Polynomials in SymPy

Ádám Gyenge

Polynomials

- A polynomial is an algebraic expression consisting of terms that are powers of variables with coefficients from a specified domain.
- Example of a univariate polynomial:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

- x is an unknown
- the coefficients $a_0, \ldots, a_n$ come from a ring (e.g $\mathbb{Z}$, $\mathbb{Q}$)

Notation

Let R be a (commutative) ring.

- $R[x]$: ring of polynomials in an indeterminate x
- $R[x_1, \ldots, x_n]$: ring of polynomials in n variables

These are commutative rings with identity 1.
R is identified with the subring of constant polynomials.

Basics

Definition

- An element $r \in R$ is called a *zero divisor* if there exists an element $s \in R$ such that $rs = 0$.
- An element $r \in R$ is called a *unit* if it has a multiplicative inverse, i.e. an element $s \in R$ such that $sr = rs = 1$.
- A ring R is called an *integral domain* if it contains no zero divisors.

Basics

Proposition

Let R be an integral domain

1. $\deg p(x)q(x) = \deg p(x) + \deg q(x)$
2. *units of $R[x]$ are just the units of R*
3. *$R[x]$ is an integral domain*

Proof.

If R has no zero divisors then neither does $R[x]$: if $p(x)$ and $q(x)$ are polynomials with leading terms $a_n x^n$ and $b_m x^m$, respectively, then the leading term of $p(x)q(x)$ is $a_n b_n x^{n+m}$, and $a_n b_n \neq 0$. This proves (3) and also verifies (1). If $p(x)$ is a unit, say $p(x)q(x) = 1$ in $R[x]$, then $\deg p(x) + \deg q(x) = 0$, so both $p(x)$ and $q(x)$ are elements of R, hence are units in R since their product is 1. This proves (2). □

Defining Polynomials

In SymPy, the `Poly` class is used to define polynomials.

```
x = sp.symbols('x')

# Define a polynomial
p = sp.Poly(2*x**3 - 4*x**2 + x - 5)
```

This defines the polynomial $2x^3 - 4x^2 + x - 5$.

Properties of Polynomials

SymPy provides various methods to extract properties of a polynomial:

- `degree()` → Degree of the polynomial
- `LC()` → Leading coefficient
- `coeffs()` → List of coefficients
- `monoms()` → List of monomials

Example: Properties

```
>>> p.degree()
3
>>> p.LC()
2
>>> p.coeffs()
[2, -4, 1, -5]
>>> p.monoms()
[(3,), (2,), (1,), (0,)]
```

Operations on Polynomials

Addition and multiplication are supported for `Poly` objects.
Example:

```
p1 = sp.Poly(x**2 + 2*x + 1)
p2 = sp.Poly(3*x - 4)

sum_p = p1 + p2
product_p = p1 * p2
```

Results:

```
>>> sum_p
Poly(x**2 + 5*x - 3, x, domain='ZZ')
>>> product_p
Poly(3*x**3 + 2*x**2 - 5*x - 4, x, domain='ZZ')
```

Polynomial Domains

Commutative rings with no zero divisors are called *integral domains* (shortly, domains).
Polynomials can be defined over various domains such as:

- Integers ($\mathbb{Z}$)
- Rationals ($\mathbb{Q}$)
- Modular arithmetic ($\mathbb{Z}_n$)

```
# Polynomial over integers
p_int = sp.Poly(2*x**2 + 3*x - 10, domain='ZZ')

# Polynomial over rationals
p_rat = sp.Poly(2*x**2 + 3/4*x - 10, domain='QQ')

# Polynomial over modular arithmetic
p_mod = sp.Poly(2*x**2 + 3*x - 10, modulus=7)
```

Behaviour

Domains affect arithmetic and the behavior of polynomials. For example, in modular arithmetic, coefficients are reduced modulo n.

```
>>> p_int
Poly(2*x**2 + 3*x - 10, x, domain='ZZ')

>>> p_rat
Poly(2*x**2 + 3/4*x - 10, x, domain='QQ')

>>> p_mod
Poly(2*x**2 + 3*x - 3, x, modulus=7)
```

Conversion Between Types

- We have met the general symbolic expressions of SymPy.
- The advantage of the `Poly` type is that it is more suitable for most calculations with polynomials.
- However, it is easy to convert between general symbolic expressions and polynomial objects.
- This conversion if also done automatically in the background by a number of functions.

```
# Define an expression
expr = (x + 1)**2 - x**2 - 2*x - 1

# Convert to polynomial
p = sp.Poly(expr, x)

# Convert back to an expression
expr_from_poly = p.as_expr()
```

Division

Recall division of polynomials.
Given f and g, we are looking for q and r, such that $f = gq + r$ with $\deg(r) < \deg(q)$.
Division with remainder can be done using `div()`:

```
# Division over the rationals
(q_rat,r_rat) = sp.div(5*x**2 + 10*x + 3,2*x + 3)
```

```
# Division over the integers
(q_int,r_int) = sp.div(5*x**2 + 10*x + 3,2*x + 3,
   domain='ZZ')
```

Example: Division

In the second example we see that SymPy can perform polynomial division even in $\mathbb{Z}[x]$. But as this ring is no longer Euclidean the degree of the remainder doesn't need to be smaller than that of f. As 2 doesn't divide 5 in $\mathbb{Z}$, $2x$ doesn't divide $5x^2$ in $\mathbb{Z}[x]$, even if the degree is smaller.

```
>>> q_rat
5*x/2 + 5/4

>>> r_rat
-3/4

>>> q_int
0

>>> r_int
5*x**2 + 10*x + 3
```

GCD and LCM

The `gcd()` and `lcm()` methods compute the greatest common divisor and least common multiple:

```
>>> sp.gcd(x*y**2 + x**2*y, x**2*y**2)
x*y

>>> sp.lcm(x*y**2 + x**2*y, x**2*y**2)
x**3*y**2 + x**2*y**3
```

Classes of commutative rings

- A *Euclidean Domain* is a ring where the division algorithm exists, allowing a form of division with remainder.
- A *Principal Ideal Domain (PID)* is a ring in which every ideal is generated by a single element.
- A *Unique Factorization Domain (UFD)* is a ring where every element can be uniquely factored into irreducible elements, similar to prime factorization in integers.

fields $\subset$ Euclidean domains $\subset$ PIDs $\subset$ UFDs $\subset$ integral domains

Roots

- `solve()` is a general solving function which can find roots. Does not give multiplicities.
- `roots()` computes the symbolic roots of a univariate polynomial; will fail for most high-degree polynomials (five or greater)
- `nroots()` computes numerical approximations of the roots of any polynomial whose coefficients can be numerically evaluated, whether the coefficients are rational or irrational
- `real_roots()` can find all the real roots exactly of a polynomial of arbitrarily large degree; because it finds only the real roots, it can be more efficient than functions that find all roots.
- `all_roots()` can find all the roots exactly of a polynomial of arbitrarily large degree
- `factor()` factors a polynomial into irreducibles and can reveal that roots lie in the coefficient ring

Roots

Example:

```
poly_expr = 5*x**2 + 10*x + 3

# Find roots of a polynomial
roots_poly = sp.roots(poly_expr)
```

The `roots()` function calculates the roots of a polynomial. The result is a dictionary where keys are the roots, and values represent their multiplicities.

```
>>> roots_poly
{-1 - sqrt(10)/5: 1, -1 + sqrt(10)/5: 1}
```

Root Finding for Polynomials

Goal: find the roots of a polynomial

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$$

That is, find x such that

$$p(x) = 0.$$

Analytical formulas exist only for small degrees.

- degree 1–2: simple formulas
- degree 3–4: complicated formulas
- degree ≥ 5: no general formula using radicals

Therefore we typically use **numerical root-finding methods**.

Newton's Method

Let $f(x)$ be a differentiable function.
Starting from an initial guess x_0, we iterate

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Geometric interpretation

- Compute the tangent line of f at x_n
- Take the intersection of the tangent with the x-axis
- This becomes the next approximation x_{n+1}

If the initial guess is sufficiently close to the root, the convergence is typically **quadratic**.
Advantages:

- Very fast convergence near the root
- Easy derivative computation
- Works for high-degree polynomials

Potential issues:

- Bad initial guess $\Rightarrow$ divergence
- Derivative close to zero

Newton's Method in SciPy

SciPy provides numerical root finders.

```
from scipy.optimize import newton

def f(x):
    return x**3 - x - 2

root = newton(f, 1.5)

print(root)
```

Features:

- implements Newton's method
- optionally accepts derivative
- very efficient numerical implementation

Irreducible Polynomials and Factoring

- A polynomial is *irreducible* over a domain if it cannot be factored into lower-degree polynomials with coefficients in that domain.
- Over the reals, linear polynomials and irreducible quadratics $ax^2 + bx + c$ (where $b^2 - 4ac < 0$) are the building blocks.
- Factoring a polynomial involves breaking it into a product of irreducible polynomials.

Example:

$$\begin{aligned} x^4 - 5x^2 + 4 &= (x^2 - 4)(x^2 - 1) \\ &= (x - 2)(x + 2)(x - 1)(x + 1) \end{aligned}$$

Factoring

The function `factor()` splits a polynomial into terms of lower degree.

```
>>> sp.factor(z**2 + 2)
z**2 + 2
```

The factoring highly depends on the domain we work over. In the above example, $z^2 + 2$ cannot be split further as it is *irreducible* over the integers.

Ring extensions

Let A be a ring.

Definition

Let $B \subset A$ be a subset. If B is a ring (with the ring operations from A), then B is a subring of A, and A is a *(ring) extension* of B.

Example

1. $\mathbb{Z} \subset \mathbb{R}$
2. $\mathbb{Z}[i] \subset \mathbb{C}$, where

$$\mathbb{Z}[i] = \{a + bi : a, b \in \mathbb{Z}\}$$

   is the ring of Gaussian integers.
3. In general, given any element $b \in B$ we have the subring

$$\langle A \cup \{b\} \rangle = \{f(b) : f \in A[x]\}$$

Factoring over extension

- We can use the argument `extension` to extend the domain of definition of a polynomial when factoring.
- In the following example we adjoin all roots of the polynomial to obtain a factorisation into linear terms:

```
>>> sp.factor(z**2 + 2,
    extension=sp.roots(z**2 + 2))
(z - sqrt(2)*I)*(z + sqrt(2)*I)
```

- A specific case is when we factor over the Gaussian numbers. Then we can set either `extension=I` or `gaussian=True`.

Factoring over finite fields

For finite fields we can use the argument `modulus`. In the next example we factor the polynomial $z^2 + 2$ over $\mathbb{Z}_3$.

```
>>> sp.factor(z**2 + 2, modulus=3)
(z - 1)*(z + 1)
```

Berlekamp Algorithm (Idea)

Goal: factor a polynomial

$$f(x) \in \mathbb{F}_p[x]$$

Key observation

In a finite field $\mathbb{F}_p$ every element satisfies $a^p = a$.

Therefore we look for polynomials $g(x)$ such that

$$g(x)^p \equiv g(x) \pmod{f(x)}.$$

Berlekamp idea

- Consider polynomials modulo $f(x)$.
- The map

  $$g(x) \mapsto g(x)^p \pmod{f(x)}$$

  acts as a linear transformation.
- Construct the **Berlekamp matrix**.
- Compute the kernel of $Q - I$.
- Elements of this kernel reveal nontrivial factors of $f(x)$.

Berlekamp Algorithm: Small Example

Factor the polynomial over $\mathbb{F}_2$:

$$f(x) = x^4 + x^3 + x + 1$$

Step 1: Work modulo $f(x)$

Compute powers of x^2 (since $p = 2$):

$$x^2, \quad x^4 \equiv x^3 + x + 1 \pmod{f(x)}$$

Step 2: Build the Berlekamp matrix

The coefficients of these polynomials form the **Berlekamp matrix** Q.

We then compute the kernel of $Q - I$ over $\mathbb{F}_2$.

Step 3: Use kernel vectors

Kernel vectors give polynomials $g(x)$ satisfying

$$g(x)^2 \equiv g(x) \pmod{f(x)}.$$

Computing $\gcd(f(x), g(x) - a)$ produces factors.

$$f(x) = (x^2 + x + 1)(x^2 + 1)$$

Multivariate Polynomials

A lot of operations can also be performed on multivariate polynomials, so we can work with them similarly to univariate polynomials
Example:

```
x, y = sp.symbols('x y')
f = 3*x**2*y + 2*x*y**2 - y**3 + 7
g = 6*x*y + 4

(q, r) = sp.div(f, g)
```

Result of Division

```
>>> q
x/2

>>> r
2*x*y**2 - 2*x - y**3 + 7
```

Rational function

Definition

- A *rational function* is an expression of the form:

$$f(x) = \frac{p(x)}{q(x)},$$

  where $p(x)$ and $q(x)$ are polynomials, and $q(x) \neq 0$.
- The *domain* of f is the set of all values of x for which the denominator $q(x)$ is not zero.

Rational functions

When $p(x)$ and $q(x)$ have non-constant polynomial greatest divisor $r(x)$, then setting $p(x) = r(x)p_1(x)$ and $q(x) = r(x)q_1(x)$ produces a rational function

$$f_1(x) = \frac{p_1(x)}{q_1(x)}$$

which may have a larger domain than f, and is equal to f on the domain of f.

Simplifying rational function

```
p = x**2 - 1
q = x - 1

# Simplify rational polynomial
rational_expr = p / q
simplified_expr = rational_expr.simplify()
```

The result is:

```
>>> simplified_expr
x + 1
```

Partial Fraction Decomposition

Given: A rational function $\frac{P(x)}{Q(x)}$, where $P(x)$ and $Q(x)$ are polynomials with $\deg P < \deg Q$.

Step 1: Factor $Q(x)$ into irreducible factors.

Step 2: Write the decomposition based on the factors:

- $(x-a)^k$ leads to terms of the form $\frac{A_1}{x-a} + \frac{A_2}{(x-a)^2} + \cdots + \frac{A_k}{(x-a)^k}$.
- $(x^2+bx+c)^m$ leads to terms of the form $\frac{Bx+C}{x^2+bx+c} + \ldots$.

Example

Decompose $\frac{2x+3}{(x-1)(x+2)}$:

$$\frac{2x+3}{(x-1)(x+2)} = \frac{A}{x-1} + \frac{B}{x+2}$$

Multiply by $(x-1)(x+2)$ and solve for A and B.

Partial Fractions Decomposition

Use the `apart()` method:

```
>>> sp.apart((x**2 + 2*x + 3)/(x**3 + 4*x**2 + 5*x + 2))
3/(x + 2) - 2/(x + 1) + 2/(x + 1)**2
```