Computer Algebra with SymPy

Ádám Gyenge

Example of Floating-Point Inaccuracy

Floating-point arithmetic can lead to small but significant errors in calculations.

```
>>> (1 / 49) * 49
0.9999999999999999
```

This example shows how numerical inaccuracies arise from machine precision limitations.

Why SymPy?

- Machine precision floating-point numbers may not be exact.
- Accumulated small errors can lead to significant inaccuracies.
- This may lead to big economical costs (e.g. Ariane 5, US$362 million)
- Some computations require exact results.
- Symbolic computation allows for analytic solutions and deeper understanding of mathematical relationships.

What is a Computer Algebra System?

- Symbolic manipulation of mathematical expressions instead of numerical computation.
- Enables exact solutions, simplifications, and symbolic calculus.
- Example: SymPy is a Python library for symbolic mathematics.
- SymPy is lightweight and requires only Python, making it accessible and versatile.

History of Computer Algebra Systems

Evolution of Computer Algebra Systems (CAS):

- 1960s: Development of first CAS like MACSYMA at MIT.
- 1970s: Emergence of REDUCE, a system for symbolic algebra.
- 1980s: Commercial systems such as Mathematica and Maple introduced.
- 1990s: Open-source systems like Maxima and Axiom gained popularity.
- 2000s: Python-based libraries like SymPy introduced for integration with modern programming.

Importance: CAS have revolutionized mathematics by enabling precise symbolic computations.

Installing and importing SymPy

1. Install SymPy using pip to get started:

   ```
   pip install sympy
   ```

2. Import SymPy into your Python script:

   ```
   import sympy as sp
   ```

Defining Symbols

Define symbolic variables to use in mathematical expressions:

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

Symbols are the foundation for creating and manipulating expressions in SymPy.
The `symbols()` function takes a string as an argument, where each symbol is separated by a space.

Creating Expressions

Use symbols to create mathematical expressions:

```
expr = 2*x + 3*y
print(expr)
```

Output: $2x + 3y$

Pretty Printing

- SymPy can format mathematical expressions in a way that resembles traditional mathematical notation.
- For this, we call the `init_printing()` method to enable *pretty printing*.
- This method automatically selects the best printing format depending on the environment, such as LaTeX for Jupyter notebooks or Unicode for standard terminals.
- Formatting enhances readability and improves the presentation of complex expressions.

Pretty Printing: example

Consider a basic power expression:

```
expr = (x + y)**3
```

We can compare the usual and the pretty printing of this expression:

```
>>> expr
(x + y)**3
>>> sp.init_printing()
>>> expr
       3
(x + y)
```

Latex printing

It is also possible to convert an expression into LaTeX with the `latex()` function:

```
>>> sp.latex(expr)
\left(x + y\right)^{3}
```

Simplification

Simplify complex expressions to their reduced forms:

```
expr = (x**2 + 2*x + 1)/(x + 1)
simplified_expr = sp.simplify(expr)
```

Output: $x + 1$

Expanding Expressions

Expand factored expressions:

```
expr = (x + 1)*(x + 2)
expanded_expr = sp.expand(expr)
```

Output: $x^2 + 3x + 2$

Factoring Expressions

Factor algebraic expressions:

```
expr = x**2 + 3*x + 2
factored_expr = sp.factor(expr)
```

Output: $(x+1)(x+2)$

Substitution

Replace symbols with specific values:

```
expr = 2*x + 3*y
result = expr.subs({x: 1, y: 2})
```

Output: 8

- `subs()` is a member function of the expression itself
- To perform substitution on an expression, pass a list or dictionary of `(old, new)` pairs.

Symbolic Substitution

```
# Define an expression
expr = 2*x + 3*y

# Substitute x with (z + 1)
new_expr = expr.subs(x, z + 1)
```

This replaces x with $(z + 1)$ in the expression.

- Result:

$$2(z + 1) + 3y = 2z + 2 + 3y$$

- Code output:

```
>>> new_expr
3*y + 2*z + 2
```

Effect of substitution

It is important to note about `subs()` that it always returns a new expression. The reason for this is that SymPy objects are immutable. That means that `subs` does not modify it in-place.

```
>>> expr = sin(x)
>>> expr.subs(x, 0)
0
>>> expr
sin(x)
```

We see that performing `expr.subs(x, 0)` leaves `expr` unchanged.

Floating Point Evaluation

Evaluate expressions with numerical approximations:

```
expr = sp.sqrt(2)
expr.evalf()
expr.evalf(30)
```

Output: 1.4142135623731 and 1.41421356237309504880168872421

Detour: OOP in Python

Classes define the blueprint for creating objects. They encapsulate data (attributes) and behavior (methods).

- `__init__()`: Constructor method to initialize attributes.
- **Methods**: Functions defined inside a class that operate on its attributes.
- **Instance Attributes**: Variables unique to each instance.
- **Class Attributes**: Shared across all instances.

Example:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def display(self):
        return f"{self.brand} {self.model}"

car1 = Car("Toyota", "Corolla")
print(car1.display())  # Toyota Corolla
```

Subclasses and inheritance

Inheritance allows a new class to derive properties and methods from an existing class.

- **Base Class (Parent)**: The class being inherited from.
- **Derived Class (Child or Subclass)**: The class inheriting the properties.
- `super()`: Calls methods from the parent class.

Example:

```
class ElectricCar(Car):
    def __init__(self, brand, model, battery_size):
        super().__init__(brand, model)
        self.battery_size = battery_size

    def battery_info(self):
        return f"{self.brand}, {self.model},
        {self.battery_size} kWh"

ev = ElectricCar("Tesla", "Model 3", 75)
print(ev.battery_info())  # Tesla, Model 3, 75 kWh
```

Internal Representation of Expressions

- SymPy's symbolic expression system defines expressions in a symbolic tree representation.
- We can see what an expression looks like internally by using the function `srepr()`.

```
expr = x**3 + 3*x + 2
print(sp.srepr(expr))
```

Output: `Add(Pow(Symbol('x'), Integer(3)), Mul(Integer(3), Symbol('x')), Integer(2))`

Tree Representation

We can also visualize the structure of an expression as a tree using the `print_tree()` function:

```
sp.print_tree(expr, assumptions=False)
```

Output:

```
Add: x**3 + 3*x + 2
+-Integer: 2
+-Pow: x**3
| +-Symbol: x
| +-Integer: 3
+-Mul: 3*x
  +-Integer: 3
  +-Symbol: x
```

The Expr class

- Add, Pow and Mul are subclasses of the class Expr
- simplify(), expand(), factor(), subs(old, new) and evalf() are methods of the class Expr
- Some other useful methods:
  - as_coefficients_dict(): Returns coefficients of terms as a dictionary.
  - free_symbols: Returns the set of variables in the expression.

Sympy objects

- All symbols are instances of the class `Symbol`.
- For the number in the expression, 2, we got `Integer(2)`.
- `Integer` is the SymPy class for integers. It is similar to the Python built-in type int, except that `Integer` is more compatible with other SymPy types.

Sympify

The `sympify()` function converts Python objects, such as strings, numbers, or lists, into SymPy expressions:

```
# Convert string to SymPy expression
expr = sp.sympify("x**2 + 2*x + 1")
```

This results in

```
>>> expr
x**2 + 2*x + 1
```

Nested objects

The `sympify()` method also supports nested data structures like lists or tuples:

```
nested = sp.sympify(["x**2", "2*x + 1"])
```

yields

```
>>> nested
[x**2, 2*x + 1]
```

Lambda Functions in Python

- Anonymous functions defined using the `lambda` keyword.
- Used for creating small, single-expression functions without a formal `def` block.

General Form:

```
lambda arguments:  expression
```

- **Arguments**: Input parameters.
- **Expression**: Single output expression (computed and returned).

Example

Simple Lambda Function:

```
# Adds 10 to a number
add_ten = lambda x: x + 10
print(add_ten(5))  # Output: 15
```

Use Cases

- **Single-use functions** (e.g., inside `map()`, `filter()`, `sorted()`).
- Simplifies code for small, concise tasks.

```
numbers = [1, 2, 3, 4]
squared = map(lambda x: x**2, numbers)
print(list(squared))  # Output: [1, 4, 9, 16]
```

Lambdify

- The `lambdify()` function translates SymPy expressions into numerical functions e.g. for fast evaluation.
- It bridges symbolic computation with numerical libraries like `math`, `NumPy`, or `SciPy`,
- Enables symbolic expressions to be evaluated efficiently over arrays or numerical inputs.

```
# Create a numerical function
f = sp.lambdify(x**2 + 2*x + 1, expr)
```

Here, `lambdify` converts the expression $x^2 + 2x + 1$ into a Python function `f` that can be evaluated at any value:

```
>>> f(3)
16
```

Using NumPy with Lambdify

The `modules` argument in `lambdify` allows specifying the numerical library used for evaluation:

```
import numpy as np

# Create a numerical function using NumPy
f_np = sp.lambdify(x, expr, modules="numpy")
```

Then we can evaluate the obtained numerical function on NumPy arrays:

```
>>> f_np(np.array([0, 1, 2, 3]))
array([ 1,  4,  9, 16])
```

Algebraic Functions

Common algebraic functions include:

- Square Root: `sp.sqrt(x)`
- Logarithms: `sp.log(x)`, `sp.log10(x)`
- Powers: `x**3` or `sp.pow(x, 3)`

Trigonometric Functions

SymPy supports trigonometric functions:

- Sine: `sp.sin(x)`
- Cosine: `sp.cos(x)`
- Tangent: `sp.tan(x)`

Hyperbolic Functions

Hyperbolic functions and their inverses:

- Sinh: `sp.sinh(x)`
- Cosh: `sp.cosh(x)`
- Inverse Sinh: `sp.asinh(x)`

Further Resources

Learn more about SymPy:

- Official Documentation: `https://docs.sympy.org/`
- Community Forums and Tutorials.