8 Corps finis et théorie des nombres

Ce chapitre décrit l'utilisation de Sage en théorie des nombres, pour manipuler des objets sur des anneaux ou corps finis (§8.1), pour tester la primalité (§8.2) ou factoriser un entier (§8.3) ; enfin nous discutons quelques applications (§8.4).

8.1 Anneaux et corps finis

Les anneaux et corps finis sont un objet fondamental en théorie des nombres, et en calcul symbolique en général. En effet, de nombreux algorithmes de calcul formel se ramènent à des calculs sur des corps finis, puis on exploite l'information obtenue via des techniques comme la remontée de Hensel ou la reconstruction par les restes chinois. Citons par exemple l'algorithme de Cantor-Zassenhaus pour la factorisation de polynôme univarié à coefficients entiers, qui commence par factoriser le polynôme donné sur un corps fini.

8.1.1 Anneau des entiers modulo n

En Sage, l'anneau $\mathbb{Z}/n\mathbb{Z}$ des entiers modulo n se définit à l'aide du constructeur `IntegerModRing` (ou plus simplement `Integers`). Tous les objets construits à partir de ce constructeur et leurs dérivés sont systématiquement réduits modulo n, et ont donc une forme canonique, c'est-à-dire

que deux variables représentant la même valeur modulo n ont la même représentation interne. Dans certains cas bien particuliers, il est plus efficace de retarder les réductions modulo n, par exemple si on multiplie des matrices avec de tels coefficients ; on préférera alors travailler avec des entiers, et effectuer les réductions modulo n « à la main » via `a % n`. Attention, le module n n'apparaît pas explicitement dans la valeur affichée :

```
sage: a=IntegerModRing(15)(3); b=IntegerModRing(17)(3);
      print a, b
3 3
sage: a == b
False
```

Une conséquence est que si l'on « copie-colle » des entiers modulo n, on perd l'information sur n. Étant donnée une variable contenant un entier modulo n, on retrouve l'information sur n via les méthodes `base_ring` ou `parent`, et la valeur de n via la méthode `characteristic` :

```
sage: R=a.base_ring(); R
Ring of integers modulo 15
sage: R.characteristic()
15
```

Les opérateurs de base (addition, soustraction, multiplication) sont surchargés pour les entiers modulo n, et appellent les fonctions correspondantes, de même que les entiers sont automatiquement convertis, dès lors qu'un des opérandes est un entier modulo n :

```
sage: print a+a, a-17, a*a+1, a^3
6 1 10 12
```

Quant à l'inversion $1/a \bmod n$ ou la division $b/a \bmod n$, Sage l'effectue quand elle est possible, sinon il renvoie une erreur `ZeroDivisionError`, i.e., quand a et n ont un pgcd non-trivial :

```
sage: 1/(a+1)
4
sage: 1/a
ZeroDivisionError: Inverse does not exist.
```

Pour obtenir la valeur de a — en tant qu'entier — à partir du résidu $a \bmod n$, on peut utiliser la méthode `lift` ou bien `ZZ` :

```
sage: z=lift(a); y=ZZ(a); print y, type(y), y==z
3 <type 'sage.rings.integer.Integer'> True
```

L'*ordre additif* de a modulo n est le plus petit entier $k > 0$ tel que $ka = 0 \bmod n$. Il vaut $k = n/g$ où $g = \text{pgcd}(a, n)$, et est donné par la méthode `additive_order` (on voit au passage qu'on peut aussi utiliser `Mod` ou `mod` pour définir les entiers modulo n) :

```
sage: [Mod(x,15).additive_order() for x in range(0,15)]
[1, 15, 15, 5, 15, 3, 5, 15, 15, 5, 3, 15, 5, 15, 15]
```

L'*ordre multiplicatif* de a modulo n, pour a premier avec n, est le plus petit entier $k > 0$ tel que $a^k = 1 \bmod n$. (Si a a un diviseur commun p avec n, alors $a^k \bmod n$ est un multiple de p quel que soit k.) Si cet ordre multiplicatif égale $\varphi(n)$, à savoir l'ordre du groupe multiplicatif modulo n, on dit que a est un *générateur* de ce groupe. Ainsi pour $n = 15$, il n'y a pas de générateur, puisque l'ordre maximal est $4 < 8 = \varphi(15)$:

```
sage: [[x,Mod(x,15).multiplicative_order()]
        for x in range(1,15) if gcd(x,15)==1]
[[1, 1], [2, 4], [4, 2], [7, 4], [8, 4], [11, 2], [13, 4], [14, 2]]
```

Voici un exemple avec $n = p$ premier, où 3 est générateur :

```
sage: p=10^20+39; mod(2,p).multiplicative_order()
50000000000000000019
sage: mod(3,p).multiplicative_order()
100000000000000000038
```

Une opération importante sur $\mathbb{Z}/n\mathbb{Z}$ est l'*exponentiation modulaire*, qui consiste à calculer $a^e \bmod n$. Le crypto-système RSA repose sur cette opération. Pour calculer efficacement $a^e \bmod n$, les algorithmes les plus efficaces nécessitent de l'ordre de $\log e$ multiplications ou carrés modulo n. Il est crucial de réduire systématiquement tous les calculs modulo n, au lieu de calculer d'abord a^e en tant qu'entier, comme le montre l'exemple suivant :

```
sage: n=3^100000; a=n-1; e=100
sage: %timeit (a^e) % n
5 loops, best of 3: 387 ms per loop
sage: %timeit power_mod(a,e,n)
125 loops, best of 3: 3.46 ms per loop
```

8.1.2 Corps finis

Les corps finis[1] en Sage se définissent à l'aide du constructeur `FiniteField` (ou plus simplement `GF`). On peut aussi bien construire les *corps premiers* `GF(p)` avec p premier que les corps composés `GF(q)` avec $q = p^k$, p premier et $k > 1$ un entier. Comme pour les anneaux, les objets créés dans un tel corps ont une forme canonique, par conséquent une réduction est effectuée à chaque opération. Les corps finis jouissent des mêmes propriétés que les anneaux (§8.1.1), avec en plus la possibilité d'inverser un élément non nul :

```
sage: R = GF(17); [1/R(x) for x in range(1,17)]
[1, 9, 6, 13, 7, 3, 5, 15, 2, 12, 14, 10, 4, 11, 8, 16]
```

Un corps non premier $\mathbb{F}_{p^k}$ avec p premier et $k > 1$ est isomorphe à l'anneau quotient des polynômes de $\mathbb{F}_p[x]$ modulo un polynôme f unitaire et irréductible de degré k. Dans ce cas, Sage demande un nom pour le *générateur* du corps, c'est-à-dire la variable x :

```
sage: R = GF(9,name='x'); R
Finite Field in x of size 3^2
```

Ici, Sage a choisi automatiquement le polynôme f :

```
sage: R.polynomial()
x^2 + 2*x + 2
```

Les éléments du corps sont alors représentés par des polynômes $a_{k-1}x^{k-1} + \cdots + a_1x + a_0$, où les a_i sont des éléments de $\mathbb{F}_p$:

```
sage: [r for r in R]
[0, 2*x, x + 1, x + 2, 2, x, 2*x + 2, 2*x + 1, 1]
```

On peut aussi imposer à Sage le polynôme irréductible f :

```
sage: Q.<x> = PolynomialRing(GF(3))
sage: R2 = GF(9,name='x',modulus=x^2+1); R2
Finite Field in x of size 3^2
```

Attention cependant, car si les deux instances `R` et `R2` créées ci-dessus sont isomorphes à $\mathbb{F}_9$, l'isomorphisme n'est pas explicite :

```
sage: p = R(x+1); R2(p)
TypeError: unable to coerce from a finite field other than the
prime subfield
```

[1] En français, le corps fini à q éléments est noté usuellement $\mathbb{F}_q$, alors qu'en anglais on utilise plutôt GF(q). On utilise ici la notation française pour désigner le concept mathématique, et la notation anglaise pour désigner du code Sage.

8.1.3 Reconstruction rationnelle

Le problème de la *reconstruction rationnelle* constitue une jolie application des calculs modulaires. Étant donné un résidu a modulo m, il s'agit de trouver un « petit » rationnel x/y tel que $x/y \equiv a \bmod m$. Si on sait qu'un tel petit rationnel existe, au lieu de calculer directement x/y en tant que rationnel, on calcule x/y modulo m, ce qui donne le résidu a, puis on retrouve x/y par reconstruction rationnelle. Cette seconde approche est souvent plus efficace, car on remplace des calculs rationnels — faisant intervenir de coûteux pgcds — par des calculs modulaires.

Lemme 1. *Soient $a, m \in \mathbb{N}$, avec $0 < a < m$. Il existe au plus une paire d'entiers $x, y \in \mathbb{Z}$ premiers entre eux tels que $x/y \equiv a \bmod m$ avec $0 < |x|, y \leqslant \sqrt{m/2}$.*

Il n'existe pas toujours de telle paire x, y, par exemple pour $a = 2$ et $m = 5$. L'algorithme de reconstruction rationnelle est basé sur l'algorithme de pgcd étendu. Le pgcd étendu de m et a calcule une suite d'entiers $a_i = \alpha_i m + \beta_i a$, où les entiers a_i décroissent, et les coefficients α_i, β_i croissent en valeur absolue. Il suffit donc de s'arrêter dès que $|a_i|, |\beta_i| < \sqrt{m/2}$, et la solution est alors $x/y = a_i/\beta_i$. Cet algorithme est disponible via la fonction `rational_reconstruction` de Sage, qui renvoie x/y lorsqu'une telle solution existe, et une erreur sinon :

```
sage: rational_reconstruction(411,1000)
-13/17
sage: rational_reconstruction(409,1000)
Traceback (most recent call last):
...
ValueError: Rational reconstruction of 409 (mod 1000) does not
exist.
```

Pour illustrer la reconstruction rationnelle, considérons le calcul du nombre harmonique $H_n = 1 + 1/2 + \cdots + 1/n$. Le calcul naïf avec des nombres rationnels est le suivant :

```
sage: def harmonic(n):
          return add([1/x for x in range(1,n+1)])
```

Or nous savons que H_n peut s'écrire sous la forme p_n/q_n avec p_n, q_n entiers, où q_n est le ppcm de $1, 2, \ldots, n$. On sait par ailleurs que $H_n \leqslant \log n + 1$, ce qui permet de borner p_n. On en déduit la fonction suivante qui détermine H_n par calcul modulaire et reconstruction rationnelle :

```
def harmonic_mod(n,m):
   return add([1/x % m for x in range(1,n+1)])
def harmonic2(n):
   q = lcm(range(1,n+1))
   pmax = RR(q*(log(n)+1))
   m = ZZ(2*pmax^2)
   m = ceil(m/q)*q + 1
   a = harmonic_mod(n,m)
   return rational_reconstruction(a,m)
```

La ligne `m = ZZ(2*pmax^2)` garantit que la reconstruction rationnelle va trouver $p \leqslant \sqrt{m/2}$, tandis que la ligne suivante garantit que m est premier avec $x = 1, 2, \ldots, n$, sinon $1/x \bmod n$ provoquerait une erreur.

```
sage: harmonic(100) == harmonic2(100)
True
```

Sur cet exemple, la fonction `harmonic2` n'est pas plus efficace que la fonction `harmonic`, mais elle illustre bien notre propos. Dans certains cas, il n'est pas nécessaire de connaître une borne rigoureuse sur x et y, une estimation « à la louche » suffit, car on peut vérifier facilement par ailleurs que x/y est la solution cherchée.

On peut généraliser la reconstruction rationnelle avec un numérateur x et un dénominateur y de tailles différentes (voir par exemple la section 5.10 du livre [vzGG03]).

8.1.4 Restes chinois

Une autre application utile des calculs modulaires est ce qu'on appelle communément les « restes chinois ». Étant donnés deux modules m et n premiers entre eux, soit x un entier inconnu tel que $x \equiv a \bmod m$ et $x \equiv b \bmod n$. Alors le *théorème des restes chinois* permet de reconstruire de façon unique la valeur de x modulo le produit mn. En effet, on déduit de $x \equiv a \bmod m$ que x s'écrit sous la forme $x = a + \lambda m$ avec $\lambda \in \mathbb{Z}$. En remplaçant cette valeur dans $x \equiv b \bmod n$, on obtient $\lambda \equiv \lambda_0 \bmod n$, où $\lambda_0 = (b - a)/m \bmod n$. Il en résulte $x = x_0 + \mu nm$, où $x_0 = a + \lambda_0 m$.

On a décrit ici la variante la plus simple des « restes chinois ». On peut également considérer le cas où m et n ne sont pas premiers entre eux, ou bien le cas de plusieurs moduli $m_1, m_2, \ldots, m_k$. La commande Sage pour trouver x_0 à partir de a, b, m, n est `crt(a,b,m,n)` :

```
sage: a=2; b=3; m=5; n=7; lambda0=(b-a)/m % n; a+lambda0*m
17
sage: crt(2,3,5,7)
17
```

Reprenons l'exemple du calcul de H_n. Calculons d'abord $H_n \bmod m_i$ pour $i = 1, 2, \ldots, k$, ensuite nous déduisons $H_n \bmod (m_1 \cdots m_k)$ par restes chinois, enfin nous retrouvons H_n par reconstruction rationnelle :

```
def harmonic3(n):
   q = lcm(range(1,n+1))
   pmax = RR(q*(log(n)+1))
   B = ZZ(2*pmax^2)
   m = 1; a = 0; p = 2^63
   while m<B:
      p = next_prime(p)
      b = harmonic_mod(n,p)
      a = crt(a,b,m,p)
      m = m*p
   return rational_reconstruction(a,m)
sage: harmonic(100) == harmonic3(100)
True
```

La fonction `crt` de Sage fonctionne aussi quand les moduli m et n ne sont pas premiers entre eux. Soit $g = \gcd(m, n)$, il y a une solution si et seulement si $a \bmod g \equiv b \bmod g$:

```
sage: crt(15,1,30,4)
45
sage: crt(15,2,30,4)
...
ValueError: No solution to crt problem since gcd(30,4) does not
divide 15-2
```

Une application plus complexe des restes chinois est présentée dans l'exercice 26.

8.2 Primalité

Tester si un entier est premier est une des opérations fondamentales d'un logiciel de calcul symbolique. Même si l'utilisateur ne s'en rend pas

compte, de tels tests sont effectués plusieurs milliers de fois par seconde par le logiciel. Par exemple pour factoriser un polynôme de $\mathbb{Z}[x]$ on commence par le factoriser dans $\mathbb{F}_p[x]$ pour un nombre premier p, il faut donc trouver un tel p.

Deux grandes classes de tests de primalité existent. Les plus efficaces sont des tests de *pseudo-primalité*, et sont en général basés sur des variantes du petit théorème de Fermat, qui dit que si p est premier, alors tout entier $0 < a < p$ est un générateur du groupe multiplicatif $(\mathbb{Z}/p\mathbb{Z})^*$, donc $a^{p-1} \equiv 1 \bmod p$. On utilise en général une petite valeur de a $(2, 3, \ldots)$ pour accélérer le calcul de $a^{p-1} \bmod p$. Si $a^{p-1} \neq 1 \bmod p$, p n'est certainement pas premier. Si $a^{p-1} \equiv 1 \bmod p$, on ne peut rien conclure : on dit alors que p est pseudo-premier en base a. L'intuition est qu'un entier p qui est pseudo-premier pour plusieurs bases a de grandes chances d'être premier (voir cependant ci-dessous). Les tests de pseudo-primalité ont en commun que quand ils renvoient `False`, le nombre est certainement composé, par contre quand ils renvoient `True`, on ne peut rien conclure.

La seconde classe est constituée des tests de *vraie primalité*. Ces tests renvoient toujours une réponse correcte, mais peuvent être moins efficaces que les tests de pseudo-primalité, notamment pour les nombres qui sont pseudo-premiers en de nombreuses bases, et en particulier pour les nombres vraiment premiers. De nombreux logiciels ne fournissent qu'un test de pseudo-primalité, voire pire le nom de la fonction correspondante (`isprime` par exemple) laisse croire à l'utilisateur que c'est un test de (vraie) primalité. Sage fournit deux fonctions distinctes : `is_pseudoprime` pour la pseudo-primalité, et `is_prime` pour la primalité :

```
sage: p=previous_prime(2^400)
sage: %timeit is_pseudoprime(p)
625 loops, best of 3: 1.07 ms per loop
sage: %timeit is_prime(p)
5 loops, best of 3: 485 ms per loop
```

Nous voyons sur cet exemple que le test de primalité est bien plus coûteux ; quand c'est possible, on préfèrera `is_pseudoprime`.

Certains algorithmes de primalité fournissent un *certificat*, qui peut être vérifié indépendamment, souvent de manière plus efficace que le test lui-même. Sage ne fournit pas de tel certificat dans la version actuelle.

Les *nombres de Carmichael* sont des entiers composés qui sont pseudo-premiers dans toutes les bases. Le petit théorème de Fermat ne permet

donc pas de les distinguer des nombres premiers, quel que soit le nombre de bases essayées. Le plus petit nombre de Carmichael est $561 = 3 \cdot 11 \cdot 17$. Un nombre de Carmichael a au moins trois facteurs premiers. En effet, supposons que $n = pq$ soit un nombre de Carmichael, avec p, q premiers, $p < q$; par définition des nombres de Carmichael, on a pour tout $1 \leqslant a < q$ l'égalité $a^{n-1} \equiv 1$ modulo n, et par suite modulo q, ce qui implique que $n-1$ est multiple de $q-1$. L'entier n est nécessairement de la forme $q + \lambda q(q-1)$, puisqu'il est multiple de q et $n-1$ multiple de $q-1$, ce qui est incompatible avec $n = pq$ puisque $p < q$. Si $n = pqr$, alors il suffit que $a^{n-1} \equiv 1 \bmod p$ — et de même pour q et r, puisque par restes chinois on aura alors $a^{n-1} \equiv 1 \bmod n$. Une condition suffisante est que $n-1$ soit multiple de $p-1$, $q-1$ et $r-1$:

```
sage: [560 % (x-1) for x in [3,11,17]]
[0, 0, 0]
```

Exercice 23. Écrire une fonction Sage comptant les nombres de Carmichael $pqr \leqslant n$, avec p, q, r premiers impairs distincts. Combien trouvez-vous pour $n = 10^4, 10^5, 10^6, 10^7$? (Richard Pinch a compté 20138200 nombres de Carmichael inférieurs à 10^{21}.)

Enfin, pour itérer une opération sur des nombres premiers dans un intervalle, il vaut mieux utiliser la construction `prime_range`, qui construit une table via un crible, plutôt qu'une boucle avec `next_probable_prime` ou `next_prime` :

```
def count_primes1(n):
   return add([1 for p in range(n+1) if is_prime(p)])
def count_primes2(n):
   return add([1 for p in range(n+1) if is_pseudoprime(p)])
def count_primes3(n):
   s=0; p=2
   while p <= n: s+=1; p=next_prime(p)
   return s
def count_primes4(n):
   s=0; p=2
   while p <= n: s+=1; p=next_probable_prime(p)
   return s
def count_primes5(n):
   s=0
   for p in prime_range(n): s+=1
```

```
   return s
sage: %timeit count_primes1(10^5)
5 loops, best of 3: 674 ms per loop
sage: %timeit count_primes2(10^5)
5 loops, best of 3: 256 ms per loop
sage: %timeit count_primes3(10^5)
5 loops, best of 3: 49.2 ms per loop
sage: %timeit count_primes4(10^5)
5 loops, best of 3: 48.6 ms per loop
sage: %timeit count_primes5(10^5)
125 loops, best of 3: 2.67 ms per loop
```

8.3 Factorisation et logarithme discret

On dit qu'un entier a est un carré — ou résidu quadratique — modulo n s'il existe x, $0 \leqslant x < n$, tel que $a \equiv x^2 \bmod n$. Sinon, on dit que a est un non-résidu quadratique modulo n. Lorsque $n = p$ est premier, ce test peut se décider efficacement grâce au calcul du symbole de Jacobi de a et p, noté $(a|p)$, qui peut prendre les valeurs $\{-1, 0, 1\}$, où $(a|p) = 0$ signifie que a est multiple de p, et $(a|p) = 1$ (respectivement $(a|p) = -1$) signifie que a est (respectivement n'est pas) un carré modulo p. La complexité du calcul du symbole de Jacobi $(a|n)$ est essentiellement la même que celle du calcul du pgcd de a et n, à savoir $O(M(\ell)\log\ell)$ où ℓ est la taille de n. Cependant toutes les implantations du symbole de Jacobi — voire du pgcd — n'ont pas cette complexité (`a.jacobi(n)` calcule $(a|n)$) :

```
sage: p=(2^42737+1)//3; a=3^42737
sage: %timeit a.gcd(p)
125 loops, best of 3: 4.3 ms per loop
sage: %timeit a.jacobi(p)
25 loops, best of 3: 26.1 ms per loop
```

Lorsque n est composé, trouver les solutions de $x^2 \equiv a \bmod n$ est aussi difficile que factoriser n. Toutefois le symbole de Jacobi, qui est relativement facile à calculer, donne une information partielle. En effet, si $(a|n) = -1$, il n'y a pas de solution, car une solution vérifie nécessairement $(a|p) = 1$ pour tous les facteurs premiers p de n, donc $(a|n) = 1$.

Le logarithme discret. Soit n un entier positif, g un *générateur* du groupe multiplicatif modulo n et a premier avec n, $0 < a < n$. Par définition

du fait que g est un générateur, il existe un entier x tel que $g^x = a \bmod n$. Le problème du *logarithme discret* consiste à trouver un tel entier x. La méthode `log` permet de résoudre ce problème :

```
sage: p=10^10+19; a=mod(17,p); a.log(2)
6954104378
sage: mod(2,p)^6954104378
17
```

Les meilleurs algorithmes connus pour calculer un logarithme discret sont de même ordre de complexité — en fonction de la taille de n — que ceux pour factoriser n. Cependant l'implantation actuelle en Sage du logarithme discret est peu efficace :

```
sage: p=10^20+39; a=mod(17,p)
sage: time r=a.log(3)
CPU times: user 89.63 s, sys: 1.70 s, total: 91.33 s
```

Suites aliquotes. La *suite aliquote* associée à un entier positif n est la suite (s_k) définie par récurrence : $s_0 = n$ et $s_{k+1} = \sigma(s_k) - s_k$, où $\sigma(s_k)$ est la somme des diviseurs de s_k, i.e., s_{k+1} est la suite des diviseurs *propres* de s_k, c'est-à-dire sans s_k lui-même. On arrête l'itération lorsque $s_k = 1$ — alors s_{k-1} est premier — ou lorsque la suite (s_k) décrit un cycle. Par exemple en partant de $n = 30$ on obtient :

$$30, 42, 54, 66, 78, 90, 144, 259, 45, 33, 15, 9, 4, 3, 1.$$

Lorsque le cycle est de longueur un, on dit que l'entier correspondant est *parfait*, par exemple $6 = 1 + 2 + 3$ et $28 = 1 + 2 + 4 + 7 + 14$ sont parfaits. Lorsque le cycle est de longueur deux, on dit que les deux entiers en question sont *amicaux*, comme 220 et 284. Lorsque le cycle est de longueur trois ou plus, les entiers formant ce cycle sont dits *sociables*.

Exercice 24. Calculer la suite aliquote commençant par 840, afficher les 5 premiers et 5 derniers éléments, et tracer le graphe de $\log_{10} s_k$ en fonction de k. (On pourra utiliser la fonction `sigma`.)

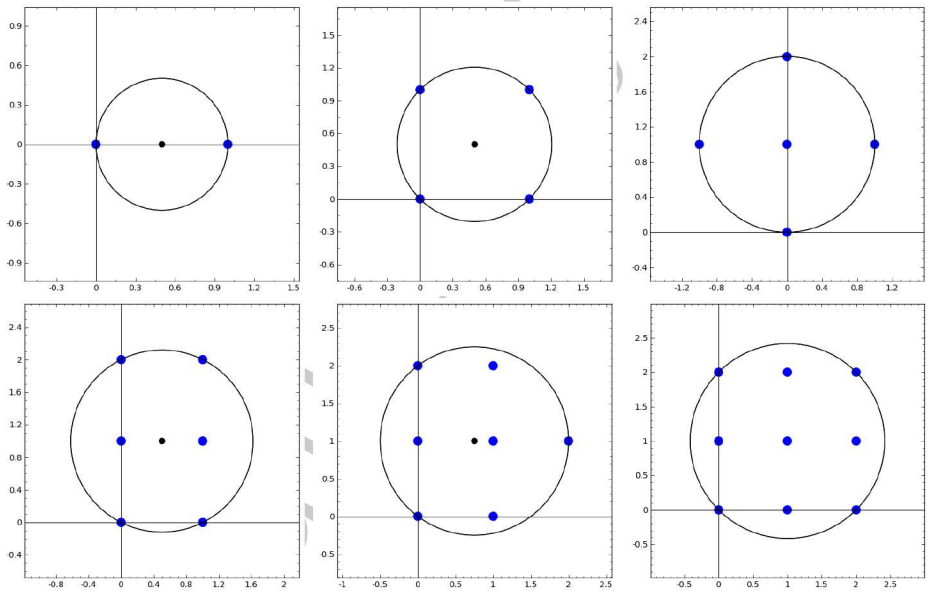
8.4 Applications

8.4.1 La constante δ

La constante δ est une généralisation en dimension deux de la constante γ d'Euler. Elle est définie comme suit :

$$\delta = \lim_{n\to\infty} \left(\sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log n \right), \tag{8.1}$$

où r_k est le rayon du plus petit disque du plan affine $\mathbb{R}^2$ contenant au moins k points de $\mathbb{Z}^2$. Par exemple $r_2 = 1/2$, $r_3 = r_4 = \sqrt{2}/2$, $r_5 = 1$, $r_6 = \sqrt{5}/2$, $r_7 = 5/4$, $r_8 = r_9 = \sqrt{2}$, ...



Exercice 25. 1. Écrire une fonction qui prend en entrée un entier positif k, et renvoie le rayon r_k et le centre (x, y) d'un plus petit disque contenant au moins k points de $\mathbb{Z}^2$.

2. Écrire une fonction dessinant le cercle correspondant à r_k avec les $m \geqslant k$ points de $\mathbb{Z}^2$ inclus.

3. En utilisant l'encadrement

$$\frac{\sqrt{\pi(k-6)+2}-\sqrt{2}}{\pi} < r_k < \sqrt{\frac{k-1}{\pi}}, \tag{8.2}$$

calculer une approximation de δ avec une erreur bornée par 0.1.

8.4.2 Calcul d'intégrale multiple via reconstruction rationnelle

Cette application est inspirée de l'article *Robust Mathematical Methods for Extremely Rare Events* de Bernard Beauzamy. Soient k et $n_1, n_2, \ldots, n_k$ des entiers positifs ou nuls. On veut calculer l'intégrale

$$I = \int_V x_1^{n_1} x_2^{n_2} \cdots x_k^{n_k} \mathrm{d}x_1 \mathrm{d}x_2 \ldots \mathrm{d}x_k,$$

où le domaine d'intégration est défini par $V = \{x_1 \geqslant x_2 \geqslant \cdots \geqslant x_k \geqslant 0, x_1 + \cdots + x_k \leqslant 1\}$.

Exercice 26. Sachant que I est un nombre rationnel, mettre au point un algorithme utilisant la reconstruction rationnelle et/ou les restes chinois pour calculer I. On implantera cet algorithme en Sage et on l'appliquera au cas où $[n_1, \ldots, n_{31}] =$

$$[9, 7, 8, 11, 6, 3, 7, 6, 6, 4, 3, 4, 1, 2, 2, 1, 1, 1, 2, 0, 0, 0, 3, 0, 0, 0, 0, 1, 0, 0, 0].$$