*God made the integers, all else is the work of man.*
Leopold Kronecker (1823 - 1891)

# 6 Finite Fields and Elementary Number Theory

This chapter describes the use of Sage for elementary number theory, for working with objects related to finite fields (§6.1), for primality testing (§6.2) and integer factorisation (§6.3); we will also discuss some applications (§6.4).

## 6.1 Finite Fields and Rings

Finite rings and fields are basic objects, both in number theory and throughout computer algebra. Indeed, many algorithms in computer algebra involve computations over finite fields, where one can exploit the information obtained using techniques such as Hensel lifting, or reconstruction using the Chinese Remainder Theorem. As an example, we can mention the Cantor-Zassenhaus algorithm for factoring univariate polynomials with integer coefficients, which begins by factoring the polynomial over a finite field.

### 6.1.1 The Ring of Integers Modulo $n$

In Sage, the ring $\mathbb{Z}/n\mathbb{Z}$ of integers modulo $n$ is defined using the constructor `IntegerModRing` (or, more simply, `Integers`). All objects constructed using this constructor and those derived from them are systematically reduced modulo $n$, and so have a canonical (or normal) form: that is to say, two variables representing the same value modulo $n$ also have the same internal representation. In certain very special situations, it may be more efficient to delay these reductions modulo $n$; for example, if one multiplies matrices with such coefficients, one would then rather work with integers, and carry out the reductions modulo $n$ "by hand"

using `a % n`. Note that the modulus $n$ does not appear explicitly in the displayed value:

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

One consequence of this is that when one uses "cut-and-paste" to copy integers modulo $n$, one loses information about $n$. Given a variable whose value is an integer modulo $n$, one can recover information about $n$ using the methods `base_ring` or `parent`, and the value of $n$ using the method `characteristic`:

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

The basic operations (addition, subtraction and multiplication) are overloaded for integers modulo $n$, and call the appropriate functions; also, integers are converted automatically when one of the operands is an integer modulo $n$:

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

For inversion, $1/a \bmod n$, or division, $b/a \bmod n$, Sage carries out the operation if possible; otherwise, i.e., when $a$ and $n$ have a nontrivial common factor, a `ZeroDivisionError` is raised:

```
sage: 1/(a+1)
4
sage: 1/a
Traceback (most recent call last):
  ...
ZeroDivisionError: Inverse does not exist.
```

To obtain the value of $a$ as an integer from its residue $a \bmod n$, one can use the method `lift` or even `ZZ`:

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

The *additive order* of $a$ modulo $n$ is the smallest integer $k > 0$ such that $ka = 0 \bmod n$. It is equal to $k = n/g$, where $g = \gcd(a, n)$, and is given by the method `additive_order` (we will see later that one can also use `Mod` or `mod` to define integers 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]
```

The *multiplicative order* of $a$ modulo $n$, for $a$ coprime[1] to $n$, is the smallest integer $k > 0$ such that $a^k = 1 \bmod n$. (If $a$ had a common divisor $p$ with $n$, then

[1] "coprime" and "relatively prime" are synonymous.

$a^k \bmod n$ would be a multiple of $p$ for all $k$.) If this multiplicative order equals $\varphi(n)$, which is the order of the multiplicative group modulo $n$, one says that $a$ is a *generator* of this group. Thus for $n = 15$, there is no generator, since the maximal order is $4 < 8 = \varphi(15)$:

```
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]]
```

Here is an example with $n = p$ prime, where 3 is a generator:

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

An important operation on $\mathbb{Z}/n\mathbb{Z}$ is *modular exponentiation*, which means to calculate $a^e \bmod n$. The RSA crypto-system relies on this operation. To calculate $a^e \bmod n$, the most efficient algorithms require of the order of $\log e$ multiplications or squarings modulo $n$. It is crucial to reduce all calculations modulo $n$ systematically, and not compute $a^e$ first as an integer, as the following example shows:

```
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
```

### 6.1.2 Finite Fields

Finite fields[2] are defined using the constructor `FiniteField`, or more simply `GF`. As well as constructing *prime fields* `GF(p)` with $p$ prime, one can construct *non-prime finite fields* `GF(q)` with $q = p^k$, where $p$ is prime and $k > 1$ an integer. As with rings, objects created in such a field have a canonical representation, and reduction is carried out at each arithmetic operation. Finite fields have the same properties as rings (§6.1.1), with in addition the possibility of inverting each non-zero element:

```
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]
```

A non-prime finite field $\mathbb{F}_{p^k}$ with $p$ prime and $k > 1$ is isomorphic to the quotient ring of polynomials in $\mathbb{F}_p[x]$ modulo a monic irreducible polynomial $f$ of degree $k$. In this case, Sage will provide a name for the *generator* of the field, that is, the variable $x$, or the user can provide a name:

[2]The finite field with $q$ elements is either denoted $\mathbb{F}_q$, or GF($q$) (where "GF" stands for "Galois Field"). Here we will use the notation $\mathbb{F}_q$ for the mathematical object, and the notation GF($q$) in Sage code.

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

Here, Sage has automatically chosen the polynomial $f$:

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

Field elements are thus represented by polynomials in the generator $x$, $a_{k-1}x^{k-1} + \cdots + a_1 x + a_0$, with coefficients $a_i$ which are elements of $\mathbb{F}_p$:

```
sage: Set([r for r in R])
{0, 1, 2, x, x + 1, x + 2, 2*x, 2*x + 1, 2*x + 2}
```

One can also make Sage use a specific irreducible polynomial $f$:

```
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
```

Be careful: even though the two fields `R` and `R2` created above are both isomorphic to $\mathbb{F}_9$, Sage provides no isomorphism between them automatically:

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

### 6.1.3 Rational Reconstruction

The problem of *rational reconstruction* is a useful application of modular methods. Given a residue $a$ modulo $m$, it involves finding a "small" rational number $x/y$ such that $x/y \equiv a \bmod m$. If one knows that such a small rational number exists, instead of computing $x/y$ directly as a rational number, one may instead compute $x/y$ modulo $m$, which gives the residue $a$, and then one recovers $x/y$ via rational reconstruction. This second approach is often more efficient, since one has replaced computations with rationals, possibly involving costly gcd calculations, by modular calculations.

LEMMA. Let $a, m \in \mathbb{N}$, with $0 < a < m$. There exists at most one pair of coprime integers $x, y \in \mathbb{Z}$ such that $x/y \equiv a \bmod m$ with $0 < |x|, y \leq \sqrt{m/2}$.

Such a pair $x, y$ does not always exist: for example, take $a = 2$ and $m = 5$. The rational reconstruction algorithm is based on the extended Euclidean algorithm. The extended gcd of $m$ and $a$ computes a sequence of integers $a_i = \alpha_i m + \beta_i a$, where the $a_i$ are decreasing, and the coefficients $\alpha_i, \beta_i$ increase in absolute value. It therefore suffices to stop as soon as $|a_i|, |\beta_i| \leq \sqrt{m/2}$, and the solution is then $x/y = a_i/\beta_i$. This algorithm is implemented in the Sage function `rational_reconstruction`, which returns $x/y$ when a solution exists, raising an error if not:

```
sage: rational_reconstruction(411,1000)
```

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

To illustrate the use of rational reconstruction, consider the computation of the Harmonic numbers $H_n = 1 + 1/2 + \cdots + 1/n$. A naive calculation using rational numbers would be as follows:

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

Now we know that $H_n$ can be written in the form $p_n/q_n$ with integers $p_n, q_n$, where $q_n = \mathrm{lcm}(1, 2, \ldots, n)$. We also know that $H_n \leq \log n + 1$, which allows us to bound $p_n$. This leads to the following function, which finds $H_n$ using modular arithmetic and rational reconstruction:

```
sage: def harmonic_mod(n,m):
....:     return add([1/x % m for x in range(1,n+1)])
sage: 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)
```

In this example, the function `harmonic2` is no more efficient than the original function `harmonic`, but it illustrates the method. It is not always necessary to know a rigorous bound for $x$ and $y$, as a rough estimate "by eye" will suffice, provided that one is able to verify easily that $x/y$ is the correct solution.

One can generalise the method of rational reconstruction to handle numerators $x$ and denominators $y$ of different sizes; see for example Section 5.10 of the book [vzGG03].

### 6.1.4 The Chinese Remainder Theorem

Another useful application of modular arithmetic involves the use of the *Chinese Remainder Theorem*, or CRT, commonly called "Chinese remaindering". Given two coprime moduli $m$ and $n$, and two residue classes $a \bmod m$ and $b \bmod n$, we seek an integer $x$ such that $x \equiv a \bmod m$ and $x \equiv b \bmod n$. The Chinese Remainder Theorem enables us to recover $x$ uniquely modulo the product $mn$. To see how this works, one deduces from $x \equiv a \bmod m$ that $x$ has the form $x = a + \lambda m$ with $\lambda \in \mathbb{Z}$. Substituting into $x \equiv b \bmod n$, one obtains $\lambda \equiv \lambda_0 \bmod n$, where $\lambda_0 = (b - a)/m \bmod n$. Hence $x = x_0 + \mu nm$, where $x_0 = a + \lambda_0 m$, and $\mu$ is an arbitrary integer.

Here we have presented the simplest variant of the Chinese Remainder Theorem. One can also consider the case of several moduli $m_1, m_2, \ldots, m_k$. The Sage command for finding $x_0$, given $a, b, m, n$, is `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
```

Let us return to the computation of $H_n$. We first compute $H_n \bmod m_i$ for $i = 1, 2, \ldots, k$, and then obtain $H_n \bmod m_1 \cdots m_k$ by Chinese remaindering, finally recovering the value of $H_n$ by rational reconstruction:

```
sage: def harmonic3(n):
....:     q = lcm(range(1,n+1))
....:     pmax = RR(q*(log(n)+1))
....:     B = ZZ(2*pmax^2)
....:     a = 0; m = 1; 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
```

The Sage function `crt` may also be used when the moduli $m$ and $n$ are not coprime. If $g = \gcd(m, n)$, then a solution exists if and only if $a \equiv b \bmod g$:

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

A more complicated application of the Chinese Remainder Theorem is given in Exercise 23.

## 6.2 Primality

Testing whether an integer is prime is a fundamental operation for a symbolic computer software package. Even if the user is not aware of it, such tests are carried out thousands of times per second by the software. For example, to factor a polynomial in $\mathbb{Z}[x]$, one starts by factoring it in $\mathbb{F}_p[x]$ for some prime number $p$, and one must therefore find a suitable prime.

| Useful commands | |
|---|---|
| Ring of integers modulo $n$ | `IntegerModRing`$(n)$ |
| Finite field with $q$ elements | `GF`$(q)$ |
| Pseudo-primality test | `is_pseudoprime`$(n)$ |
| Primality test | `is_prime`$(n)$ |

TABLE 6.1 – Review.

There are two main classes of primality test. The most efficient are *pseudo-primality* tests, and are in general based on forms of Fermat's Little Theorem, which says that if $p$ is prime, then every integer $a$ with $0 < a < p$ is an element of the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^*$, and hence $a^{p-1} \equiv 1 \bmod p$. One uses small values of $a$ $(2, 3, \ldots)$ to speed up the computation of $a^{p-1} \bmod p$. If $a^{p-1} \not\equiv 1 \bmod p$, then $p$ is certainly not prime. If $a^{p-1} \equiv 1 \bmod p$, one cannot conclude either that $p$ is or is not prime; we say that $p$ is a (Fermat) *pseudo-prime to base* $a$. The intuition is that an integer $p$ which is a pseudo-prime to many bases has a greater chance of being prime (but see below). Pseudo-primality tests share the property that when they return the verdict `False`, the number is certainly composite, whereas when they return `True`, no definite conclusion is possible.

The second class consists of *true primality* tests. These tests always return a correct answer, but can be less efficient than pseudo-primality tests, especially for numbers that are pseudo-primes to many bases, and in particular for actual primes. Many software packages only provide pseudo-primality tests, despite the name of the corresponding function (`isprime`, for example) sometimes leading the user to believe that a true primality test is provided. Sage provides two different functions: `is_pseudoprime` for pseudo-primality, and `is_prime` for true primality:

```
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
```

We see in this example that the primality test is more costly; when possible, therefore, one prefers to use `is_pseudoprime`.

Some primality testing algorithms provide a *certificate*, which allows an independent subsequent verification of the result, often more efficiently than the test itself. Sage does not provide such a certificate in the current release, but one can construct one using Pocklington's Theorem:

THEOREM. Let $n > 1$ be an odd integer such that $n - 1 = FR$, with $F \geq \sqrt{n}$. If for each prime factor $p$ of $F$, there exists $a$ such that $a^{n-1} \equiv 1 \bmod n$ and $a^{(n-1)/p} - 1$ is coprime to $n$, then $n$ is prime.

Consider for example $n = 2^{31} - 1$. The factorisation of $n - 1$ is $2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$. One can take $F = 151 \cdot 331$, and $a = 3$ satisfies the condition for both

factors $p = 151$ and $p = 331$. Hence it suffices to prove the primality of 151 and 331 in order to deduce that $n$ is prime. This test uses modular exponentiation in an important way.

**Carmichael numbers**

*Carmichael numbers* are composite integers $n$ that are pseudo-primes to all bases coprime to $n$. Fermat's Little Theorem is insufficient to distinguish these from primes, however many bases are tested. The smallest Carmichael number is $561 = 3 \cdot 11 \cdot 17$. A Carmichael number must have at least three prime factors: for suppose that $n = pq$ is a Carmichael number, with $p, q$ primes and $p < q$; by definition of Carmichael numbers, if $a$ is a primitive root modulo $q$ then $a^{n-1} \equiv 1$ modulo $n$ implies that the same congruence also holds modulo $q$, and hence that $n-1$ is a multiple of $q-1$. Then $n$ must be of the form $q + \lambda q(q-1)$, since it is a multiple of $q$ and $n-1$ is a multiple of $q-1$; now $n = pq$ implies $p = \lambda(q-1) + 1$, which contradicts $p < q$. If $n = pqr$, then $n$ is a Carmichael number if $a^{n-1} \equiv 1 \bmod p$, and similarly modulo $q$ and $r$, since then the Chinese Remainder Theorem implies that $a^{n-1} \equiv 1 \bmod n$. So a sufficient condition is that $n-1$ is divisible by each of $p-1$, $q-1$ and $r-1$:

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

**Exercise 20.** Write a Sage function to count the Carmichael numbers $n = pqr \leq N$, with $p, q, r$ distinct odd primes. How many do you find for $N = 10^4, 10^5, 10^6, 10^7$? (Richard Pinch has counted 20138200 Carmichael numbers less than $10^{21}$.)

Finally, in order to repeat an operation on all prime numbers in an interval, it is better to employ the construction `prime_range`, which constructs a table of primes using a sieve, than to simply use a loop with `next_probable_prime` or `next_prime`:

```
sage: def count_primes1(n):
....:     return add([1 for p in range(n+1) if is_prime(p)])
sage: %timeit count_primes1(10^5)
5 loops, best of 3: 674 ms per loop
```

The function is faster if one uses `is_pseudoprime` instead of `is_prime`:

```
sage: def count_primes2(n):
....:     return add([1 for p in range(n+1) if is_pseudoprime(p)])
sage: %timeit count_primes2(10^5)
5 loops, best of 3: 256 ms per loop
```

In this example, it is worth using a loop rather than constructing a list of $10^5$ elements, and again `is_pseudoprime` is faster than `is_prime`:

```
sage: def count_primes3(n):
....:     s = 0; p = 2
```

```
....:     while p <= n: s += 1; p = next_prime(p)
....:     return s
sage: %timeit count_primes3(10^5)
5 loops, best of 3: 49.2 ms per loop
sage: def count_primes4(n):
....:     s = 0; p = 2
....:     while p <= n: s += 1; p = next_probable_prime(p)
....:     return s
sage: %timeit count_primes4(10^5)
5 loops, best of 3: 48.6 ms per loop
```

Using the iterator `prime_range` is faster still:

```
sage: def count_primes5(n):
....:     s = 0
....:     for p in prime_range(n): s += 1
....:     return s
sage: %timeit count_primes5(10^5)
125 loops, best of 3: 2.67 ms per loop
```

## 6.3 Factorisation and Discrete Logarithms

One says that an integer $a$ is a square, or a quadratic residue, modulo $n$ if there exists $x$ such that $a \equiv x^2 \bmod n$. If not, one says that $a$ is a quadratic non-residue[3] modulo $n$. When $n = p$ is prime, there is a test to decide efficiently whether $a$ is a quadratic residue, using the computation of the Jacobi symbol of $a$ and $p$, denoted $(a|p)$, which takes the values $\{-1, 0, 1\}$, where $(a|p) = 0$ when $a$ is a multiple of $p$, and $(a|p) = 1$ (respectively, $(a|p) = -1$) when $a$ is (respectively, is not) a square modulo $p$. The complexity of computing the Jacobi symbol $(a|n)$ is essentially the same as that of computing the gcd of $a$ and $n$, namely $O(M(\ell)\log\ell)$ where $\ell$ is the size of $n$, and $M(\ell)$ is the cost of multiplying two integers of size $\ell$. However, implementations of Jacobi symbols — as of gcds — do not all have this complexity (here, `a.jacobi(n)` computes $(a|n)$):

```
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
```

When $n$ is composite, finding solutions to $x^2 \equiv a \bmod n$ is as hard as factorising $n$. Moreover, the Jacobi symbol, which is relatively simple to compute, only gives partial information: if $(a|n) = -1$ then there is no solution, since the existence of a solution implies $(a|p) = 1$ for all prime factors $p$ of $n$, hence $(a|n) = 1$; but $(a|n) = +1$ does not imply that $a$ is a square modulo $n$ when $n$ is composite.

[3]This terminology is traditional, though "non-quadratic residue" would be more logical.

Let $n$ be a positive integer, let $g$ be a *generator* of the multiplicative group modulo $n$ (we assume here that $n$ is such that this group is cyclic), and let $a$ be coprime to $n$. By definition of the fact that $g$ is a generator, there is an integer $x$ such that $g^x = a \bmod n$. The *discrete logarithm problem* consists of finding such an integer $x$. The `log` method gives a solution to this problem:

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

The best known algorithms for computing discrete logarithms have the same order of complexity, as a function of the size of $n$, as those for factoring $n$. However, the current implementation of discrete logarithms in Sage is not very efficient:

```
sage: p = 10^37+43; a = mod(17,p)
sage: time r = a.log(2)
CPU times: user 1min 32s, sys: 64 ms, total: 1min 32s
Wall time: 1min 34s
```

**Aliquot sequences**

The *aliquot sequence* associated to a positive integer $n$ is the recurrent sequence $(s_k)$ defined by: $s_0 = n$ and $s_{k+1} = \sigma(s_k) - s_k$, where $\sigma(s_k)$ is the sum of the positive divisors of $n$, i.e., $s_{k+1}$ is the sum of the *proper* divisors of $s_k$, excluding $s_k$ itself. The iteration stops when $s_k = 1$, so $s_{k-1}$ is prime, or when the sequence $(s_k)$ enters a cycle. For example, starting from $n = 30$ one obtains:

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

When the cycle has length one, we say that the starting integer is *perfect*, for example $6 = 1 + 2 + 3$ and $28 = 1 + 2 + 4 + 7 + 14$ are perfect. When the cycle has length two, the two integers in the cycle are called *amicable* and form an *amicable pair*, for example 220 and 284. When the cycle has length three or more, the integers in the cycle are called *sociable*.

**Exercise 21.** Calculate the aliquot sequence starting with 840, take the 5 first and 5 last terms, and draw the graph of $\log_{10} s_k$ as a function of $k$ (you can use the function `sigma`).

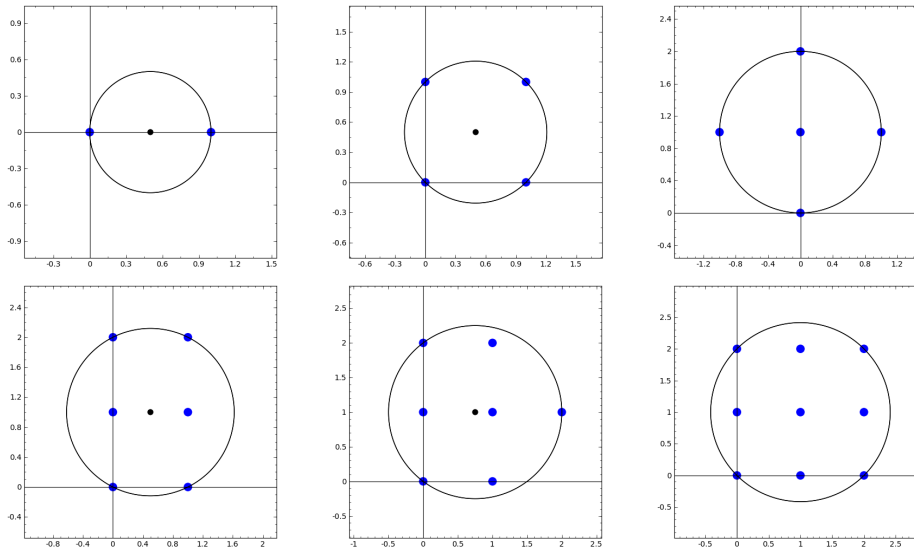
## 6.4 Applications

### 6.4.1 The Constant $\delta$

The constant $\delta$ is a two-dimensional generalisation of Euler's constant $\gamma$. It is defined as follows:

$$\delta = \lim_{n\to\infty} \left( \sum_{k=2}^{n} \frac{1}{\pi r_k^2} - \log n \right), \tag{6.1}$$

where $r_k$ is the radius of the smallest closed disc in the affine plane $\mathbb{R}^2$ containing at least $k$ points of $\mathbb{Z}^2$. For example, $r_2 = 1/2$, $r_3 = r_4 = \sqrt{2}/2$, $r_5 = 1$, $r_6 = \sqrt{5}/2$, $r_7 = 5/4$, and $r_8 = r_9 = \sqrt{2}$:



**Exercise 22** (Masser-Gramain constant)**.** 1. Write a function which takes as input a positive integer $k$, and returns the radius $r_k$ and the centre $(x_k, y_k)$ of a minimal disc, of radius $r_k$, containing at least $k$ points of $\mathbb{Z}^2$. You may assume that $r_k < \sqrt{k/\pi}$.

2. Write a function which draws the circle with centre $(x_k, y_k)$ and radius $r_k$, together with $m \geq k$ points of $\mathbb{Z}^2$, as above.

3. Using the bounding inequalities

$$\frac{\sqrt{\pi(k-6)+2} - \sqrt{2}}{\pi} < r_k < \sqrt{\frac{k-1}{\pi}}, \tag{6.2}$$

calculate an approximation of $\delta$ with an error at most 0.3.

### 6.4.2 Computation of a Multiple Integral

This application was inspired by the article [Bea09]. Let $k$ and $n_1, n_2, \ldots, n_k$ be non-negative integers. We wish to compute the integral

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

where the domain of integration is defined by $V = \{x_1 \geq x_2 \geq \cdots \geq x_k \geq 0, x_1 + \cdots + x_k \leq 1\}$. For example, for $k = 2$, $n_1 = 3$, $n_2 = 5$, one finds the value

$$I = \int_{x_2=0}^{1/2} \int_{x_1=x_2}^{1-x_2} x_1^3 x_2^5 \, \mathrm{d}x_1 \, \mathrm{d}x_2 = \frac{13}{258048}.$$

**Exercise 23.** Given that $I$ is a rational number, develop an algorithm using rational reconstruction and/or the Chinese Remainder Theorem to calculate $I$. Implement the algorithm in Sage, and apply it to the case $[n_1, \ldots, n_{31}] =$

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