(E)
    liste_d = diviseurs(p)[:-1]
    dic = {}
    dic[PointInf()]=0
    for P in E:
        if not(P in dic):
            flag = True
            for e in liste_d:
                if mult(e,P,C).isInf():
                    flag = False
                    Q=copy(P)
                    for i in range(e-1):
                        dic[Q]=0
                        Q=somme(P,Q,C)
            if flag:
                return P
    return PointInf()
```

ECC

```
from random import randint
from ECC_calcul import *
from ECC_tools import *
from ECC_calcul import *
from ECC_generateur import *


### Algorithme de cryptage ###


def cryptage(P,cle):
    (C,E,q,g,h) = cle
    a = randint(1,q)
    c1 = mult(a,g,C)
    c2 = somme(P,mult(a,h,C),C)
    return (c1,c2)


### Algorithme de décryptage ###


def decryptage(m,cle,cle_prive):
    (C,E,q,g,h) = cle
    (c1,c2) = m
    return somme(c2,inverse(mult(cle_prive,c1,C),C),C)
```

ECC illustr

```
import matplotlib.pyplot as plt
from ECC_tools import *
from ECC_courbe import *


def illustr(C):

    E = EC(C)
    E1 = [e.x for e in E[1::]] #Liste des abscisses
    E2 = [e.y for e in E[1::]] #Liste des ordonnées

    ''' Création du graphe '''
    premier = "p="+str(C.prem)
    plt.title(premier)

    p2=C.prem//2
    a=2
    plt.axis([-a*p2,a*p2,-a*p2,a*p2])

    plt.clf()
    plt.plot(E1,E2,'.',markersize=2, color='#ffb859',
             label='Courbe (a=' + str(C.a) + ',b=' +str(C.b) +')')
    plt.grid()
    plt.legend()
    plt.show()
```