7 Polynomials

This chapter will discuss univariate polynomials and related objects, mainly rational functions and formal power series. We will first see how to perform with Sage some transformations like the Euclidean division of polynomials, factorisation into irreducible polynomials, root isolation, or partial fraction decomposition. All these transformations will take into account the ring or field where the polynomial coefficients live: Sage enables us to compute in polynomial rings $A[x]$, in their quotient $A[x]/\langle P(x)\rangle$, in fraction fields $K(x)$ or in formal power series rings $A[[x]]$ for a whole set of base rings.

Operations on polynomials also have some unexpected applications. How to automatically guess the next term of the sequence

$$1, 1, 2, 3, 8, 11, 39...?$$

For example, using the Padé approximation of rational functions, presented in Section 7.4.3! How to easily get a series expansion of the solutions of the equation $e^{xf(x)} = f(x)$? An answer can be found in Section 7.5.3.

We assume in general that the reader is used to playing with polynomials and rational functions at the first year university level. However, we will discuss more advanced subjects. How to prove that the solutions of the equation $x^5 - x - 1$ cannot be expressed by radicals? It suffices to compute its Galois group, as explained in Section 7.3.4. The corresponding parts are not used elsewhere in this book, and the reader may skip them. Finally, this chapter gives a few examples with algebraic and p-adic numbers.

Multivariate polynomials are discussed in Chapter 9.

Playing with polynomial rings, $R = A[x]$	
construction (dense repr.)	`R.<x> = A[]` *or* `R.<x> = PolynomialRing(A)`
e.g. $\mathbb{Z}[x]$, $\mathbb{Q}[x]$, $\mathbb{R}[x]$, $\mathbb{Z}/n\mathbb{Z}[x]$	`ZZ['x'], QQ['x'], RR['x'], Integers(n)['x']`
construction (sparse repr.)	`R.<x> = PolynomialRing(A, sparse=True)`
accessing the base ring A	`R.base_ring()`
accessing the variable x	`R.gen()` *or* `R.0`
tests (integral, noetherian...)	`R.is_integral_domain(), R.is_noetherian(), ...`

TABLE 7.1 – Polynomial rings.

7.1 Polynomial Rings

7.1.1 Introduction

We have seen in Chapter 2 how to perform computations on *symbolic expressions*, elements of the "symbolic ring" SR. Some of the methods available for these expressions, for example degree, are suited for polynomials:

```
sage: x = var('x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print("{} is of degree {}".format(p, p.degree(x)))
(x^4 - 1)*(2*x + 1)*(x + 2) is of degree 6
```

In some computer algebra systems, like Maple or Maxima, representing polynomials as particular symbolic expressions is the usual way to play with them. Like Axiom, Magma or MuPAD, Sage also allows to manipulate polynomials in a more algebraic way, and "knows" how to compute in rings like $\mathbb{Q}[x]$ or $\mathbb{Z}/4\mathbb{Z}\,[x, y, z]$.

Hence, to reproduce the above example in a well-defined polynomial ring, we assign to the Python variable x the *unknown of the polynomial ring in x with rational coefficients*, given by polygen(QQ, 'x'), instead of the *symbolic variable x* returned[1] by var('x'):

```
sage: x = polygen(QQ, 'x'); p = (2*x+1)*(x+2)*(x^4-1)
sage: print("{} is of degree {}".format(p, p.degree()))
2*x^6 + 5*x^5 + 2*x^4 - 2*x^2 - 5*x - 2 is of degree 6
```

We notice that the polynomial is automatically expanded. The "algebraic" polynomials are always represented in normal form. This is a crucial difference with respect to the polynomials in SR. In particular, when two algebraic polynomials are mathematically equal, their computer representation is the same, and a comparison coefficient by coefficient is enough to check their equality.

The available functions on algebraic polynomials are much wider and more efficient that those on (polynomial) symbolic expressions.

7.1.2 Building Polynomial Rings

Polynomials in Sage, like many other algebraic objects, have in general coefficients in a commutative ring. This is the point of view of this book, however most of the examples will have coefficients in a field. In the whole chapter, the letters A and K respectively correspond to a commutative ring and to a field.

The first step to perform a computation in an algebraic structure R is often to build R itself. We build $\mathbb{Q}[x]$ with

```
sage: R = PolynomialRing(QQ, 'x')
sage: x = R.gen()
```

The `'x'` on the first line is a character string, which is the name of the indeterminate, or *generator* of the ring. The `x` on the second line is a Python variable in which one stores the generator; using the same name makes the code easier to read. The object stored in the variable `x` represents the polynomial $x \in \mathbb{Q}[x]$. Its parent (the *parent* of a Sage object is the algebraic structure "from which it comes", see §5.1) is the ring `QQ['x']`:

```
sage: x.parent()
Univariate Polynomial Ring in x over Rational Field
```

The polynomial $x \in \mathbb{Q}[x]$ is considered different from $x \in A[x]$ for a base ring $A \neq \mathbb{Q}$, and also different from those, like $t \in \mathbb{Q}[t]$, whose indeterminate has a different name.

The expression `PolynomialRing(QQ, 't')` might also be written `QQ['t']`. We often combine this abbreviation with the construction `S.<g> = ...`, which simultaneously assigns a structure to the variable `S` and its generator to the variable `g`. The construction of the ring $\mathbb{Q}[x]$ and of its indeterminate then reduces to `R.<x> = QQ[]`. The form `x = polygen(QQ, 'x')` seen above is equivalent to

```
sage: x = PolynomialRing(QQ, 'x').gen()
```

Let us mention that we can choose between several memory representations when we construct a polynomial ring. The differences between representations are discussed in §7.6.

Exercise 24 (Variables and indeterminates).

1. How would you define `x` and `y` to obtain the following results?

   ```
   sage: x^2 + 1
   y^2 + 1
   sage: (y^2 + 1).parent()
   Univariate Polynomial Ring in x over Rational Field
   ```

2. After the instructions

   ```
   sage: Q.<x> = QQ[]; p = x + 1; x = 2; p = p + x
   ```

   what is the value of `p`?

Polynomials with polynomial coefficients

In Sage, we can define polynomial rings with coefficients in any commutative ring, including another polynomial ring. But beware that rings $A[x][y]$ constructed this way differ from the true polynomial rings with several variables like $A[x, y]$. The latter, presented in Chapter 9, are better suited for usual computations. Indeed, working in $A[x][y][\dots]$ introduces an asymmetry between the variables.

However, in some cases we precisely want to have one main variable, and the other variables as parameters. The `polynomial` method of multivariate polynomials allows us to isolate one variable, more or less like the `collect` method of symbolic expressions. For example, to compute the reciprocal of a given polynomial with respect to one of its variables:

```
sage: R.<x,y,z,t> = QQ[]; p = (x+y+z*t)^2
sage: p.polynomial(t).reverse()
(x^2 + 2*x*y + y^2)*t^2 + (2*x*z + 2*y*z)*t + z^2
```

Here, `p.polynomial(t)` creates a univariate polynomial in the variable `t` and with coefficients in `QQ[x,y,z]`, to which we then apply the `reverse` method.

The other conversions between $A[x, y, \dots]$ and $A[x][y][\dots]$ work as expected:

```
sage: x = polygen(QQ); y = polygen(QQ[x], 'y')
sage: p = x^3 + x*y + y + y^2; p
y^2 + (x + 1)*y + x^3
sage: q = QQ['x,y'](p); q
x^3 + x*y + y^2 + y
sage: QQ['x']['y'](q)
y^2 + (x + 1)*y + x^3
```

7.1.3 Polynomials

Creation and Basic Arithmetic. After the instruction `R.<x> = QQ[]`, the expressions constructed from `x` and rational constants with operations `+` and `*` are elements of $\mathbb{Q}[x]$. For example, in `p = x + 2`, Sage automatically determines that the value of the variable `x` and the integer 2 can be seen as elements of $\mathbb{Q}[x]$. The addition routine of polynomials in $\mathbb{Q}[x]$ is thus called; it builds and returns the polynomial $x + 2 \in \mathbb{Q}[x]$.

Another way to build a polynomial is to enumerate its coefficients:

```
sage: def rook_polynomial(n, var='x'):
....:     return ZZ[var]([binomial(n, k)^2 * factorial(k)
....:                                    for k in (0..n) ])
```

The above function constructs polynomials whose coefficient of x^k is the number of ways to put k rooks on an $n \times n$ chessboard, so that two rooks cannot

[1] A little difference here: while `var('x')` is equivalent to `x = var('x')` in interactive use, `polygen(QQ, 'x')` alone does not change the value of the Python variable `x`.

Accessing data, syntactic operations	
indeterminate x	`p.variables()`, `p.variable_name()`
coefficient of x^k	`p[k]`
leading coefficient	`p.leading_coefficient()`
degree	`p.degree()`
list of coefficients	`p.list()` *or* `p.coefficients(sparse=False)`
list of *non-zero* coefficients	`p.coefficients()`
dictionary degree $\mapsto$ coefficient	`p.dict()`
tests (monic, constant...)	`p.is_monic()`, `p.is_constant()`, ...
Basic arithmetic	
operations $p+q$, $p-q$, $p \times q$, p^k	`p + q`, `p - q`, `p * q`, `p^k`
substitution $x := a$	`p(a)` *or* `p.subs(a)`
derivative	`p.derivative()` *or* `p.diff()`
Transformations	
transformation of coefficients	`p.map_coefficients(f)`
change of base ring $A[x] \to B[x]$	`p.change_ring(B)` *or* `B['x'](p)`
reciprocal polynomial	`p.reverse()`

TABLE 7.2 – Basic operations on polynomials $p, q \in A[x]$.

capture each other; this explains the name of the function. The parentheses after `ZZ[var]` force the conversion of a given object into an element of this ring. The conversion of a list $[a_0, a_1, \dots]$ into an element of `ZZ['x']` yields the polynomial $a_0 + a_1\,x + \cdots \in \mathbb{Z}[x]$.

Global View on Polynomial Operations. The elements of a polynomial ring are represented by Python objects from the class `Polynomial`, or from derived classes. The main operations[2] available for these objects are summarised in Tables 7.2 to 7.5. For example, we query the degree of a polynomial with the `degree` method. Similarly, `p.subs(a)` or simply `p(a)` yields the value of p at the point a, but also computes the composition $p \circ a$ when a itself is a polynomial, and more generally evaluates a polynomial of $A[x]$ at an element of an A-algebra:

```
sage: p = R.random_element(degree=4) # a random polynomial
sage: p
-4*x^4 - 52*x^3 - 1/6*x^2 - 4/23*x + 1
sage: p.subs(x^2)
-4*x^8 - 52*x^6 - 1/6*x^4 - 4/23*x^2 + 1
sage: p.subs(matrix([[1,2],[3,4]]))
```

[2]There are many other operations. Those tables omit too advanced functionalities, some specialised variants of methods we mention, and numerous methods common to all ring elements, and even to all Sage objects, which have no particular interest on polynomials. Note however that some specialised methods (for example `p.rescale(a)`, equivalent to `p(a*x)`) are often more efficient than more general methods that could replace them.

```
[-375407/138  -273931/69]
[ -273931/46  -598600/69]
```

We will come back to the content of the last two tables in Sections 7.2.1 and 7.3.

Change of Ring. The exact list of available operations, their meaning and their efficiency heavily depend on the base ring. For example, the polynomials in `GF(p)['x']` have a method `small_roots` which returns their small roots with respect to the characteristic p; those in `QQ['x']` do not have such a method, since it makes no sense. The `factor` method exists for all polynomials, but raises an exception `NotImplementedError` for polynomials with coefficients in `SR` or in $\mathbb{Z}/4\mathbb{Z}$. This exception means that this operation is not available in Sage for this kind of object, despite having a mathematical meaning.

It is very useful to be able to juggle the different rings of coefficients on which we might consider a given polynomial. Applied to a polynomial in $A[x]$, the method `change_ring(B)` returns its image in $B[x]$, when a natural method to convert the coefficients exists. The conversion is often given by a canonical morphism from A to B: in particular, `change_ring` might be used to extend the base ring to gain additional algebraic properties. Here for example, the polynomial p is irreducible over the rationals, but it factors on $\mathbb{R}$:

```
sage: x = polygen(QQ)
sage: p = x^2 - 16*x + 3
sage: p.factor()
x^2 - 16*x + 3
sage: p.change_ring(RDF).factor()
(x - 15.810249675906654) * (x - 0.18975032409334563)
```

The `RDF` domain is that of “machine floating-point numbers”, and is discussed in Chapter 11. The obtained factorisation is approximate; it does not suffice to recover the original polynomial. To represent real roots of polynomials with integer coefficients in a way that enables exact computations, we use the domain `AA` of real algebraic numbers. We will see some examples in the following sections.

The same method `change_ring` allows to reduce a polynomial in $\mathbb{Z}[x]$ modulo a prime number:

```
sage: p.change_ring(GF(3))
x^2 + 2*x
```

Conversely, if $B \subset A$ and if the coefficients of p are in fact in B, we also call `change_ring` to recover p in $B[x]$.

Iteration. More generally, one often needs to apply a given transformation to all coefficients of a polynomial. The method `map_coefficients` is designed for this. Applied to a polynomial $p \in A[x]$ with parameter a function f, it returns the polynomial obtained by applying f to all *non-zero* coefficients of p. In general, f is an anonymous function defined using the `lambda` construction (see §3.3.2). Here is for example how one can compute the conjugate of a polynomial with complex coefficients:

Divisibility and Euclidean division	
divisibility test $p \mid q$	`p.divides(q)`
multiplicity of a divisor $q^k \mid p$	`k = p.valuation(q)`
Euclidean division $p = qd + r$	`q, r = p.quo_rem(d)` *or* `q = p//d, r = p%d`
pseudo-division $a^k p = qd + r$	`q, r, k = p.pseudo_divrem(d)`
greatest common divisor	`p.gcd(q), gcd([p1, p2, p3])`
least common multiple	`p.lcm(q), lcm([p1, p2, p3])`
extended gcd $g = up + vq$	`g, u, v = p.xgcd(q)` *or* `xgcd(p, q)`
"Chinese remainder" $c \equiv a \bmod p$, $c \equiv b \bmod q$	`c = crt(a, b, p, q)`
Miscellaneous	
interpolation $p(x_i) = y_i$	`p = R.lagrange_polynomial([(x1,y1), ...])`
content of $p \in \mathbb{Z}[x]$	`p.content()`

TABLE 7.3 – Polynomial arithmetic.

```
sage: QQi.<myI> = QQ[I]    # myI is the i of QQi, I that of SR
sage: R.<x> = QQi[]; p = (x + 2*myI)^3; p
x^3 + 6*I*x^2 - 12*x - 8*I
sage: p.map_coefficients(lambda z: z.conjugate())
x^3 - 6*I*x^2 - 12*x + 8*I
```

Here, we can also write `p.map_coefficients(conjugate)`, since `conjugate(z)` has the same effect as `z.conjugate` for $z \in \mathbb{Q}[i]$. Calling explicitly a method of the object `z` is more robust: the code then works for all objects having a `conjugate()` method, and only for those.

7.2 Euclidean Arithmetic

Apart from the sum and product, the most elementary operations on polynomials are the Euclidean division and the greatest common divisor computation. The corresponding operators and methods (Table 7.3) mimic those on integers. However, quite often, these operations are hidden by an additional abstraction layer: quotient of rings (§7.2.2) where each arithmetic operation involves an implicit Euclidean division, rational functions (§7.4) whose normalisation implies some gcd computations...

7.2.1 Divisibility

Divisions. The Euclidean division works in a field, and more generally in a commutative ring when the leading coefficient of the divisor is invertible, since this coefficient is the only one from the base ring by which it is required to divide:

```
sage: R.<t> = Integers(42)[]; (t^20-1) % (t^5+8*t+7)
22*t^4 + 14*t^3 + 14*t + 6
```

Operations on polynomial rings

The parents of polynomial objects, i.e., the rings $A[x]$, are themselves first class Sage objects. Let us briefly see how to use them.

A first family of methods enables us to construct particular polynomials, to draw random ones, or to enumerate families, here those of degree exactly 2 over $\mathbb{F}_2$:

```
sage: list(GF(2)['x'].polynomials(of_degree=2))
[x^2, x^2 + 1, x^2 + x, x^2 + x + 1]
```

We will call some of these methods in the examples of the next sections, to build objects on which we will work. Chapter 15 explains more generally how to enumerate finite sets with Sage.

Secondly, the system "knows" some basic facts for each polynomial ring. We can check whether a given object is a ring, if it is noetherian:

```
sage: A = QQ['x']
sage: A.is_ring() and A.is_noetherian()
True
```

or if $\mathbb{Z}$ is a sub-ring of $\mathbb{Q}[x]$, and for which values of n the ring $\mathbb{Z}/n\mathbb{Z}$ is integral:

```
sage: ZZ.is_subring(A)
True
sage: [n for n in range(20)
....:     if Integers(n)['x'].is_integral_domain()]
[0, 2, 3, 5, 7, 11, 13, 17, 19]
```

These capabilities largely rely on the Sage *category* system (see also §5.2.3). Polynomial rings belong to a number of "categories", like the category of sets, that of Euclidean rings, and many more:

```
sage: R.categories()
[Category of euclidean domains,
 Category of principal ideal domains,
 ...
 Category of sets with partial maps, Category of objects]
```

This reflects that any polynomial ring is also a set, a Euclidean domain, and so on. The system can thus automatically transfer to polynomial rings the general properties of objects from these different categories.

When the leading coefficient is not invertible, we can still define a *pseudo Euclidean division* (pseudo-division for short): let A be a commutative ring, $p, d \in A[x]$, and a the leading coefficient of d. Then there exists two polynomials $q, r \in A[x]$, with $\deg r < \deg d$, and an integer $k \leq \deg p - \deg d + 1$ such that

$$a^k p = qd + r.$$

The pseudo-division is given by the `pseudo_divrem` method.

To perform an exact division, we also use the Euclidean quotient operator `//`. Indeed, dividing by a non-constant polynomial with `/` returns a result of type rational function (see §7.4), or fails when this makes no sense:

```
sage: ((t^2+t)//t).parent()
Univariate Polynomial Ring in t over Ring of integers modulo 42
sage: (t^2+t)/t
Traceback (most recent call last):
...
TypeError: self must be an integral domain.
```

Exercise 25. Usually, in Sage, polynomials in $\mathbb{Q}[x]$ are represented on the monomial basis $(x^n)_{n\in\mathbb{N}}$. Chebyshev polynomials T_n, defined by $T_n(\cos\theta) = \cos(n\theta)$, form a family of orthogonal polynomials and thus a basis of $\mathbb{Q}[x]$. The first Chebyshev polynomials are

```
sage: x = polygen(QQ); [chebyshev_T(n, x) for n in (0..4)]
[1, x, 2*x^2 - 1, 4*x^3 - 3*x, 8*x^4 - 8*x^2 + 1]
```

Write a function taking as input an element of $\mathbb{Q}[x]$ and returning the coefficients of its decomposition in the basis $(T_n)_{n\in\mathbb{N}}$.

Exercise 26 (Division by increasing powers)**.** Let $n \in \mathbb{N}$ and $u, v \in A[x]$, with $v(0)$ invertible. Then a unique pair (q, r) of polynomials exists in $A[x]$ with $\deg q \leq n$ such that $u = qv + x^{n+1}r$. Write a function which computes q and r by an analogue of the Euclidean division algorithm. How would you perform this computation in the easiest way, using available Sage functions?

GCD. Sage is able to compute the gcd of polynomials over a field, thanks to the Euclidean structure of $K[x]$, but also on some other rings, including the integers:

```
sage: S.<x> = ZZ[]; p = 2*(x^10-1)*(x^8-1)
sage: p.gcd(p.derivative())
2*x^2 - 2
```

We can prefer the more symmetric expression `gcd(p,q)`, which yields the same result as `p.gcd(q)`. It is though slightly less natural in Sage since it is not a general mechanism: `gcd(p,q)` calls a function of two arguments, defined manually in the source code of Sage, and which calls in turn `p.gcd`. Only some usual methods have such an associated function.

The *extended gcd*, i.e., the computation of a Bézout relation

$$g = \gcd(p, q) = ap + bq, \qquad g, p, q, a, b \in K[x]$$

is given by `p.xgcd(q)`:

```
sage: R.<x> = QQ[]; p = x^5-1; q = x^3-1
sage: print("the gcd is %s = (%s)*p + (%s)*q" % p.xgcd(q))
the gcd is x - 1 = (-x)*p + (x^3 + 1)*q
```

The `xgcd` method also exists for polynomials in `ZZ['x']`, but beware: since $\mathbb{Z}[x]$ is not a principal ideal ring, the result is in general not a Bézout relation ($ap + bq$ might be an integer multiple of the gcd)!

7.2.2 Ideals and Quotients

Ideals of $A[x]$. The ideals of polynomial rings, and the quotients by these ideals, are represented by Sage objects built from the polynomial ring by the methods `ideal` and `quo`. The product of a tuple of polynomials by a polynomial ring is interpreted as an ideal:

```
sage: R.<x> = QQ[]
sage: J1 = (x^2 - 2*x + 1, 2*x^2 + x - 3)*R; J1
Principal ideal (x - 1) of Univariate Polynomial Ring in x
over Rational Field
```

We can multiply ideals, and reduce a polynomial modulo an ideal:

```
sage: J2 = R.ideal(x^5 + 2)
sage: ((3*x+5)*J1*J2).reduce(x^10)
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

The reduced polynomial remains in this case an element of `QQ['x']`. Another way is to construct the quotient by an ideal and project the elements on it. The parent of the projected element is then in the quotient ring. The `lift` method of the quotient elements converts them back into the initial ring.

```
sage: B = R.quo((3*x+5)*J1*J2) # quo automatically names 'xbar' which is
sage: B(x^10)                  #   the generator of B image of x
421/81*xbar^6 - 502/81*xbar^5 + 842/81*xbar - 680/81
sage: B(x^10).lift()
421/81*x^6 - 502/81*x^5 + 842/81*x - 680/81
```

If K is a field, then the ring $K[x]$ is principal: the ideals are represented during computations by a generator, all this being an algebraic language for the operations seen in §7.2.1. Its principal advantage is that quotient rings can be easily used in new constructions, here that of $\big(\mathbb{F}_5[t]/\langle t^2+3\rangle\big)[x]$:

```
sage: R.<t> = GF(5)[]; R.quo(t^2+3)['x'].random_element()
(3*tbar + 1)*x^2 + (2*tbar + 3)*x + 3*tbar + 4
```

Sage also allows building non principal ideals like in $\mathbb{Z}[x]$, however the available operations are then limited — except in case of multivariate polynomials over a field, which are the subject of Chapter 9.

Exercise 27. We define the sequence $(u_n)_{n\in\mathbb{N}}$ with the initial conditions $u_n = n + 7$ for $0 \le n < 1000$, and the linear recurrence relation

$$u_{n+1000} = 23u_{n+729} - 5u_{n+2} + 12u_{n+1} + 7u_n \qquad (n \ge 0).$$

Construction of ideals and quotient rings $Q = R/J$	
ideal $\langle u, v, w\rangle$	`R.ideal(u, v, w)` *or* `(u, v, w)*R`
reduction of p modulo J	`J.reduce(p)` *or* `p.mod(J)`
quotient ring R/J, $R/\langle p\rangle$	`R.quo(J)`, `R.quo(p)`
ring whose quotient gave Q	`Q.cover_ring()`
isomorphic number field	`Q.number_field()`

Elements of $K[x]/\langle p\rangle$	
lift (section of $R \twoheadrightarrow R/J$)	`u.lift()`
minimal polynomial	`u.minpoly()`
characteristic polynomial	`u.charpoly()`
matrix	`u.matrix()`
trace	`u.trace()`

TABLE 7.4 – Ideals and quotients.

Compute the last five digits of $u_{10^{10000}}$. *Hint:* we might look at the algorithm from §3.2.4. However, this algorithm is too expensive when the order of the recurrence is large. Introduce a clever quotient of polynomial rings to avoid this issue.

Algebraic Extensions. An important special case is the quotient of $K[x]$ by an irreducible polynomial to build an algebraic extension of K. The number fields, finite extensions of $\mathbb{Q}$, are represented by the objects `NumberField`, distinct from the quotients of `QQ['x']`. When this makes sense, the method `number_field` of a quotient of polynomial rings returns the corresponding number field. The interface of number fields, more complete than that of quotient rings, is beyond the scope of this book. The non-prime finite fields $\mathbb{F}_{p^k}$, built as algebraic extensions of the prime finite fields $\mathbb{F}_p$, are described in §6.1.

7.3 Factorisation and Roots

A third level after the elementary operations and the Euclidean arithmetic concerns the decomposition of a polynomial into a product of irreducible factors, or factorisation. It is maybe where computer algebra is the most useful!

7.3.1 Factorisation

Irreducibility Test. On the algebraic side, the simplest question about the factorisation of a polynomial is whether it is irreducible. Naturally, the answer depends on the base ring. The method `is_irreducible` tells if a polynomial is irreducible in its parent ring. For example, the polynomial $3x^2 - 6$ is irreducible over $\mathbb{Q}$, but not over $\mathbb{Z}$ (why?):

```
sage: R.<x> = QQ[]; p = 3*x^2 - 6
sage: p.is_irreducible(), p.change_ring(ZZ).is_irreducible()
(True, False)
```

Factorisation. The factorisation of an *integer* of hundreds or thousands of digits is a very hard problem. In contrast, factoring a *polynomial* of degree 1000 on $\mathbb{Q}$ or $\mathbb{F}_p$ — for small p — needs only a few seconds[3]:

```
sage: p = QQ['x'].random_element(degree=1000)
sage: %timeit p.factor()
1 loop, best of 3: 2.45 s per loop
```

Here ends the algorithmic similarity between polynomials and integers we have seen in preceding sections.

Like the irreducibility test, the factorisation is performed on the base ring. For example, the factorisation of a polynomial on the integers contains a constant part, itself split into prime factors, and a product of primitive polynomials, i.e., whose coefficients are coprime:

```
sage: x = polygen(ZZ); p = 54*x^4+36*x^3-102*x^2-72*x-12
sage: p.factor()
2 * 3 * (3*x + 1)^2 * (x^2 - 2)
```

Sage is able to factor polynomials on various rings — rational, complex (approximate), finite fields and number fields in particular:

```
sage: for A in [QQ, ComplexField(16), GF(5), QQ[sqrt(2)]]:
....:     print(str(A) + ":")
....:     print(A['x'](p).factor())
Rational Field:
(54) * (x + 1/3)^2 * (x^2 - 2)
Complex Field with 16 bits of precision:
(54.00) * (x - 1.414) * (x + 0.3333)^2 * (x + 1.414)
Finite Field of size 5:
(4) * (x + 2)^2 * (x^2 + 3)
Number Field in sqrt2 with defining polynomial x^2 - 2:
(54) * (x - sqrt2) * (x + sqrt2) * (x + 1/3)^2
```

The result of a decomposition into irreducible factors is not a polynomial (since the polynomials are always in normal form, i.e., in expanded form!), but an object `f` of type `Factorization`. We obtain the `i`th factor with `f[i]`, and we get back the polynomial with `f.expand()`. The `Factorization` objects also provide methods like `gcd` and `lcm` which have the same meaning as for polynomials, but work on the factored forms.

Square-Free Decomposition. Despite its good theoretical and practical complexity, the full factorisation of a polynomial is an expensive operation. The square-free decomposition is a weaker factorisation, much easier to obtain — some gcd computations are enough — and which already brings a lot of information.

[3]On the theoretical side, we know how to factor in $\mathbb{Q}[x]$ in polynomial time, and in $\mathbb{F}_p[x]$ in probabilistic polynomial time, whereas we do not know whether integers can be factored in polynomial time.

Factorisation	
irreducibility test	`p.is_irreducible()`
factorisation	`p.factor()`
square-free factorisation	`p.squarefree_decomposition()`
square-free part $p/\gcd(p,p')$	`p.radical()`
Roots	
roots in A, in D	`p.roots()`, `p.roots(D)`
real roots	`p.roots(RR)`, `p.real_roots()`
complex roots	`p.roots(CC)`, `p.complex_roots()`
isolation of real roots	`p.roots(RIF)`, `p.real_root_intervals()`
isolation of complex roots	`p.roots(CIF)`
resultant	`p.resultant(q)`
discriminant	`p.discriminant()`
Galois group (p irreducible)	`p.galois_group()`

TABLE 7.5 – Factorisation and roots.

Let $p = \prod_{i=1}^{r} p_i^{m_i} \in K[x]$ be a polynomial that splits into a product of irreducible factors over a field K of characteristic zero. We say that p is *square-free* if all its factors p_i have multiplicity $m_i = 1$, i.e., if the roots of p in an algebraic closure of K are simple. A *square-free decomposition* is a factorisation into a product of square-free and coprime factors:

$$p = f_1 f_2^2 \dots f_s^s \qquad \text{where} \qquad f_m = \prod_{m_i = m} p_i.$$

Hence, the square-free decomposition splits the irreducible factors of p by multiplicity. The *square-free part* $f_1 \dots f_s = p_1 \dots p_r$ of p is the polynomial with simple roots which has the same roots as p, disregarding multiplicities.

7.3.2 Root Finding

The computation of the roots of a polynomial may be performed in several ways: Do we want real or complex roots? Roots in another domain? Do we want exact or approximate roots? With or without multiplicities? In a guaranteed or heuristic way? The `roots` method of a polynomial returns by default the roots in its base ring, in the form of a list of pairs (root, multiplicity):

```
sage: R.<x> = ZZ[]; p = (2*x^2-5*x+2)^2 * (x^4-7); p.roots()
[(2, 2)]
```

With a parameter, `roots(D)` returns the roots in the domain D, here the rational roots, and approximations of the ℓ-adic roots for $\ell = 19$:

```
sage: p.roots(QQ)
[(2, 2), (1/2, 2)]
sage: p.roots(Zp(19, print_max_terms=3))
```

```
[(7 + 16*19 + 17*19^2 + ... + O(19^20), 1),
 (12 + 2*19 + 19^2 + ... + O(19^20), 1),
 (10 + 9*19 + 9*19^2 + ... + O(19^20), 2),
 (2 + O(19^20), 2)]
```

This works for a large number of domains, with more or less efficiency.

In particular, selecting for D the field of algebraic numbers `QQbar` or that of real algebraic numbers `AA` enables us to compute exactly the complex or real roots of a polynomial with rational coefficients:

```
sage: roots = p.roots(AA); roots
[(-1.626576561697786?, 1), (0.500000000000000?, 2),
(1.626576561697786?, 1), (2.000000000000000?, 2)]
```

Sage plays transparently for the user with different representations of algebraic numbers. One encodes each $\alpha \in \bar{\mathbb{Q}}$ by its minimal polynomial together with a sufficiently accurate interval to distinguish α from the other roots. Therefore, despite their output, the returned roots are not just approximate values. They can be reused in exact computations:

```
sage: a = roots[0][0]^4; a.simplify(); a
7
```

Here, we have raised the first root found to the fourth power, then forced Sage to simplify the result to make it clear it equals the integer 7.

A variant of the exact resolution is to simply *isolate* the roots, i.e., determine intervals containing exactly one root each, by giving as domain `D` that of the real intervals `RIF` or complex intervals `CIF`. Among the other useful domains in the case of a polynomial with rational coefficients, let us mention `RR`, `CC`, `RDF`, `CDF`, which all correspond to approximate numerical roots, and the number fields `QQ[alpha]`.The specific methods `real_roots`, `complex_roots` and (for some base rings) `real_root_intervals` offer additional options or give slightly different results from the `roots` method. The numerical approximation and isolation of roots is discussed in more detail in §12.2.

7.3.3 Resultant

In a unique factorisation domain, the existence of a common non-constant factor between two polynomials is characterised by the nullity of their *resultant* $\operatorname{Res}(p, q)$, which is a polynomial in their coefficients. A major advantage of the resultant compared to the gcd is that it *specialises* well under ring morphisms. For example, the polynomials $x - 12$ and $x - 20$ are coprime in $\mathbb{Z}[x]$, but the nullity of their resultant

```
sage: x = polygen(ZZ); (x-12).resultant(x-20)
-8
```

modulo n shows that they have a common factor in $\mathbb{Z}/n\mathbb{Z}$ if and only if n divides 8.

Let $p = \sum_{i=0}^{m} p_i x^i$ and $q = \sum_{i=0}^{n} q_i x^i$ be two non constant polynomials in $A[x]$, with $p_m, q_n \neq 0$. The resultant of p and q is defined by

$$\operatorname{Res}(p,q) = \left|\begin{array}{ccccccc} p_m & \cdots & \cdots & p_0 & & & \\ & \ddots & & & \ddots & & \\ & & p_m & & \cdots & \cdots & p_0 \\ q_n & \cdots & & q_0 & & & \\ & \ddots & & & \ddots & & \\ & & \ddots & & & \ddots & \\ & & & q_n & & \cdots & q_0 \end{array}\right|. \tag{7.1}$$

It is the determinant, in suitable bases, of the linear map

$$\begin{array}{rcl} A_{n-1}[x] \times A_{m-1}[x] & \to & A_{m+n-1}[x] \\ u, v & \mapsto & up + vq \end{array}$$

where $A_k[x] \subset A[x]$ is the sub-module of polynomials of degree at most k. If p and q split into linear factors, their resultant may also be expressed in terms of differences of their roots:

$$\operatorname{Res}(p,q) = p_m^n q_n^m \prod_{i,j} (\alpha_i - \beta_j), \qquad \left\{ \begin{array}{l} p = p_m (x-\alpha_1)\dots(x-\alpha_m) \\ q = q_n (x-\beta_1)\dots(x-\beta_n). \end{array} \right.$$

The specialisation property mentioned above follows from the definition (7.1): if $\varphi : A \to A'$ is a ring morphism, the application of which to p and q keeps their degrees unchanged, i.e., such that $\varphi(p_m) \neq 0$ and $\varphi(q_n) \neq 0$, then we have

$$\operatorname{Res}(\varphi(p), \varphi(q)) = \varphi(\operatorname{Res}(p,q)).$$

As a consequence, $\varphi(\operatorname{Res}(p,q))$ vanishes when $\varphi(p)$ and $\varphi(q)$ share a common factor. We have seen above an example of this phenomenon, with φ the canonical projection from $\mathbb{Z}$ to $\mathbb{Z}/n\mathbb{Z}$.

The most common usage of the resultant concerns the case where the base ring itself is a polynomial ring: $p, q \in A[x]$ with $A = K[a_1, \dots, a_k]$. In particular, given $\alpha_1, \dots, \alpha_k \in K$, let us consider the specialisation

$$\begin{array}{rrcl} \varphi : & B[a_1, \dots, a_k] & \to & K \\ & q(a_1, \dots, a_k) & \mapsto & q(\alpha_1, \dots, \alpha_k). \end{array}$$

We see that the resultant $\operatorname{Res}(p,q)$ vanishes at $(\alpha_1, \dots, \alpha_k)$ if and only if the specialisations $\varphi(p), \varphi(q) \in K[x]$ share a common factor, *assuming that* one of the leading terms of p and q does not vanish in $(\alpha_1, \dots, \alpha_k)$.

For example, the discriminant of $p \in \mathbb{Q}[x]$ of degree m is defined by

$$\operatorname{disc}(p) = (-1)^{m(m-1)/2} \operatorname{Res}(p, p')/p_m.$$

This definition generalises the classical discriminants of degree two and three polynomials:

```
sage: R.<a,b,c,d> = QQ[]; x = polygen(R); p = a*x^2+b*x+c
sage: p.resultant(p.derivative())
-a*b^2 + 4*a^2*c
sage: p.discriminant()
b^2 - 4*a*c
sage: (a*x^3 + b*x^2 + c*x + d).discriminant()
b^2*c^2 - 4*a*c^3 - 4*b^3*d + 18*a*b*c*d - 27*a^2*d^2
```

Since the discriminant of p is, up to a normalisation, the resultant of p and its derivative, it vanishes if and only if p has a multiple root in $\mathbb{C}$.

7.3.4 Galois Group

The Galois group of an irreducible polynomial $p \in \mathbb{Q}[x]$ is an algebraic object which describes some of the "symmetries" of the roots of p. It is a central object in the theory of algebraic equations. In particular, the equation $p(x) = 0$ is solvable by radicals — i.e., its roots can be expressed from coefficients of p using the four operations and the nth root — if and only if the Galois group of p is *solvable*.

Sage allows the computation of the Galois group of polynomials with rational coefficients of moderate degree, and performs several operations on the obtained groups. Both Galois theory and the group theory functionalities of Sage go beyond the scope of this book. Let us simply apply without more explanations Galois' theorem on the solvability by radicals. The following computation[4] shows that the roots of $x^5 - x - 1$ cannot be expressed using radicals:

```
sage: x = polygen(QQ); G = (x^5 - x - 1).galois_group(); G
Transitive group number 5 of degree 5
sage: G.is_solvable()
False
```

It is one of the simplest examples of this situation, since polynomials of degree less than or equal to 4 are always solvable by radicals, as well as obviously those of the form $x^5 - a$. By looking at the generators of G seen as a permutation group, we recognise that $G \simeq \mathfrak{S}_5$, which can be easily verified:

```
sage: G.gens()
[(1,2,3,4,5), (1,2)]
sage: G.is_isomorphic(SymmetricGroup(5))
True
```

7.4 Rational Functions

7.4.1 Construction and Basic Properties

The division of two polynomials (on an integral ring) produces a rational function. Its parent is the fraction field of the polynomial ring, obtained with `Frac(R)`:

[4]This computation requires a table of finite groups which is not in the default installation of Sage, but we can upload and automatically install it with the command `sage -i database_gap` (it might be needed to restart Sage after the installation).

Rational functions	
fraction field $K(x)$	`Frac(K['x'])`
numerator	`r.numerator()`
denominator	`r.denominator()`
simplification (modifies `r`)	`r.reduce()`
partial fraction decomposition	`r.partial_fraction_decomposition()`
rational reconstruction of s mod m	`s.rational_reconstruct(m)`

Truncated power series	
ring $A[[t]]$	`PowerSeriesRing(A, 'x', default_prec=n)`
ring $A((t))$	`LaurentSeriesRing(A, 'x', default_prec=n)`
coefficient $[x^k]\,f(x)$	`f[k]`
truncation	`x + O(x^n)`
precision	`f.prec()`
derivative, antiderivative (vanishes at 0)	`f.derivative()`, `f.integral()`
usual operations $\sqrt{f}$, $\exp f$, ...	`f.sqrt()`, `f.exp()`, ...
reciprocal ($f \circ g = g \circ f = x$)	`g = f.reverse()`
solution of $y' = ay + b$	`a.solve_linear_de(precision, b)`

Table 7.6 – Objects constructed from polynomials.

```
sage: x = polygen(RR); r = (1 + x)/(1 - x^2); r.parent()
Fraction Field of Univariate Polynomial Ring in x over Real
Field with 53 bits of precision
sage: r
(x + 1.00000000000000)/(-x^2 + 1.00000000000000)
```

We see that the simplification is not automatic. This is because `RR` is an *inexact* ring, i.e., its elements are approximations of mathematical objects. The `reduce` method puts the fraction in reduced form. It does not return a new object, but modifies the existing fraction:

```
sage: r.reduce(); r
1.00000000000000/(-x + 1.00000000000000)
```

On an exact ring, in contrast, rational functions are automatically reduced.

The operations on rational functions are analogous to those on polynomials. Those having a meaning in both cases (substitution, derivative, factorisation...) may be used in the same manner. Table 7.6 enumerates some other useful methods. The partial fraction decomposition and the rational reconstruction deserve some explanations.

7.4.2 Partial Fraction Decomposition

Sage computes the partial fraction decomposition of a rational function a/b in `Frac(K['x'])` from the factorisation of b in `K['x']`. It is therefore the partial fraction decomposition on K. The result contains a polynomial part p and a list of rational functions whose denominators are powers of irreducible factors of b:

```
sage: R.<x> = QQ[]; r = x^10 / ((x^2-1)^2 * (x^2+3))
sage: poly, parts = r.partial_fraction_decomposition()
sage: poly
x^4 - x^2 + 6
sage: for part in parts: part.factor()
(17/32) * (x - 1)^-1
(1/16) * (x - 1)^-2
(-17/32) * (x + 1)^-1
(1/16) * (x + 1)^-2
(-243/16) * (x^2 + 3)^-1
```

We have thus obtained the partial fraction decomposition on the rationals

$$r = \frac{x^{10}}{(x^2-1)^2(x^2+3)} = x^4 - x^2 + 6 + \frac{\frac{17}{32}}{x-1} + \frac{\frac{1}{16}}{(x-1)^2} - \frac{\frac{17}{32}}{x+1} + \frac{\frac{1}{16}}{(x+1)^2} - \frac{\frac{243}{16}}{x^2+3}.$$

This is also clearly the partial fraction decomposition of r on the real numbers.

However, on the complex numbers, the denominator of the last term is not irreducible, hence the rational function can be further decomposed. We can compute the partial fraction decomposition on the complex numbers numerically:

```
sage: C = ComplexField(15)
sage: Frac(C['x'])(r).partial_fraction_decomposition()
(x^4 - x^2 + 6.000, [0.5312/(x - 1.000), 0.06250/(x^2 - 2.000*x + 1.000)
  ,
4.385*I/(x - 1.732*I), (-4.385*I)/(x + 1.732*I),
(-0.5312)/(x + 1.000), 0.06250/(x^2 + 2.000*x + 1.000)])
```

We obtain the exact decomposition on $\mathbb{C}$ in the same manner, by replacing `C` by `QQbar`. Doing the computation on `AA`, we would get the decomposition on the reals, even when all real roots of the denominator are not rational.

7.4.3 Rational Reconstruction

As for integers in §6.1.3, the rational reconstruction also exists for polynomials with coefficients in $A = \mathbb{Z}/n\mathbb{Z}$. Given $m, s \in A[x]$, the command

```
sage: s.rational_reconstruct(m, dp, dq)
```

computes when possible polynomials $p, q \in A[x]$ such that

$$qs \equiv p \mod m, \qquad \deg p \le d_p, \quad \deg q \le d_q.$$

For simplicity, let us restrict ourselves to the case where n is prime. Such a relation with q and m coprime implies $p/q = s$ in $A[x]/\langle m\rangle$, which explains the "rational reconstruction" name.

The rational reconstruction problem translates into a linear system on the coefficients of p and q, and a simple dimension argument shows that a non-trivial solution exists as soon as $d_p + d_q \ge \deg m - 1$. A solution with q and m coprime does not always exist (for example, the solutions of $p \equiv qx \mod x^2$ with $\deg p \le 0$, $\deg q \le 1$ are the constant multiples of $(p, q) = (0, x)$), but `rational_reconstruct` looks rather for solutions q coprime to m.

Padé Approximants. The case $m = x^n$ is called Padé approximant. A Padé approximant of type $(k, n-k)$ of a formal power series $f \in K[[x]]$ is a rational function $p/q \in K(x)$ such that $\deg p \leq k-1$, $\deg q \leq n-k$, $q(0) = 1$, and $p/q = f + O(x^n)$. We then have $p/q \equiv f \bmod x^n$.

Let us start with a symbolic example. The following commands compute a Padé approximant of the series $f = \sum_{i=0}^{\infty} (i+1)^2 x^i$ with coefficients in $\mathbb{Z}/101\mathbb{Z}$:

```
sage: A = Integers(101); R.<x> = A[]
sage: f6 = sum( (i+1)^2 * x^i for i in (0..5) ); f6
36*x^5 + 25*x^4 + 16*x^3 + 9*x^2 + 4*x + 1
sage: num, den = f6.rational_reconstruct(x^6, 1, 3); num/den
(100*x + 100)/(x^3 + 98*x^2 + 3*x + 100)
```

By expanding back into power series the rational function found, we see that not only the terms correspond up the term in x^5, but even the next term "is correct"!

```
sage: S = PowerSeriesRing(A, 'x', 7); S(num)/S(den)
1 + 4*x + 9*x^2 + 16*x^3 + 25*x^4 + 36*x^5 + 49*x^6 + O(x^7)
```

Indeed, f itself is a rational function: we have $f = (1+x)/(1-x)^3$. The truncated expansion `f6`, together with bounds on the degrees of the numerator and denominator, is enough to represent it without any ambiguity. From this point of view, the computation of Padé approximants is the converse of the series expansion of power series: it allows us to go back from this alternative representation to the usual one as quotient of two polynomials.

An Analytic Example. Historically, Padé approximants do not come from this kind of symbolic reasoning, but from the approximation theory of analytic functions. Indeed, the Padé approximants of the series expansion of an analytic function often approximate the function better than series truncations. When the degree of the denominator is large enough, Padé approximants can even give good approximations outside the convergence disc of the series. We sometimes say that they "swallow the poles". Figure 7.1, which shows the convergence of the approximants of type $(2k, k)$ of the tangent function around 0, illustrates this phenomenon.

Although `rational_reconstruct` is restricted to polynomials on $\mathbb{Z}/n\mathbb{Z}$, it is possible to use it to compute Padé approximants with rational coefficients, and obtain that figure. The simplest way is to first perform the rational reconstruction modulo a large enough prime:

```
sage: x = var('x'); s = tan(x).taylor(x, 0, 20)
sage: p = previous_prime(2^30); ZpZx = Integers(p)['x']
sage: Qx = QQ['x']
```

```
sage: num, den = ZpZx(s).rational_reconstruct(ZpZx(x)^10,4,5)
sage: num/den
(1073741779*x^3 + 105*x)/(x^4 + 1073741744*x^2 + 105)
```

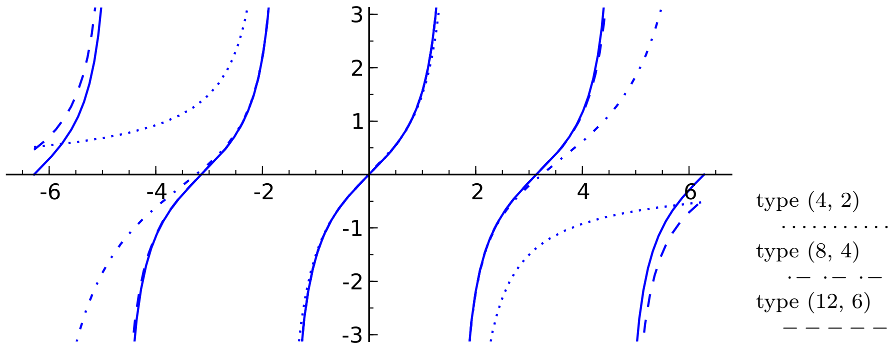

Figure 7.1 – The tangent function and some Padé approximants on $[-2\pi, 2\pi]$.

then to lift the solution found. The following function lifts an element a from $\mathbb{Z}/p\mathbb{Z}$ into an integer of absolute value at most $p/2$.

```
sage: def lift_sym(a):
....:     m = a.parent().defining_ideal().gen()
....:     n = a.lift()
....:     if n <= m // 2: return n
....:     else: return n - m
```

We then get:

```
sage: Qx(map(lift_sym, num))/Qx(map(lift_sym, den))
(-10*x^3 + 105*x)/(x^4 - 45*x^2 + 105)
```

When the wanted coefficients are too large for this technique, we can perform the computation modulo several primes, and apply the "Chinese Remainder Theorem" to obtain a solution with integer coefficients, as explained in §6.1.4. Another possibility is to compute a recurrence relation with constant coefficients which is satisfied by the series coefficients. This computation is almost equivalent to a Padé approximant (see Exercise 28), but the Sage function `berlekamp_massey` is able to perform it on any field.

Let us make the preceding computation more automatic, by writing a function which directly computes the approximant with rational coefficients, under favorable assumptions:

```
sage: def mypade(pol, n, k):
....:     x = ZpZx.gen();
....:     n,d = ZpZx(pol).rational_reconstruct(x^n, k-1, n-k)
....:     return Qx(map(lift_sym, n))/Qx(map(lift_sym, d))
```

It then suffices to call `plot` on the results of this function (converted into elements of `SR`, since `plot` is not able to draw directly the graph of an "algebraic" rational function) to obtain the graph of Figure 7.1:

```
sage: add(
....:     plot(expr, -2*pi, 2*pi, ymin=-3, ymax=3,
....:             linestyle=sty, detect_poles=True, aspect_ratio=1)
....:     for (expr, sty) in [
....:         (tan(x), '-'),
....:         (SR(mypade(s, 4,  2)), ':' ),
....:         (SR(mypade(s, 8,  4)), '-.'),
....:         (SR(mypade(s, 12, 6)), '--') ])
```

The following exercises demonstrate two other classical applications of the rational reconstruction.

Exercise 28. 1. Show that if $(u_n)_{n\in\mathbb{N}}$ satisfies a linear recurrence with constant coefficients, then the power series $\sum_{n\in\mathbb{N}} u_n z^n$ is a rational function. How would you interpret the numerator and denominator?

2. Guess the next terms of the sequence

$$1, 1, 2, 3, 8, 11, 34, 39, 148, 127, 662, 339, 3056, 371, 14602, -4257, \dots,$$

by using `rational_reconstruct`. Find again the result with the `berlekamp_massey` function.

Exercise 29 (Cauchy interpolation)**.** Find a rational function $r = p/q \in \mathbb{F}_{17}(x)$ such that $r(0) = -1$, $r(1) = 0$, $r(2) = 7$, $r(3) = 5$, with p of minimal degree.

7.5 Formal Power Series

A formal power series is a power series considered as a simple sequence of coefficients, without considering convergence. More precisely, if A is a commutative ring, we call formal power series of indeterminate x with coefficients in A the formal sums $\sum_{n=0}^{\infty} a_n x^n$ where (a_n) is any sequence of elements of A. Together with the natural addition and multiplication operations

$$\sum_{n=0}^{\infty} a_n x^n + \sum_{n=0}^{\infty} b_n x^n = \sum_{n=0}^{\infty} (a_n + b_n) x^n,$$

$$\Big(\sum_{n=0}^{\infty} a_n x^n\Big)\Big(\sum_{n=0}^{\infty} b_n x^n\Big) = \sum_{n=0}^{\infty}\Big(\sum_{i+j=n} a_i b_j\Big) x^n,$$

the formal power series constitute a ring named $A[[x]]$.

In a computer algebra system, these series are useful to represent analytic functions for which we have no closed form. As always, the computer performs some computations, but it is the user's responsibility to give them a mathematical meaning. In particular, she/he should make sure that the considered series are convergent (if needed).

Formal power series also appear frequently in combinatorics, in the form of generating series. We will see such an example in §15.1.2.

7.5.1 Operations on Truncated Power Series

The ring $\mathbb{Q}[[x]]$ of formal power series is constructed by

```
sage: R.<x> = PowerSeriesRing(QQ)
```

or in short `R.<x> = QQ[[]]`[5]. The elements of `A[['x']]` are truncated power series, i.e., objects of the form

$$f = f_0 + f_1\,x + \cdots + f_{n-1}\,x^{n-1} + O(x^n).$$

They play the role of approximations of infinite "mathematical" series, much like elements of `RR` are approximations of real numbers. The `A[['x']]` ring is thus an inexact ring.

Each series has its own order of truncation[6] and the precision automatically follows through computations:

```
sage: R.<x> = QQ[[]]
sage: f = 1 + x + O(x^2); g = x + 2*x^2 + O(x^4)
sage: f + g
1 + 2*x + O(x^2)
sage: f * g
x + 3*x^2 + O(x^3)
```

Series with infinite precision do exist, they correspond exactly to polynomials:

```
sage: (1 + x^3).prec()
+Infinity
```

A default precision is used when it is necessary to truncate an exact result. It is given at the ring creation, or afterwards with the `set_default_prec` method:

```
sage: R.<x> = PowerSeriesRing(Reals(24), default_prec=4)
sage: 1/(1 + RR.pi() * x)^2
1.00000 - 6.28319*x + 29.6088*x^2 - 124.025*x^3 + O(x^4)
```

As a consequence of the above, it is not possible to test the mathematical equality between two series. This is an important difference between these objects and the other classes of objects seen in this chapter. Sage thus considers two elements of `A[['x']]` as equal as soon as they match up to the *smallest* of their precisions:

```
sage: R.<x> = QQ[[]]
sage: 1 + x + O(x^2) == 1 + x + x^2 + O(x^3)
True
```

Warning: this implies that the test `O(x^2) == 0` returns true.

The basic arithmetic operations on series work as for polynomials. We also have some usual functions, for example `f.exp()` when $f(0) = 0$, as well as the

[5]Or from $\mathbb{Q}[x]$, by `QQ['x'].completion('x')`.

[6]In some sense, this is the main difference between a polynomial modulo x^n and a series truncated at order n: the operations on these two objects are analogous, but the elements of $A[[x]]/\langle x^n\rangle$ have all the same "precision".

derivative and antiderivative functions. Hence, an asymptotic expansion when $x \to 0$ of

$$\frac{1}{x^2} \exp\Big(\int_0^x \sqrt{\frac{1}{1+t}}\, \mathrm{d}t\Big)$$

is given by

```
sage: (1/(1+x)).sqrt().integral().exp() / x^2 + O(x^4)
x^-2 + x^-1 + 1/4 + 1/24*x - 1/192*x^2 + 11/1920*x^3 + O(x^4)
```

Here, only terms up to x^3 appear in the result, since `+ O(x^4)` explicitly asks to truncate to order 4. However, the intermediate computations are performed to the default precision 20, which we can check by omitting the `O(x^4)` term. To get even more terms, we can increase the precision of intermediate computations.

This example also demonstrates that if $f, g \in K[[x]]$ and $g(0) = 0$, the quotient f/g yields an object of type *formal Laurent series*. Contrary to the Laurent series in complex analysis, of the form $\sum_{n=-\infty}^{\infty} a_n x^n$, the formal Laurent series are sums of the form $\sum_{n=-N}^{\infty} a_n x^n$, with a finite number of terms of negative exponent. This restriction is mandatory for the product of two formal series: without it, each product coefficient would be the sum of an infinite series.

7.5.2 Solutions of an Equation: Series Expansions

Given a differential equation whose exact solutions are too complex to compute or to deal with, or simply which does not admit a closed-form solution, an alternative is often to look for solutions in the form of series expansions. We usually first determine solutions of the equation in the space of formal power series, and if necessary we conclude using a convergence argument that the constructed series solutions make sense analytically. Sage may be of great help for the first step.

Let us consider for example the differential equation

$$y'(x) = \sqrt{1+x^2}\, y(x) + \exp(x), \qquad y(0) = 1.$$

This equation has a unique formal power series solution, whose first terms might be computed by

```
sage: (1+x^2).sqrt().solve_linear_de(prec=6, b=x.exp())
1 + 2*x + 3/2*x^2 + 5/6*x^3 + 1/2*x^4 + 7/30*x^5 + O(x^6)
```

Moreover, Cauchy's theorem on the existence of solutions to linear differential equations with analytic coefficients ensures that this series converges for $|x| < 1$: its sum thus provides an analytic solution on the complex unit disc.

This approach is not limited to differential equations. The functional equation $e^{xf(x)} = f(x)$ is more complex, at least since it is not linear. Nevertheless, this is a fixed-point equation, we can try to refine a (formal) solution iteratively:

```
sage: S.<x> = PowerSeriesRing(QQ, default_prec=5)
sage: f = S(1)
sage: for i in range(5):
....:     f = (x*f).exp()
....:     print(f)
```

```
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + O(x^5)
```

What happens here? The solutions of $e^{xf(x)} = f(x)$ in $\mathbb{Q}[[x]]$ are the fixed points of the transform $\Phi : f \mapsto e^{xf}$. If a sequence of iterates of the form $\Phi^n(a)$ converges, its limit is necessarily a solution to the equation. Conversely, let us write $f(x) = \sum_{n=0}^{\infty} f_n\, x^n$, and let us expand in series both sides:

$$\begin{aligned}\sum_{n=0}^{\infty} f_n\, x^n &= \sum_{k=0}^{\infty} \frac{1}{k!}\Big(x\sum_{j=0}^{\infty} f_j\, x^j\Big)^k \\ &= \sum_{n=0}^{\infty}\Bigg(\sum_{k=0}^{\infty}\frac{1}{k!}\sum_{\substack{j_1,\dots,j_k\in\mathbb{N}\\ j_1+\dots+j_k=n-k}} f_{j_1}f_{j_2}\dots f_{j_k}\Bigg)x^n.\end{aligned} \tag{7.2}$$

Ignoring the details of the formula, the important fact is that f_n might be computed from the preceding coefficients $f_0, \dots, f_{n-1}$, as we see by isolating the coefficients on both sides. Hence, each iteration of Φ yields a new correct term.

Exercise 30. Compute the series expansion to order 15 of $\tan x$ near zero, from the differential equation $\tan' = 1 + \tan^2$.

7.5.3 Lazy Power Series

The fixed-point phenomenon motivates the introduction of a new kind of formal power series called *lazy* power series. They are not truncated series, but infinite series; the "lazy" adjective means that coefficients are computed on demand only. As a counterpart, we can only represent series whose coefficients are computable: essentially, combinations of basic series and some solutions of equations for which relations like (7.2) exist. For example, the series `lazy_exp` defined by

```
sage: L.<x> = LazyPowerSeriesRing(QQ)
sage: lazy_exp = x.exponential(); lazy_exp
O(1)
```

is an object which contains in its internal representation all the information needed to compute the series expansion of $\exp x$ to any order. Its output is initially `O(1)` since no coefficient was computed so far. If we ask for the coefficient of x^5, the corresponding computation is performed, and the computed coefficients are stored in memory:

```
sage: lazy_exp[5]
1/120
sage: lazy_exp
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
```

Let us go back to the equation $e^{xf(x)} = f(x)$ to see how it can be solved with lazy series. We first try to reproduce the above computation in the ring `QQ[['x']]`:

```
sage: f = L(1)  # the constant lazy series 1
sage: for i in range(5):
....:     f = (x*f).exponential()
....:     f.compute_coefficients(5) # forces the computation
....:     print(f)                  # of the first coefficients
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + O(x^6)
1 + x + 3/2*x^2 + 5/3*x^3 + 41/24*x^4 + 49/30*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 101/24*x^4 + 63/10*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 49/5*x^5 + O(x^6)
1 + x + 3/2*x^2 + 8/3*x^3 + 125/24*x^4 + 54/5*x^5 + O(x^6)
```

The obtained expansions are of course the same as above[7]. However the value of `f` at each iteration is now an infinite series, whose coefficients can be computed on demand. All these intermediate series are kept in memory. The computation of each one is automatically done at the required precision in order to yield, for example, the coefficient of x^7 in the last iterate when one asks for it:

```
sage: f[7]
28673/630
```

With the code of §7.5.2, accessing `f[7]` would have raised an error, since the index 7 is larger than the truncation order of the series `f`.

However, the value returned by `f[7]` is the coefficient of x^7 in the iterate $\Phi^5(1)$, and not in the solution! The power of lazy series is the possibility to directly get the limit, by defining f itself as a lazy series:

```
sage: from sage.combinat.species.series import LazyPowerSeries
sage: f = LazyPowerSeries(L, name='f')
sage: f.define((x*f).exponential())
sage: f.coefficients(8)
[1, 1, 3/2, 8/3, 125/24, 54/5, 16807/720, 16384/315]
```

The iterative computation did "work" thanks to the relation (7.2). Under the hood, Sage deduces from the recursive definition `f.define((x*f).exponential())` a similar formula, which enables it to compute coefficients by recurrence.

7.6 Computer Representation of Polynomials

A given mathematical object — the polynomial p, with coefficients in A — might be encoded in very different ways on a computer. While the result of a mathematical operation on p is clearly independent of the representation, the corresponding

[7]We observe however that Sage sometimes has incoherent conventions: the `exp` method for truncated series is now called `exponential`, and `compute_coefficients(5)` computes the coefficients up to order 5 included, whereas `default_prec=5` gave series truncated after the coefficient of x^4.

Sage objects might behave differently. The choice of representation impacts the possible operations, the exact form of their results, and particularly the efficiency of the computations.

Dense or Sparse Representation. Two principal ways exist for representing polynomials. In a *dense* representation, the coefficients of $p = \sum_{i=0}^{n} p_i\, x^i$ are stored in a table $[p_0, \ldots, p_n]$ indexed by the exponents. A *sparse* representation only stores the non-zero coefficients: the polynomial is encoded by a set of pairs exponent-coefficient (i, p_i), stored in a list, or better, in a dictionary indexed by the exponents (see §3.3.9).

For polynomials that really are dense, i.e., whose coefficients are mostly non-zero, the dense representation uses less memory and enables faster computations. It saves the encoding of the exponents and of the internal data structures of the dictionary: it only stores what is strictly necessary, the coefficients. Moreover, accessing an element and iterating on elements are faster in a table than in a dictionary. Conversely, the sparse representation enables us to efficiently compute with polynomials that we could not even store in memory with a dense representation:

```
sage: R = PolynomialRing(ZZ, 'x', sparse=True)
sage: p = R.cyclotomic_polynomial(2^50); p, p.derivative()
(x^562949953421312 + 1, 562949953421312*x^562949953421311)
```

As shown by the preceding example, the representation is a characteristic of the polynomial ring, chosen at its construction. The "dense" polynomial $x \in \mathbb{Q}[x]$ and the "sparse" polynomial $x \in \mathbb{Q}[x]$ thus have different parents. The default representation of univariate polynomials is dense. The option `sparse=True` of `PolynomialRing` enables us to build a polynomial ring with sparse representation.

In addition, some details of the representation vary according to the kind of coefficients. The same holds for the code used to perform basic operations. Indeed, Sage provides a *generic* polynomial implementation which works on any commutative ring, but also optimised variants for some particular types of coefficients. These variants bring some additional features, and above all are much more efficient than the generic version. They call for this purpose some specialised external libraries, like FLINT or NTL in the case of $\mathbb{Z}[x]$.

To complete huge computations successfully, it is very important to work whenever possible in polynomial rings with efficient implementations. The help page output by `p?` for a polynomial `p` indicates which implementation it uses. The choice of the implementation often depends on the base ring and the representation. The `implementation` option of `PolynomialRing` enables us to choose a particular implementation when several are possible.

Symbolic Expressions. The symbolic expressions discussed in Chapters 1 and 2 (i.e., the elements of `SR`) provide a third representation of polynomials. They are a natural choice when a computation mixes polynomials and more diverse expressions, as it is often the case in analysis. The flexibility they offer is sometimes useful even in a fully algebraic context. For example, the polynomial

A little bit of theory

To get the best out of fast operations on polynomials, it is good to have an idea of their algorithmic complexity. We briefly discuss this for the reader with some algorithmic knowledge. We limit ourselves to the case of dense polynomials.

Additions, subtractions and other direct operations on coefficients are performed in linear time with respect to the degrees of the considered polynomials. Their practical efficiency thus depends essentially on the easy access to the coefficients, and therefore on the internal data structure.

The critical operation is multiplication. Indeed, not only is this a basic arithmetic operation, but other operations use algorithms whose complexity depends essentially on that of multiplication. For example, given two polynomials of degree at most n, we can compute their Euclidean division at the cost of $O(1)$ multiplications, or their gcd at that of $O(\log n)$ multiplications.

Good news: we know how to multiply polynomials in quasi-linear time. More precisely, the best known complexity over any ring is $O(n \log n \log\log n)$ operations in the base ring. It relies on generalisations of the famous Schönhage-Strassen algorithm, which attains the same complexity for integer multiplication. By comparison, the method used by hand to multiply polynomials requires of the order of n^2 operations.

In practice, the fast multiplication algorithms are competitive for large enough degrees, as well as corresponding methods for the division. The libraries called by Sage for some kinds of coefficients use such advanced algorithms: this explains why Sage is able to efficiently work with polynomials of huge degree on some coefficient rings.

$(x+1)^{10^{10}}$, once expanded, is dense, but it is not necessary (nor desirable!) to expand it in order to differentiate it or evaluate it numerically.

Beware however: as opposed to algebraic polynomials, symbolic polynomials (in `SR`) are not attached to a particular polynomial ring, and are not put in canonical form. A given polynomial might have a lot of different forms, it is the user's responsibility to perform the needed conversions between them. In the same vein, the `SR` domain groups together all symbolic expressions, without any distinction between polynomials and other expressions, but we can explicitly check whether a given symbolic expression `f` is polynomial in the variable `x` by `f.is_polynomial(x)`.