

Algebra with SymPy

Ádám Gyenge

Outline

Linear Algebra

Solving Equations

Gröbner Bases

Rings, Ideals, and Modules

Number Theory

Defining Matrices in SymPy

- ▶ SymPy provides the `Matrix` class for defining and manipulating matrices.
- ▶ Entries can be numbers, symbols, expressions, etc.:

```
# Define symbols
```

```
a, b, c, d = sp.symbols('a b c d')
```

```
# Define symbolic matrices
```

```
M = sp.Matrix([[a, b],  
               [c, d]])
```

```
N = sp.Matrix([[0, 1],  
               [1, 0]])
```

- ▶ Results in $M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ and $N = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$.

Matrix Operations in SymPy

- ▶ SymPy supports the usual matrix operations.
- ▶ Examples:

```
# Transpose of a matrix
```

```
M.T
```

```
# Output: Matrix([[a, c], [b, d]])
```

```
# Scalar multiplication
```

```
2 * M
```

```
# Output: Matrix([[2*a, 2*b], [2*c, 2*d]])
```

```
# Element-wise addition
```

```
M + N
```

```
# Output: Matrix([[a, b + 1], [c + 1, d]])
```

Matrix Multiplication and Determinant

- ▶ Multiplication of matrices:

`M * N`

Output: `Matrix([[b, a], [d, c]])`

- ▶ Compute determinant:

`M.det()`

Output: `a*d - b*c`

Matrix Inversion and Eigenvalues

- ▶ Compute inverse if the determinant is non-zero:

```
M.inv()  
# Output: Matrix([  
# [ d/(a*d - b*c), -b/(a*d - b*c)],  
# [-c/(a*d - b*c),  a/(a*d - b*c)])
```

- ▶ Eigenvalues and eigenvectors:

```
M.eigenvals()  
M.eigenvects()
```

Solving Equations with SymPy

- ▶ Use `Eq` for defining equations.
- ▶ Use `solve()` for solving algebraic equations.
- ▶ For example, consider a quadratic equation

```
x = sp.symbols('x')
equation = sp.Eq(x**2 - 5*x + 6, 0)
solution = sp.solve(equation, x)
# Output: [2, 3]
```

Systems of Equations

- We can also solve systems of linear equations:

```
x, y = sp.symbols('x y')  
equations = (sp.Eq(2*x + y, 10), sp.Eq(3*x - y, 5))  
solution = sp.solve(equations, (x, y))  
# Output: {x: 3, y: 4}
```


Nonlinear equations

SymPy can also be used to solve some nonlinear equations.

For example, consider solving the following example:

```
# Define a nonlinear equation:  $x^3 - 6x^2 + 11x - 6 = 0$   
equation = sp.Eq(x**3 - 6*x**2 + 11*x - 6, 0)
```

```
# Solve the nonlinear equation  
solution = sp.solve(equation, x)
```

Here the cubic equation $x^3 - 6x^2 + 11x - 6 = 0$ is solved, and the roots of the equation are $x = 1$, $x = 2$, and $x = 3$.

```
>>> solution  
[1, 2, 3]
```

Trigonometric Equations

Here's an example of solving a simple trigonometric equation:

```
# Define the trigonometric equation:  $\sin(x) = 0$   
equation = sp.Eq(sp.sin(x), 0)
```

```
# Solve the trigonometric equation  
solution = sp.solve(equation, x)
```

In this case, the equation $\sin(x) = 0$ is solved, and the solutions are $x = 0$ and $x = \pi$.

```
>>> solution  
[0, pi]
```

Ideals

Let $(R, +, \cdot)$ be a (commutative) ring with 1.

Definition

A subset $I \subseteq R$ is an *ideal* if it is a subgroup for $+$ and has the property $aI \subseteq I$ for all $a \in R$. That is, $x \in I$ and $a \in R$ implies $ax \in I$.

- ▶ The quotient group R/I inherits a uniquely defined multiplication from R which makes it into a ring.
- ▶ The mapping

$$\phi : R \rightarrow R/I, \quad a \mapsto a + I$$

is a surjective ring homomorphism

- ▶ Fact: There is a one-to-one order preserving correspondence between ideals J of R that contain I , and ideals $\bar{J}$ of R/I given by $J = \phi^{-1}(\bar{J})$.

Generators

- **Generating Set:** An ideal I can be expressed as:

$$I = \langle f_1, f_2, \dots, f_k \rangle = \left\{ \sum_{i=1}^k r_i f_i \mid r_i \in R \right\}.$$

- **Principal Ideal:** If an ideal I has a single generator f , it is called a *principal ideal*:

$$I = \langle f \rangle = \{r \cdot f \mid r \in R\}.$$

Example: In $\mathbb{Z}$, $4\mathbb{Z} = \{4k \mid k \in \mathbb{Z}\}$ is a principal ideal generated by 4.

Gröbner Bases

Let $R = k[x_1, x_2, \dots, x_n]$ be a polynomial ring over a field k with n variables.

Definition (Buchberger, 1960s)

A *Gröbner basis* for an ideal $I \subseteq R$ is a finite generating set $G = \{g_1, g_2, \dots, g_t\} \subseteq I$ such that the leading term of any polynomial in I (with respect to a chosen monomial order) is divisible by the leading term of some g_i in G . Formally, this means that for every $f \in I$, if $\text{LT}(f)$ denotes the leading term of f , then there exists $g_i \in G$ such that $\text{LT}(g_i) \mid \text{LT}(f)$.

Gröbner Bases in Sympy

The `groebner()` method provides functions to compute Gröbner bases and perform related operations. Example:

```
# Define variables
x, y, z = sp.symbols('x y z')

# Define a system of polynomial equations
equations = [x**2 + y**2 - 1, x*y - z**2]

# Compute Groebner basis
gb = sp.groebner(equations)
```

Gröbner Bases in Sympy

The result is an object of type `GroebnerBasis` storing the elements of the basis of the ideal generated by the input polynomials as well as other data used or obtained during the computation:

```
>>> gb
GroebnerBasis([x**2 + y**2 - 1, x*y - z**2,
               x*z**2 + y**3 - y, y**4 - y**2 + z**4],
               x, y, z, domain='ZZ', order='lex')
```

Monomial ordering

Definition

A *monomial order* is a total order on the set of all (monic) monomials in a given polynomial ring, satisfying the property of respecting multiplication: if $u \leq v$ and w is any monomial, then $uw \leq vw$.

Example

- ▶ **Lexicographic Order (lex):** Monomials are ordered lexicographically, e.g., $x^2y > xy^2 > y^3$.
- ▶ **Graded Lexicographic Order (grlex):** Monomials are ordered by total degree, with ties resolved lexicographically.
- ▶ **Graded Reverse Lexicographic Order (grevlex):** Monomials are ordered by total degree, with ties resolved using reverse lexicographic order.

In SymPy, use the argument `order` of `groebner()`.

Monomial orders in SymPy

Lexicographic order

```
>>> sp.groebner(equations,order='lex')
GroebnerBasis([x**2 + y**2 - 1, x*y - z**2,
               x*z**2 + y**3 - y, y**4 - y**2 + z**4],
               x, y, z, domain='ZZ', order='lex')
```

Graded lexicographic order

```
>>> sp.groebner(equations,order='grlex')
GroebnerBasis([y**4 + z**4 - y**2,
               x*z**2 + y**3 - y, x**2 + y**2 - 1, x*y - z**2],
               x, y, z, domain='ZZ', order='grlex')
```

Graded reverse lexicographic order

```
>>> sp.groebner(equations,order='grevlex')
GroebnerBasis([y**3 + x*z**2 - y,
               x**2 + y**2 - 1, x*y - z**2],
               x, y, z, domain='ZZ', order='grevlex')
```

Applications of Gröbner Bases

- ▶ Test if a polynomial belongs to an ideal:

```
f = 2*x**3 + y**3 + 3*y  
result = gb.contains(f)  
# Output: True
```

- ▶ Test if an ideal is zero-dimensional:

```
gb.is_zero_dimensional  
# Output: True
```

Rings

In SymPy, the `Domain` class serves as the base class for constructing various types of rings.

The following types of domains are available:

1. `ZZ` for integers
2. `QQ` for rational
3. `GF(p)` for finite fields of prime order.
4. `RR` for real (floating point) numbers.
5. `CC` for complex (floating point) numbers.
6. `QQ(a)` for algebraic number fields.
7. `K[x]` for polynomial rings.
8. `K(x)` for rational function fields.
9. `EX` for arbitrary expressions.

Example of rings

For example, we can define the polynomials with integer coefficients, denoted $\mathbb{Z}[x]$, and the ring of polynomials with rational coefficients, denoted $\mathbb{Q}[x, y]$ as follows:

```
# Define variables
x, y = symbols('x y')

# Define rings
integer_poly_ring = sp.ZZ.old_poly_ring(x)
rational_poly_ring = sp.QQ.old_poly_ring(x, y)
```

This yields:

```
>>> integer_poly_ring
ZZ[x]
```

```
>>> rational_poly_ring
QQ[x,y]
```

Methods of Domain

- ▶ `convert(element, target_domain)`: Converts an element to another domain.
- ▶ `gcd(a, b)`: Computes the greatest common divisor of two elements.
- ▶ `lcm(a, b)`: Computes the least common multiple of two elements.
- ▶ `is_unit(element)`: Checks if an element is a unit (invertible).
- ▶ `factor(element)`: Factors an element into irreducible components.
- ▶ `add(a, b)`, `mul(a, b)`, `pow(a, n)`: Performs addition, multiplication, and exponentiation in the domain.
- ▶ `zero`, `one`: Returns the additive and multiplicative identities, respectively.

Structural properties

The `Domain` class also provides methods to determine the structural properties of algebraic domains, such as:

- ▶ The `is_Field` property checks if the domain is a field, meaning every non-zero element has a multiplicative inverse.
- ▶ The `is_PID` property verifies whether the domain is a PID, where every ideal is generated by a single element.

```
>>> sp.QQ.is_Field  
True
```

```
>>> sp.QQ.is_PID  
True
```

```
>>> sp.ZZ.is_Field  
False
```

```
>>> sp.ZZ.is_PID  
True
```

Ideals

We can construct an ideal in a ring using the `ideal()` method. As arguments, we need to specifying the generators of the ideal:

```
>>> rational_poly_ring.ideal(y**2-x**3)
<-x**3 + y**2>
```

The above code creates the ideal $\langle y^2 - x^3 \rangle$ in the ring $\mathbb{Q}[x, y]$.

Operations on ideals

► Sum of Ideals:

$$I + J = \{f + g \mid f \in I, g \in J\}.$$

The sum $I + J$ is the smallest ideal containing both I and J .

► Product of Ideals:

$$I \cdot J = \left\{ \sum_{k=1}^m f_k g_k \mid f_k \in I, g_k \in J, m \in \mathbb{N} \right\}.$$

The product $I \cdot J$ consists of all finite sums of products of elements from I and J .

► Intersection of Ideals:

$$I \cap J = \{f \in R \mid f \in I \text{ and } f \in J\}.$$

The intersection $I \cap J$ is the set of all elements common to both I and J .

Operations on ideals

SymPy also allows computation of common operations involving ideals:

```
>>> I_1=rational_poly_ring.ideal(y**2)
```

```
>>> I_2=rational_poly_ring.ideal(x**3)
```

```
# Sum of ideals
```

```
>>> I_1+I_2
```

```
<y**2,x**3>
```

```
# Product of ideals
```

```
>>> I_1*I_2
```

```
<x**3*y**2>
```

```
# Intersection of ideals
```

```
>>> I_1.intersect(I_2)
```

```
<x**3*y**2>
```

Quotient w.r.t an ideal

We can also form the quotient of a ring modulo an ideal. For this, we use the `quotient()` method or, simply, the sign `/`:

```
>>> rational_poly_ring/[x**2]  
QQ[x,y]/<x**2>
```

Modules

Let R be a ring (commutative, with unity 1_R).

Definition

An R -module is a set M equipped with two operations:

- ▶ Addition: $+: M \times M \rightarrow M$, such that M is an abelian group under $+$.
- ▶ Scalar multiplication: $\cdot: R \times M \rightarrow M$, satisfying:
 - ▶ $r \cdot (x + y) = r \cdot x + r \cdot y$ for all $r \in R, x, y \in M$.
 - ▶ $(r + s) \cdot x = r \cdot x + s \cdot x$ for all $r, s \in R, x \in M$.
 - ▶ $(r \cdot s) \cdot x = r \cdot (s \cdot x)$ for all $r, s \in R, x \in M$.
 - ▶ $1_R \cdot x = x$ for all $x \in M$.

Intuition: Modules generalize the concept of vector spaces, but over arbitrary rings instead of fields. **Examples:**

- ▶ $\mathbb{Z}^n$ is a module over $\mathbb{Z}$.
- ▶ $R[x]$, the ring of polynomials over R , is a module over R .
- ▶ Vector spaces are modules over fields.

Free modules

An module over a ring R is *free* if it is isomorphic to R^n for some n . In SymPy, we can work with free modules and modules constructed from them.

```
# Defining a free module
```

```
free_mod = rational_poly_ring.free_module(4)
```

```
# Defining a submodule
```

```
sub_mod = free_mod.submodule([1,x,y**2,x*y],[0,1,0,x**2])
```

The result is:

```
>>> free_mod
```

```
QQ[x,y]**4
```

```
>>> sub_mod
```

```
<[1, x, y**2, x*y], [0, 1, 0, x**2]>
```

Integers and prime numbers

SymPy offers several utilities for analyzing and generating prime numbers, as well as factorizing integers into their prime components.

To check the primality of a number, factorize it, or generate primes within a range, we use the following:

```
# Check if a number is prime
```

```
>>> sp.isprime(17)
```

```
True
```

```
# Factorize an integer
```

```
>>> sp.factorint(60)
```

```
{2: 2, 3: 1, 5: 1}
```

```
# Generate prime numbers within a range
```

```
list(sp.primerange(10, 30))
```

```
[11, 13, 17, 19, 23, 29]
```

Divisors and modular arithmetic

```
# Find all divisors of an integer
>>> sp.divisors(28)
[1, 2, 4, 7, 14, 28]

# Compute modular inverse
>>> sp.mod_inverse(3, 7)
5
```

Here, `divisors` lists all divisors of a number, while `mod_inverse` computes the modular inverse when it exists. Modular inverses are particularly important in cryptography and solving congruences. The last result follows from $3 * 5 \equiv 1 \pmod{7}$.

Diophantine equations

Diophantine equations seek integer solutions to polynomial equations. SymPy's diophantine function can solve a variety of such equations, including linear, quadratic, and Pell's equations.

```
# Define variables
x, y, z = sp.symbols('x y z')

# Solve a Diophantine equation
solutions = sp.diophantine(x**2 + y**2 - z**2)
# Solutions include integer triples representing
    Pythagorean triples
```

Result:

```
>>> solutions
{(2*p*q, p**2 - q**2, p**2 + q**2)}
```

Chinese Remainder Theorem

Theorem: Let $n_1, n_2, \dots, n_k$ be pairwise coprime positive integers, and let $N = n_1 n_2 \cdots n_k$. Then for any integers $a_1, a_2, \dots, a_k$, the system of congruences

$$x \equiv a_1 \pmod{n_1}$$

$$x \equiv a_2 \pmod{n_2}$$

$$\vdots$$

$$x \equiv a_k \pmod{n_k}$$

has a unique solution modulo N .

Example: Solve $x \equiv 2 \pmod{3}$, $x \equiv 1 \pmod{5}$, $x \equiv 3 \pmod{7}$.

Chinese Remainder Theorem

In SymPy, this is implemented through the `crt` function.

```
# Solve a system of modular congruences
moduli = [3, 5, 7]
remainders = [2, 1, 3]
solution = sp.crt(moduli, remainders)  # (101, 105)
```

Here, the solution (101, 105) implies that $x \equiv 101 \pmod{105}$ satisfies the given congruences. The `crt` function handles both the solution and the modulus.