```
>>> R.dup_content(f)
2
```

sympy.polys.densetools.**dmp_ground_content**(*f*, *u*, *K*)

Compute the GCD of coefficients of `f` in `K[X]`.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_content(f)
2
```

sympy.polys.densetools.**dup_primitive**(*f*, *K*)

Compute content and the primitive form of `f` in `K[x]`.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x = ring("x", ZZ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

```
>>> R, x = ring("x", QQ)
>>> f = 6*x**2 + 8*x + 12
```

```
>>> R.dup_primitive(f)
(2, 3*x**2 + 4*x + 6)
```

sympy.polys.densetools.**dmp_ground_primitive**(*f*, *u*, *K*)

Compute content and the primitive form of `f` in `K[X]`.

Examples

```
>>> from sympy.polys import ring, ZZ, QQ
```

```
>>> R, x,y = ring("x,y", ZZ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

```
>>> R, x,y = ring("x,y", QQ)
>>> f = 2*x*y + 6*x + 4*y + 12
```

```
>>> R.dmp_ground_primitive(f)
(2, x*y + 3*x + 2*y + 6)
```

sympy.polys.densetools.**dup_extract**(*f*, *g*, *K*)

Extract common content from a pair of polynomials in `K[x]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_extract(6*x**2 + 12*x + 18, 4*x**2 + 8*x + 12)
(2, 3*x**2 + 6*x + 9, 2*x**2 + 4*x + 6)
```

sympy.polys.densetools.**dmp_ground_extract**(*f*, *g*, *u*, *K*)

Extract common content from a pair of polynomials in `K[X]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_ground_extract(6*x*y + 12*x + 18, 4*x*y + 8*x + 12)
(2, 3*x*y + 6*x + 9, 2*x*y + 4*x + 6)
```

sympy.polys.densetools.**dup_real_imag**(*f*, *K*)

Return bivariate polynomials `f1` and `f2`, such that `f = f1 + f2*I`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dup_real_imag(x**3 + x**2 + x + 1)
(x**3 + x**2 - 3*x*y**2 + x - y**2 + 1, 3*x**2*y + 2*x*y - y**3 + y)
```

sympy.polys.densetools.**dup_mirror**(*f*, *K*)

Evaluate efficiently the composition `f(-x)` in `K[x]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_mirror(x**3 + 2*x**2 - 4*x + 2)
-x**3 + 2*x**2 + 4*x + 2
```

sympy.polys.densetools.**dup_scale**(*f*, *a*, *K*)

Evaluate efficiently composition `f(a*x)` in `K[x]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_scale(x**2 - 2*x + 1, ZZ(2))
4*x**2 - 4*x + 1
```

sympy.polys.densetools.**dup_shift**(*f*, *a*, *K*)

Evaluate efficiently Taylor shift `f(x + a)` in `K[x]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_shift(x**2 - 2*x + 1, ZZ(2))
x**2 + 2*x + 1
```

sympy.polys.densetools.**dup_transform**(*f*, *p*, *q*, *K*)

Evaluate functional transformation `q**n * f(p/q)` in `K[x]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_transform(x**2 - 2*x + 1, x**2 + 1, x - 1)
x**4 - 2*x**3 + 5*x**2 - 4*x + 4
```

sympy.polys.densetools.**dmp_compose**(*f*, *g*, *u*, *K*)

Evaluate functional composition `f(g)` in `K[X]`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_compose(x*y + 2*x + y, y)
y**2 + 3*y
```

sympy.polys.densetools.**dup_decompose**(*f*, *K*)

Computes functional decomposition of `f` in `K[x]`.

Given a univariate polynomial `f` with coefficients in a field of characteristic zero, returns list `[f_1, f_2, ..., f_n]`, where:

```
f = f_1 o f_2 o ... f_n = f_1(f_2(... f_n))
```

and `f_2, ..., f_n` are monic and homogeneous polynomials of at least second degree.

Unlike factorization, complete functional decompositions of polynomials are not unique, consider examples:

1. `f o g = f(x + b) o (g - b)`
2. `x**n o x**m = x**m o x**n`
3. `T_n o T_m = T_m o T_n`

where `T_n` and `T_m` are Chebyshev polynomials.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_decompose(x**4 - 2*x**3 + x**2)
[x**2, x**2 - x]
```

References

[R695]

sympy.polys.densetools.**dmp_lift**(*f*, *u*, *K*)

Convert algebraic coefficients to integers in K[X].

Examples

```
>>> from sympy.polys import ring, QQ
>>> from sympy import I
```

```
>>> K = QQ.algebraic_field(I)
>>> R, x = ring("x", K)
```

```
>>> f = x**2 + K([QQ(1), QQ(0)])*x + K([QQ(2), QQ(0)])
```

```
>>> R.dmp_lift(f)
x**8 + 2*x**6 + 9*x**4 - 8*x**2 + 16
```

sympy.polys.densetools.**dup_sign_variations**(*f*, *K*)

Compute the number of sign variations of f in K[x].

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> R.dup_sign_variations(x**4 - x**2 - x + 1)
2
```

sympy.polys.densetools.**dmp_clear_denoms**(*f*, *u*, *K0*, *K1=None*, *convert=False*)

Clear denominators, i.e. transform K_0 to K_1.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + QQ(1,3)*y + 1
```

```
>>> R.dmp_clear_denoms(f, convert=False)
(6, 3*x + 2*y + 6)
>>> R.dmp_clear_denoms(f, convert=True)
(6, 3*x + 2*y + 6)
```

sympy.polys.densetools.**dmp_revert**(*f*, *g*, *u*, *K*)

Compute f**(-1) mod x**n using Newton iteration.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

Manipulation of dense, univariate polynomials with finite field coefficients

Functions in this module carry the prefix `gf_`, referring to the classical name "Galois Fields" for finite fields. Note that many polynomial factorization algorithms work by reduction to the finite field case, so having special implementations for this case is justified both by performance, and by the necessity of certain methods which do not even make sense over general fields.

sympy.polys.galoistools.**gf_crt**(*U*, *M*, *K=None*)

Chinese Remainder Theorem.

Given a set of integer residues `u_0,...,u_n` and a set of co-prime integer moduli `m_0,..,m_n`, returns an integer `u`, such that `u = u_i mod m_i` for `i = ``0,...,n`.

Examples

Consider a set of residues `U = [49, 76, 65]` and a set of moduli `M = [99, 97, 95]`. Then we have:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt

>>> gf_crt([49, 76, 65], [99, 97, 95], ZZ)
639985
```

This is the correct result because:

```
>>> [639985 % m for m in [99, 97, 95]]
[49, 76, 65]
```

Note: this is a low-level routine with no error checking.

See also:

sympy.ntheory.modular.crt **(page 1509)**
a higher level crt routine

sympy.ntheory.modular.solve_congruence (page 1511)

sympy.polys.galoistools.**gf_crt1**(*M*, *K*)

First part of the Chinese Remainder Theorem.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt1
```

```
>>> gf_crt1([99, 97, 95], ZZ)
(912285, [9215, 9405, 9603], [62, 24, 12])
```

sympy.polys.galoistools.**gf_crt2**(*U*, *M*, *p*, *E*, *S*, *K*)

Second part of the Chinese Remainder Theorem.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_crt2
```

```
>>> U = [49, 76, 65]
>>> M = [99, 97, 95]
>>> p = 912285
>>> E = [9215, 9405, 9603]
>>> S = [62, 24, 12]
```

```
>>> gf_crt2(U, M, p, E, S, ZZ)
639985
```

sympy.polys.galoistools.**gf_int**(*a*, *p*)

Coerce `a mod p` to an integer in the range `[-p/2, p/2]`.

Examples

```
>>> from sympy.polys.galoistools import gf_int
```

```
>>> gf_int(2, 7)
2
>>> gf_int(5, 7)
-2
```

sympy.polys.galoistools.**gf_degree**(*f*)

Return the leading degree of `f`.

Examples

```
>>> from sympy.polys.galoistools import gf_degree
```

```
>>> gf_degree([1, 1, 2, 0])
3
>>> gf_degree([])
-1
```

sympy.polys.galoistools.**gf_LC**(*f*, *K*)

Return the leading coefficient of `f`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_LC
```

```
>>> gf_LC([3, 0, 1], ZZ)
3
```

sympy.polys.galoistools.**gf_TC**(*f*, *K*)

Return the trailing coefficient of `f`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_TC
```

```
>>> gf_TC([3, 0, 1], ZZ)
1
```

sympy.polys.galoistools.**gf_strip**(*f*)

Remove leading zeros from `f`.

Examples

```
>>> from sympy.polys.galoistools import gf_strip
```

```
>>> gf_strip([0, 0, 0, 3, 0, 1])
[3, 0, 1]
```

sympy.polys.galoistools.**gf_trunc**(*f*, *p*)

Reduce all coefficients modulo `p`.

Examples

```
>>> from sympy.polys.galoistools import gf_trunc
```

```
>>> gf_trunc([7, -2, 3], 5)
[2, 3, 3]
```

sympy.polys.galoistools.**gf_normal**(*f*, *p*, *K*)

Normalize all coefficients in K.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_normal
```

```
>>> gf_normal([5, 10, 21, -3], 5, ZZ)
[1, 2]
```

sympy.polys.galoistools.**gf_from_dict**(*f*, *p*, *K*)

Create a GF(p)[x] polynomial from a dict.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_from_dict
```

```
>>> gf_from_dict({10: ZZ(4), 4: ZZ(33), 0: ZZ(-1)}, 5, ZZ)
[4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4]
```

sympy.polys.galoistools.**gf_to_dict**(*f*, *p*, *symmetric=True*)

Convert a GF(p)[x] polynomial to a dict.

Examples

```
>>> from sympy.polys.galoistools import gf_to_dict
```

```
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5)
{0: -1, 4: -2, 10: -1}
>>> gf_to_dict([4, 0, 0, 0, 0, 0, 3, 0, 0, 0, 4], 5, symmetric=False)
{0: 4, 4: 3, 10: 4}
```

sympy.polys.galoistools.**gf_from_int_poly**(*f*, *p*)

Create a GF(p)[x] polynomial from Z[x].

Examples

```
>>> from sympy.polys.galoistools import gf_from_int_poly
```

```
>>> gf_from_int_poly([7, -2, 3], 5)
[2, 3, 3]
```

sympy.polys.galoistools.**gf_to_int_poly**(*f*, *p*, *symmetric=True*)

Convert a `GF(p)[x]` polynomial to `Z[x]`.

Examples

```
>>> from sympy.polys.galoistools import gf_to_int_poly
```

```
>>> gf_to_int_poly([2, 3, 3], 5)
[2, -2, -2]
>>> gf_to_int_poly([2, 3, 3], 5, symmetric=False)
[2, 3, 3]
```

sympy.polys.galoistools.**gf_neg**(*f*, *p*, *K*)

Negate a polynomial in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_neg
```

```
>>> gf_neg([3, 2, 1, 0], 5, ZZ)
[2, 3, 4, 0]
```

sympy.polys.galoistools.**gf_add_ground**(*f*, *a*, *p*, *K*)

Compute `f + a` where `f` in `GF(p)[x]` and `a` in `GF(p)`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_ground
```

```
>>> gf_add_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 1]
```

sympy.polys.galoistools.**gf_sub_ground**(*f*, *a*, *p*, *K*)

Compute `f - a` where `f` in `GF(p)[x]` and `a` in `GF(p)`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_ground
```

```
>>> gf_sub_ground([3, 2, 4], 2, 5, ZZ)
[3, 2, 2]
```

sympy.polys.galoistools.**gf_mul_ground**(*f*, *a*, *p*, *K*)

Compute `f * a` where `f` in `GF(p)[x]` and `a` in `GF(p)`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul_ground
```

```
>>> gf_mul_ground([3, 2, 4], 2, 5, ZZ)
[1, 4, 3]
```

sympy.polys.galoistools.**gf_quo_ground**(*f*, *a*, *p*, *K*)

Compute `f/a` where `f` in `GF(p)[x]` and `a` in `GF(p)`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo_ground
```

```
>>> gf_quo_ground(ZZ.map([3, 2, 4]), ZZ(2), 5, ZZ)
[4, 1, 2]
```

sympy.polys.galoistools.**gf_add**(*f*, *g*, *p*, *K*)

Add polynomials in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add
```

```
>>> gf_add([3, 2, 4], [2, 2, 2], 5, ZZ)
[4, 1]
```

sympy.polys.galoistools.**gf_sub**(*f*, *g*, *p*, *K*)

Subtract polynomials in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub
```

```
>>> gf_sub([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 2]
```

sympy.polys.galoistools.**gf_mul**(*f*, *g*, *p*, *K*)

Multiply polynomials in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_mul
```

```
>>> gf_mul([3, 2, 4], [2, 2, 2], 5, ZZ)
[1, 0, 3, 2, 3]
```

sympy.polys.galoistools.**gf_sqr**(*f*, *p*, *K*)

Square polynomials in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqr
```

```
>>> gf_sqr([3, 2, 4], 5, ZZ)
[4, 2, 3, 1, 1]
```

sympy.polys.galoistools.**gf_add_mul**(*f*, *g*, *h*, *p*, *K*)

Returns `f + g*h` where `f`, `g`, `h` in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_add_mul
>>> gf_add_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[2, 3, 2, 2]
```

sympy.polys.galoistools.**gf_sub_mul**(*f*, *g*, *h*, *p*, *K*)

Compute `f - g*h` where `f`, `g`, `h` in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sub_mul
```

```
>>> gf_sub_mul([3, 2, 4], [2, 2, 2], [1, 4], 5, ZZ)
[3, 3, 2, 1]
```

sympy.polys.galoistools.**gf_expand**(*F*, *p*, *K*)

Expand results of *factor()* (page 2373) in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_expand
```

```
>>> gf_expand([([3, 2, 4], 1), ([2, 2], 2), ([3, 1], 3)], 5, ZZ)
[4, 3, 0, 3, 0, 1, 4, 1]
```

sympy.polys.galoistools.**gf_div**(*f*, *g*, *p*, *K*)

Division with remainder in `GF(p)[x]`.

Given univariate polynomials `f` and `g` with coefficients in a finite field with `p` elements, returns polynomials `q` and `r` (quotient and remainder) such that `f = q*g + r`.

Consider polynomials `x**3 + x + 1` and `x**2 + x` in GF(2):

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_div, gf_add_mul

>>> gf_div(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
([1, 1], [1])
```

As result we obtained quotient `x + 1` and remainder 1, thus:

```
>>> gf_add_mul(ZZ.map([1]), ZZ.map([1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 0, 1, 1]
```

References

[R696], [R697]

sympy.polys.galoistools.**gf_rem**(*f*, *g*, *p*, *K*)

Compute polynomial remainder in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rem
```

```
>>> gf_rem(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1]
```

sympy.polys.galoistools.**gf_quo**(*f*, *g*, *p*, *K*)

Compute exact quotient in GF(p)[x].

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_quo
```

```
>>> gf_quo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
[1, 1]
>>> gf_quo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

sympy.polys.galoistools.**gf_exquo**(*f*, *g*, *p*, *K*)

Compute polynomial quotient in GF(p)[x].

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_exquo
```

```
>>> gf_exquo(ZZ.map([1, 0, 3, 2, 3]), ZZ.map([2, 2, 2]), 5, ZZ)
[3, 2, 4]
```

```
>>> gf_exquo(ZZ.map([1, 0, 1, 1]), ZZ.map([1, 1, 0]), 2, ZZ)
Traceback (most recent call last):
...
ExactQuotientFailed: [1, 1, 0] does not divide [1, 0, 1, 1]
```

sympy.polys.galoistools.**gf_lshift**(*f*, *n*, *K*)

Efficiently multiply f by x**n.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lshift
```

```
>>> gf_lshift([3, 2, 4], 4, ZZ)
[3, 2, 4, 0, 0, 0, 0]
```

sympy.polys.galoistools.**gf_rshift**(*f*, *n*, *K*)

Efficiently divide `f` by `x**n`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_rshift
```

```
>>> gf_rshift([1, 2, 3, 4, 0], 3, ZZ)
([1, 2], [3, 4, 0])
```

sympy.polys.galoistools.**gf_pow**(*f*, *n*, *p*, *K*)

Compute `f**n` in `GF(p)[x]` using repeated squaring.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow
```

```
>>> gf_pow([3, 2, 4], 3, 5, ZZ)
[2, 4, 4, 2, 2, 1, 4]
```

sympy.polys.galoistools.**gf_pow_mod**(*f*, *n*, *g*, *p*, *K*)

Compute `f**n` in `GF(p)[x]/(g)` using repeated squaring.

Given polynomials `f` and `g` in `GF(p)[x]` and a non-negative integer `n`, efficiently computes `f**n (mod g)` i.e. the remainder of `f**n` from division by `g`, using the repeated squaring algorithm.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_pow_mod
```

```
>>> gf_pow_mod(ZZ.map([3, 2, 4]), 3, ZZ.map([1, 1]), 5, ZZ)
[]
```

References

[R698]

sympy.polys.galoistools.**gf_gcd**(*f*, *g*, *p*, *K*)

Euclidean Algorithm in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcd
```

```
>>> gf_gcd(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 3]
```

sympy.polys.galoistools.**gf_lcm**(*f*, *g*, *p*, *K*)

Compute polynomial LCM in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_lcm
```

```
>>> gf_lcm(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
[1, 2, 0, 4]
```

sympy.polys.galoistools.**gf_cofactors**(*f*, *g*, *p*, *K*)

Compute polynomial GCD and cofactors in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_cofactors
```

```
>>> gf_cofactors(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 3]), 5, ZZ)
([1, 3], [3, 3], [2, 1])
```

sympy.polys.galoistools.**gf_gcdex**(*f*, *g*, *p*, *K*)

Extended Euclidean Algorithm in `GF(p)[x]`.

Given polynomials `f` and `g` in `GF(p)[x]`, computes polynomials `s`, `t` and `h`, such that `h = gcd(f, g)` and `s*f + t*g = h`. The typical application of EEA is solving polynomial diophantine equations.

Consider polynomials `f = (x + 7) (x + 1)`, `g = (x + 7) (x**2 + 1)` in `GF(11)[x]`. Application of Extended Euclidean Algorithm gives:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_gcdex, gf_mul, gf_add

>>> s, t, g = gf_gcdex(ZZ.map([1, 8, 7]), ZZ.map([1, 7, 1, 7]), 11, ZZ)
>>> s, t, g
([5, 6], [6], [1, 7])
```

As result we obtained polynomials `s = 5*x + 6` and `t = 6`, and additionally `gcd(f, g) = x + 7`. This is correct because:

```
>>> S = gf_mul(s, ZZ.map([1, 8, 7]), 11, ZZ)
>>> T = gf_mul(t, ZZ.map([1, 7, 1, 7]), 11, ZZ)

>>> gf_add(S, T, 11, ZZ) == [1, 7]
True
```

References

[R699]

sympy.polys.galoistools.**gf_monic**(*f*, *p*, *K*)

Compute LC and a monic polynomial in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_monic
```

```
>>> gf_monic(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [1, 4, 3])
```

sympy.polys.galoistools.**gf_diff**(*f*, *p*, *K*)

Differentiate polynomial in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_diff
```

```
>>> gf_diff([3, 2, 4], 5, ZZ)
[1, 2]
```

sympy.polys.galoistools.**gf_eval**(*f*, *a*, *p*, *K*)

Evaluate `f(a)` in `GF(p)` using Horner scheme.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_eval
```

```
>>> gf_eval([3, 2, 4], 2, 5, ZZ)
0
```

sympy.polys.galoistools.**gf_multi_eval**(*f*, *A*, *p*, *K*)

Evaluate `f(a)` for `a` in `[a_1, ..., a_n]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_multi_eval
```

```
>>> gf_multi_eval([3, 2, 4], [0, 1, 2, 3, 4], 5, ZZ)
[4, 4, 0, 2, 0]
```

sympy.polys.galoistools.**gf_compose**(*f*, *g*, *p*, *K*)

Compute polynomial composition `f(g)` in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose
```

```
>>> gf_compose([3, 2, 4], [2, 2, 2], 5, ZZ)
[2, 4, 0, 3, 0]
```

sympy.polys.galoistools.**gf_compose_mod**(*g*, *h*, *f*, *p*, *K*)

Compute polynomial composition `g(h)` in `GF(p)[x]/(f)`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_compose_mod
```

```
>>> gf_compose_mod(ZZ.map([3, 2, 4]), ZZ.map([2, 2, 2]), ZZ.map([4, 3]),␣
↪5, ZZ)
[4]
```

sympy.polys.galoistools.**gf_trace_map**(*a*, *b*, *c*, *n*, *f*, *p*, *K*)

Compute polynomial trace map in `GF(p)[x]/(f)`.

Given a polynomial `f` in `GF(p)[x]`, polynomials `a`, `b`, `c` in the quotient ring `GF(p)[x]/(f)` such that `b = c**t (mod f)` for some positive power `t` of `p`, and a positive integer `n`, returns a mapping:

```
a -> a**t**n, a + a**t + a**t**2 + ... + a**t**n (mod f)
```

In factorization context, `b = x**p mod f` and `c = x mod f`. This way we can efficiently compute trace polynomials in equal degree factorization routine, much faster than with other methods, like iterated Frobenius algorithm, for large degrees.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_trace_map
```

```
>>> gf_trace_map([1, 2], [4, 4], [1, 1], 4, [3, 2, 4], 5, ZZ)
([1, 3], [1, 3])
```

References

[R700]

sympy.polys.galoistools.**gf_random**(*n*, *p*, *K*)

Generate a random polynomial in `GF(p)[x]` of degree `n`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_random
>>> gf_random(10, 5, ZZ)
[1, 2, 3, 2, 1, 1, 1, 2, 0, 4, 2]
```

sympy.polys.galoistools.**gf_irreducible**(*n*, *p*, *K*)

Generate random irreducible polynomial of degree `n` in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible
>>> gf_irreducible(10, 5, ZZ)
[1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]
```

sympy.polys.galoistools.**gf_irreducible_p**(*f*, *p*, *K*)

Test irreducibility of a polynomial `f` in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_irreducible_p
```

```
>>> gf_irreducible_p(ZZ.map([1, 4, 2, 2, 3, 2, 4, 1, 4, 0, 4]), 5, ZZ)
True
>>> gf_irreducible_p(ZZ.map([3, 2, 4]), 5, ZZ)
False
```

sympy.polys.galoistools.**gf_sqf_p**(*f*, *p*, *K*)

Return `True` if `f` is square-free in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_p
```

```
>>> gf_sqf_p(ZZ.map([3, 2, 4]), 5, ZZ)
True
>>> gf_sqf_p(ZZ.map([2, 4, 4, 2, 2, 1, 4]), 5, ZZ)
False
```

sympy.polys.galoistools.**gf_sqf_part**(*f*, *p*, *K*)

Return square-free part of a `GF(p)[x]` polynomial.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_sqf_part
```

```
>>> gf_sqf_part(ZZ.map([1, 1, 3, 0, 1, 0, 2, 2, 1]), 5, ZZ)
[1, 4, 3]
```

sympy.polys.galoistools.**gf_sqf_list**(*f*, *p*, *K*, *all=False*)

Return the square-free decomposition of a `GF(p)[x]` polynomial.

Given a polynomial `f` in `GF(p)[x]`, returns the leading coefficient of `f` and a square-free decomposition `f_1**e_1 f_2**e_2 ... f_k**e_k` such that all `f_i` are monic polynomials and `(f_i, f_j)` for `i != j` are co-prime and `e_1 ... e_k` are given in increasing order. All trivial terms (i.e. `f_i = 1`) are not included in the output.

Consider polynomial `f = x**11 + 1` over `GF(11)[x]`:

```
>>> from sympy.polys.domains import ZZ

>>> from sympy.polys.galoistools import (
...     gf_from_dict, gf_diff, gf_sqf_list, gf_pow,
... )
...
```

(continues on next page)

(continued from previous page)

```
>>> f = gf_from_dict({11: ZZ(1), 0: ZZ(1)}, 11, ZZ)
```

Note that `f'(x) = 0`:

```
>>> gf_diff(f, 11, ZZ)
[]
```

This phenomenon does not happen in characteristic zero. However we can still compute square-free decomposition of `f` using `gf_sqf()`:

```
>>> gf_sqf_list(f, 11, ZZ)
(1, [([1, 1], 11)])
```

We obtained factorization `f = (x + 1)**11`. This is correct because:

```
>>> gf_pow([1, 1], 11, 11, ZZ) == f
True
```

References

[R701]

sympy.polys.galoistools.**gf_Qmatrix**(*f*, *p*, *K*)

Calculate Berlekamp's `Q` matrix.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix
```

```
>>> gf_Qmatrix([3, 2, 4], 5, ZZ)
[[1, 0],
 [3, 4]]
```

```
>>> gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 0, 0],
 [0, 4, 0, 0],
 [0, 0, 1, 0],
 [0, 0, 0, 4]]
```

sympy.polys.galoistools.**gf_Qbasis**(*Q*, *p*, *K*)

Compute a basis of the kernel of `Q`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_Qmatrix, gf_Qbasis
```

```
>>> gf_Qbasis(gf_Qmatrix([1, 0, 0, 0, 1], 5, ZZ), 5, ZZ)
[[1, 0, 0, 0], [0, 0, 1, 0]]
```

```
>>> gf_Qbasis(gf_Qmatrix([3, 2, 4], 5, ZZ), 5, ZZ)
[[1, 0]]
```

sympy.polys.galoistools.**gf_berlekamp**(*f*, *p*, *K*)

Factor a square-free `f` in `GF(p)[x]` for small `p`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_berlekamp
```

```
>>> gf_berlekamp([1, 0, 0, 0, 1], 5, ZZ)
[[1, 0, 2], [1, 0, 3]]
```

sympy.polys.galoistools.**gf_zassenhaus**(*f*, *p*, *K*)

Factor a square-free `f` in `GF(p)[x]` for medium `p`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_zassenhaus
```

```
>>> gf_zassenhaus(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

sympy.polys.galoistools.**gf_shoup**(*f*, *p*, *K*)

Factor a square-free `f` in `GF(p)[x]` for large `p`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_shoup
```

```
>>> gf_shoup(ZZ.map([1, 4, 3]), 5, ZZ)
[[1, 1], [1, 3]]
```

sympy.polys.galoistools.**gf_factor_sqf**(*f*, *p*, *K*, *method=None*)

Factor a square-free polynomial `f` in `GF(p)[x]`.

Examples

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor_sqf
```

```
>>> gf_factor_sqf(ZZ.map([3, 2, 4]), 5, ZZ)
(3, [[1, 1], [1, 3]])
```

sympy.polys.galoistools.**gf_factor**(*f*, *p*, *K*)

Factor (non square-free) polynomials in `GF(p)[x]`.

Given a possibly non square-free polynomial `f` in `GF(p)[x]`, returns its complete factorization into irreducibles:

```
f_1(x)**e_1 f_2(x)**e_2 ... f_d(x)**e_d
```

where each `f_i` is a monic polynomial and `gcd(f_i, f_j) == 1`, for `i != j`. The result is given as a tuple consisting of the leading coefficient of `f` and a list of factors of `f` with their multiplicities.

The algorithm proceeds by first computing square-free decomposition of `f` and then iteratively factoring each of square-free factors.

Consider a non square-free polynomial `f = (7*x + 1) (x + 2)**2` in `GF(11)[x]`. We obtain its factorization into irreducibles as follows:

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.galoistools import gf_factor

>>> gf_factor(ZZ.map([5, 2, 7, 2]), 11, ZZ)
(5, [([1, 2], 1), ([1, 8], 2)])
```

We arrived with factorization `f = 5 (x + 2) (x + 8)**2`. We did not recover the exact form of the input polynomial because we requested to get monic factors of `f` and its leading coefficient separately.

Square-free factors of `f` can be factored into irreducibles over `GF(p)` using three very different methods:

Berlekamp
: efficient for very small values of `p` (usually `p < 25`)

Cantor-Zassenhaus
: efficient on average input and with “typical” `p`

Shoup-Kaltofen-Gathen
: efficient with very large inputs and modulus

If you want to use a specific factorization method, instead of the default one, set `GF_FACTOR_METHOD` with one of `berlekamp`, `zassenhaus` or `shoup` values.

References

[R702]

sympy.polys.galoistools.**gf_value**(*f*, *a*)

Value of polynomial 'f' at 'a' in field R.

Examples

```
>>> from sympy.polys.galoistools import gf_value
```

```
>>> gf_value([1, 7, 2, 4], 11)
2204
```

sympy.polys.galoistools.**gf_csolve**(*f*, *n*)

To solve f(x) congruent 0 mod(n).

n is divided into canonical factors and f(x) cong 0 mod(p**e) will be solved for each factor. Applying the Chinese Remainder Theorem to the results returns the final answers.

Examples

Solve [1, 1, 7] congruent 0 mod(189):

```
>>> from sympy.polys.galoistools import gf_csolve
>>> gf_csolve([1, 1, 7], 189)
[13, 49, 76, 112, 139, 175]
```

References

[R703]

Manipulation of sparse, distributed polynomials and vectors

Dense representations quickly require infeasible amounts of storage and computation time if the number of variables increases. For this reason, there is code to manipulate polynomials in a *sparse* representation. The Ring object and elements are implemented by the classes *PolyRing* (page 2553) and *PolyElement* (page 2554).

In commutative algebra, one often studies not only polynomials, but also *modules* over polynomial rings. The polynomial manipulation module provides rudimentary low-level support for finitely generated free modules. This is mainly used for Groebner basis computations (see there), so manipulation functions are only provided to the extend needed. They carry the prefix `sdm_`. Note that in examples, the generators of the free module are called $f_1, f_2, \ldots$.

sympy.polys.distributedmodules.**sdm_monomial_mul**(*M*, *X*)

Multiply tuple `X` representing a monomial of $K[X]$ into the tuple `M` representing a monomial of F.

Examples

Multiplying xy^3 into xf_1 yields $x^2y^3f_1$:

```
>>> from sympy.polys.distributedmodules import sdm_monomial_mul
>>> sdm_monomial_mul((1, 1, 0), (1, 3))
(1, 2, 3)
```

sympy.polys.distributedmodules.**sdm_monomial_deg**(*M*)

Return the total degree of M.

Examples

For example, the total degree of x^2yf_5 is 3:

```
>>> from sympy.polys.distributedmodules import sdm_monomial_deg
>>> sdm_monomial_deg((5, 2, 1))
3
```

sympy.polys.distributedmodules.**sdm_monomial_divides**(*A*, *B*)

Does there exist a (polynomial) monomial X such that XA = B?

Examples

Positive examples:

In the following examples, the monomial is given in terms of x, y and the generator(s), f_1, f_2 etc. The tuple form of that monomial is used in the call to sdm_monomial_divides. Note: the generator appears last in the expression but first in the tuple and other factors appear in the same order that they appear in the monomial expression.

$A = f_1$ divides $B = f_1$

```
>>> from sympy.polys.distributedmodules import sdm_monomial_divides
>>> sdm_monomial_divides((1, 0, 0), (1, 0, 0))
True
```

$A = f_1$ divides $B = x^2yf_1$

```
>>> sdm_monomial_divides((1, 0, 0), (1, 2, 1))
True
```

$A = xyf_5$ divides $B = x^2yf_5$

```
>>> sdm_monomial_divides((5, 1, 1), (5, 2, 1))
True
```

Negative examples:

$A = f_1$ does not divide $B = f_2$

```
>>> sdm_monomial_divides((1, 0, 0), (2, 0, 0))
False
```

$A = xf_1$ does not divide $B = f_1$

```
>>> sdm_monomial_divides((1, 1, 0), (1, 0, 0))
False
```

$A = xy^2 f_5$ does not divide $B = yf_5$

```
>>> sdm_monomial_divides((5, 1, 2), (5, 0, 1))
False
```

sympy.polys.distributedmodules.**sdm_LC**(*f*, *K*)

Returns the leading coeffcient of `f`.

sympy.polys.distributedmodules.**sdm_to_dict**(*f*)

Make a dictionary from a distributed polynomial.

sympy.polys.distributedmodules.**sdm_from_dict**(*d*, *O*)

Create an sdm from a dictionary.

Here `O` is the monomial order to use.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 1, 0): QQ(1), (1, 0, 0): QQ(2), (0, 1, 0): QQ(0)}
>>> sdm_from_dict(dic, lex)
[((1, 1, 0), 1), ((1, 0, 0), 2)]
```

sympy.polys.distributedmodules.**sdm_add**(*f*, *g*, *O*, *K*)

Add two module elements `f`, `g`.

Addition is done over the ground field `K`, monomials are ordered according to `O`.

Examples

All examples use lexicographic order.

$(xyf_1) + (f_2) = f_2 + xyf_1$

```
>>> from sympy.polys.distributedmodules import sdm_add
>>> from sympy.polys import lex, QQ
>>> sdm_add([((1, 1, 1), QQ(1))], [((2, 0, 0), QQ(1))], lex, QQ)
[((2, 0, 0), 1), ((1, 1, 1), 1)]
```

$(xyf_1) + (-xyf_1)$ = 0`

```
>>> sdm_add([((1, 1, 1), QQ(1))], [((1, 1, 1), QQ(-1))], lex, QQ)
[]
```

$(f_1) + (2f_1) = 3f_1$

```
>>> sdm_add([((1, 0, 0), QQ(1))], [((1, 0, 0), QQ(2))], lex, QQ)
[((1, 0, 0), 3)]
```

$(yf_1) + (xf_1) = xf_1 + yf_1$

```
>>> sdm_add([((1, 0, 1), QQ(1))], [((1, 1, 0), QQ(1))], lex, QQ)
[((1, 1, 0), 1), ((1, 0, 1), 1)]
```

sympy.polys.distributedmodules.**sdm_LM**(*f*)

Returns the leading monomial of `f`.

Only valid if $f \neq 0$.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_LM, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(1), (4, 0, 1): QQ(1)}
>>> sdm_LM(sdm_from_dict(dic, lex))
(4, 0, 1)
```

sympy.polys.distributedmodules.**sdm_LT**(*f*)

Returns the leading term of `f`.

Only valid if $f \neq 0$.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_LT, sdm_from_dict
>>> from sympy.polys import QQ, lex
>>> dic = {(1, 2, 3): QQ(1), (4, 0, 0): QQ(2), (4, 0, 1): QQ(3)}
>>> sdm_LT(sdm_from_dict(dic, lex))
((4, 0, 1), 3)
```

sympy.polys.distributedmodules.**sdm_mul_term**(*f*, *term*, *O*, *K*)

Multiply a distributed module element `f` by a (polynomial) term `term`.

Multiplication of coefficients is done over the ground field `K`, and monomials are ordered according to `O`.

Examples

$0f_1 = 0$

```
>>> from sympy.polys.distributedmodules import sdm_mul_term
>>> from sympy.polys import lex, QQ
>>> sdm_mul_term([((1, 0, 0), QQ(1))], ((0, 0), QQ(0)), lex, QQ)
[]
```

$x0 = 0$

```
>>> sdm_mul_term([], ((1, 0), QQ(1)), lex, QQ)
[]
```

$(x)(f_1) = xf_1$

```
>>> sdm_mul_term([((1, 0, 0), QQ(1))], ((1, 0), QQ(1)), lex, QQ)
[((1, 1, 0), 1)]
```

$(2xy)(3xf_1 + 4yf_2) = 8xy^2 f_2 + 6x^2 y f_1$

```
>>> f = [((2, 0, 1), QQ(4)), ((1, 1, 0), QQ(3))]
>>> sdm_mul_term(f, ((1, 1), QQ(2)), lex, QQ)
[((2, 1, 2), 8), ((1, 2, 1), 6)]
```

sympy.polys.distributedmodules.**sdm_zero**()

Return the zero module element.

sympy.polys.distributedmodules.**sdm_deg**(*f*)

Degree of `f`.

This is the maximum of the degrees of all its monomials. Invalid if `f` is zero.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_deg
>>> sdm_deg([((1, 2, 3), 1), ((10, 0, 1), 1), ((2, 3, 4), 4)])
7
```

sympy.polys.distributedmodules.**sdm_from_vector**(*vec*, *O*, *K*, ***opts*)

Create an sdm from an iterable of expressions.

Coefficients are created in the ground field `K`, and terms are ordered according to monomial order `O`. Named arguments are passed on to the polys conversion code and can be used to specify for example generators.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_from_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ, lex
>>> sdm_from_vector([x**2+y**2, 2*z], lex, QQ)
[((1, 0, 0, 1), 2), ((0, 2, 0, 0), 1), ((0, 0, 2, 0), 1)]
```

sympy.polys.distributedmodules.**sdm_to_vector**(*f*, *gens*, *K*, *n=None*)

Convert sdm `f` into a list of polynomial expressions.

The generators for the polynomial ring are specified via `gens`. The rank of the module is guessed, or passed via `n`. The ground field is assumed to be `K`.

Examples

```
>>> from sympy.polys.distributedmodules import sdm_to_vector
>>> from sympy.abc import x, y, z
>>> from sympy.polys import QQ
>>> f = [((1, 0, 0, 1), QQ(2)), ((0, 2, 0, 0), QQ(1)), ((0, 0, 2, 0),␣
↪QQ(1))]
>>> sdm_to_vector(f, [x, y, z], QQ)
[x**2 + y**2, 2*z]
```

Polynomial factorization algorithms

Many variants of Euclid's algorithm:

Classical remainder sequence

Let K be a field, and consider the ring $K[X]$ of polynomials in a single indeterminate X with coefficients in K. Given two elements f and g of $K[X]$ with $g \neq 0$ there are unique polynomials q and r such that $f = qg + r$ and $\deg(r) < \deg(g)$ or $r = 0$. They are denoted by $\mathrm{quo}(f, g)$ (*quotient*) and $\mathrm{rem}(f, g)$ (*remainder*), so we have the *division identity*

$$f = \mathrm{quo}(f, g)g + \mathrm{rem}(f, g).$$

It follows that every ideal I of $K[X]$ is a principal ideal, generated by any element $\neq 0$ of minimum degree (assuming I non-zero). In fact, if g is such a polynomial and f is any element of I, $\mathrm{rem}(f, g)$ belongs to I as a linear combination of f and g, hence must be zero; therefore f is a multiple of g.

Using this result it is possible to find a greatest common divisor (gcd) of any polynomials $f, g, \ldots$ in $K[X]$. If I is the ideal formed by all linear combinations of the given polynomials with coefficients in $K[X]$, and d is its generator, then every common divisor of the polynomials also divides d. On the other hand, the given polynomials are multiples of the generator d; hence d is a gcd of the polynomials, denoted $\gcd(f, g, \ldots)$.

An algorithm for the gcd of two polynomials f and g in $K[X]$ can now be obtained as follows. By the division identity, $r = \mathrm{rem}(f, g)$ is in the ideal generated by f and g, as well as f is in the ideal generated by g and r. Hence the ideals generated by the pairs (f, g) and (g, r) are the same. Set $f_0 = f$, $f_1 = g$, and define recursively $f_i = \mathrm{rem}(f_{i-2}, f_{i-1})$ for $i \geq 2$. The recursion ends after a finite number of steps with $f_{k+1} = 0$, since the degrees of the polynomials are strictly decreasing. By the above remark, all the pairs (f_{i-1}, f_i) generate the same ideal. In particular, the ideal generated by f and g is generated by f_k alone as $f_{k+1} = 0$. Hence $d = f_k$ is a gcd of f and g.

The sequence of polynomials $f_0, f_1, \ldots, f_k$ is called the *Euclidean polynomial remainder sequence* determined by (f, g) because of the analogy with the classical Euclidean algorithm for the gcd of natural numbers.

The algorithm may be extended to obtain an expression for d in terms of f and g by using the full division identities to write recursively each f_i as a linear combination of f and g. This leads to an equation

$$d = uf + vg \qquad (u, v \in K[X])$$

analogous to Bézout's identity in the case of integers.

sympy.polys.euclidtools.**dmp_half_gcdex**(*f*, *g*, *u*, *K*)

Half extended Euclidean algorithm in $F[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

sympy.polys.euclidtools.**dmp_gcdex**(*f*, *g*, *u*, *K*)

Extended Euclidean algorithm in $F[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

sympy.polys.euclidtools.**dmp_invert**(*f*, *g*, *u*, *K*)

Compute multiplicative inverse of f modulo g in $F[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x = ring("x", QQ)
```

sympy.polys.euclidtools.**dmp_euclidean_prs**(*f*, *g*, *u*, *K*)

Euclidean polynomial remainder sequence (PRS) in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

Simplified remainder sequences

Assume, as is usual, that the coefficient field K is the field of fractions of an integral domain A. In this case the coefficients (numerators and denominators) of the polynomials in the Euclidean remainder sequence tend to grow very fast.

If A is a unique factorization domain, the coefficients may be reduced by cancelling common factors of numerators and denominators. Further reduction is possible noting that a gcd of polynomials in $K[X]$ is not unique: it may be multiplied by any (non-zero) constant factor.

Any polynomial f in $K[X]$ can be simplified by extracting the denominators and common factors of the numerators of its coefficients. This yields the representation $f = cF$ where $c \in K$ is the *content* of f and F is a *primitive* polynomial, i.e., a polynomial in $A[X]$ with coprime coefficients.

It is possible to start the algorithm by replacing the given polynomials f and g with their primitive parts. This will only modify $\operatorname{rem}(f,g)$ by a constant factor. Replacing it with its primitive part and continuing recursively we obtain all the primitive parts of the polynomials in the Euclidean remainder sequence, including the primitive $\gcd(f,g)$.

This sequence is the *primitive polynomial remainder sequence*. It is an example of *general polynomial remainder sequences* where the computed remainders are modified by constant multipliers (or divisors) in order to simplify the results.

sympy.polys.euclidtools.**dmp_primitive_prs**(*f*, *g*, *u*, *K*)

Primitive polynomial remainder sequence (PRS) in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

Subresultant sequence

The coefficients of the primitive polynomial sequence do not grow exceedingly, but the computation of the primitive parts requires extra processing effort. Besides, the method only works with fraction fields of unique factorization domains, excluding, for example, the general number fields.

Collins [Collins67] realized that the so-called *subresultant polynomials* of a pair of polynomials also form a generalized remainder sequence. The coefficients of these polynomials are expressible as determinants in the coefficients of the given polynomials. Hence (the logarithm of) their size only grows linearly. In addition, if the coefficients of the given polynomials are in the subdomain A, so are those of the subresultant polynomials. This means that the subresultant sequence is comparable to the primitive remainder sequence without relying on unique factorization in A.

To see how subresultants are associated with remainder sequences recall that all polynomials h in the sequence are linear combinations of the given polynomials f and g

$$h = uf + vg$$

with polynomials u and v in $K[X]$. Moreover, as is seen from the extended Euclidean algorithm, the degrees of u and v are relatively low, with limited growth from step to step.

Let $n = \deg(f)$, and $m = \deg(g)$, and assume $n \geq m$. If $\deg(h) = j < m$, the coefficients of the powers X^k ($k > j$) in the products uf and vg cancel each other. In particular, the products must have the same degree, say, l. Then $\deg(u) = l - n$ and $\deg(v) = l - m$ with a total of $2l - n - m + 2$ coefficients to be determined.

On the other hand, the equality $h = uf + vg$ implies that $l - j$ linear combinations of the coefficients are zero, those associated with the powers X^i ($j < i \leq l$), and one has a given non-zero value, namely the leading coefficient of h.

To satisfy these $l - j + 1$ linear equations the total number of coefficients to be determined cannot be lower than $l - j + 1$, in general. This leads to the inequality $l \geq n + m - j - 1$. Taking $l = n + m - j - 1$, we obtain $\deg(u) = m - j - 1$ and $\deg(v) = n - j - 1$.

In the case $j = 0$ the matrix of the resulting system of linear equations is the Sylvester matrix $S(f,g)$ associated to f and g, an $(n+m) \times (n+m)$ matrix with coefficients of f and g as entries.

Its determinant is the resultant $\text{res}(f, g)$ of the pair (f, g). It is non-zero if and only if f and g are relatively prime.

For any j in the interval from 0 to m the matrix of the linear system is an $(n+m-2j)\times(n+m-2j)$ submatrix of the Sylvester matrix. Its determinant $s_j(f, g)$ is called the j th *scalar subresultant* of f and g.

If $s_j(f, g)$ is not zero, the associated equation $h = uf+vg$ has a unique solution where $\deg(h) = j$ and the leading coefficient of h has any given value; the one with leading coefficient $s_j(f, g)$ is the j th *subresultant polynomial* or, briefly, *subresultant* of the pair (f, g), and denoted $S_j(f, g)$. This choice guarantees that the remainining coefficients are also certain subdeterminants of the Sylvester matrix. In particular, if f and g are in $A[X]$, so is $S_j(f, g)$ as well. This construction of subresultants applies to any j between 0 and m regardless of the value of $s_j(f, g)$; if it is zero, then $\deg(S_j(f, g)) < j$.

The properties of subresultants are as follows. Let $n_0 = \deg(f)$, $n_1 = \deg(g)$, $n_2, \ldots, n_k$ be the decreasing sequence of degrees of polynomials in a remainder sequence. Let $0 \le j \le n_1$; then

- $s_j(f, g) \neq 0$ if and only if $j = n_i$ for some i.
- $S_j(f, g) \neq 0$ if and only if $j = n_i$ or $j = n_i - 1$ for some i.

Normally, $n_{i-1} - n_i = 1$ for $1 < i \le k$. If $n_{i-1} - n_i > 1$ for some i (the *abnormal* case), then $S_{n_{i-1}-1}(f, g)$ and $S_{n_i}(f, g)$ are constant multiples of each other. Hence either one could be included in the polynomial remainder sequence. The former is given by smaller determinants, so it is expected to have smaller coefficients.

Collins defined the *subresultant remainder sequence* by setting

$$f_i = S_{n_{i-1}-1}(f, g) \quad (2 \le i \le k).$$

In the normal case, these are the same as the $S_{n_i}(f, g)$. He also derived expressions for the constants γ_i in the remainder formulas

$$\gamma_i f_i = \text{rem}(f_{i-2}, f_{i-1})$$

in terms of the leading coefficients of $f_1, \ldots, f_{i-1}$, working in the field K.

Brown and Traub [BrownTraub71] later developed a recursive procedure for computing the coefficients γ_i. Their algorithm deals with elements of the domain A exclusively (assuming $f, g \in A[X]$). However, in the abnormal case there was a problem, a division in A which could only be conjectured to be exact.

This was subsequently justified by Brown [Brown78] who showed that the result of the division is, in fact, a scalar subresultant. More specifically, the constant appearing in the computation of f_i is $s_{n_{i-2}}(f, g)$ (Theorem 3). The implication of this discovery is that the scalar subresultants are computed as by-products of the algorithm, all but $s_{n_k}(f, g)$ which is not needed after finding $f_{k+1} = 0$. Completing the last step we obtain all non-zero scalar subresultants, including the last one which is the resultant if this does not vanish.

`sympy.polys.euclidtools.`**`dmp_inner_subresultants`**`(f, g, u, K)`

Subresultant PRS algorithm in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y␣
↪+ 16
```

```
>>> prs = [f, g, a, b]
>>> sres = [[1], [1], [3, 0, 0, 0, 0], [-3, 0, 0, -12, 1, 0, -54, 8, 729,
↪ -216, 16]]
```

```
>>> R.dmp_inner_subresultants(f, g) == (prs, sres)
True
```

sympy.polys.euclidtools.**dmp_subresultants**(*f*, *g*, *u*, *K*)

Computes subresultant PRS of two polynomials in $K[X]$.

Examples


```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y␣
↪+ 16
```

```
>>> R.dmp_subresultants(f, g) == [f, g, a, b]
True
```

sympy.polys.euclidtools.**dmp_prs_resultant**(*f*, *g*, *u*, *K*)

Resultant algorithm in $K[X]$ using subresultant PRS.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> a = 3*x*y**4 + y**3 - 27*y + 4
>>> b = -3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y␣
↪+ 16
```

```
>>> res, prs = R.dmp_prs_resultant(f, g)
```

```
>>> res == b              # resultant has n-1 variables
False
>>> res == b.drop(x)
True
>>> prs == [f, g, a, b]
True
```

sympy.polys.euclidtools.**dmp_zz_modular_resultant**(*f*, *g*, *p*, *u*, *K*)

Compute resultant of f and g modulo a prime p.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_modular_resultant(f, g, 5)
-2*y**2 + 1
```

sympy.polys.euclidtools.**dmp_zz_collins_resultant**(*f*, *g*, *u*, *K*)

Collins's modular resultant algorithm in $Z[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = x + y + 2
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_zz_collins_resultant(f, g)
-2*y**2 - 5*y + 1
```

sympy.polys.euclidtools.**dmp_qq_collins_resultant**(*f*, *g*, *u*, *K0*)

Collins's modular resultant algorithm in $Q[X]$.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x + y + QQ(2,3)
>>> g = 2*x*y + x + 3
```

```
>>> R.dmp_qq_collins_resultant(f, g)
-2*y**2 - 7/3*y + 5/6
```

sympy.polys.euclidtools.**dmp_resultant**(*f*, *g*, *u*, *K*, *includePRS=False*)

Computes resultant of two polynomials in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> f = 3*x**2*y - y**3 - 4
>>> g = x**2 + x*y**3 - 9
```

```
>>> R.dmp_resultant(f, g)
-3*y**10 - 12*y**7 + y**6 - 54*y**4 + 8*y**3 + 729*y**2 - 216*y + 16
```

sympy.polys.euclidtools.**dmp_discriminant**(*f*, *u*, *K*)

Computes discriminant of a polynomial in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y,z,t = ring("x,y,z,t", ZZ)
```

```
>>> R.dmp_discriminant(x**2*y + x*z + t)
-4*y*t + z**2
```

sympy.polys.euclidtools.**dmp_rr_prs_gcd**(*f*, *g*, *u*, *K*)

Computes polynomial GCD using subresultants over a ring.

Returns `(h, cff, cfg)` such that `a = gcd(f, g)`, `cff = quo(f, h)`, and `cfg = quo(g, h)`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_rr_prs_gcd(f, g)
(x + y, x + y, x)
```

sympy.polys.euclidtools.**dmp_ff_prs_gcd**(*f*, *g*, *u*, *K*)

Computes polynomial GCD using subresultants over a field.

Returns `(h, cff, cfg)` such that `a = gcd(f, g)`, `cff = quo(f, h)`, and `cfg = quo(g, h)`.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,2)*x**2 + x*y + QQ(1,2)*y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_ff_prs_gcd(f, g)
(x + y, 1/2*x + 1/2*y, x)
```

sympy.polys.euclidtools.**dmp_zz_heu_gcd**(*f*, *g*, *u*, *K*)

Heuristic polynomial GCD in $Z[X]$.

Given univariate polynomials f and g in $Z[X]$, returns their GCD and cofactors, i.e. polynomials `h`, `cff` and `cfg` such that:

```
h = gcd(f, g), cff = quo(f, h) and cfg = quo(g, h)
```

The algorithm is purely heuristic which means it may fail to compute the GCD. This will be signaled by raising an exception. In this case you will need to switch to another GCD method.

The algorithm computes the polynomial GCD by evaluating polynomials f and g at certain points and computing (fast) integer GCD of those evaluations. The polynomial GCD is recovered from the integer image by interpolation. The evaluation process reduces f and g variable by variable into a large integer. The final step is to verify if the interpolated polynomial is the correct GCD. This gives cofactors of the input polynomials as a side effect.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_zz_heu_gcd(f, g)
(x + y, x + y, x)
```

References

[R704]

sympy.polys.euclidtools.**dmp_qq_heu_gcd**(*f*, *g*, *u*, *K0*)

Heuristic polynomial GCD in $Q[X]$.

Returns `(h, cff, cfg)` such that `a = gcd(f, g)`, `cff = quo(f, h)`, and `cfg = quo(g, h)`.

Examples

```
>>> from sympy.polys import ring, QQ
>>> R, x,y, = ring("x,y", QQ)
```

```
>>> f = QQ(1,4)*x**2 + x*y + y**2
>>> g = QQ(1,2)*x**2 + x*y
```

```
>>> R.dmp_qq_heu_gcd(f, g)
(x + 2*y, 1/4*x + 1/2*y, 1/2*x)
```

sympy.polys.euclidtools.**dmp_inner_gcd**(*f*, *g*, *u*, *K*)

Computes polynomial GCD and cofactors of f and g in $K[X]$.

Returns `(h, cff, cfg)` such that `a = gcd(f, g)`, `cff = quo(f, h)`, and `cfg = quo(g, h)`.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_inner_gcd(f, g)
(x + y, x + y, x)
```

sympy.polys.euclidtools.**dmp_gcd**(*f*, *g*, *u*, *K*)

Computes polynomial GCD of f and g in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_gcd(f, g)
x + y
```

sympy.polys.euclidtools.**dmp_lcm**(*f*, *g*, *u*, *K*)

Computes polynomial LCM of f and g in $K[X]$.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> f = x**2 + 2*x*y + y**2
>>> g = x**2 + x*y
```

```
>>> R.dmp_lcm(f, g)
x**3 + 2*x**2*y + x*y**2
```

sympy.polys.euclidtools.**dmp_content**(*f*, *u*, *K*)

Returns GCD of multivariate coefficients.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_content(2*x*y + 6*x + 4*y + 12)
2*y + 6
```

sympy.polys.euclidtools.**dmp_primitive**(*f*, *u*, *K*)

Returns multivariate content and a primitive polynomial.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y, = ring("x,y", ZZ)
```

```
>>> R.dmp_primitive(2*x*y + 6*x + 4*y + 12)
(2*y + 6, x + 2)
```

sympy.polys.euclidtools.**dmp_cancel**(*f*, *g*, *u*, *K*, *include=True*)

Cancel common factors in a rational function f/g.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x,y = ring("x,y", ZZ)
```

```
>>> R.dmp_cancel(2*x**2 - 2, x**2 - 2*x + 1)
(2*x + 2, x - 1)
```

Polynomial factorization in characteristic zero:

sympy.polys.factortools.**dmp_trial_division**(*f*, *factors*, *u*, *K*)

Determine multiplicities of factors for a multivariate polynomial using trial division.

sympy.polys.factortools.**dmp_zz_mignotte_bound**(*f*, *u*, *K*)

Mignotte bound for multivariate polynomials in $K[X]$.

sympy.polys.factortools.**dup_zz_hensel_step**(*m*, *f*, *g*, *h*, *s*, *t*, *K*)

One step in Hensel lifting in $Z[x]$.

Given positive integer m and $Z[x]$ polynomials f, g, h, s and t such that:

```
f = g*h (mod m)
s*g + t*h = 1 (mod m)

lc(f) is not a zero divisor (mod m)
lc(h) = 1

deg(f) = deg(g) + deg(h)
deg(s) < deg(h)
deg(t) < deg(g)
```

returns polynomials G, H, S and T, such that:

```
f = G*H (mod m**2)
S*G + T*H = 1 (mod m**2)
```

References

[R705]

sympy.polys.factortools.**dup_zz_hensel_lift**(*p*, *f*, *f_list*, *l*, *K*)

Multifactor Hensel lifting in $Z[x]$.

Given a prime p, polynomial f over $Z[x]$ such that $lc(f)$ is a unit modulo p, monic pair-wise coprime polynomials f_i over $Z[x]$ satisfying:

```
f = lc(f) f_1 ... f_r (mod p)
```

and a positive integer l, returns a list of monic polynomials $F_1, F_2, \ldots, F_r$ satisfying:

```
f = lc(f) F_1 ... F_r (mod p**l)

F_i = f_i (mod p), i = 1..r
```

References

[R706]

sympy.polys.factortools.**dup_zz_zassenhaus**(*f*, *K*)

Factor primitive square-free polynomials in $Z[x]$.

sympy.polys.factortools.**dup_zz_irreducible_p**(*f*, *K*)

Test irreducibility using Eisenstein's criterion.

sympy.polys.factortools.**dup_cyclotomic_p**(*f*, *K*, *irreducible=False*)

Efficiently test if `f` is a cyclotomic polynomial.

Examples

```
>>> from sympy.polys import ring, ZZ
>>> R, x = ring("x", ZZ)
```

```
>>> f = x**16 + x**14 - x**10 + x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(f)
False
```

```
>>> g = x**16 + x**14 - x**10 - x**8 - x**6 + x**2 + 1
>>> R.dup_cyclotomic_p(g)
True
```