

Eisenstein

January 17, 2024

1 L’anneau des entiers d’Eisenstein

Marc Lorenzi

18 décembre 2023

```
[1]: import matplotlib.pyplot as plt
import random
import math
import shanks
```

1.1 1. Un sous-anneau de $\mathbb{C}$

1.1.1 1.1 Introduction

On note j une des deux racines cubiques non réelles de 1. L’ensemble

$$\mathbb{Z}[j] = \{x + jy : (x, y) \in \mathbb{Z}^2\}$$

est un sous-anneau de $\mathbb{C}$. Les éléments de $\mathbb{Z}[j]$ sont les *entiers d’Eisenstein*.

On représente les entiers d’Eisenstein en Python par des objets de la classe `Zj`. Pour tous $x, y, x', y' \in \mathbb{Z}$,

$$\begin{aligned}(x + jy) \pm (x' + jy') &= (x \pm x') + j(y \pm y') \\ (x + jy) \times (x' + jy') &= (xx' - yy') + j(xy' + x'y - yy')\end{aligned}$$

On en déduit le code des opérations élémentaires.

On a $\mathbb{Z} \subset \mathbb{Z}[j]$. La classe `Zj` en tient compte : on peut opérer entre des entiers (de type `int`) et des objets de la classe `Zj`.

Nous verrons que l’anneau $\mathbb{Z}[j]$ peut être muni d’une division euclidienne : la méthode `__floordiv__` fait appel à une fonction `quotient` que nous écrirons plus tard.

Nous parlerons un peu plus loin des méthodes `conj`, `trace` et `norme`.

La méthode la plus compliquée de la classe `Zj` est la méthode `__repr__` qui permet à un objet de cette classe de s'afficher correctement. Il faut tenir compte de plusieurs facteurs (nullité ou pas des composantes, signes, composante égale à 1).

```
[2]: class Zj:

    def __init__(self, x, y=0):
        self.x = x
        self.y = y

    def __hash__(self):
        return hash((self.x, self.y))

    def __repr__(self):
        if self.x == 0:
            if self.y == 0: return '0'
            elif self.y == -1: return '-j'
            elif self.y == 1: return 'j'
            else: return str(self.y) + 'j'
        else:
            s = str(self.x)
            if self.y == 0: return s
            elif self.y < 0:
                if self.y != -1: return s + str(self.y) + 'j'
                else: return s + '-j'
            else:
                if self.y != 1: return s + '+' + str(self.y) + 'j'
                else: return s + '+j'

    def __eq__(self, b):
        if isinstance(b, int): b = Zj(b)
        return self.x == b.x and self.y == b.y

    def __req__(self, b): return self == b

    def __add__(self, b):
        if isinstance(b, int): b = Zj(b)
        x1, y1 = self.x, self.y
        x2, y2 = b.x, b.y
        return Zj(x1 + x2, y1 + y2)

    def __radd__(self, b): return self + b

    def __neg__(self): return Zj(-self.x, -self.y)

    def __sub__(self, b): return self + (-b)
    def __rsub__(self, b): return b + (-self)
```

```

def __mul__(self, b):
    if isinstance(b, int): b = Zj(b)
    x1, y1 = self.x, self.y
    x2, y2 = b.x, b.y
    x = x1 * x2 - y1 * y2
    y = x1 * y2 + x2 * y1 - y1 * y2
    return Zj(x, y)

def __rmul__(self, b): return self * b

def __floordiv__(self, b):
    if isinstance(b, int): b = Zj(b)
    return quotient(self, b)

def __rfloordiv__(self, b):
    b = Zj(b)
    return b // self

def __mod__(self, b):
    if isinstance(b, int): b = Zj(b)
    return self - (self // b) * b

def __rmod__(self, b):
    b = Zj(b)
    return b % self

def __pow__(self, n):
    if n < 0: raise Exception('Not implemented')
    else:
        z = Zj(1)
        y = Zj(self.x, self.y)
        while n != 0:
            if n % 2 != 0: z = z * y
            n = n // 2
            y = y * y
        return z

def conj(self):
    return Zj(self.x - self.y, -self.y)

def trace(self):
    return (self + self.conj()).x

def norme(self):
    return (self * self.conj()).x

```

Définissons la variable globale J.

```
[3]: J = Zj(0, 1)
     print(J)
```

j

On a $j^2 + j + 1 = 0$ et $j^3 = 1$.

```
[4]: print(J ** 2 + J + 1, J ** 3)
```

0 1

La méthode `__pow__` utilise une exponentiation rapide. On peut donc calculer instantanément de très grandes puissances.

```
[5]: print(J ** (10 ** 10))
```

j

1.1.2 1.2 Une représentation graphique

Dessignons quelques points de l'anneau $\mathbb{Z}[j]$. La fonction `vers_complexe` prend en paramètre un entier d'Eisenstein $a = x + jy$ où $x, y \in \mathbb{Z}$. On prend ici $j = -\frac{1}{2} + i\frac{\sqrt{3}}{2}$. La fonction renvoie le couple de flottants $(x - \frac{y}{2}, y\frac{\sqrt{3}}{2})$.

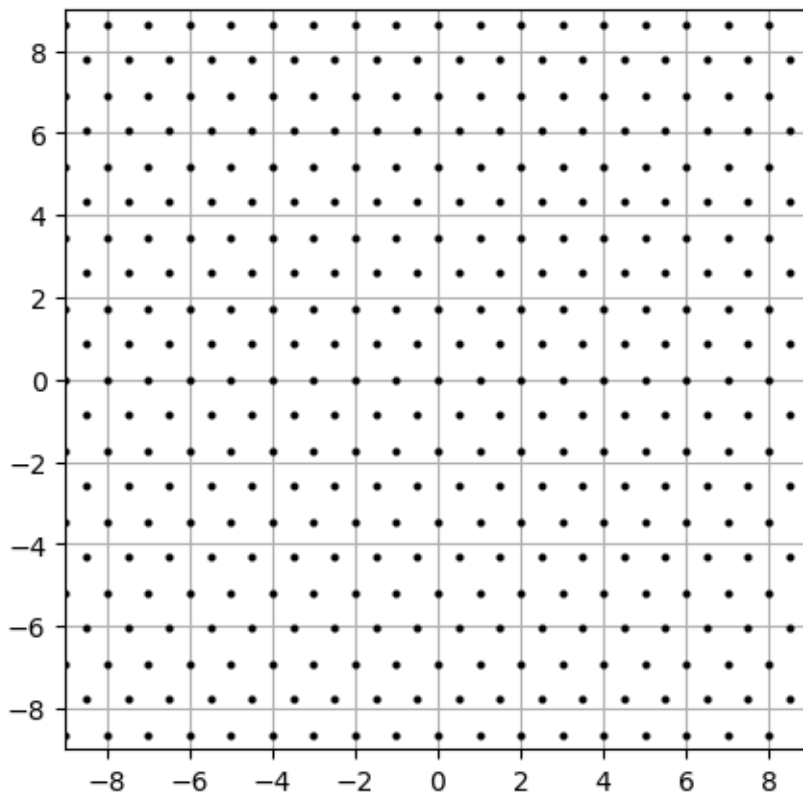
```
[6]: def vers_complexe(a):
     x = a.x - a.y / 2
     y = a.y * math.sqrt(3) / 2
     return (x, y)
```

La fonction `tracer_Zj` dessine les éléments de $\mathbb{Z}[j]$ dont les parties réelle et imaginaire appartiennent à $[-N, N]$.

```
[7]: def tracer_Zj(N):
     N2 = int(1.6 * N)
     xs, ys = [], []
     for x in range(-N2, N2 + 1):
         for y in range(-N2, N2 + 1):
             u, v = vers_complexe(Zj(x, y))
             if abs(u) <= N and abs(v) <= N:
                 xs.append(u)
                 ys.append(v)
     plt.xlim(-N, N)
     plt.ylim(-N, N)
     plt.plot(xs, ys, 'ok', ms=2)
     plt.grid()
```

```
[8]: plt.rcParams['figure.figsize'] = (5, 5)
```

```
[9]: tracer_Zj(9)
plt.savefig('figures/Zj.png', bbox_inches='tight')
```



1.1.3 1.3 Conjugué, trace, norme

Soit $a = x + jy \in \mathbb{Z}[j]$ où $x, y \in \mathbb{Z}$. Remarquons que $\bar{j} = j^2 = -j - 1$. On a donc

$$\bar{a} = (x - y) - jy \in \mathbb{Z}[j]$$

La classe $\mathbb{Z}j$ définit la méthode `conj` qui renvoie le conjugué d'un entier d'Eisenstein. Voici le conjugué de j .

```
[10]: print(J.conj())
```

-1-j

Définition. Pour tout $a \in \mathbb{Z}[j]$,

- La *trace* de a est $Tr(a) = a + \bar{a}$.
- La *norme* de a est $N(a) = a \times \bar{a}$.

On a $Tr(a) \in \mathbb{Z}$ et $N(a) = |a|^2 \in \mathbb{N}$.

Proposition. Pour tout $a \in \mathbb{Z}[j]$,

$$a^2 - Tr(a)a + N(a) = 0$$

Démonstration. Soit $a \in \mathbb{Z}[j]$.

$$a^2 - Tr(a)a + N(a) = a^2 - (a + \bar{a})a + a\bar{a} = 0$$

Ainsi, tout élément de $\mathbb{Z}[j]$ est racine d'une équation de degré 2 du type $x^2 - Sx + P = 0$ où $S, P \in \mathbb{Z}$.

La classe `Zj` définit les méthodes `trace` et `norme`.

Voici la trace et la norme de j et $\bar{j}$.

```
[11]: print(J.trace(), J.norme())
      print(J.conj().trace(), J.conj().norme())
```

```
-1 1
```

```
-1 1
```

Effectivement, j et $\bar{j}$ sont les deux racines de l'équation $a^2 + a + 1 = 0$.

1.1.4 1.4 Éléments « aléatoires » de $\mathbb{Z}[j]$

La fonction `random_Zj` renvoie un élément $a = x + jy$ de $\mathbb{Z}[j]$ où x et y sont deux entiers aléatoires appartenant à $[-N, N]$.

```
[12]: def random_Zj(N):
      x = random.randint(-N, N + 1)
      y = random.randint(-N, N + 1)
      return Zj(x, y)
```

```
[13]: print(random_Zj(10 ** 10))
```

```
6252262384+1956867042j
```

1.1.5 1.5 Les inversibles de $\mathbb{Z}[j]$

On montre facilement que pour tout $a \in \mathbb{Z}[j]$, a est inversible si et seulement si $N(a) = 1$. En posant $a = x + jy$ où $x, y \in \mathbb{Z}$, ceci équivaut à

$$x^2 - xy + y^2 = 1$$

ou encore

$$\left(x - \frac{y}{2}\right)^2 + \frac{3}{4}y^2 = 1$$

Si a est inversible, alors $\frac{3}{4}y^2 \leq 1$ donc, comme $y \in \mathbb{Z}$, $y = -1, 0$ ou 1 .

- Si $y = -1$ alors $(x + \frac{1}{2})^2 = \frac{1}{4}$, donc $x = -\frac{1}{2} \pm \frac{1}{2}$. Ainsi, $x = 0$ ou $x = -1$ et donc $a = -j$ ou $a = -1 - j = -j^2$.
- Si $y = 1$ alors $(x - \frac{1}{2})^2 = \frac{1}{4}$, donc $x = \frac{1}{2} \pm \frac{1}{2}$. Ainsi, $x = 0$ ou $x = 1$ et donc $a = j$ ou $a = 1 + j = -j^2$.
- Si $y = 0$ alors $x^2 = 1$, donc $x = \pm 1$. Ainsi, $a = 1$ ou $a = -1$.

Inversement, les 6 nombres que nous venons de mettre en évidence sont inversibles : ce sont les racines sixièmes de l'unité. Le groupe des inversibles de l'anneau $\mathbb{Z}[j]$ est donc le groupe

$$\mathcal{U}_6 = \{\pm 1, \pm j, \pm j^2\}$$

```
[14]: U6 = [1 + 0 * J, -J ** 2, J, -1 + 0 * J, J ** 2, -J]
      print(U6)
```

```
[1, 1+j, j, -1, -1-j, -j]
```

Dessignons $\mathcal{U}_6$.

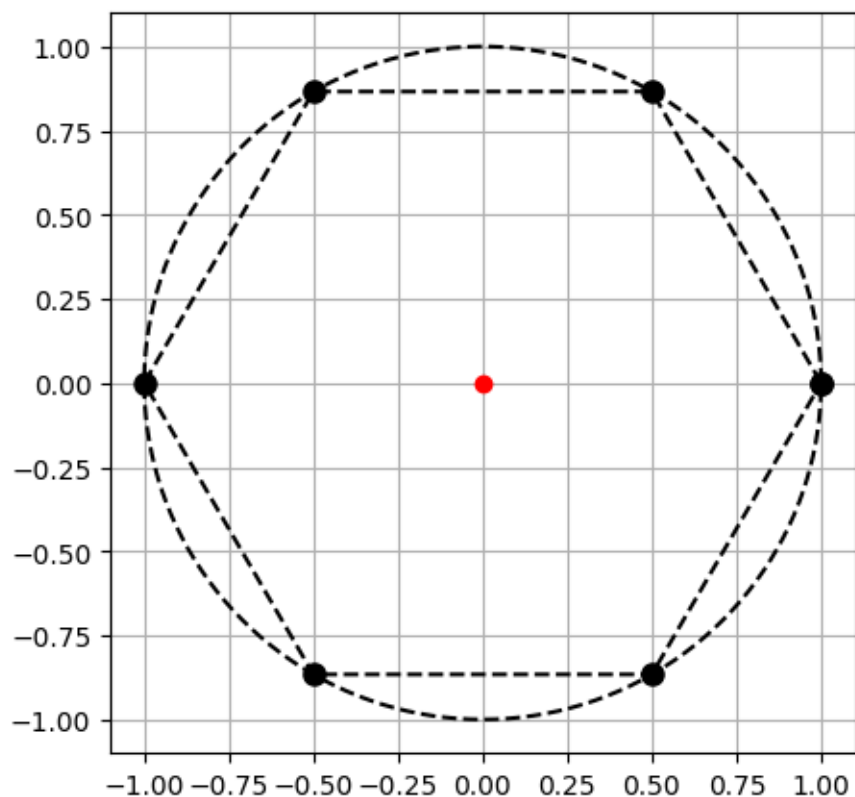
La fonction `tracer_cercle` trace le cercle de centre O et de rayon r . On peut passer à cette fonction des paramètres supplémentaires à destination de la fonction `pyplot.plot`.

```
[15]: def tracer_cercle(r, **opt):
      xs = [r * math.cos(2 * k * math.pi / 1000) for k in range(1001)]
      ys = [r * math.sin(2 * k * math.pi / 1000) for k in range(1001)]
      plt.plot(xs, ys, color='k', **opt)
```

```
[16]: def tracer_inversibles():
      zs = [vers_complexe(u) for u in U6]
      xs = [x for (x, y) in zs]
      ys = [y for (x, y) in zs]
      xs.append(zs[0][0])
      ys.append(zs[0][1])
      plt.plot(xs, ys, '--ok', ms=8)
      tracer_cercle(1, ls='--')
      plt.plot([0], [0], 'or', ms=6)
      plt.grid()
```

```
[17]: plt.rcParams['figure.figsize'] = (5, 5)
```

```
[18]: tracer_inversibles()
      plt.savefig('figures/inversibles.png', bbox_inches='tight')
```



1.2 2. Division euclidienne

1.2.1 2.1 Divisibilité

Définition. Soient $a, b \in \mathbb{Z}[j]$ a divise b s'il existe $c \in \mathbb{Z}[j]$ tel que $b = ac$. Nous noterons $a \mid b$.

La relation $\mid$ est réflexive et transitive. Si $a, b \in \mathbb{Z}[j]$ on a $a \mid b$ et $b \mid a$ si et seulement si il existe $u \in \mathcal{U}_6$ tel que $b = au$. Nous dirons que a et b sont *associés*.

La fonction `associes` prend en paramètre un entier d'Eisenstein a . Elle renvoie la liste des associés de a .

```
[19]: def associes(a):
      return [a * u for u in U6]
```

```
[20]: a = random_Zj(1000)
      print(a)
      print(associes(a))
```

```
322+491j
[322+491j, -169+322j, -491-169j, -322-491j, 169-322j, 491+169j]
```


Les associés de a sont les sommets d'un hexagone régulier. Au moins un de ces sommets est de la forme $x + jy$ où x et y sont positifs. La fonction `associe_privilegie` renvoie celui qui a la plus petite partie imaginaire.

```
[21]: def associe_privilegie(a):
    bs = []
    for u in U6:
        b = a * u
        if b.x >= 0 and b.y >= 0: bs.append(b)
    b0 = bs[0]
    for b in bs:
        if b.y < b0.y: b0 = b
    return b0
```

```
[22]: a = random_Zj(1000)
print(a)
print(associes(a))
print(associe_privilegie(a))
```

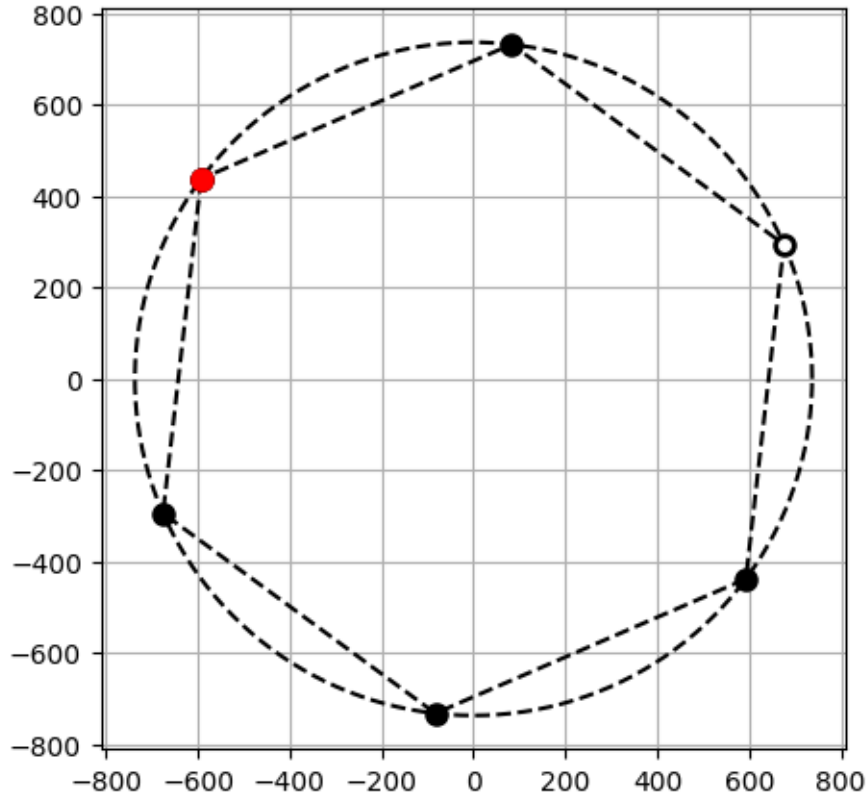
```
698-129j
[698-129j, 827+698j, 129+827j, -698+129j, -827-698j, -129-827j]
827+698j
```

La fonction `tracer_associes` dessine les associés de a . Elle dessine le point a en rouge et l'associé privilégié de a en blanc.

```
[23]: def tracer_associes(a):
    zs = [vers_complexe(b) for b in associes(a)]
    xs = [x for (x, y) in zs]
    ys = [y for (x, y) in zs]
    xs.append(zs[0][0])
    ys.append(zs[0][1])
    plt.plot(xs, ys, '--ok', ms=8)
    tracer_cercle(math.sqrt(a.norme()), ls='--')
    u, v = vers_complexe(a)
    plt.plot([u], [v], 'or', ms=8)
    b = associe_privilegie(a)
    u, v = vers_complexe(b)
    plt.plot([u], [v], 'ow', ms=4)
    plt.grid()
```

```
[24]: a = random_Zj(1000)
print(a)
tracer_associes(a)
```

```
-340+506j
```



1.2.2 2.2 Une division euclidienne dans $\mathbb{Z}[j]$

Soient $a, b \in \mathbb{Z}[j]$. Supposons $b \neq 0$. Soit $z = a/b = x + jy$, où $x, y \in \mathbb{Q}$. Soient $\alpha, \beta \in \mathbb{Z}$ tels que $|x - \alpha| \leq 1/2$ et $|y - \beta| \leq 1/2$. Soit $q = \alpha + j\beta \in \mathbb{Z}[j]$. On a alors

$$N(z - q) = |z - q|^2 = (x - \alpha)^2 + (y - \beta)^2 - (x - \alpha)(y - \beta) \leq \frac{3}{4}$$

De là, en posant $r = a - bq$,

$$N(r) = N(b)N\left(\frac{a}{b} - q\right) \leq \frac{3}{4}N(b)$$

La fonction `approx_int` prend en paramètres deux entiers x et y . Elle renvoie l'entier $n = \lfloor x/y + 1/2 \rfloor$. On a

$$\left| \frac{x}{y} - n \right| \leq \frac{1}{2}$$

```
[25]: def approx_int(x, y):
      return (2 * x + y) // (2 * y)
```

```
[26]: print(approx_int(24, 7), approx_int(25, 7))
```

3 4

La fonction `quotient` prend en paramètres deux entiers d'Eisenstein a et b . Elle renvoie un quotient de la division euclidienne de a par b . Pour déterminer ce quotient on utilise

$$\frac{a}{b} = \frac{a\bar{b}}{N(b)}$$

```
[27]: def quotient(a, b):
      c = a * b.conj()
      N = b.norme()
      return Zj(approx_int(c.x, N), approx_int(c.y, N))
```

Nous pouvons maintenant écrire `a // b` pour calculer un quotient. La classe `Zj` définit à partir de là `a % b`. On a $a = (a // b) b + (a \% b)$.

```
[28]: a = 59 + 43 * J
      b = 28 + 51 * J
      q = a // b
      r = a % b
      print(q, r)
      print(b * q + r)
      print(b.norme(), r.norme())
```

```
-j 8+20j
59+43j
1957 304
```

1.2.3 2.3 Pgcd de deux entiers d'Eisenstein

L'anneau $\mathbb{Z}[j]$ est un *anneau euclidien*. Il est donc aussi *principal* : ses idéaux sont les ensembles de la forme $a\mathbb{Z}[j]$ où $a \in \mathbb{Z}[j]$.

Étant donnés $a, b \in \mathbb{Z}[j]$, un pgcd de a et b est un entier d'Eisenstein δ tel que

$$a\mathbb{Z}[j] + b\mathbb{Z}[j] = \delta\mathbb{Z}[j]$$

Si $(a, b) \neq (0, 0)$, a et b possèdent 6 pgcd qui sont associés.

L'algorithme d'Euclide permet de calculer des pgcd. Étant donnés deux entiers d'Eisenstein a et b , on définit par récurrence une suite $(r_n)_{n \geq 0}$ d'éléments de $\mathbb{Z}[j]$ en posant $r_0 = a$, $r_1 = b$, puis, pour tout $n \in \mathbb{N}$,

- si $r_{n+1} \neq 0$ alors r_{n+2} est « le » reste de la division euclidienne de r_n par r_{n+1} .
- sinon $r_{n+2} = 0$.

Il existe un entier $n_0 \geq 1$ tel que $r_{n_0} = 0$. Un pgcd de a et b est alors r_{n_0-1} .

La fonction `pgcd` fait le travail.

```
[29]: def pgcd(a, b):
      while b != 0:
          a, b = b, a % b
      return associe_privilegie(a)
```

```
[30]: a = 59 + 43 * J
      b = 28 + 51 * J
      print(pgcd(a, b))
```

5+3j

Pour obtenir les détails du calculs du pgcd, voici une fonction `pgcd_dbg`.

```
[31]: def pgcd_dbg(a, b):
      cpt = 2
      print('%3s%10s%10s%10s' % ('n', 'q', 'r', 'N(r)'))
      print('%3d%10s%10s%10d' % (0, '', a, a.norme()))
      print('%3d%10s%10s%10d' % (1, '', b, b.norme()))
      while b != 0:
          q = a // b
          a, b = b, a % b
          print('%3d%10s%10s%10d' % (cpt, q, b, b.norme()))
          cpt += 1
```

```
[32]: a = 59 + 43 * J
      b = 28 + 51 * J
      pgcd_dbg(a, b)
```

n	q	r	N(r)
0		59+43j	2793
1		28+51j	1957
2	-j	8+20j	304
3	2	12+11j	133
4	2+j	-5-3j	19
5	-3-j	0	0

Si nous voulons *tous* les pgcd de a et b , rien n'est plus facile.

```
[33]: a = 59 + 43 * J
      b = 28 + 51 * J
      d = pgcd(a, b)
      print([d * u for u in U6])
```

[5+3j, 2+5j, -3+2j, -5-3j, -2-5j, 3-2j]

La fonction `restes_euclide` prend en paramètres deux entiers d'Eisenstein a et b . Elle renvoie la liste des restes de l'algorithme d'Euclide, y compris le premier reste nul.

```
[34]: def restos_euclide(a, b):  
    rs = [a, b]  
    while rs[-1] != 0:  
        rs.append(rs[-2] % rs[-1])  
    return rs
```

```
[35]: a = 59 + 43 * J  
b = 28 + 51 * J  
rs = restos_euclide(a, b)  
for r in rs:  
    print('%-10s%10d' % (r, r.norme()))
```

59+43j	2793
28+51j	1957
8+20j	304
12+11j	133
-5-3j	19
0	0

1.2.4 2.4 Complexité

Tentons un calcul de pgcd avec deux entiers d'Eisenstein aléatoires. On affiche

- Sur la colonne de gauche, les restes de l'algorithme d'Euclide.
- Sur la colonne de droite, les normes de ces restes.

```
[36]: a = random_Zj(10 ** 10)  
b = random_Zj(10 ** 10)  
rs = restos_euclide(a, b)  
for r in rs:  
    print('%-25s%25d' % (r, r.norme()))
```

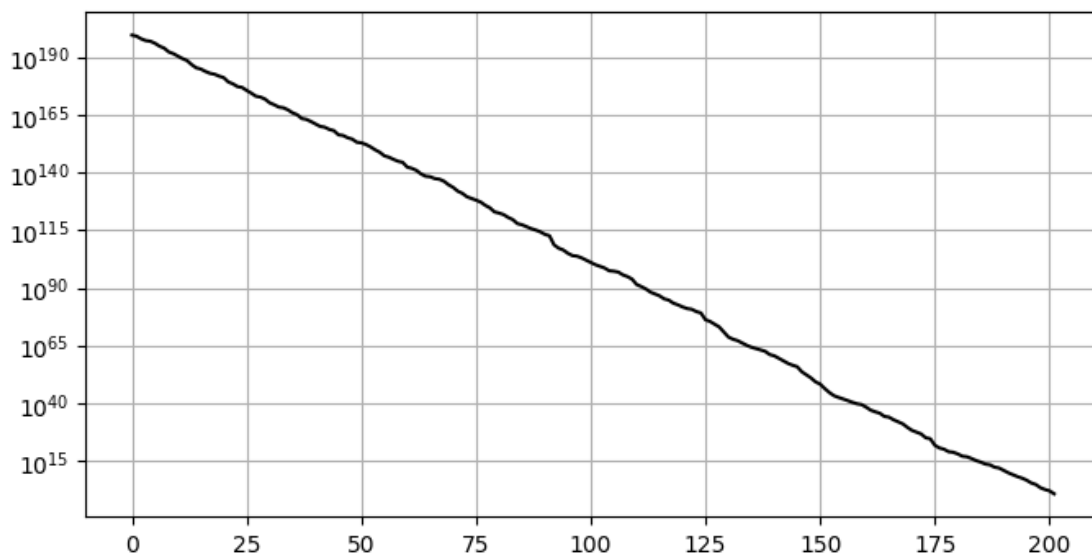
1761941690+7860805301j	51046417922384858011
-2167924163-8601682014j	60041034365897242483
-405982473-740876713j	412937112117028849
-188553106+96120768j	62915345180662468
70932937+17024141j	4113730607201733
-12638950+22351458j	941829691917364
952121-8253759j	76889649301561
-576707-1457698j	1616809782567
-265015-116139j	52942640461
51534+16253j	2082331063
-23598+407j	566635639
3931-6938j	90861883

2071+1331j	3304101
87+164j	20197
6-8j	148
5+2j	19
j	1
0	0

Les normes des restes décroissent très rapidement. Affichons graphiquement ces normes.

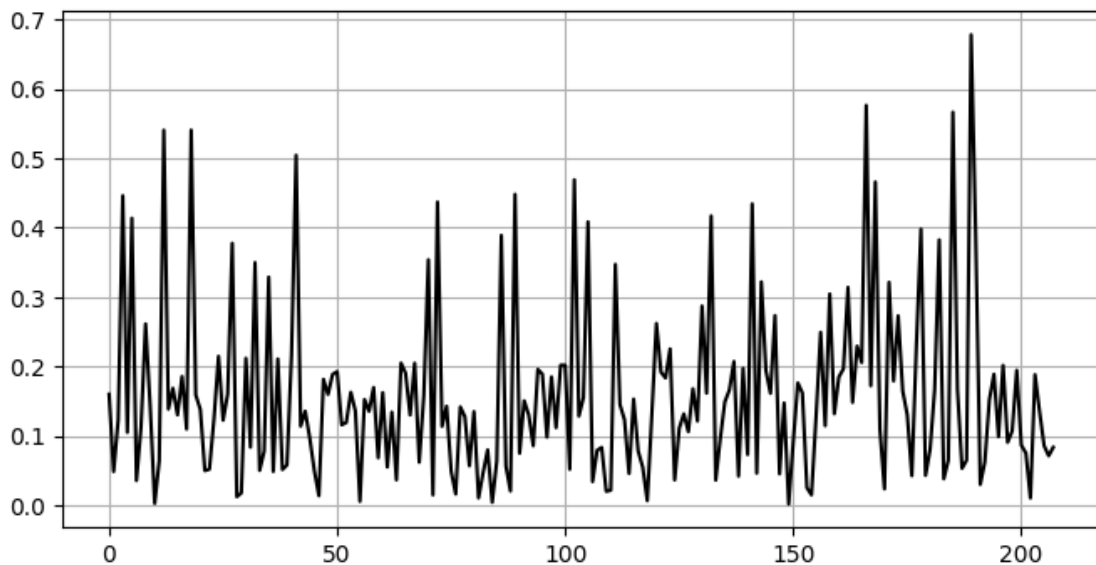
```
[37]: plt.rcParams['figure.figsize'] = (8, 4)
```

```
[38]: a = random_Zj(10 ** 100)
b = random_Zj(10 ** 100)
rs = restes_euclide(a, b)
n = len(rs)
xs = [rs[k].norme() for k in range(1, n - 1)]
plt.semilogy(xs, 'k')
plt.grid()
```



Reprenons l'expérience en affichant les quotients $\frac{N(r_{k+1})}{N(r_k)}$ des normes de deux restes successifs. Comme nous l'avons déjà vu, ces quotients sont inférieurs ou égaux à $\frac{3}{4}$.

```
[39]: a = random_Zj(10 ** 100)
b = random_Zj(10 ** 100)
rs = restes_euclide(a, b)
n = len(rs)
xs = [rs[k + 1].norme() / rs[k].norme() for k in range(1, n - 2)]
plt.plot(xs, 'k')
plt.grid()
```



Quelle est la complexité de l'algorithme d'Euclide dans l'anneau $\mathbb{A}$? Notons n_0 le plus petit entier tel que $r_{n_0} = 0$. On a pour tout $k \in \llbracket 1, n_0 - 1 \rrbracket$,

$$N(r_{k+1}) \leq \frac{3}{4} N(r_k)$$

Il en résulte facilement que pour tout $k \in \llbracket 1, n_0 \rrbracket$,

$$N(r_k) \leq \left(\frac{3}{4}\right)^{k-1} N(b)$$

Prenons $k = n_0 - 1$. On a $N(r_{n_0-1}) \geq 1$, donc

$$1 \leq \left(\frac{3}{4}\right)^{n_0-2} N(b)$$

De là,

$$\left(\frac{4}{3}\right)^{n_0-2} \leq N(b)$$

et donc

$$n_0 \leq 2 + \log_{4/3} N(b) = 2 + 2 \log_{4/3} |b|$$

Il est sans doute possible de faire mieux que cela, mais nous nous contenterons de ce résultat :

Le nombre de divisions de l'algorithme d'Euclide est $O(\log |b|)$.

1.3 3. Factorisation

1.3.1 3.1 Entiers d'Eisenstein premiers

L'anneau $\mathbb{A}$ est un anneau euclidien. Il est donc aussi *factoriel*, c'est à dire qu'il possède une propriété d'existence et d'unicité de décomposition en produit de nombres premiers. C'est de cette dernière notion que nous allons parler ici.

Définition. Soit $a \in \mathbb{A}$. L'entier d'Eisenstein a est *irréductible* si

- $a \notin \mathcal{U}$.
- Pour tous $b, c \in \mathbb{A}$, si $a = bc$ alors $b \in \mathcal{U}$ ou $c \in \mathcal{U}$.

Définition. Soit $a \in \mathbb{A}$. L'entier d'Eisenstein a est *premier* si

- $a \neq 0$.
- $a \notin \mathcal{U}$.
- Pour tous $b, c \in \mathbb{A}$, si a divise bc alors $a|b$ ou $a|c$.

Dans l'anneau $\mathbb{A}$, l'irréductibilité et la primalité sont deux notions équivalentes. Quels sont les entiers d'Eisenstein premiers ? Nous admettons les deux résultats ci-dessous.

Proposition. Soit $p \in \mathbb{Z}$. p est premier dans $\mathbb{A}$ si et seulement si p est premier dans $\mathbb{Z}$ et $|p| \equiv 2 \pmod{3}$.

Proposition. Soit $a \in \mathbb{A}$. On suppose que a n'est pas associé à un entier relatif. Alors, a est premier dans $\mathbb{A}$ si et seulement si $N(a)$ est un entier premier.

Par exemple, l'entier 3 n'est pas premier dans $\mathbb{Z}[j]$. En effet, $3 = (2 + j)(2 + \bar{j})$ et aucun des deux entiers d'Eisenstein du produit n'est inversible (en fait, ils sont premiers).

```
[40]: print((2 + J) * (2 + J.conj()))  
      print((2 + J).norme(), (2 + J.conj()).norme())
```

```
3  
3 3
```

Écrivons une fonction qui décide si un entier d'Eisenstein est premier. Commençons par une fonction `est_entier_premier` qui décide si un entier $p \in \mathbb{Z}$ est premier dans l'anneau $\mathbb{Z}$. Cette fonction n'est pas efficace, sa complexité en pire cas est $O(\sqrt{p})$.

```
[41]: def est_entier_premier(p):  
      p = abs(p)  
      if p <= 1: return False  
      else:  
          k = 2  
          while k * k <= p and p % k != 0:  
              k = k + 1  
          return k * k > p  
  
[42]: print([p for p in range(100) if est_entier_premier(p)])
```


[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

La fonction `est_associe_entier` prend en paramètre un entier d'Eisenstein. Si l'un des associés de a est un entier relatif b , La fonction renvoie `(True, |b|)`. Sinon, elle renvoie `(False, None)`.

```
[43]: def est_associe_entier(a):
        for u in U6:
            b = a * u
            if b.y == 0: return (True, abs(b.x))
        return (False, None)
```

La fonction `est_premier_eisenstein` décide si son paramètre est un entier d'Eisenstein premier.

```
[44]: def est_premier_eisenstein(a):
        ok, p = est_associe_entier(a)
        if ok: return p % 3 == 2 and est_entier_premier(p)
        else: return est_entier_premier(a.norme())
```

```
[45]: print([p for p in range(1000) if est_premier_eisenstein(Zj(p))])
```

[2, 5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89, 101, 107, 113, 131, 137, 149, 167, 173, 179, 191, 197, 227, 233, 239, 251, 257, 263, 269, 281, 293, 311, 317, 347, 353, 359, 383, 389, 401, 419, 431, 443, 449, 461, 467, 479, 491, 503, 509, 521, 557, 563, 569, 587, 593, 599, 617, 641, 647, 653, 659, 677, 683, 701, 719, 743, 761, 773, 797, 809, 821, 827, 839, 857, 863, 881, 887, 911, 929, 941, 947, 953, 971, 977, 983]

```
[46]: print(est_premier_eisenstein(3 + J))
        print(est_premier_eisenstein(5 + J))
```

True
False

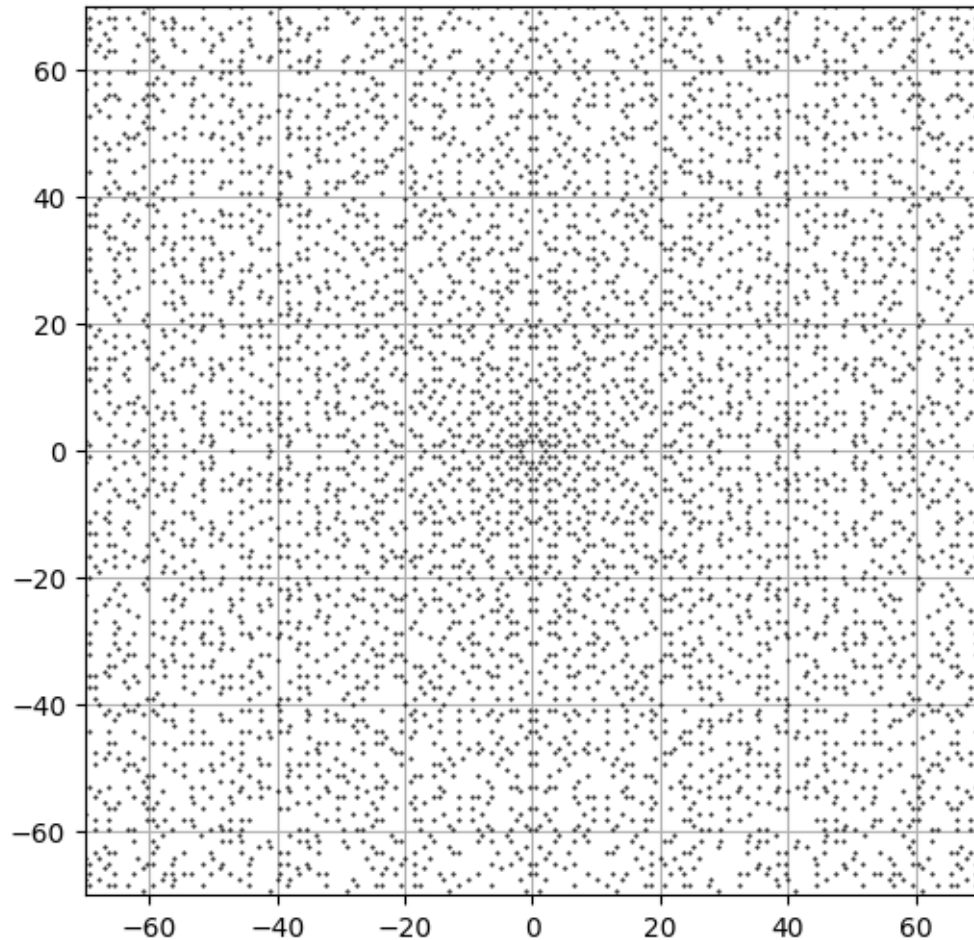
La fonction `tracer_premiers` dessine tous les entiers d'Eisenstein premiers dont les parties réelle et imaginaire appartiennent à $[-N, N]$.

```
[47]: def tracer_premiers(N):
        N2 = int(1.6 * N)
        xs, ys = [], []
        for x in range(-N2, N2 + 1):
            for y in range(-N2, N2 + 1):
                u, v = vers_complexe(Zj(x, y))
                if abs(u) <= N2 and abs(v) <= N2:
                    if est_premier_eisenstein(Zj(x, y)):
                        xs.append(u)
                        ys.append(v)
        plt.plot(xs, ys, 'ok', ms=0.5)
```

```
plt.xlim(-N, N)
plt.ylim(-N, N)
plt.grid()
```

```
[48]: plt.rcParams['figure.figsize'] = (6, 6)
```

```
[49]: tracer_premiers(70)
plt.savefig('figures/premiers.png', bbox_inches='tight')
```



1.3.2 4.2 Factorisation

Soit $a \in \mathbb{Z}[j]$. Supposons que nous savons factoriser a dans $\mathbb{Z}[j]$. Les diviseurs premiers de a sont de deux types :

- Des entiers premiers p_i congrus à 2 modulo 3
- Des entiers d'Eisenstein π_i non associés à un entier et dont la norme est un entier premier.

Les diviseurs de $N(a)$ sont alors

- Les p_i^2
- Les entiers naturels $\pi_i \bar{\pi}_i$

Ainsi, si l'on sait factoriser a , on sait aussi factoriser $N(a)$. Inversement, supposons que l'on sait factoriser $N(a)$ dans $\mathbb{Z}$. Soit π un entier d'Eisenstein premier. Si π divise a alors π divise $a\bar{a} = N(a)$. Inversement, si π divise $N(a)$ alors, comme π est premier, π divise a ou π divise $\bar{a}$. Et donc, π divise a ou $\bar{\pi}$ divise a .

Pour factoriser a il suffit (et il faut !) donc de factoriser $N(a)$ dans $\mathbb{Z}$. Puis, pour chacun des facteurs premiers p entiers de $N(a)$, factoriser p dans $\mathbb{Z}[j]$.

- Si $p = 3$ c'est facile : $3 = (2 + j)(1 - j)$.
- Si $|p| \equiv 2 \pmod{3}$ alors p est aussi premier dans $\mathbb{Z}[j]$.
- Si $|p| \equiv 1 \pmod{3}$ alors $p = \pi \bar{\pi}$ où π est premier dans $\mathbb{Z}[j]$. Nous verrons un peu plus loin comment trouver efficacement un tel π .

La fonction `factoriser` prend en paramètre un entier d'Eisenstein a . Elle renvoie une liste de couples (π_i, k_i) où les π_i sont les entiers d'Eisenstein premiers distincts qui divisent a et les k_i sont les puissances correspondantes. Une exception est à faire pour le dernier élément de la liste, qui est un couple $(u, 1)$ où $u \in \mathcal{U}_6$. On a précisément

$$a = u \prod \pi_i^{k_i}$$

```
[50]: def factoriser(a):
    fs = []
    while not (a in U6):
        p = diviseur_entier_premier(a.norme())
        if p % 3 == 2:
            k = 0
            while a % p == 0:
                k = k + 1
                a = a // p
            fs.append((p, k))
        else:
            u = diviseur_eisenstein(p)
            if a % u != 0: u = u.conj()
            k = 0
            while a % u == 0:
                k = k + 1
                a = a // u
            fs.append((u, k))
    fs.append((a, 1))
    return fs
```

Il reste à écrire deux fonctions :

- `diviseur_entier_premier` qui trouve un diviseur entier premier d'un entier naturel supérieur ou égal à 2. Nous en écrivons une version naïve.

- `diviseur_eisenstein` qui prend en paramètre un entier premier p non congru à 2 modulo 3 et qui renvoie un entier d'Eisenstein u premier qui divise p

```
[51]: def diviseur_entier_premier(n):
    p = 2
    while p * p <= n and n % p != 0:
        p = p + 1
    if p * p > n:
        return n
    else:
        return p
```

```
[52]: diviseur_entier_premier(77)
```

```
[52]: 7
```

Passons à ce qui est le moins évident. Soit p un entier premier positif congru à 1 modulo 3. Supposons trouvé un entier $a \in [0, p-1]$ tel que $a^2 - a + 1 \equiv 0 \pmod{p}$. Nous admettrons ici le résultat suivant.

Proposition. Soit $\pi = \text{pgcd}(a + j, p)$. Alors, π est un entier d'Eisenstein premier qui divise p .

Reste à savoir comment trouver a . Il s'agit de résoudre dans $\mathbb{Z}/p\mathbb{Z}$ l'équation

$$(E) \ a^2 - a + 1 = 0$$

Le discriminant de cette équation est -3 . Il se trouve que comme $p \equiv 1 \pmod{3}$, -3 est un carré modulo p . Pour trouver une racine carrée δ de -3 on peut utiliser par exemple l'algorithme de Tonelli-Shanks. Cet algorithme est implémenté dans le module `shanks`. Une fois calculé δ , on a

$$a = (1 \pm \delta) * 2^{-1}$$

et l'inverse de 2 dans $\mathbb{Z}/p\mathbb{Z}$ est $\frac{p+1}{2}$.

La fonction `diviseur_eisenstein` fait le travail.

```
[53]: def diviseur_eisenstein(p):
    if p == 3: return 2 + J
    else:
        r = shanks.racine(p - 3, p)
        a = ((1 + r) * ((p + 1) // 2)) % p
        return pgcd(a + J, p)
```

```
[54]: diviseur_eisenstein(19)
```

```
[54]: 5+2j
```

```
[55]: print(19 // (5 + 2 * J), 19 % (5 + 2 * J))
```

3-2j 0

Nous pouvons maintenant factoriser des entiers d'Eisenstein.

```
[56]: a = random_Zj(10 ** 6)
      print(a)
      fs = factoriser(a)
      print(fs)
```

```
-283172-978654j
[(2, 1), (1-2j, 1), (6+5j, 1), (18875-15247j, 1), (-j, 1)]
```

La fonction `verifier` prend une liste renvoyée par `factoriser`. Elle en fait le produit. On devrait évidemment retrouver l'entier d'Eisenstein original.

```
[57]: def verifier(fs):
      a = Zj(1)
      for (p, k) in fs:
          a = a * p ** k
      return a
```

```
[58]: a = random_Zj(10 ** 6)
      print(a)
      fs = factoriser(a)
      print(fs)
      print(verifier(fs))
```

```
-968619+260560j
[(9-13j, 1), (27921-39373j, 1), (-1-j, 1)]
-968619+260560j
```

1.3.3 4.3 Diviseurs

Maintenant que nous savons factoriser les entiers d'Eisenstein, il est facile de déterminer tous leurs diviseurs. La fonction `diviseurs_aux` prend en paramètre une liste de couples (p_i, k_i) renvoyée par la fonction `factoriser`. Elle effectue tous les produits possibles et renvoie la liste des résultats.

```
[59]: def diviseurs_aux(fs):
      if fs == []:
          return [Zj(1)]
      else:
          p, k = fs[0]
          ds = diviseurs_aux(fs[1:])
          return [p ** j * d for j in range(k + 1) for d in ds]
```

La fonction `diviseurs principaux` renvoie la liste des diviseurs de a à inversible près.

```
[60]: def diviseurs_principaux(a):
      fs = factoriser(a)[: -1]
```

```
return diviseurs_aux(fs)
```

Pour avoir la liste de *tous* les diviseurs de a , il suffit de multiplier par les éléments de $\mathcal{U}_6$.

```
[61]: def diviseurs(a):  
      ds = diviseurs_principaux(a)  
      return [u * d for u in U6 for d in ds]
```

```
[62]: a = 3 + J  
      print(diviseurs_principaux(a))  
      print(diviseurs(a))  
      print(len(diviseurs(a)))
```

```
[1, 1-2j]  
[1, 1-2j, 1+j, 3+j, j, 2+3j, -1, -1+2j, -1-j, -3-j, -j, -2-3j]  
12
```

```
[63]: a = 5 + J  
      print(diviseurs_principaux(a))  
      print(diviseurs(a))  
      print(len(diviseurs(a)))
```

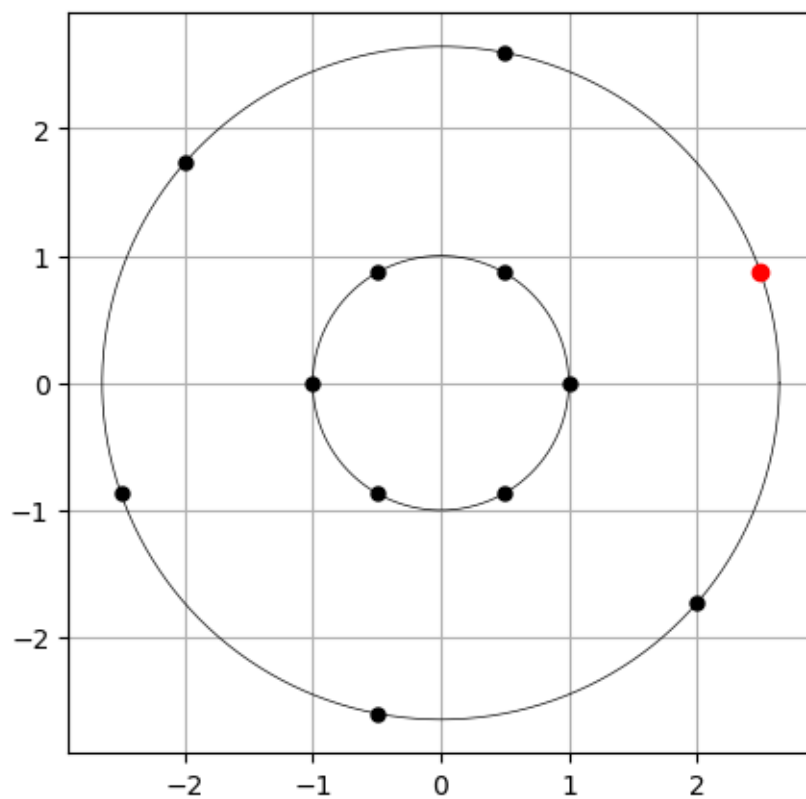
```
[1, 3+2j, 2+j, 4+5j]  
[1, 3+2j, 2+j, 4+5j, 1+j, 1+3j, 1+2j, -1+4j, j, -2+j, -1+j, -5-j, -1, -3-2j,  
-2-j, -4-5j, -1-j, -1-3j, -1-2j, 1-4j, -j, 2-j, 1-j, 5+j]  
24
```

La fonction `tracer_diviseurs` dessine les diviseurs de l'entier d'Eisenstein a .

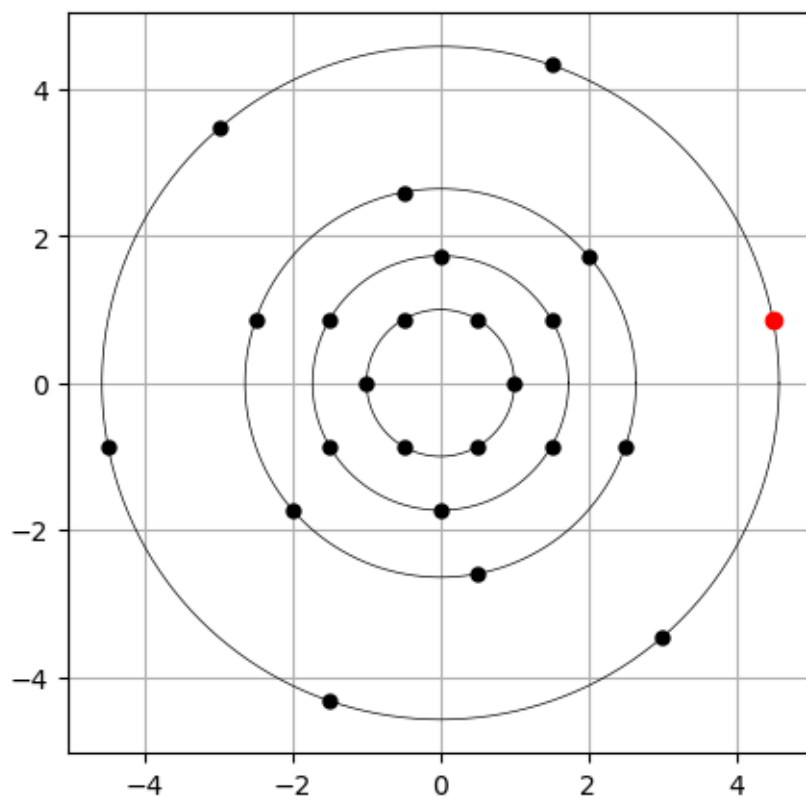
```
[64]: def tracer_diviseurs(a):  
      ds = diviseurs(a)  
      rs = {math.sqrt(d.norme()) for d in ds}  
      for r in rs:  
          tracer_cercle(r, ls='-', lw=0.5)  
      zs = [vers_complexe(d) for d in ds]  
      xs = [x for (x, y) in zs]  
      ys = [y for (x, y) in zs]  
      plt.plot(xs, ys, 'ok', ms=5)  
      x, y = vers_complexe(a)  
      plt.plot([x], [y], 'or')  
      plt.grid()
```

```
[65]: plt.rcParams['figure.figsize'] = (5, 5)
```

```
[66]: tracer_diviseurs(3 + J)  
      plt.savefig('figures/diviseurs_3j.png', bbox_inches='tight')
```

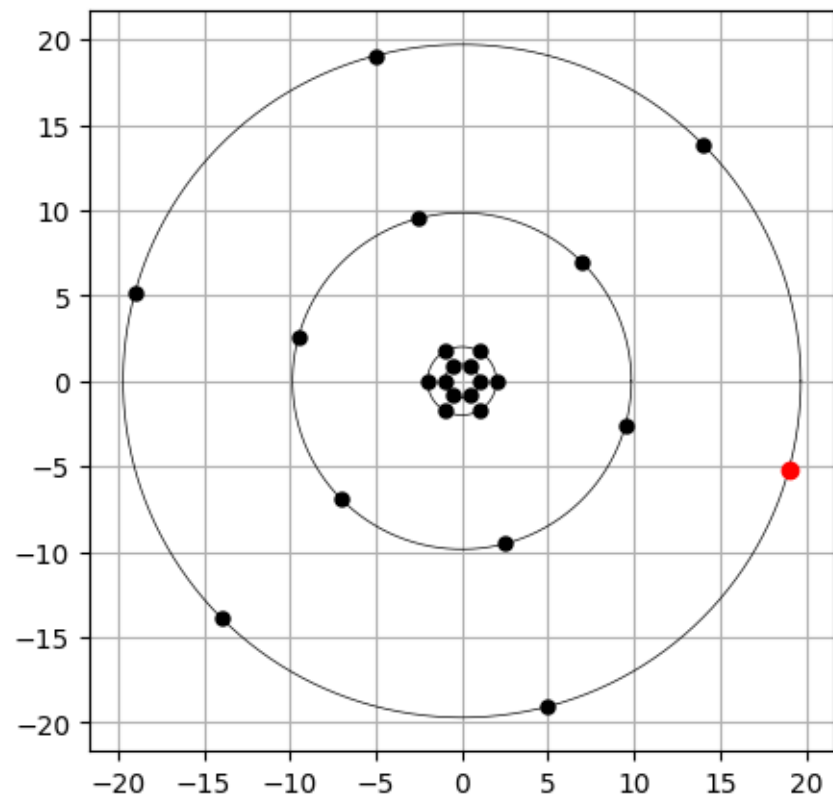


```
[67]: tracer_diviseurs(5 + J)
plt.savefig('figures/diviseurs_5j.png', bbox_inches='tight')
```



```
[68]: a = random_Zj(50)
      print(a)
      print(diviseurs_principaux(a))
      tracer_diviseurs(a)
```

```
16-6j
[1, 8-3j, 2, 16-6j]
```

[]: