# `polylib` — library for polynomial manipulation

## Table of contents

The library `polylib` provides functions for polynomials, i. e. such who accept polynomials as arguments as well as such returning polynomials. Many of the first kind also accept polynomial expressions.

## `polylib::Dpoly` – differential operator for polynomials

If `f` is a polynomial in the indeterminates `x1` through xn, `polylib::Dpoly([i1,..,ik], f)` computes the $k$-th partial derivative $\frac{\partial^k f}{\partial x_{i1} \ldots \partial x_{ik}}$.

`polylib::Dpoly(f)` returns the derivative of `f` with respect to its only variable for an univariate polynomial `f`.

**Call(s):**

- `polylib::Dpoly(f)`
- `polylib::Dpoly(indexlist, f)`

**Parameters:**

| | | |
|---|---|---|
| `f` | — | polynomial |
| `indexlist` | — | list of positive integers |

**Return Value:** `polylib::Dpoly` returns a polynomial in the same indeterminates and over the same coefficient ring as the input.

**Overloadable by:** `f`

**Related Functions:** `D`, `diff`

---

**Details:**

- If some element of `indexlist` is greater than the number of indeterminates of `f`, the zero polynomial is returned.
- `polylib::Dpoly([ ], p)` returns `p`.
- If the coefficients of the polynomial are elements of a domain `d`, then this domain must have the method `"intmult": d::intmult(e,i)`, that must calculate the integer multiple of a domain element `e` and a positive integer `i`.
- `polylib::Dpoly` is a function of the system kernel.

---

**Example 1.** We differentiate a univariate polynomial with respect to its only indeterminate. In this case, we may leave out the first argument.

```
>> polylib::Dpoly(poly(2*x^2 + x + 1));

                        poly(4 x + 1, [x])
```

**Example 2.** No we differentiate a bivariate polynomial, and must specify the indeterminate in this case.

```
>> polylib::Dpoly([1], poly(x^2*y + 3*x + y, [x, y]));

                        poly(2 x y + 3, [x, y])
```

**Example 3.** It is also possible to compute second or higher partial derivatives.

```
>> polylib::Dpoly([1, 2], poly(x^2*y + 3*x + y, [x, y]));

                          poly(2 x, [x, y])
```

**Changes:**

- `polylib::Dpoly` used to be `Dpoly`.

---

## `polylib::Poly` – domain of polynomials

`polylib::Poly([x1,...,xn], R)` creates the ring of polynomials in the unknowns `x1` through `xn` over the coefficient ring `R`. If the argument `R` is missing, `Expr` is used.

**Domain:**

- `polylib::Poly([x1, ...]<, R>)`

**Parameters:**

`x1` — unknown
`R` — admissible coefficient ring for polynomials. See `poly`.

---

**Details:**

- `polylib::Poly` is a facade domain; it has no domain elements. It serves only as a coefficient ring for polynomials.
- The attempt to create an element of `polylib::Poly` results in a `DOM_POLY`.
- The arithmetical operations of the domain are realized by the corresponding kernel methods.

**Related Domains:** `Dom::DistributedPolynomial`

**Entries:**

| | |
|---|---|
| `zero` | the zero polynomial |
| `one` | the constant polynomial one |
| `indets` | list of unknowns |
| `coeffRing` | the coefficient ring `R` |

---

**Example 1.** `polylib::Poly` can be used for defining polynomials in `x` whose coefficients are polynomials in `y`. Such polynomials must not be confused with bivariate polynomials in `x` and `y`.

```
>> delete x,y: e:= x*(y^2*2 + y) + 3*y:
   poly(e, [x, y]);  poly(e, [x], polylib::Poly([y]))

                          2
                poly(2 x y  + x y + 3 y, [x, y])

                  2
     poly((y + 2 y ) x + 3 y, [x], polylib::Poly([y], Expr))
```

**Changes:**

- `polylib::Poly` used to be `Poly`.

---

## `polylib::cyclotomic` – cyclotomic polynomials

`polylib::cyclotomic(n, x)` computes the $n$-th cyclotomic polynomial, expressed in the indeterminate $x$.

**Call(s):**

- `polylib::cyclotomic(n, x)`

**Parameters:**

`n` — positive integer
`x` — identifier

**Return Value:** a polynomial over `Expr` in the indeterminate `x`.

**Related Functions:** `numlib::phi`

---

**Details:**

- The $n$-th cyclotomic polynomial is defined to be the minimal polynomial of any $n$-th primitive root of unity over the field of rational numbers.

---

**Example 1.** We compute the 20th cyclotomic polynomial.

```
>> polylib::cyclotomic(20, z);

                   8     6     4     2
             poly(z  - z  + z  - z  + 1, [z])
```

**Changes:**

- `polylib::cyclotomic` is a new function.

---

## `polylib::decompose` – functional decomposition of a polynomial

`polylib::decompose(p,x)` returns a sequence of polynomials $q_1,\dots,q_n$ such that $p(x) = q_1(\dots q_n(x)\dots)$.

**Call(s):**

- `polylib::decompose(p)`
- `polylib::decompose(p, x)`

**Parameters:**

`p` — polynomial or polynomial expression
`x` — one of the indeterminates of the polynomial `p`

**Return Value:** If a decomposition is possible, `polylib::decompose` returns it as an expression sequence, each element being of the same type as the input. If no decomposition is possible, the input is returned.

**Overloadable by:** `p`

**Related Functions:** `factor`

---

**Details:**

- The second argument may be left out if the polynomial is univariate, as in Example 1.
- If a polynomial has several decompositions, it is not specified which of them is returned.

---

**Example 1.** In the simplest case, an univariate polynomial is decomposed with respect to its only variable:

```
>> polylib::decompose(x^4+x^2+1)
```

```
     2       2
x + x  + 1, x
```

**Example 2.** If there are several variables, a main variable must be specified:

```
>> polylib::decompose(y*x^4+y,y);
```

```
     4
y + x  y
```

**Background:**

- A description of the algorithm behind `polylib::decompose` can be found in *Barton and Zippel*, Polynomial decomposition algorithms, Journal of Symbolic Computation, 1 (1985), pp. 159–168.

**Changes:**

- `polylib::decompose` used to be `decompose`.

---

## `polylib::divisors` – divisors of a polynomial, polynomial expression, or `Factored` element

`polylib::divisors(p)` computes the set of all monic divisors of the polynomial or polynomial expression `p`.

**Call(s):**

- `polylib::divisors(p)`
- `polylib::divisors(f)`
- `polylib::divisors(e)`

**Parameters:**

`p` — a polynomial or polynomial expression
`f` — `Factored` (return value of `factor`)
`e` — element of a domain of category `Cat::Polynomial`

**Return Value:** `polylib::divisors` returns a set of polynomials. The polynomials are from the same type as the polynomials in the argument.

**Related Functions:** `Cat::Polynomial`, `DOM_POLY`, `Dom::Polynomial`, `Dom::MultivariatePolynomial`, `Dom::UnivariatePolynomial`, `factor`, `irreducible`, `numlib::divisors`, `polylib::sqrfree`

---

**Details:**

- `polylib::divisors(f)` returns all monic divisors of a pre-factored polynomial. Cf. example 3.
- `polylib::divisors` works on polynomials of category `Cat::Polynomial` as well. Cf. example 4.

---

**Example 1.** If the argument is a polynomial, a set of polynomials is returned:

```
>> polylib::divisors(poly(x^2 - 2*x + 1))

                                             2
    {poly(1, [x]), poly(x - 1, [x]), poly(x  - 2 x + 1, [x])}
```

**Example 2.** If the argument is a polynomial expression, a set of polynomial expressions is returned:

```
>> polylib::divisors(x^2 - 1)

                                      2
                  {1, x - 1, x + 1, x  - 1}
```

**Example 3.** If the argument is of type `Factored` (a `factor` return value) a set of polynomials is returned:

```
>> p := factor(poly(x^2 - 1));
   polylib::divisors(p)

                poly(x - 1, [x]) poly(x + 1, [x])

 {poly(1, [x]), poly(x - 1, [x]), poly(x + 1, [x]),

          2
    poly(x  - 1, [x])}
```

The polynomials in the resulting set have the same type as the polynomials in the `Factored` element:

```
>> p := factor(x^2 - 1);
   polylib::divisors(p)

                         (x - 1) (x + 1)

                                      2
                    {1, x - 1, x + 1, x  - 1}
```

**Example 4.** `polylib::divisors` works on polynomials from category `Cat::Polynomial` as well:

```
>> P := Dom::Polynomial(Dom::IntegerMod(7)):
   polylib::divisors(P(x^3 + 2*x^2 + 1))

                             3              2
       {1 mod 7, (1 mod 7) x  + (2 mod 7) x  + (1 mod 7)}
```

**Changes:**

- `polylib::divisors` is a new function.

---

## `polylib::discrim` – discriminant of a polynomial

`polylib::discrim(p, x)` returns the discriminant of the polynomial `p` with respect to the variable `x`.

**Call(s):**

- `polylib::discrim(p,x)`

**Parameters:**

`x` — indeterminante
`p` — polynomial or polynomial expression

**Return Value:** `polylib::discrim` returns an element of the coefficient ring of `p`. If the coefficient ring is `Expr` or `IntMod(n)`, an expression is returned.

**Overloadable by:** `p`

**Related Functions:** `polylib::resultant`

---

**Details:**

- The function `normal` is applied to the discriminant before returning it.

---

**Example 1.** We compute the discriminant of the general quadratic equation:

```
>> polylib::discrim(a*x^2 + b*x + c, x);

                          2
                         b  - 4 a c
```

**Background:**

- The discriminant of `p` with respect to the variable `x` is defined as:

  $$(-1)^{d(d-1)/2}\mathrm{res}_x(p, p')/c,$$

  where $d$ is the degree and $c$ is the leading coefficient of $p$.

**Changes:**

- `polylib::discrim` used to be `discrim`.

---

## `polylib::elemSym` – elementary symmetric polynomials

`polylib::elemSym([x1,...,xn], k)` returns the `k`-th elementary symmetric polynomial in the given variables `x1` through `xn`.

**Call(s):**

- `polylib::elemSym(l, k)`

**Parameters:**

`l` — list of indeterminatess
`k` — positive integer

**Return Value:** The result is a polynomial over the coefficient ring `Expr`. If `k` is greater than the number of operands of `l`, `undefined` is returned.

**Related Functions:** `polylib::representByElemSym`

**Details:**

- A given list `l` is a valid first argument only if its elements can be used as indeterminates of a polynomial .

**Example 1.** The first elementary symmetric polynomial is just the sum of its variables:

```
>> polylib::elemSym([x,y,z], 1);

                   poly(x + y + z, [x, y, z])
```

**Example 2.** Indeterminates may also be e.g. trigonometric functions:

```
>> polylib::elemSym([sin(u),cos(u), exp(u)], 2);

 poly(sin(u) cos(u) + sin(u) exp(u) + cos(u) exp(u),

    [sin(u), cos(u), exp(u)])
```

**Background:**

- For more information about elementary symmetric polynomials, see v.d. Waerden, Algebra, vol. 1 .

**Changes:**

- `polylib::elemSym` is a new function.

---

## `polylib::makerat` – convert expression into rational function over a suitable field

`polylib::makerat(a)` returns two polynomials $f$ and $g$ over some extension field of the rationals and a list of substitutions such that applying the substitutions to the rational function $\frac{f}{g}$ gives `a`.

`polylib::makerat(l)` does the same for every element of a list `l` of expressions such that the same extension field is chosen for all elements of the list.

**Call(s):**

- `polylib::makerat(a)`
- `polylib::makerat(l)`

**Parameters:**

`a` — polynomial over `Expr` or arithmetical expression
`l` — list or set of polynomials over `Expr` or arithmetical expressions

**Return Value:** `polylib::makerat` returns an expression sequence consisting of three operands:

- The first operand represents the numerator (or the list/set of numerators, respectively). It is a single polynomial if the call was `polylib::makerat(a)`, otherwise it is a set or list of polynomials (the same type as the input). The polynomial(s) may have more indeterminates than the input. The coefficient ring is either `Expr` or a `Dom::AlgebraicExtension`; if it is `Expr`, the polynomial has only integer coefficients.
- The second operand represents the denominator (or the list/set of denominators, respectively). It is of the same type as the first operand.
- The third operand is a list of equations.

**Related Functions:** `rationalize`

**Example 1.** In the simplest case (integer polynomial), the numerator equals the input, the denominator equals 1, and no substitutions are necessary:

```
>> polylib::makerat(x^2+3)

                     2
            poly(x  + 3, [x]), poly(1, [x]), []
```

**Example 2.** Floating point numbers are considered transcendental:

```
>> polylib::makerat(0.27*x)

       poly(x D1, [x, D1]), poly(1, [x, D1]), [D1 = 0.27]
```

**Example 3.** Radicals are replaced by elements of algebraic extensions:

```
>> polylib::makerat(sqrt(2)/x)

 poly(D2, [x], Dom::AlgebraicExtension(Dom::Rational,

      2
    D2  - 2 = 0, D2)), poly(x, [x],

                                           2
    Dom::AlgebraicExtension(Dom::Rational, D2  - 2 = 0, D2)),

           1/2
    [D2 = 2   ]
```

**Changes:**

- `polylib::makerat` is a new function.

---

## `polylib::minpoly` – approximate minimal polynomial

`polylib::minpoly(a, n, x)` computes a univariate polynomial `f` in the variable `x` of degree `n` with integer coefficients such that `a` equals a root of `f` up to the precision given by `DIGITS`, and such that the sum of squares of its coefficients is minimal among all polynomials with this property.

**Call(s):**

- `polylib::minpoly(a, n, x)`

**Parameters:**

`a` — arithmetical expression that can be converted to a floating point number
`n` — positive integer
`x` — identifier

**Return Value:** `polylib::minpoly` returns a polynomial in `x`. Its coefficient ring is `Expr`, all of its coefficients are integers.

**Side Effects:** `polylib::minpoly` is sensitive to the environment variable `DIGITS`.

**Related Functions:** `lllint`, `stats::linReg`, `numeric::lagrange`

**Example 1.** We compute a polynomial of degree 4 that has a root close to `PI` (up to 6 decimal digits) and small integer coefficients:

```
>> DIGITS:=6: polylib::minpoly(PI, 4, x); delete DIGITS:

                    4       3       2
            poly(7 x  - 16 x  - 16 x  - 6 x - 9, [x])
```

If the root has to be even closer to `PI`, bigger coefficients are needed:

```
>> DIGITS:=20: polylib::minpoly(PI, 4, x); delete DIGITS

                  4         3        2
     poly(- 1951 x  + 6379 x  - 422 x  + 283 x - 4468, [x])
```

**Background:**

- The problem reduces to finding a shortest integer vector in the lattice $\{e_i + a^i \cdot e_{n+1}; 0 \leq i \leq n\}$, where $e_i$ denotes the vector with $e_i[j] = \delta_{i,j}$ (Kronecker symbol). This problem is solved using the algorithm of Lenstra/Lenstra/Lovasz. See Lenstra/Lenstra/Lovasz, Factoring polynomials with rational coefficients, Math. Ann. 261(1982), pp. 515–534.

**Changes:**

- `polylib::minpoly` used to be `numeric::minpoly`.

---

## `polylib::primpart` – primitive part of a polynomial

`polylib::primpart(f)` returns the primitive part of the polynomial `f`.

**Call(s):**

- `polylib::primpart(f)`
- `polylib::primpart(q)`
- `polylib::primpart(xpr` *<inds>*`)`

**Parameters:**

`f` — polynomial
`q` — rational number
`xpr` — expression
`inds` — list of identifiers

**Return Value:** `polylib::primpart` returns an object of the same type as the input, or `FAIL`.

**Related Functions:** `content`, `factor`, `gcd`, `icontent`, `irreducible`

---

**Details:**

- If the input is a `polynomial`, the greatest common divisor of its coefficients is removed. The function `gcd` must be able to calculate this gcd.
- If the first argument is an expression, it is converted into a polynomial in the indeterminates specified by the second argument, or in all of its indeterminates if no second argument is given. `polylib::primpart` returns `FAIL` if the expression cannot be converted into a polynomial.
- For a rational number, its sign is returned.

---

**Example 1.** In the following example, a bivariate polynomial is given. Its coefficients are the integers 3, 6, and 9; the primitive part is obtained by dividing the polynomial by their gcd.

```
>> polylib::primpart(poly(6*x^3*y + 3*x*y + 9*y, [x, y]));

                     3
              poly(2 x  y + x y + 3 y, [x, y])
```

However, consider the same polynomial viewed as a univariate polynomial in `x`. Its coefficients are polynomials in `y` in this case, and their gcd `3*y` is divided off.

```
>> polylib::primpart(poly(6*x^3*y + 3*x*y + 9*y, [x]));

                          3
                   poly(2 x  + x + 3, [x])
```

**Example 2.** `polylib::primpart` divides the coefficients by their gcd, but does not normalize the result. This must be done explicitly:

```
>> polylib::primpart(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x])

                        3                        2
                       x  (9 y + 6)   x (4 y + 6 y )
                       ------------ + --------------
                         3 y + 2        3 y + 2

>> normal(polylib::primpart(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x]))

                                          3
                             2 x y + 3 x
```

**Background:**

- The primitive part of a polynomial $f$ is a polynomial $g$ whose coefficients are relatively prime such that $f = rg$ for some element $r$ of the coefficient ring.

**Changes:**

- `polylib::primpart` used to be `primpart`.

## `polylib::primitiveElement` – primitive element for tower of field extensions

For given field extensions $F = K(\alpha)$ and $G = F(\beta)$, `polylib::primitiveElement(F, G)` returns a list consisting of a simple algebraic extension of $K$ that is $K$-isomorphic to $G$, a symbol for a primitive element of that extension, and the images of $\alpha$ and $\beta$ under some fixed $K$–isomorphism.

**Call(s):**

- `polylib::primitiveElement(F, G)`

**Parameters:**

`F` — a field created by `Dom::AlgebraicExtension`
`G` — a field created by `Dom::AlgebraicExtension` with ground field `F`

**Return Value:** The list returned consists of four operands:

- a field `H` of type `AlgebraicExtension` over the same ground field as `F`;
- an identifier that equals the entry `H::variable`;
- an object of type `H` that satisfies the minimal polynomial for $\alpha$;
- an object of type `H` that satisfies the minimal polynomial for $\beta$.

**Related Functions:** `polylib::splitfield`, `Dom::AlgebraicExtension`

---

**Details:**

- It is presumed that the extension is separable. Otherwise, it may happen that the algorithm does not terminate.

---

**Example 1.** Since the rational numbers are perfect, extensions of them can always be handled:

```
>> F:=Dom::AlgebraicExtension(Dom::Rational, sqrt2^2-2):
   G:=Dom::AlgebraicExtension(F, sqrt3^2-3):
```

Now $G = Q(\sqrt{2}, \sqrt{3})$, and we use `polylib::primitiveElement` to find a primitive element for `G`:

```
>> polylib::primitiveElement(F, G)

 --
 |
 |   Dom::AlgebraicExtension(Dom::Rational,
 --

                                              3            3 -
-
      4        2                     9 X1   X1   11 X1   X1   |
    X1  - 10 X1  + 1 = 0, X1), X1, - ---- + ---, ----- - -
--  |
                                      2      2     2      2  -
-
```

This means that a primitive element $X_1$ of the extension is determined by its minimal polynomial $X_1^4 - 10X_1^2 + 1$. The last two operands of the list are field elements whose squares are 2 and 3, respectively.

**Example 2.** The function works also for subdomains of `AlgebraicExtension`, e.g. Galois fields.

```
>> F:=Dom::GaloisField(7,2):
   G:=Dom::GaloisField(F,2):
   polylib::primitiveElement(F, G)

 [Dom::AlgebraicExtension(Dom::IntegerMod(7),

              2       3     4
    3 X5 - X5  + 2 X5  + X5  - 1 = 0, X5), X5,

                   2       3
    - 3 X5 + 3 X5  - 3 X5  - 2, X5]
```

**Background:**

- The chosen primitive element is $\alpha + s\beta$, where $s$ is a positive integer.

---

## `polylib::randpoly` – creates a random polynomial

`polylib::randpoly()` returns a univariate random polynomial with integer coefficients; the global identifier `x` is used as an unknown.

`polylib::randpoly(list)` returns a random polynomial in all unknowns given in `list`.

`polylib::randpoly(list, ring)` returns a random polynomial in the identifiers given in `list` over the coefficient ring `ring`.

**Call(s):**

- `polylib::randpoly()`
- `polylib::randpoly(list <, `*`Degree`*`= n> <, `*`Terms`*`= k> <,`*`Coeffs`*`= f>)`
- `polylib::randpoly(ring <, `*`Degree`*`= n> <, `*`Terms`*`= k> <,`*`Coeffs`*`= f>)`
- `polylib::randpoly(list, ring <, `*`Degree`*`= n> <, `*`Terms`*`= k> <,`*`Coeffs`*`= f>)`

**Parameters:**

`list` — list of unknowns
`ring` — coefficient ring

**Options:**

*Degree*= `k` — determines the maximum degree the result can have in each variable. `k` must be a non–negative integer. Default is 6.

*Terms*= `t` — makes `polylib::randpoly` generate the sum of `t` random terms. `t` must be a positive integer or `infinity`. If `t` equals `infinity`, `polylib::randpoly` returns a dense polynomial. Default is 5.

*Coeffs*= `f` — Create the coefficients of the result by calling `f()`.

**Return Value:** a polynomial in the given unknowns over the given ring. If no list of indeterminates is given, `[x]` is used. If no ring is given, `Expr` is used.

**Related Functions:** `poly`, `random`

---

**Details:**

- See `poly` for a detailed description of possible unknowns and coefficient rings.
- The polynomial is created by adding the given number (or the default number) of terms, unless the option `Terms=infinity` is given. Each term is created by generating its coefficient and its degree vector at random. Since the same degree vector may be generated several times, the actual number of terms in the result can be smaller than the value of the option `Terms` .
- The random coefficients are generated by calling `f()`, if the option `Coeffs=f` is given. If this option is missing and `ring` is `Expr` the coefficients will be random integers in the range $-999\ldots999$. If `ring` is a user-defined domain, it must have a method `"random"` to create the coefficients if no function is given.

---

**Example 1.** We generate a univariate random polynomial and use the default values for the options. Six random monomials are generated and added. Since it is unlikely that each of the possible exponents 0 to 5 is generated exactly once, the result is very likely to have less than six terms.

```
>> polylib::randpoly([z])

                 5        4        3
       poly(663 z  - 764 z  - 806 z  + 381 z, [z])
```

**Example 2.** We create a bivariate random polynomial over the finite field with 7 elements. This works because `Dom::IntegerMod` has a `"random"` slot that generates random elements:

```
>> polylib::randpoly([x,y],Dom::IntegerMod(7),Degree=3,Terms=4);

            3        2  2
    poly(2 x  y + 6 x  y  + 5 x, [x, y], Dom::IntegerMod(7))
```

**Changes:**

- `polylib::randpoly` used to be `randpoly`.

---

## `polylib::realroots` – **isolate all real roots of a real univariate polynomial**

`polylib::realroots(p)` returns intervals isolating the real roots of the real univariate polynomial `p`.

`polylib::realroots(p, eps)` returns refined intervals approximating the real roots of `p` to the relative precision given by `eps`.

**Call(s):**

- `polylib::realroots(p)`
- `polylib::realroots(p, eps)`

**Parameters:**

`p` — a univariate polynomial: either an expression or a polyomial of domain type `DOM_POLY`.
`eps` — a (small) positive real number determining the size of the returned intervals.

**Return Value:** A list of lists $[[a_1, b_1], [a_2, b_2], \dots]$ with rational numbers $a_i \leq b_i$ is returned. Lists with $a_i = b_i$ represent exact rational roots. Lists with $a_i < b_i$ represent open intervals containing exactly one real root. If the polynomial has no real roots, then the empty list `[ ]` is returned.

**Side Effects:** The function is sensitive to the environment variable `DIGITS`, if there are non-integer or non-rational coefficients in the polynomial. Any such coefficient is replaced by a rational number approximating the coefficient to `DIGITS` significant decimal places.

**Related Functions:** `numeric::fsolve`, `numeric::polyroots`, `numeric::realroot`, `numeric::realroots`

**Details:**

- All coefficients of `p` must be real and numerical, i.e., either integers, rationals or floating point numbers. Numerical symbolic objects such as `sqrt(2)`, `exp(10*PI)` etc. are accepted, if they can be converted to real floating point numbers via `float`. The same holds for the precision goal `eps`.

- The isolating intervals are ordered such that their centers are increasing, i.e., $a_i + b_i < a_{i+1} + b_{i+1}$.

- The number `nops(realroots(p))` of intervals is the number of real roots of `p`. Multiple roots are counted only once. Cf. Example 3.

- Isolating intervals may be quite large. The optional argument `eps` may be used to refine the intervals such that they approximate the real roots to a relative precision `eps`. With this argument the returned intervals satisfy $b_i - a_i \le \texttt{eps} \left|\frac{a_i+b_i}{2}\right|$ , i.e., each center $(a_i + b_i)/2$ approximates a root with a relative precision `eps/2`.

- Some care should be taken when trying to obtain highly accurate approximations of the roots via small values of `eps`. Internally, bisectioning with exact rational arithmetic is used to locate the roots to the precision `eps`. This process may take much more time than determining the isolating intervals without using the second argument `eps` in `polylib::realroots`. It may be faster to use moderate values of `eps` to obtain first approximations of the roots via `polylib::realroots`. These approximations may then be improved by a fast numerical solver such as `numeric::fsolve` with an appropriately high value of `DIGITS`. Cf. Example 6. However, note that `polylib::realroots` will always succeed in locating the roots to the desired precision eventually. Numerical solvers may fail or return a root not belonging to the interval which was used for the initial approximation.

- Unexpected results may be obtained when the polynomial contains irrational coefficients. Internally, any such coefficient $c$ is converted to a floating point number. This float is then replaced by an approximating rational number $r$ satisfying $|r - c| \le 10^{-\texttt{DIGITS}}|c|$. Finally, `polylib::realroots` returns rigorous bounds for the real roots of the rationalized polynomial. Despite the fact that all coefficients are approximated correctly to `DIGITS` decimal places this may change the roots drastically. In particular, multiple roots or clusters of poorly separated simple roots are very sensitive to small perturbations in the coefficients of the polynomial. Cf. Examples 4 and 5.

**Example 1.** We use a polynomial expression as input to `polylib::realroots`:

```
>> p := (x - 1/3)*(x - 1)*(x - 4/3)*(x - 2)*(x - 17):

>> polylib::realroots(p)

           [[0, 1], [1, 1], [1, 2], [2, 2], [16, 32]]
```

The roots 1 and 2 are found exactly: the corresponding intervals have length 0. The other isolating intervals are quite large. We refine the intervals such that they approximate the roots to 12 decimal places. Note that this is independent of the current value of `DIGITS`, because no floating point arithmetic is used:

```
>> polylib::realroots(p, 10^(-12))

 [[1466015503701/4398046511104, 733007751851/2199023255552],

    [1, 1], [1466015503701/1099511627776,

    733007751851/549755813888], [2, 2], [17, 17]]
```

We convert these exact bounds for the real roots to floating point approximations. Note that with the default value of `DIGITS=10` we ignore 2 of the 12 correct digits the rational bounds could potentially give:

```
>> map(%, map, float)

 [[0.3333333333, 0.3333333333], [1.0, 1.0],

    [1.333333333, 1.333333333], [2.0, 2.0], [17.0, 17.0]]

>> delete p:
```

**Example 2.** Orthogonal polynomials of degree $n$ have $n$ simple real roots. We consider the Legendre polynomial of degree 5, available in the library `orthpoly` for orthogonal polynomials:

```
>> polylib::realroots(orthpoly::legendre(5, x), 10^(-DIGITS)):

>> map(%, float@op, 1)

 [-0.9061798459, -0.5384693101, 0.0, 0.5384693101, 0.9061798459]
```

**Example 3.** We consider a polynomial with a multiple root:

```
>> p := poly((x - 1/3)^3*(x - 1), [x])
```

```
                4       3         2
         poly(x  - 2 x  + 4/3 x  - 10/27 x + 1/27, [x])
```

Note that only one isolating interval $[0, 1]$ is returned for the triple root $1/3$:

```
>> polylib::realroots(p)
```

```
                        [[0, 1], [1, 1]]
```

```
>> delete p:
```

**Example 4.** We consider a polynomial with non-rational roots:

```
>> p := (x - 3)^2*(x - PI)^2:
```

Converting the result of `polylib::realroots` to floating point numbers one sees that the exact roots `3, 3, PI, PI` are approximated only to 3 decimal places:

```
>> map(polylib::realroots(p, 10^(-10)), map, float)
```

```
 [[2.998807805, 2.998807805], [3.001213582, 3.001213583],

    [3.140323518, 3.140323519], [3.142840401, 3.142840401]]
```

This is caused by the internal rationalization of the coefficients of `p`. Information on the rationalized polynomial may be optained by a builtin `userinfo` command:

```
>> setuserinfo(polylib::realroots, 1):
```

```
>> polylib::realroots(p, 10^(-10))
```

```
 Info: polynomial rationalized to poly(x^4 - 12283185307/1000..)
 ...
```

The intervals returned by `polylib::realroots(p, 10^(-10))` correctly locate the 4 exact roots of this rationalized polynomial to a precision of 10 digits. However, because all 4 roots are close, the small perturbations of the coefficients introduced by rationalization have a drastic effect on the location of the roots. In particular, rationalization splits the two original double roots into 4 simple roots.

```
>> setuserinfo(polylib::realroots, 0): delete p:
```

**Example 5.** We consider a further example involving non-exact coefficients. First we approximate the roots of a polynomial with exact coefficients:

```
>> p1 := (x - 1/3)^3*(x - 4/3):
```

```
>> map(polylib::realroots(p1, 10^(-10)), map, float)

   [[0.3333333333, 0.3333333333], [1.333333333, 1.333333333]]
```

Now we introduce roundoff errors by replacing one entry by a floating point approximation:

```
>> p2 := (x - 1.0/3)^3*(x - 4/3):
```

```
>> map(polylib::realroots(p2, 10^(-10)),map,float)

   [[0.3332481323, 0.3332481323], [1.333333333, 1.333333333]]
```

In this example rationalization caused the triple root `1/3` to split into one real root and two complex conjugate roots.

```
>> delete p1, p2:
```

**Example 6.** We want to approximate roots to a precision of 1000 digits:

```
>> p := x^5 - 129/20*x^4 + 69/5*x^3 - 14*x^2 + 12*x - 8:
```

We recommend not to obtain the result directly by `polylib::realroots(p,10^(-1000))`, because the internal bisectioning process for refining crude isolating intervals converges only linearly. Instead, we compute first approximations of the roots to a precision of 10 digits:

```
>> approx := map(polylib::realroots(p, 10^(-10)), float@op, 1)

             [1.489177599, 1.752191733, 3.255184556]
```

These values are used as starting points for a numerical root finder. The internal Newton search in `numeric::fsolve` converges quadratically and yields the high precision results much faster than `polylib::realroots`:

```
>> DIGITS := 1000: Roots := []:
>> for x0 in approx do
       Roots := append(Roots, numeric::fsolve([p = 0], [x = x0]));
   end_for

 [[x = 1.4891775988468702813389161146738446438 94...],

     [x = 1.752191733304413195335101727880090131407...],

     [x = 3.255184555797733438479691333705558491124...]]
```

```
>> delete approx, Digits, Roots, x0:
```

**Changes:**

- `polylib::realroots` is a new function.

---

## `polylib::representByElemSym` – represent symmetric by elementary symmetric polynomials

`polylib::representByElemSym(f, [x1,...,xn])` returns a polynomial `g` in the identifiers $x_1$ through $x_n$ such that replacing each `xi` by the `i`-th elementary symmetric polynomial gives `f`.

**Call(s):**

- `polylib::representByElemSym(f, l)`

**Parameters:**

`f` — symmetric polynomial
`l` — list of indeterminates

**Return Value:** The result is a polynomial having the same coefficient ring as `f`.

**Related Functions:** `polylib::elemSym`

---

**Details:**

- The list `l` must have as many operands as `f` has indeterminates.
- The input must be symmetric; this is not checked.

---

**Example 1.** The symmetric polynomial $x^2 + y^2$ can be written as $(x + y)^2 - 2(x \cdot y)$:

```
>> polylib::representByElemSym(poly(x^2+y^2), [u,v]);

                          2
                    poly(u  - 2 v, [u, v])
```

**Example 2.** `polylib::representByElemSym` works over domains also:

```
>> f:=poly(x^2+y^2, Dom::IntegerMod(7)):
   polylib::representByElemSym(f, [u,v])

             2
        poly(u  + 5 v, [u, v], Dom::IntegerMod(7))
```

**Background:**

- It is a well-known theorem that every symmetric polynomial can be written in this way.

**Changes:**

- `polylib::representByElemSym` is a new function.

---

## `polylib::resultant` – **resultant of two polynomials**

`polylib::resultant(f, g)` returns the resultant of `f` and `g` with respect to their first variable.

`polylib::resultant(f, g, x)` returns the resultant of `f` and `g` with respect to the variable `x`.

`polylib::resultant(fexpr, gexpr, inds, x)` returns the resultant of `fexpr` and `gexpr` with respect to the variable `x`; `fexpr` and `gexpr` are viewed as polynomials in the indeterminates `inds`.

**Call(s):**

- `polylib::resultant(f, g <, x>)`
- `polylib::resultant(fexpr, gexpr <, inds><, x>)`

**Parameters:**

| | |
|---|---|
| `f, g` | — polynomials |
| `fexpr, gexpr` | — expressions |
| `x` | — indeterminate |
| `inds` | — list of indeterminates |

**Return Value:** If the input consists of polynomials in at least two variables, `polylib::resultant` returns a polynomial in one variable less than the input.

If the input consists of univariate polynomials, `polylib::resultant` returns an element of the coefficient ring.

If the input consists of expressions, `polylib::resultant` returns an expression.

**Overloadable by:** `p`, q

**Related Functions:** `polylib::discrim`, `linalg::det`, `linalg::sylvester`

---

**Details:**

- Both input polynomials must have exactly the same second and third operand, i.e. their variables and coefficient rings must be identical.
- If the arguments are expressions then these are converted into polynomials using `poly`. `polylib::resultant` returns `FAIL` if the expressions cannot be converted.
- If the argument `inds` is missing, the input expressions are converted into polynomials in all indeterminates occurring in at least one of them. They are *not* independently converted, hence the conversion cannot result in two polynomials with different variables causing an error. See example 1.
- If the coefficient ring is a domain, it must have a `"_divide"` method.
- If the coefficient ring is `Expr`, `polylib::resultant` returns an expression if called with two univariate polynomials. See example 2.
- For polynomials over `IntMod(n)`, the computation may stop with an error if `n` is not prime.

---

**Example 1.** If the input consists of expressions, the sets of indeterminates occurring in the expressions need not coincide:

```
>> polylib::resultant(a*x + c, c*x + d, x);

                                   2
                            a d - c
```

**Example 2.** If the coefficient ring of two univariate input polynomials is `Expr`, the result is an expression:

```
>> polylib::resultant(poly(x^2 -1), poly(x + 1));

                                0
```

**Background:**

- The resultant of two polynomials is defined to be the determinant of their Sylvester matrix. A call to `polylib::resultant` is more efficient than consecutive calls to `linalg::sylvester` and `linalg::det`.

**Changes:**

- `polylib::resultant` used to be `resultant`.

---

## `polylib::sortMonomials` – sorting monomials with respect to a term ordering

`polylib::sortMonomials(f, ord)` returns a list of all monomials constituting the polynomial `f`, sorted in descending order with respect to `ord`.

**Call(s):**

- `polylib::sortMonomials(f)`
- `polylib::sortMonomials(f, vars)`
- `polylib::sortMonomials(f, ord)`
- `polylib::sortMonomials(f, vars, ord)`

**Parameters:**

`f` — polynomial or polynomial expression
`vars` — nonempty list of identifiers
`ord` — monomial ordering

**Return Value:** a list of polynomials or expressions of the same type as `f`.

**Overloadable by:** `f`

**Related Functions:** `Dom::MonomOrdering`, `lmonomial`, `nthmonomial`

---

**Details:**

- A monomial ordering may be: one of the identifiers `LexOrder`, `DegreeOrder`, `DegInvLexOrder`; or an object of type `Dom::MonomOrdering` or convertible to that type; or any object returning a number when called as `ord(m1,m2)` for two degree vectors `m1` and `m2`. A degree vector is a list of integers, as returned by `degreevec`.
- If no order is given, the lexikographical order is used.
- If no list of variables is given, all indeterminates of `f` are used.
- Given two degree vectors, `m1` is considered to be greater than `m2` if and only if `ord(m1,m2)` is positive .

---

**Example 1.** The monomials of the polynomial below are compared using a monomial ordering from `Dom::MonomOrdering` .

```
>> polylib::sortMonomials(poly(x^2+x*y^3+2, [x,y]), DegRevLex(2))

               3                   2
     [poly(x y , [x, y]), poly(x , [x, y]), poly(2, [x, y])]
```

**Changes:**

- `polylib::sortMonomials` is a new function.

---

## `polylib::splitfield` – the splitting field of a polynomial

Given $p \in K[X]$, `polylib::splitfield(p)` returns a simple field extension $F$ of $K$ and some elements $\alpha_1, \ldots, \alpha_n$ of $F$, such that $\prod_{i=1}^{n} X - \alpha_i$ is an associate of $p$, and such that $F$ is the smallest extension of $K$ containing all of the $\alpha_i$.

**Call(s):**

- `polylib::splitfield(p)`

**Parameters:**

`p` — univariate polynomial over a field or univariate polynomial expression

**Return Value:** `polylib::splitfield` returns a list of two operands: the first one is the splitting field of the polynomial, i.e. a `Dom::AlgebraicExtension` of the coefficient ring; the second one is a list of all roots of the polynomial in the splitting field, each root followed by its multiplicity.

**Related Functions:** `factor`, `evalp`

---

**Details:**

- If the input is a polynomial expression, as in Example 1, it is treated as a polynomial over the rationals.
- The polynomial `p` need not be irreducible.
- The name for the primitive element of the field extension is generated using `genident` and is therefore different in every call of `polylib::splitfield`, even if the same polynomial is passed.
- MuPAD must be able to factor polynomials over the coefficient field of `p`.

The coefficient field must be perfect. Otherwise, it may happen that `polylib::splitfield` does not terminate.

---

**Example 1.** We adjoin $\sqrt{-1}$ to the rationals:

```
>> polylib::splitfield(x^2+1)

                                           2
 [Dom::AlgebraicExtension(Dom::Rational, X1  + 1 = 0, X1),

    [X1, 1, -X1, 1]]
```

**Example 2.** A call to `polylib::splitfield` becomes more interesting for polynomials for of degree at least 3:

```
>> polylib::splitfield(x^3-2)

 --
 |                                            6
 |  Dom::AlgebraicExtension(Dom::Rational, X4  + 108 = 0, X4),
 --

    --        4       4              4    -- --
    |  X4   X4      X4        X4   X4      |  |
    |  -- - ---, 1, ---, 1, - -- - ---, 1  |  |
    -- 2    36      18        2    36      -- --
```

**Example 3.** In this example, we work over the field of univariate rational functions (the quotient field of the univariate polynomials) over the rationals:

```
>> R:=Dom::DistributedPolynomial([x], Dom::Rational):
   F:=Dom::Fraction(R):
   f:=poly(y^3-x,[y],F):
   polylib::splitfield(f)

 --
 |
 |  Dom::AlgebraicExtension(Dom::Fraction(
 --

    Dom::DistributedPolynomial([x], Dom::Rational, LexOrder)),

                          --        4        4              4
        2     6           |  X6   X6       X6        X6   X6
    27 x  + X6  = 0, X6), |  -- - ----, 1, ---, 1, - -- - -
---,
                          -- 2    18 x     9 x       2    18 x

      -- --
       |  |
    1  |  |
      -- --
```

**Changes:**

- `polylib::splitfield` is a new function.

---

## `polylib::sqrfree` – square-free factorization of polynomials

`polylib::sqrfree(f)` returns the square-free factorization of `f`, that is, a factorization of $f$ in the form $f = u \cdot p_1^{e_1} \cdot \ldots \cdot p_r^{e_r}$ with primitive and pairwise different square-free divisors $p_i$.

**Call(s):**

- `polylib::sqrfree(f <, recollect>)`

**Parameters:**

`f` — a polynomial or an arithmetical expression
`recollect` — `TRUE` or `FALSE`

**Return Value:** a factored object, i.e., an object of the domain type `Factored`.

**Related Functions:** `content`, `factor`, `Factored`, `irreducible`, `polylib::primpart`

---

**Details:**

- `polylib::sqrfree(f)` returns the square-free factorization of the polynomial $f$, that is, a factorization of $f$ in the form $f = u \cdot f_1^{e_1} \cdot \ldots \cdot f_r^{e_r}$ with primitive and pairwise different square-free divisors $f_i$ (i.e., $\gcd(f_i, f_j) = 1$ for $i \neq j$).

  $u$ is a unit of the coefficient ring of $f$, and $e_i$ are positive integers.

- The result of `polylib::sqrfree` is an object of the domain type `Factored`. Let `g:= polylib::sqrfree(f)` be such an object. It is represented internally as the list `[u, f1, e1, ..., fr, er]` of odd length ($= r+1$).

You may extract the unit $u$, the factors $f_i$ as well as the exponents $e_i$ by the ordinary index operator `[ ]`, i.e., `g[1] = u, g[2] = f1, g[3] = e1, ...`.

For example, to extract all square-free divisors of $f$, enter `g[2*i] $ i = 1..nops(g) div 2`. The same can be achieved with the call `Factored::factors( g )`, and `Factored::exponents( g )` returns a list of the exponents $e_i$ ($1 \leq i \leq r$) of the square-free factorization of $f$.

The call `convert( g,DOM_LIST )` gives the internal representation of a factored object, i.e., the list as described above.

Note that the result of `polylib::sqrfree` is printed as an expression and behaves like that. As an example, the result of `polylib::sqrfree( x^2+2*x+1 )` is the object printed as `(x+1)^2` which is of type `"_power"`.

Please read the help page of `Factored` for details.

- The call `polylib::sqrfree(f, FALSE)` returns a square-free factorization of `f`, where the exponents $e_i$ need not be pairwise different.

- `polylib::sqrfree` can handle univariate and multivariate polynomials over `Expr`, residue class rings `IntMod(p)` with prime modulus `p`, domains representing a unique factorization domain of characteristic zero, and finite fields.

- If the argument of `polylib::sqrfree` is an expression, its numerator and denominator are converted into polynomials in all occurring indeterminates.

  These polynomials are regarded as polynomials over some extension of the rational numbers (i.e., over `Expr`, see `poly`). The choice of that extension follows the same rules as in the case of the function `factor`.

  Factors of the denominator of an expression are indicated by negative multiplicities.

---

**Example 1.** The factors in a squarefree factorization are pairwise relatively prime, but they need not be irreducible:

```
>> polylib::sqrfree(
     2 - 2*x - 6*x^4 + 6*x^5 + 6*x^8 - 6*x^9 -2*x^12 + 2*x^13
   )

                              4        2    3     3
                     2 (x - 1)  (x + x  + x  + 1)
```

**Example 2.** Even if a factorization into irreducibles has been found, irreducible factors with the same multiplicity are collected again:

```
>> polylib::sqrfree( x^6 + x^4*y*6 + x^2*y^2*9 )

                                    2  2
                         (x (3 y + x ))
```

You can avoid this by giving a second argument:

```
>> polylib::sqrfree( x^6 + x^4*y*6 + x^2*y^2*9, FALSE)

                          2         2 2
                         x  (3 y + x )
```

**Example 3.** `polylib::sqrfree` works also for polynomials:

```
>> polylib::sqrfree( poly(2 + 5*x + 4*x^2 + x^3) )

                               2
               poly(x + 1, [x])  poly(x + 2, [x])
```

**Changes:**

- `polylib::sqrfree` used to be `sqrfree`.
- the return value of `polylib::sqrfree` is an object of the domain `Factored`.