

# SYMPY: SYMBOLIC COMPUTING IN PYTHON

ONDŘEJ ČERTÍK\*, ISURU FERNANDO†, AND ASHUTOSH SABOO‡

## 1. Introduction.

## 2. Architecture.

**2.1. Basic Usage.** Being built on Python, SymPy requires that all variable names be defined before they can be used. The statement

```
>>> from sympy import *
```

will import all SymPy functions into the global Python namespace. All the examples in this paper assume that this has been run.

Additionally, symbolic variables, called symbols, must be assigned to Python variables before they can be used. This is typically done through the `symbols` function, which creates multiple symbols at once. For instance,

```
>>> x, y, z = symbols('x y z')
```

creates three symbols named `x`, `y`, and `z`, assigned to Python variables of the same name. The Python variable names that symbols are assigned to are immaterial—we could have just as well have written `a, b, c = symbols('x y z')`. All the examples in this paper will assume that the symbols `x`, `y`, and `z` have been assigned as above.

Expressions are created from symbols using Python syntax, which mirrors usual mathematical notation. Note that in Python, exponentiation is `**`.

```
>>> (x**2 - 2*x + 3)/y
```

```
(x**2 - 2*x + 3)/y
```

**2.2. The Core.** The core of a computer algebra system (CAS) refers to the module that is in charge of resending symbolic expressions and performing basic manipulations with them. In SymPy, every symbolic expression is an instance of a Python class. Expressions are represented by expression trees. The operators are represented by the type of an expression and the child nodes are stored in the `args` attribute. A leaf node in the expression tree has an empty `args`. The `args` attribute is provided by the class `Basic`, which is a superclass of all SymPy objects and provides common methods to all SymPy tree-elements. For example, consider the expression  $xy + 2$ :

```
>>> from sympy import *
```

```
>>> x, y = symbols('x y')
```

```
>>> expr = x*y + 2
```

The expression `expr` is an addition, so it is of type `Add`. The child nodes of `expr` are `x*y` and `2`.

```
>>> type(expr)
```

```
<class 'sympy.core.add.Add'>
```

```
>>> expr.args
```

```
(2, x*y)
```

We can dig further into the expression tree to see the full expression. For example, the first child node, given by `expr.args[0]` is `2`. Its class is `Integer`, and it has empty `args`, indicating that it is a leaf node.

---

\*Los Alamos National Laboratory ([ondrej.certik@gmail.com](mailto:ondrej.certik@gmail.com)).

†University of Moratuwa ([isuru.11@cse.mrt.ac.lk](mailto:isuru.11@cse.mrt.ac.lk)).

‡Birla Institute of Technology and Science, Pilani, K.K. Birla Goa Campus ([ashutosh.saboo@gmail.com](mailto:ashutosh.saboo@gmail.com)).

```

42 >>> expr.args[0]
43 2
44 >>> type(expr.args[0])
45 <class 'sympy.core.numbers.Integer'>
46 >>> expr.args[0].args
47 ()

```

The function `srepr` gives a string representing a valid Python code, containing all the nested class constructor calls to create the given expression.

```

50 >>> srepr(expr)
51 "Add(Mul(Symbol('x'), Symbol('y')), Integer(2))"

```

Every SymPy expression satisfies a key invariant, namely, `expr.func(*expr.args) == expr`.<sup>1</sup> This means that expressions are rebuildable from their `args`. Here, we note that in SymPy, the `==` operator represents exact structural equality, not mathematical equality. This allows one to test if any two expressions are equal to one another as expression trees.

Python allows classes to overload operators. The Python interpreter translates the above `x*y + 2` to, roughly, `(x.__mul__(y)).__add__(2)`. `x` and `y`, returned from the `symbols` function, are `Symbol` instances. The `2` in the expression is processed by Python as a literal, and is stored as Python's builtin `int` type. When `2` is called by the `__add__` method, it is converted to the SymPy type `Integer(2)`. In this way, SymPy expressions can be built in the natural way using Python operators and numeric literals.

One must be careful in one particular instance. Python does not have a builtin rational literal type. Given a fraction of integers such as `1/2`, Python will perform floating point division and produce `0.5`.<sup>2</sup> Python uses eager evaluation, so expressions like `x + 1/2` will produce `x + 0.5`, and by the time any SymPy function sees the `1/2` it has already been converted to `0.5` by Python. However, for a CAS like SymPy, one typically wants to work with exact rational numbers whenever possible. Working around this is simple, however: one can wrap one of the integers with `Integer`, like `x + Integer(1)/2`, or using `x + Rational(1, 2)`. SymPy provides a function `S` which can be used to convert objects to SymPy types with minimal typing, such as `x + S(1)/2`. This gotcha is a small downside to using Python directly instead of a custom domain specific language (DSL), and we consider it to be worth it for the advantages listed above.

**2.3. Assumptions.** An important feature of the SymPy core is the assumptions system. The assumptions system allows users to specify that symbols have certain common mathematical properties, such as being positive, imaginary, or integer. SymPy is careful to never perform simplifications on an expression unless the assumptions allow them. For instance, the identity  $\sqrt{x^2} = x$  holds if  $x$  is nonnegative ( $x \geq 0$ ). If  $x$  is real, the identity  $\sqrt{x^2} = |x|$  holds. However, for general complex  $x$ , no such identity holds.

By default, SymPy performs all calculations assuming that variables are complex valued. This assumption makes it easier to treat mathematical problems in full generality.

```

86 >>> x = Symbol('x')

```

<sup>1</sup>`expr.func` is used instead of `type(expr)` to allow the function of an expression to be distinct from its actual Python class. In most cases the two are the same.

<sup>2</sup>This is the behavior in Python 3. In Python 2, `1/2` will perform integer division and produce `0`, unless one uses `from __future__ import division`.

```

87 >>> sqrt(x**2)
88 sqrt(x**2)

```

89 By assuming symbols are complex by default, SymPy avoids performing mathematically invalid operations. However, in many cases users will wish to simplify expressions containing terms like  $\sqrt{x^2}$ .

92 Assumptions are set on `Symbol` objects when they are created. For instance `Symbol('x', positive=True)` will create a symbol named `x` that is assumed to be positive.

```

93 >>> x = Symbol('x', positive=True)
94 >>> sqrt(x**2)
95 x

```

98 Some common assumptions that SymPy allows are `positive`, `negative`, `real`, `nonpositive`, `nonnegative`, `real`, `integer`, and `commutative`<sup>3</sup>. Assumptions on any object can be checked with the `is_assumption` attributes, like `x.is_positive`.

101 Assumptions are only needed to restrict a domain so that certain simplifications can be performed. It is not required to make the domain match the input of a function. For instance, one can create the object  $\sum_{n=0}^m f(n)$  as `Sum(f(n), (n, 0, m))` without setting `integer=True` when creating the `Symbol` object `n`.

105 The assumptions system additionally has deductive capabilities. The assumptions use a three-valued logic using the Python builtin objects `True`, `False`, and `None`. `None` represents the “unknown” case. This could mean that the given assumption could be either true or false under the given information, for instance, `Symbol('x', real=True).is_positive` will give `None` because a real symbol might be positive or it might not. It could also mean not enough is implemented to compute the given fact, for instance, `(pi + E).is_irrational` gives `None`, because SymPy does not know how to determine if  $\pi + e$  is rational or irrational, indeed, it is an open problem in mathematics.

114 Basic implications between the facts are used to deduce assumptions. For instance, the assumptions system knows that being an integer implies being rational, so `Symbol('x', integer=True).is_rational` returns `True`. Furthermore, expressions compute the assumptions on themselves based on the assumptions of their arguments. For instance, if `x` and `y` are both created with `positive=True`, then `(x + y).is_positive` will be `True`.

120 SymPy also has an experimental assumptions system where facts are stored separate from objects, and deductions are made with a SAT solver. We will not discuss this system here.

123 **2.4. Extensibility.** Extensibility is an important feature for SymPy. Because the same language, Python, is used both for the internal implementation and the external usage by users, all the extensibility capabilities available to users are also used by functions that are part of SymPy.

127 The typical way to create a custom SymPy object is to subclass an existing SymPy class, generally either `Basic`, `Expr`, or `Function`. All SymPy classes used for expression trees<sup>4</sup> should be subclasses of the base class `Basic`, which defines some basic methods for symbolic expression trees. `Expr` is the subclass for mathematical expressions that can be added and multiplied together. Instances of `Expr` typically

<sup>3</sup>If  $A$  and  $B$  are Symbols created with `commutative=False` then SymPy will keep  $A \cdot B$  and  $B \cdot A$  distinct.

<sup>4</sup>Some internal classes, such as those used in the polynomial module, do not follow this rule for efficiency reasons.

represent complex numbers, but may also include other “rings” like matrix expressions. Not all SymPy classes are subclasses of `Expr`. For instance, logic expressions, such as `And(x, y)` are subclasses of `Basic` but not of `Expr`.

The `Function` class is a subclass of `Expr` which makes it easier to define mathematical functions called with arguments. This includes named functions like `sin(x)` and `log(x)` as well as undefined functions like `f(x)`. Subclasses of `Function` should define a class method `eval`, which returns values for which the function should be automatically evaluated, and `None` for arguments that shouldn’t be automatically evaluated.

The behavior of classes in SymPy with various other SymPy functions is defined by defining a relevant `_eval_*` method on the class. For instance, an object can tell SymPy’s `diff` function how to take the derivative of itself by defining the `_eval_derivative(self, x)` method. The most common `_eval_*` methods relate to the assumptions. `_eval_is_assumption` defines the assumptions for *assumption*.

As an example of the notions presented in this section, we present below a stripped down version of the gamma function  $\Gamma(x)$  from SymPy, which evaluates itself on positive integer arguments, has the positive and real assumptions defined, can be rewritten in terms of factorial with `gamma(x).rewrite(factorial)`, and can be differentiated. `fdiff` is a convenience method for subclasses of `Function`. `fdiff` returns the derivative of the function without worrying about the chain rule. `self.func` is used throughout instead of referencing `gamma` explicitly so that potential subclasses of `gamma` can reuse the methods.

```
from sympy import Integer, Function, floor, factorial, polygamma

class gamma(Function)
    @classmethod
    def eval(cls, arg):
        if isinstance(arg, Integer) and arg.is_positive:
            return factorial(arg - 1)

    def _eval_is_real(self):
        x = self.args[0]
        # noninteger means real and not integer
        if x.is_positive or x.is_noninteger:
            return True

    def _eval_is_positive(self):
        x = self.args[0]
        if x.is_positive:
            return True
        elif x.is_noninteger:
            return floor(x).is_even

    def _eval_rewrite_as_factorial(self, z):
        return factorial(z - 1)

    def fdiff(self, argindex=1):
        from sympy.core.function import ArgumentIndexError
        if argindex == 1:
            return self.func(self.args[0])*polygamma(0, self.args[0])
```





277 converge numerically).

278 Due to this generic approach, particular combinations of hypergeometric functions  
 279 can be specified easily. The implementation of the Meijer G-function takes only a few  
 280 dozen lines of code, yet covers the whole input domain in a robust way. The Meijer  
 281 G-function instance  $G_{1,3}^{3,0}(0; \frac{1}{2}, -1, -\frac{3}{2}|x)$  is a good test case [31]; past versions of  
 282 both Maple and Mathematica produced incorrect numerical values for large  $x > 0$ .  
 283 Here, mpmath automatically removes the internal singularity and compensates for  
 284 cancellations (amounting to 656 bits of precision when  $x = 10000$ ), giving correct  
 285 values:

```
286 >>> mpmath.mp.dps = 15
287 >>> mpmath.meijerg([], [0], [[-0.5, -1, -1.5], []], 10000)
288 mpf('2.4392576907199564e-94')
```

289 Equivalently, with SymPy's interface this function can be evaluated as:

```
290 >>> meijerg([], [0], [[-S(1)/2, -1, -S(3)/2], []], 10000).evalf()
291 2.43925769071996e-94
```

292 We highlight the generalized hypergeometric functions and the Meijer G-function,  
 293 due to those functions' frequent appearance in closed forms for integrals and sums  
 294 [todo: crossref symbolic integration]. Via mpmath, SymPy has relatively good sup-  
 295 port for evaluating sums and integrals numerically, using two complementary ap-  
 296 proaches: direct numerical evaluation, or first computing a symbolic closed form  
 297 involving special functions. [example?]

298 **3.2. Numerical simplification.** The `nsimplify` function in SymPy (a wrapper  
 299 of `identify` in mpmath) attempts to find a simple symbolic expression that evaluates  
 300 to the same numerical value as the given input. It works by applying a few simple  
 301 transformations (including square roots, reciprocals, logarithms and exponentials) to  
 302 the input and, for each transformed value, using the PSLQ algorithm [16] to search  
 303 for a matching algebraic number or optionally a linear combination of user-provided  
 304 base constants (such as  $\pi$ ).

```
305 >>> x = 1 / (sin(pi/5)+sin(2*pi/5)+sin(3*pi/5)+sin(4*pi/5))*2
306 >>> nsimplify(x)
307 -2*sqrt(5)/5 + 1
308 >>> nsimplify(pi, tolerance=0.01)
309 22/7
310 >>> nsimplify(1.783919626661888, [pi], tolerance=1e-12)
311 pi/(-1/3 + 2*pi/3)
```

312 **4. Features.** SymPy has an extensive feature set that encompasses too much to  
 313 cover in-depth here. Bedrock areas, such a Calculus, receive their own sub-sections  
 314 below. Additionally, Table 1 describes other capabilities present in the SymPy code  
 315 base. This gives a sampling from the breadth of topics and application domains that  
 316 SymPy services.

Table 1: SymPy Features and Descriptions

Feature	Description
Discrete Math	Summations, products, binomial coefficients, prime number tools, integer factorization, Diophantine equation solving, and boolean logic representation, equivalence testing, and inference.

Concrete Math	Tools for determining whether summation and product expressions are convergent, absolutely convergent, hypergeometric, and other properties. May also compute Gosper's normal form [28] for two univariate polynomials.
Plotting	Hooks for visualizing expressions via matplotlib [?] or as text drawings when lacking a graphical back-end.
Geometry	Allows the creation of 2D geometrical entities, such as lines and circles. Enables queries on these entities, including asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines.
Statistics	Support for a random variable type as well as the ability to declare this variable from prebuilt distribution functions such as Normal, Exponential, Coin, Die, and other custom distributions.
Polynomials	Computes polynomial algebras over various coefficient domains ranging from the simple (e.g., polynomial division) to the advanced (e.g., Gröbner bases [7] and multivariate factorization over algebraic number domains).
Sets	Representations of empty, finite, and infinite sets. This includes special sets such as for all natural, integer, and complex numbers.
Series	Implements series expansion, sequences, and limit of sequences. This includes special series, such as Fourier and power series.
Vectors	Provides basic vector math and differential calculus with respect to 3D Cartesian coordinate systems.
Matrices	Tools for creating matrices of symbols and expressions. This is capable of both sparse and dense representations and performing symbolic linear algebraic operations (e.g., inversion and factorization).
Combinatorics & Group Theory	Implements permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [25], and Prufer sequences [10].
Code Generation	Enables generation of compilable and executable code in a variety of different programming languages directly from expressions. Target languages include C, Fortran, Julia, JavaScript, Mathematica, Matlab and Octave, Python, and Theano.
Tensors	Symbolic manipulation of indexed objects.
Lie Algebras	Represents Lie algebras and root systems.
Cryptography	Represents block and stream ciphers, including shift, Affine, substitution, Vigenere's, Hill's, bifid, RSA, Kid RSA, linear-feedback shift registers, and Elgamal encryption



Special Functions	Implements a number of well known special functions, including Dirac delta, Gamma, Beta, Gauss error functions, Fresnel integrals, Exponential integrals, Logarithmic integrals, Trigonometric integrals, Bessel, Hankel, Airy, B-spline, Riemann Zeta, Dirichlet eta, polylogarithm, Lerch transcendent, hypergeometric, elliptic integrals, Mathieu, Jacobi polynomials, Gegenbauer polynomial, Chebyshev polynomial, Legendre polynomial, Hermite polynomial, Laguerre polynomial, and spherical harmonic functions.
-------------------	---

**4.1. Simplification.** The generic way to simplify an expression is by calling the `simplify` function. It must be emphasized that simplification is not an unambiguously defined mathematical operation [14]. The `simplify` function applies several simplification routines along with some heuristics to make the output expression as “simple” as possible.

It is often preferable to apply more directed simplification functions. These apply very specific rules to the input expression, and are often able to make guarantees about the output (for instance, the `factor` function, given a polynomial with rational coefficients in several variables, is guaranteed to produce a factorization into irreducible factors). Table 4.1 lists some common simplification functions.

<code>expand</code>	expand the expression
<code>factor</code>	factor a polynomial into irreducibles
<code>collect</code>	collect polynomial coefficients
<code>cancel</code>	rewrite a rational function as $p/q$ with common factors canceled
<code>apart</code>	compute the partial fraction decomposition of a rational function
<code>trigsimp</code>	simplify trigonometric expressions [17]

Substitutions are performed through the `.subs` method, which is sensible to some mathematical properties while matching, such as associativity, commutativity, additive and multiplicative inverses, and matching of powers.

**4.2. Calculus.** Derivatives can be computed with the `diff` function.

```
>>> diff(sin(x), x)
cos(x)
```

Unevaluated `Derivative` objects are also supported.

```
>>> expr = Derivative(sin(x), x)
>>> expr
Derivative(sin(x), x)
```

Unevaluated expressions can be evaluated with the `doit` method.

```
>>> expr.doit()
cos(x)
```

Integrals can be analogously calculated either with the `integrate` function, or the unevaluated `Integral` objects.

```
>>> integrate(sin(x), x)
-cos(x)
>>> expr = Integral(sin(x), x)
>>> expr
Integral(sin(x), x)
>>> expr.doit()
```

```

349 -cos(x)
350 Definite integration can be calculated with the same method, by specifying a range
351 of the integration variable. The following computes  $\int_0^1 \sin(x) dx$ .
352 >>> integrate(sin(x), (x, 0, 1))
353 -cos(1) + 1
354 SymPy implements a combination of the Risch algorithm [13], table lookups, a
355 reimplementation of Manuel Bronstein's "Poor Man's Integrator" [12], and an algo-
356 rithm for computing integrals based on Meijer G-functions. These allow SymPy to
357 compute a wide variety of indefinite and definite integrals.
358 Summations and products are also supported, via the evaluated summation and
359 product and unevaluated Sum and Product, and use the same syntax as integrate.
360 Summations are computed using a combination of Gosper's algorithm and an algo-
361 rithm that uses Meijer G-functions. Products are computed via some heuristics.
362 The limit module implements the Gruntz algorithm [19] for computing symbolic
363 limits. For example, the following computes  $\lim_{x \rightarrow \infty} x \sin(\frac{1}{x}) = 1$  (note that  $\infty$  is oo in
364 SymPy).
365 >>> limit(x*sin(1/x), x, oo)
366 1
367 As a more complicated example, SymPy computes  $\lim_{x \rightarrow 0} \left( 2e^{\frac{1-\cos(x)}{\sin(x)}} - 1 \right)^{\frac{\sinh(x)}{\operatorname{atan}^2(x)}} = e$ .
368 >>> limit((2*E**((1-cos(x))/sin(x))-1)**(sinh(x)/atan(x)**2), x, 0)
369 E

```

**4.3. Printers.** SymPy has a rich collection of expression printers for displaying expressions to the user. By default, an interactive Python session will render the `str` form of an expression, which has been used in all the examples in this paper so far.

```

373 >>> phi0 = Symbol('phi0')
374 >>> str(Integral(sqrt(phi0), phi0))
375 Integral(sqrt(phi0 + 1), x)

```

Expressions can be printed with 2D monospace text with `pprint`. This uses Unicode characters to render mathematical symbols such as integral signs, square roots, and parentheses. Greek letters and subscripts in symbol names are rendered automatically.

Alternately, the `use_unicode=False` flag can be set, which causes the expression to be printed using only ASCII characters.

```

382 >>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
383 /
384 |
385 | -----
386 | \ /  phi0 + 1   d(phi0)
387 |
388 /

```

The function `latex` returns a  $\text{\LaTeX}$  representation of an expression.

```

390 >>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
391 \int \sqrt{\phi_0 + 1} \, d\phi_0

```

Users are encouraged to run the `init_printing` function at the beginning of interactive sessions, which automatically enables the best pretty printing supported by their environment. In the Jupyter notebook or `qtconsole` [26] the  $\text{\LaTeX}$  printer is used to render expressions using MathJax or  $\text{\LaTeX}$  if it is installed on the system. The 2D text representation is used otherwise.

Other printers such as MathML are also available. SymPy uses an extensible printer subsystem which allows users to customize the printing for any given printer, and for custom objects to define their printing behavior for any printer. SymPy’s code generation capabilities, which we will not discuss in-depth here, use the same printer model.

**4.4. Sets.** SymPy supports representation of a wide variety of mathematical sets. This is achieved by first defining abstract representations of atomic set classes and then combining and transforming them using various set operations.

Each of the set classes inherits from the base class `Set` and defines methods to check membership and calculate unions, intersections, and set differences. When these methods are not able to evaluate to atomic set classes, they are represented as abstract unevaluated objects.

SymPy has the following atomic set classes:

- `EmptySet` represents the empty set  $\emptyset$ .
- `UniversalSet` is an abstract “universal set” for which everything is a member. The union of the universal set with any set gives the universal set and the intersection gives to the other set itself.
- `FiniteSet` is functionally equivalent to Python’s built `inset` object. Its members can be any SymPy object including other sets themselves.
- `Integers` represents the set of Integers  $\mathbb{Z}$ .
- `Naturals` represents the set of Natural numbers  $\mathbb{N}$ , i.e., the set of positive integers.
- `Naturals0` represents the whole numbers, which are all the non-negative integers.
- `Range` represents a range of integers. A range is defined by specifying a start value, an end value, and a step size. Range is functionally equivalent to Python’s `range` except it supports infinite endpoints, allowing the representation of infinite ranges.
- `Interval` represents an interval of real numbers. It is specified by giving the start and end point and specifying if it is open or closed in the respective ends.

Other than unevaluated classes of Union, Intersection and Set Difference operations, we have following set classes.

- `ProductSet` defines the Cartesian product of two or more sets. The product set is useful when representing higher dimensional spaces. For example to represent a three-dimensional space we simply take the Cartesian product of three real sets.
- `ImageSet` represents the image of a function when applied to a particular set. In notation, the image set of a function  $F$  with respect to a set  $S$  is  $\{F(x)|x \in S\}$ . SymPy uses image sets to represent sets of infinite solutions equations such as  $\sin(x) = 0$ .
- `ConditionSet` represents subset of a set whose members satisfies a particular condition. In notation, the condition set of the set  $S$  with respect to the condition  $H$  is  $\{x|H(x), x \in S\}$ . SymPy uses condition sets to represent the set of solutions of equations and inequalities, where the equation or the inequality is the condition and the set is the domain being solved over.

A few other classes are implemented as special cases of the classes described above. The set of real numbers, `Reals` is implemented as a special case of `Interval`,  $(-\infty, \infty)$ . `ComplexRegion` is implemented as a special case of `ImageSet`. `ComplexRegion`

446 supports both polar and rectangular representation of regions on the complex plane.

447 **4.5. Solvers.** SymPy has module of equation solvers for symbolic equations.  
 448 There are two submodules to solve algebraic equations in SymPy, referred to as old  
 449 solve function, `solve`, and new solve function, `solveset`. Solveset is introduced with  
 450 several design changes with respect to old `solve` function to resolve the issues with  
 451 old `solve` function, for example old `solve` function's input API has many flags which  
 452 are not needed and they make it hard for the user and the developers to work on  
 453 solvers. In contrast to old solve function, the `solveset` has a clean input API, It  
 454 only asks for the much needed information from the user, following are the function  
 455 signatures of old and new solve function:

```
456 solve(f, *symbols, **flags) # old solve function
457 solveset(f, symbol, domain) # new solve function
```

458 The old `solve` function has an inconsistent output API for various types of inputs,  
 459 whereas the `solveset` has a canonical output API which is achieved using sets. It  
 460 can consistently return various types of solutions.

461 • Single solution

```
462 >>> solveset(x - 1)
```

```
463 >>> {1}
```

464 • Finite set of solution, quadratic equation

```
465 >>> solveset(x**2 - pi**2, x)
```

```
466 {-pi, pi}
```

467 • No Solution

```
468 >>> solveset(1, x)
```

```
469 EmptySet()
```

470 • Interval of solution

```
471 >>> solveset(x**2 - 3 > 0, x, domain=S.Reals)
```

```
472 (-oo, -sqrt(3)) U (sqrt(3), oo)
```

473 • Infinitely many solutions

```
474 >>> solveset(sin(x) - 1, x, domain=S.Reals)
```

```
475 ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers())
```

```
476 >>> solveset(x - x, x, domain=S.Reals)
```

```
477 (-oo, oo)
```

```
478 >>> solveset(x - x, x, domain=S.Complexes)
```

```
479 S.Complexes
```

480 • Linear system: finite and infinite solution for determined, under determined  
 481 and over determined problems.

```
482 >>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
```

```
483 >>> b = Matrix([3, 6, 9])
```

```
484 >>> linsolve((A, b), x, y, z)
```

```
485 {(-1,2,0)}
```

```
486 >>> linsolve(Matrix([[1, 1, 1, 1], [1, 1, 2, 3]]), (x, y, z))
```

```
487 {(-y - 1, y, 2)}
```

488 The new solve i.e. `solveset` is under active development and is a planned replace-  
 489 ment for `solve`, Hence there are some features which are implemented in `solve` and is  
 490 not yet implemented in `solveset`. The table below show the current state of old and  
 491 new solve functions.

492

Solveset vs Solve		
Feature	solve	solveset
Consistent Output API	No	Yes
Consistent Input API	No	Yes
Univariate	Yes	Yes
Linear System	Yes	Yes (linsolve)
Non Linear System	Yes	Not yet
Transcendental	Yes	Not yet

Below are some of the examples of old **solve** function:

- Non Linear (multivariate) System of Equation: Intersection of a circle and a parabola.

```
>>> solve([x**2 + y**2 - 16, 4*x - y**2 + 6], x, y)
[(-2 + sqrt(14), -sqrt(-2 + 4*sqrt(14))),
 (-2 + sqrt(14), sqrt(-2 + 4*sqrt(14))),
 (-sqrt(14) - 2, -I*sqrt(2 + 4*sqrt(14))),
 (-sqrt(14) - 2, I*sqrt(2 + 4*sqrt(14)))]
```

- Transcendental Equation

```
>>> solve(x + log(x)**2 - 5*(x + log(x)) + 6, x)
[LambertW(exp(2)), LambertW(exp(3))]
>>> solve(x**3 + exp(x))
[-3*LambertW((-1)**(2/3)/3)]
```

Diophantine equations play a central and an important role in number theory. A Diophantine equation has the form,  $f(x_1, x_2, \dots, x_n) = 0$  where  $n \geq 2$  and  $x_1, x_2, \dots, x_n$  are integer variables. If we can find  $n$  integers  $a_1, a_2, \dots, a_n$  such that  $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$  satisfies the above equation, we say that the equation is solvable.

Currently, following five types of Diophantine equations can be solved using SymPy's Diophantine module.

- Linear Diophantine equations:  $a_1x_1 + a_2x_2 + \dots + a_nx_n = b$
- General binary quadratic equation:  $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation:  $ax^2 + by^2 + cz^2 + dxy + eyz + fzx = 0$
- Extended Pythagorean equation:  $a_1x_1^2 + a_2x_2^2 + \dots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
- General sum of squares:  $x_1^2 + x_2^2 + \dots + x_n^2 = k$

When an equation is fed into Diophantine module, it factors the equation (if possible) and solves each factor separately. Then all the results are combined to create the final solution set. Following examples illustrate some of the basic functionalities of the Diophantine module.

```
>>> from sympy import symbols
>>> x, y, z = symbols("x, y, z", integer=True)

>>> diophantine(2*x + 3*y - 5)
set([(3*t_0 - 5, -2*t_0 + 5)])

>>> diophantine(2*x + 4*y - 3)
set()

>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
set([(2, 1), (5, 1)])
```

```

536 >>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
537 set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])
538
539 >>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
540 set([(16*p**2 + 28*p*q + 20*q**2, 3*p**2 + 38*p*q - 25*q**2, 4*p**2 - 24*p*q + 68*q**2)])
541
542 >>> from sympy.abc import a, b, c, d, e, f
543 >>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
544 set([(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5, 420*t2*t5, 60*t3*t5, 210*t4*t5, 4
545
546 >>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)
547 set([(8, 4, 4, 4, 0, 0)])

```

548 **4.6. Matrices.** SymPy supports matrices with symbolic expressions as elements.■

```

549 >>> x, y = symbols('x y')
550 >>> A = Matrix(2, 2, [x, x + y, y, x])
551 >>> A
552 Matrix([
553 [ x, x + y],
554 [ y, x]])

```

555 All SymPy matrix types can do linear algebra including matrix addition, multipli-  
556 cation, exponentiation, computing determinant, solving linear systems and comput-  
557 ing inverses using LU decomposition, LDL decomposition, Gauss-Jordan elimination,  
558 Cholesky decomposition, Moore-Penrose pseudoinverse, and adjugate matrix.

559 All operations are computed are computed symbolically. Eigenvalues are com-  
560 puted by generating the characteristic polynomial using the Berkowitz algorithm and  
561 then solving it using polynomial routines. Diagonalizable matrices can be diagonalized  
562 first to compute the eigenvalues.

```

563 >>> A.eigenvals()
564 {x - sqrt(y*(x + y)): 1, x + sqrt(y*(x + y)): 1}

```

565 Internally these matrices store the elements as a list making it a dense repre-  
566 sentation. For storing sparse matrices, the `SparseMatrix` class can be used. Sparse  
567 matrices store the elements in a dictionary of keys (DoK) format.

568 SymPy also supports matrices with symbolic dimension values. `MatrixSymbol`  
569 represents a matrix with dimensions  $m \times n$ , where  $m$  and  $n$  can be symbolic. Matrix  
570 addition and multiplication, scalar operations, matrix inverse and transpose are stored  
571 symbolically as matrix expressions.

```

572 >>> m, n, p = symbols("m, n, p", integer=True)
573 >>> R = MatrixSymbol("R", m, n)
574 >>> S = MatrixSymbol("S", n, p)
575 >>> T = MatrixSymbol("t", m, p)
576 >>> U = R*S + 2*T
577 >>> u.shape
578 (m, p)
579 >>> U[0, 1]
580 2*T[0, 1] + Sum(R[0, _k]*S[_k, 1], (_k, 0, n - 1))

```

581 Block matrices are also supported in SymPy. `BlockMatrix` elements can be any  
582 matrix expression which includes explicit matrices, matrix symbols, and block matri-  
583 ces. All functionalities of matrix expressions are also present in `BlockMatrix`.

```

584 >>> n, m, l = symbols('n m l')

```

```

585 >>> X = MatrixSymbol('X', n, n)
586 >>> Y = MatrixSymbol('Y', m, m)
587 >>> Z = MatrixSymbol('Z', n, m)
588 >>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
589 >>> B
590 Matrix([
591 [X, Z],
592 [0, Y]])
593 >>> B[0, 0]
594 X[0, 0]
595 >>> B.shape
596 (m + n, m + n)

```

597 **5. Domain Specific Features.** SymPy includes several packages that allow  
598 users to solve domain specific problems. For example, a comprehensive physics pack-  
599 age is included that is useful for solving problems in classical mechanics, optics,  
600 and quantum mechanics along with support for manipulating physical quantities with  
601 units.

602 **5.1. Vector Algebra.** The `sympy.physics.vector` package provides reference  
603 frame, time, and space aware vector and dyadic objects that allow for three dimen-  
604 sional operations such as addition, subtraction, scalar multiplication, inner and outer  
605 products, cross products, etc. Both of these objects can be written in very compact  
606 notation that make it easy to express the vectors and dyadics in terms of multiple  
607 reference frames with arbitrarily defined relative orientations. The vectors are used  
608 to specify the positions, velocities, and accelerations of points, orientations, angular  
609 velocities, and angular accelerations of reference frames, and force and torques. The  
610 dyadics are essentially reference frame aware  $3 \times 3$  tensors. The vector and dyadic  
611 objects can be used for any one-, two-, or three-dimensional vector algebra and they  
612 provide a strong framework for building physics and engineering tools.

613 The following Python interpreter session showing how a vector is created using  
614 the orthogonal unit vectors of three reference frames that are oriented with respect  
615 to each other and the result of expressing the vector in the  $A$  frame. The  $B$  frame  
616 is oriented with respect to the  $A$  frame using Z-X-Z Euler Angles of magnitude  $\pi$ ,  $\frac{\pi}{2}$ ,  
617 and  $\frac{\pi}{3}$  rad, respectively whereas the  $C$  frame is oriented with respect to the  $B$  frame  
618 through a simple rotation about the  $B$  frame's  $X$  unit vector through  $\frac{\pi}{2}$  rad.

```

619 >>> from sympy import pi
620 >>> from sympy.physics.vector import ReferenceFrame
621 >>> A = ReferenceFrame('A')
622 >>> B = ReferenceFrame('B')
623 >>> C = ReferenceFrame('C')
624 >>> B.orient(A, 'body', (pi, pi / 3, pi / 4), 'zxz')
625 >>> C.orient(B, 'axis', (pi / 2, B.x))
626 >>> v = 1 * A.x + 2 * B.z + 3 * C.y
627 >>> v
628 A.x + 2*B.z + 3*C.y
629 >>> v.express(A)
630 A.x + 5*sqrt(3)/2*A.y + 5/2*A.z

```

631 **5.2. Classical Mechanics.** The `physics.mechanics` package utilizes the `physics.vector`  
632 package to populate time aware particle and rigid body objects to fully describe the

kinematics and kinetics of a rigid multi-body system. These objects store all of the information needed to derive the ordinary differential or differential algebraic equations that govern the motion of the system, i.e., the equations of motion. These equations of motion abide by Newton’s laws of motion and can handle any arbitrary kinematical constraints or complex loads. The package offers two automated methods for formulating the equations of motion based on Lagrangian Dynamics [23] and Kane’s Method [22]. Lastly, there are automated linearization routines for constrained dynamical systems based on [27].

**5.3. Quantum Mechanics.** The `sympy.physics.quantum` package provides quantum functions, states, operators, and computation of standard quantum models.

**5.4. Optics.** The `physics.optics` package provides Gaussian optics functions.

**5.5. Units.** The `physics.units` module provides around two hundred predefined prefixes and SI units that are commonly used in the sciences. Additionally, it provides the `Unit` class which allows the user to define their own units. These prefixes and units are multiplied by standard SymPy objects to make expressions unit aware, allowing for algebraic and calculus manipulations to be applied to the expressions while the units are tracked in the manipulations. The units of the expressions can be easily converted to other desired units. There is also a new units system in `sympy.physics.unitsystems` that allows the user to work in specified unit systems.

**6. Other Projects that use SymPy.** There are several projects that use SymPy as a library for implementing a part of their project, or even as a part of back-end for their application as well.

Some of them are listed below:

- **Cadabra:** Cadabra is a symbolic computer algebra system (CAS) designed specifically for the solution of problems encountered in field theory.
- **Octave Symbolic:** The Octave-Forge Symbolic package adds symbolic calculation features to GNU Octave. These include common Computer Algebra System tools such as algebraic operations, calculus, equation solving, Fourier and Laplace transforms, variable precision arithmetic and other features.
- **SymPy.jl:** Provides a Julia interface to SymPy using PyCall.
- **Mathics:** Mathics is a free, general-purpose online CAS featuring Mathematica compatible syntax and functions. It is backed by highly extensible Python code, relying on SymPy for most mathematical tasks.
- **Mathpix:** An iOS App, that uses Artificial Intelligence to detect handwritten math as input, and uses SymPy Gamma, to evaluate the math input and generate the relevant steps to solve the problem.
- **IKFast:** IKFast is a robot kinematics compiler provided by **OpenRAVE**. It analytically solves robot inverse kinematics equations and generates optimized C++ files. It uses SymPy for its internal symbolic mathematics.
- **Sage:** A CAS, visioned to be a viable free open source alternative to Magma, Maple, Mathematica and Matlab.
- **SageMathCloud:** SageMathCloud is a web-based cloud computing and course management platform for computational mathematics.
- **PyDy:** Multibody Dynamics with Python.
- **galgebra:** Geometric algebra (previously `sympy.galgebra`).
- **yt:** Python package for analyzing and visualizing volumetric data (`yt.units` uses SymPy).



- **SfePy**: Simple finite elements in Python.
- **Quameon**: Quantum Monte Carlo in Python.
- **Lcapy**: Experimental Python package for teaching linear circuit analysis.
- **Quantum Programming in Python**: Quantum 1D Simple Harmonic Oscillator and Quantum Mapping Gate.
- **LaTeX Expression project**: Easy LaTeX typesetting of algebraic expressions in symbolic form with automatic substitution and result computation.
- **Symbolic statistical modeling**: Adding statistical operations to complex physical models.

## 7. Conclusion and future work.

## 8. References.

### REFERENCES

- [1] <https://github.com/sympy/sympy/blob/master/doc/src/modules/polys/ringseries.rst>.
- [2] <https://reference.wolfram.com/language/ref/Flat.html>.
- [3] <https://reference.wolfram.com/language/ref/Orderless.html>.
- [4] <https://reference.wolfram.com/language/ref/OneIdentity.html>.
- [5] <https://reference.wolfram.com/language/tutorial/FlatAndOrderlessFunctions.html>.
- [6] *The software engineering of the wolfram system*, 2016, <https://reference.wolfram.com/language/tutorial/TheSoftwareEngineeringOfTheWolframSystem.html>.
- [7] W. W. ADAMS AND P. LOUSTAUNAU, *An introduction to Gröbner bases*, no. 3, American Mathematical Soc., 1994.
- [8] D. H. BAILEY, K. JEYABALAN, AND X. S. LI, *A comparison of three high-precision quadrature schemes*, *Experimental Mathematics*, 14 (2005), pp. 317–329.
- [9] C. M. BENDER AND S. A. ORSZAG, *Advanced Mathematical Methods for Scientists and Engineers*, Springer, 1st ed., October 1999.
- [10] N. BIGGS, E. K. LLOYD, AND R. J. WILSON, *Graph Theory, 1736-1936*, Oxford University Press, 1976.
- [11] R. P. BRENT AND P. ZIMMERMANN, *Modern Computer Arithmetic*, Cambridge University Press, version 0.5.1 ed.
- [12] M. BRONSTEIN, *Poor Man's Integrator*, <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint>.
- [13] M. BRONSTEIN, *Symbolic Integration I: Transcendental Functions*, Springer-Verlag, New York, NY, USA, 2005.
- [14] J. CARETTE, *Understanding Expression Simplification*, in ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 2004, ACM Press, pp. 72–79, <http://dx.doi.org/http://doi.acm.org/10.1145/1005285.1005298>.
- [15] R. J. FATEMAN, *A review of Mathematica*, *Journal of Symbolic Computation*, 13 (1992), pp. 545–579, [http://dx.doi.org/DOI:10.1016/S0747-7171\(10\)80011-2](http://dx.doi.org/DOI:10.1016/S0747-7171(10)80011-2).
- [16] H. R. P. FERGUSON, D. H. BAILEY, AND S. ARNO, *Analysis of PSLQ, an integer relation finding algorithm*, *Mathematics of Computation*, 68 (1999), pp. 351–369.
- [17] H. FU, X. ZHONG, AND Z. ZENG, *Automated and Readable Simplification of Trigonometric Expressions*, *Mathematical and Computer Modelling*, 55 (2006), pp. 1169–1177.
- [18] D. GOLDBERG, *What every computer scientist should know about floating-point arithmetic*, *ACM Computing Surveys (CSUR)*, 23 (1991), pp. 5–48.
- [19] D. GRUNTZ, *On Computing Limits in a Symbolic Manipulation System*, PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.
- [20] D. GRUNTZ AND W. KOEPF, *Formal power series*, (1993).
- [21] C. V. HORSER, *GMPY*, <https://pypi.python.org/pypi/gmpy2>, 2015.
- [22] T. R. KANE AND D. A. LEVINSON, *Dynamics, Theory and Applications*, McGraw Hill, 1985.
- [23] J. LAGRANGE, *Mécanique analytique*, no. v. 1 in *Mécanique analytique*, Ve Courcier, 1811.
- [24] M. MOSKEWICZ, C. MADIGAN, AND S. MALIK, *Method and system for efficient implementation of boolean satisfiability*, Aug. 26 2008, <http://www.google.co.in/patents/US7418369>. US Patent 7,418,369.
- [25] A. NIJENHUIS AND H. S. WILF, *Combinatorial Algorithms: For Computers and Calculators*, Academic Press, New York, NY, USA, second ed., 1978.
- [26] F. PÉREZ AND B. E. GRANGER, *Ipython: a system for interactive scientific computing*, Com-

- puting in Science & Engineering, 9 (2007), pp. 21–29.
- [27] D. L. PETERSON, G. GEDE, AND M. HUBBARD, *Symbolic linearization of equations of motion of constrained multibody systems*, Multibody System Dynamics, 33 (2014), pp. 143–161, <http://dx.doi.org/10.1007/s11044-014-9436-5>.
- [28] M. PETKOVŠEK, H. S. WILF, AND D. ZEILBERGER, *A = bak peters*, Wellesley, MA, (1996).
- [29] M. SOFRONIOU AND G. SPALETTA, *Precise numerical computation*, Journal of Logic and Algebraic Programming, 64 (2005), pp. 113–134.
- [30] H. TAKAHASI AND M. MORI, *Double exponential formulas for numerical integration*, Publications of the Research Institute for Mathematical Sciences, 9 (1974), pp. 721–741.
- [31] V. T. TOTH, *Maple and meijer's g-function: a numerical instability and a cure*. <http://www.vttoth.com/CMS/index.php/technical-notes/67>, 2007.

## 9. Supplement.

**9.1. The Gruntz Algorithm.** We first define comparability classes by calculating  $L$ :

$$(1) \quad L \equiv \lim_{x \rightarrow \infty} \frac{\log |f(x)|}{\log |g(x)|}$$

And then we define the  $<$ ,  $>$  and  $\sim$  operations as follows:  $f > g$  when  $L = \pm\infty$  ( $f$  is more rapidly varying than  $g$ , i.e.,  $f$  goes to  $\infty$  or 0 faster than  $g$ ,  $f$  is greater than any power of  $g$ ),  $f < g$  when  $L = 0$  ( $f$  is less rapidly varying than  $g$ ) and  $f \sim g$  when  $L \neq 0, \pm\infty$  (both  $f$  and  $g$  are bounded from above and below by suitable integral powers of the other).

Examples:

$$\begin{aligned} 2 &< x < e^x < e^{x^2} < e^{e^x} \\ 2 &\sim 3 \sim -5 \\ x &\sim x^2 \sim x^3 \sim \frac{1}{x} \sim x^m \sim -x \\ e^x &\sim e^{-x} \sim e^{2x} \sim e^{x+e^{-x}} \\ f(x) &\sim \frac{1}{f(x)} \end{aligned}$$

The Gruntz algorithm, on an example:

$$\begin{aligned} f(x) &= e^{x+2e^{-x}} - e^x + \frac{1}{x} \\ \lim_{x \rightarrow \infty} f(x) &=? \end{aligned}$$

Strategy: mrv set: the set of most rapidly varying subexpressions  $\{e^x, e^{-x}, e^{x+2e^{-x}}\}$ , the same comparability class. Take an item  $\omega$  from mrv, converging to 0 at infinity. Here  $\omega = e^{-x}$ . If not present in the mrv set, use the relation  $f(x) \sim \frac{1}{f(x)}$ .

Rewrite the mrv set using  $\omega$ :  $\{\frac{1}{\omega}, \omega, \frac{1}{\omega}e^{2\omega}\}$ , substitute back into  $f(x)$  and expand in  $\omega$ :

$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2)$$

The core idea of the algorithm:  $\omega$  is from the mrv set, so in the limit  $\omega \rightarrow 0$ :

$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \rightarrow 2 + \frac{1}{x}$$

We iterate until we get just a number, the final limit. Gruntz proved this algorithm always works and converges in his Ph.D. thesis [19].

Generally:

$$f(x) = \underbrace{O\left(\frac{1}{\omega^3}\right)}_{\infty} + \underbrace{\frac{C_{-2}(x)}{\omega^2}}_{\infty} + \underbrace{\frac{C_{-1}(x)}{\omega}}_{\infty} + C_0(x) + \underbrace{C_1(x)\omega}_0 + \underbrace{O(\omega^2)}_0$$

we look at the lowest power of  $\omega$ . The limit is one of: 0,  $\lim_{x \rightarrow \infty} C_0(x)$ ,  $\infty$ .

## 9.2. Series.

**9.2.1. Series Expansion.** SymPy is able to calculate the symbolic series expansion of an arbitrary series or expression involving elementary and special functions and multiple variables. For this it has two different implementations- the `series` method and Ring Series.

The first approach stores a series as an object of the `Basic` class. Each function has its specific implementation of its expansion which is able to evaluate the Puiseux series expansion about a specified point. For example, consider a Taylor expansion about 0:

```
>>> from sympy import symbols, series
>>> x, y = symbols('x, y')
>>> series(sin(x+y) + cos(x*y), x, 0, 2)
1 + sin(y) + x*cos(y) + O(x**2)
```

The newer and much faster[1] approach called Ring Series makes use of the observation that a truncated Taylor series, is in fact a polynomial. Ring Series uses the efficient representation and operations of sparse polynomials. The choice of sparse polynomials is deliberate as it performs well in a wider range of cases than a dense representation. Ring Series gives the user the freedom to choose the type of coefficients he wants to have in his series, allowing the use of faster operations on certain types.

For this, several low level methods for expansion of trigonometric, hyperbolic and other elementary functions like inverse of a series, calculating  $n$ th root, etc, are implemented using variants of the Newton[11] Method. All these support Puiseux series expansion. The following example demonstrates the use of an elementary function that calculates the Taylor expansion of the `sine` of a series.

```
>>> from sympy import ring
>>> from sympy.polys.ring_series import rs_sin
>>> R, x = ring('x', QQ)
>>> rs_sin(x**2 + x, x, 5)
-1/2*x**4 - 1/6*x**3 + x**2 + x
```

The function `sympy.polys.rs_series` makes use of these elementary functions to expand an arbitrary SymPy expression. It does so by following a recursive strategy of expanding the lower most functions first and then composing them recursively to calculate the desired expansion. Currently it only supports expansion about 0 and is under active development. Ring Series is several times faster than the default implementation with the speed difference increasing with the size of the series. The `sympy.polys.rs_series` takes as input any SymPy expression and hence there is no need to explicitly create a polynomial ring. An example:

```
>>> from sympy.polys.ring_series import rs_series
>>> from sympy.abc import a, b
>>> from sympy import sin, cos
>>> rs_series(sin(a + b), a, 4)
-1/2*(sin(b))*a**2 + (sin(b)) - 1/6*(cos(b))*a**3 + (cos(b))*a
```

**9.2.2. Formal Power Series.** SymPy can be used for computing the Formal Power Series of a function. The implementation is based on the algorithm described in the paper on Formal Power Series[20]. The advantage of this approach is that an explicit formula for the coefficients of the series expansion is generated rather than just computing a few terms.

The following example shows how to use `fps`:

```
>>> f = fps(sin(x), x, x0=0)
>>> f.truncate(6)
x - x**3/6 + x**5/120 + O(x**6)
>>> f[15]
-x**15/1307674368000
```

**9.2.3. Fourier Series.** SymPy provides functionality to compute Fourier Series of a function using the `fourier_series` function. Under the hood it just computes  $a_0$ ,  $a_n$ ,  $b_n$  using standard integration formulas.

Here's an example on how to compute Fourier Series in SymPy:

```
>>> L = symbols('L')
>>> f = fourier_series(2 * (Heaviside(x/L) - Heaviside(x/L - 1)) - 1, (x, 0, 2*L))
>>> f.truncate(3)
4*sin(pi*x/L)/pi + 4*sin(3*pi*x/L)/(3*pi) + 4*sin(5*pi*x/L)/(5*pi)
```

**9.3. Logic.** SymPy supports construction and manipulation of boolean expressions through the `logic` module. SymPy symbols can be used as propositional variables and also be substituted as `True` or `False`. A good number of manipulation features for boolean expressions have been implemented in the `logic` module.

**9.3.1. Constructing boolean expressions.** A boolean variable can be declared as a SymPy symbol. Python operators `&`, `|` and `~` are overloaded for logical `And`, `Or` and `negate`. Several others like `Xor`, `Implies` can be constructed with `^`, `>>` respectively. The above are just a shorthand, expressions can also be constructed by directly calling `And()`, `Or()`, `Not()`, `Xor()`, `Nand()`, `Nor()`, etc.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> e = (x & y) | z
>>> e.subs({x: True, y: True, z: False})
True
```

**9.3.2. CNF and DNF.** Any boolean expression can be converted to conjunctive normal form, disjunctive normal form and negation normal form. The API also permits to check if a boolean expression is in any of the above mentioned forms.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> to_cnf((x & y) | z)
And(Or(x, z), Or(y, z))
>>> to_dnf(x & (y | z))
Or(And(x, y), And(x, z))
>>> is_cnf((x | y) & z)
True
>>> is_dnf((x & y) | z)
True
```

**9.3.3. Simplification and Equivalence.** The module supports simplification of given boolean expression by making deductions on it. Equivalence of two expres-

```

854 sions can also be checked. If so, it is possible to return the mapping of variables of
855 two expressions so as to represent the same logical behaviour.
856 >>> from sympy import *
857 >>> a, b, c, x, y, z = symbols('a b c x y z')
858 >>> e = a & (~a | ~b) & (a | c)
859 >>> simplify(e)
860 And(Not(b), a)
861 >>> e1 = a & (b | c)
862 >>> e2 = (x & y) | (x & z)
863 >>> bool_map(e1, e2)
864 (And(Or(b, c), a), {b: y, a: x, c: z})

```

865 **9.3.4. SAT solving.** The module also supports satisfiability checking of a given  
866 boolean expression. If satisfiable, it is possible to return a model for which the ex-  
867 pression is satisfiable. The API also supports returning all possible models. The SAT  
868 solver has a clause learning DPLL algorithm implemented with watch literal scheme  
869 and VSIDS heuristic[24].

```

870 >>> from sympy import *
871 >>> a, b, c = symbols('a b c')
872 >>> satisfiable(a & (~a | b) & (~b | c) & ~c)
873 False
874 >>> satisfiable(a & (~a | b) & (~b | c) & c)
875 {b: True, a: True, c: True}

```

876 **9.4. SymPy Gamma.** SymPy Gamma is a simple web application that runs  
877 on Google App Engine. It executes and displays the results of SymPy expressions as  
878 well as additional related computations, in a fashion similar to that of Wolfram|Alpha.  
879 For instance, entering an integer will display its prime factors, digits in the base-10  
880 expansion, and a factorization diagram. Entering a function will display its docstring;  
881 in general, entering an arbitrary expression will display its derivative, integral, series  
882 expansion, plot, and roots.

883 SymPy Gamma also has several additional features than just computing the re-  
884 sults using SymPy.

- 885 • It displays integration steps, differentiation steps in detail, which can be  
886 viewed in Figure 1:



Fig. 1: Integral steps of  $\tan(x)$

- It also displays the factor tree diagrams for different numbers.
- SymPy Gamma also saves user search queries, and offers many such similar features for free, which Wolfram|Alpha only offers to its paid users.

Every input query from the user on SymPy Gamma is first, parsed by its own parser, which handles several different forms of function names, which SymPy as a library doesn't support. For instance, SymPy Gamma supports queries like `sin x`, whereas SymPy doesn't support this, and supports only `sin(x)`.

This parser converts the input query to the equivalent SymPy readable code, which is then eventually processed by SymPy and the result is finally formatted in LaTeX and displayed on the SymPy Gamma web-application.

**9.5. SymPy Live.** SymPy Live is an online Python shell, which runs on Google App Engine, that executes SymPy code. It is integrated in the SymPy documentation examples, located at this [link](#).

This is accomplished by providing a HTML/JavaScript GUI for entering source code and visualization of output, and a server part which evaluates the requested source code. It's an interactive AJAX shell, that runs SymPy code using Python on the server.

Certain Features of SymPy Live:

- It supports the exact same syntax as SymPy, hence it can be used easily, to test for outputs of various SymPy expressions.
- It can be run as a standalone app or in an existing app as an admin-only handler, and can also be used for system administration tasks, as an interactive way to try out APIs, or as a debugging aid during development.
- It can also be used to plot figures ([link](#)), and execute all kinds of expressions that SymPy can evaluate.
- SymPy Live also formats the output in LaTeX for pretty-printing the output.

**9.6. Comparison with Mathematica.** Wolfram Mathematica is a popular proprietary CAS. It features highly advanced algorithms. Mathematica has a core implemented in C++ [6] which interprets its own programming language (known as Wolfram language).

Analogously to Lisp’s S-expressions, Mathematica uses its own style of M-expressions, which are arrays of either atoms or other M-expression. The first element of the expression identifies the type of the expression and is indexed by zero, whereas the first argument is indexed by one. Notice that SymPy expression arguments are stored in a Python tuple (that is, an immutable array), while the expression type is identified by the type of the object storing the expression.

Mathematica can associate attributes to its atoms. Attributes may define mathematical properties and behavior of the nodes associated to the atom. In SymPy, the usage of static class fields is roughly similar to Mathematica’s attributes, though other programming patterns may also be used to achieve an equivalent behavior, such as class inheritance.

Unlike SymPy, Mathematica’s expressions are mutable, that is one can change parts of the expression tree without the need of creating a new object. The reactivity of Mathematica allows for a lazy updating of any references to that data structure.

Products in Mathematica are determined by some builtin node types, such as `Times`, `Dot`, and others. `Times` is overloaded by the `*` operator, and is always meant to represent a commutative operator. The other notable product is `Dot`, overloaded by the `.` operator. This product represents matrix multiplication, it is not commutative. SymPy uses the same node for both scalar and matrix multiplication, the only exception being with abstract matrix symbols. Unlike Mathematica, SymPy determines commutativity with respect to multiplication from the factor’s expression type. Mathematica puts the `Orderless` attribute on the expression type.

Regarding associative expressions, SymPy handles associativity by making associative expressions inherit the class `AssocOp`, while Mathematica specifies the `Flat` attribute on the expression type.

Mathematica relies heavily on pattern matching: even the so-called equivalent of function declaration is in reality the definition of a pattern matching generating an expression tree transformation on input expressions. Mathematica’s pattern matching is sensitive to associative[2], commutative[3], and one-identity[4] properties of its expression tree nodes[5]. SymPy has various ways to perform pattern matching. All of them play a lesser role in the CAS than in Mathematica and are basically available as a tool to rewrite expressions. The differential equation solver in SymPy somewhat relies on pattern matching to identify the kind of differential equation, but it is envisaged to replace that strategy with analysis of Lie symmetries in the future. Mathematica’s real advantage is the ability to add new overloading to the expression builder at runtime, or for specific subnodes. Consider for example

```
In[1] := Unprotect[Plus]
```



```

955
956 Out[1]= {Plus}
957
958 In[2]:= Sin[x_]^2 + Cos[y_]^2 := 1
959
960 In[3]:= x + Sin[t]^2 + y + Cos[t]^2
961
962 Out[3]= 1 + x + y
963 This expression in Mathematica defines a substitution rule that overloads the func-
964 tionality of the Plus node (the node for additions in Mathematica). The trailing
965 underscore after a symbol means that it is to be considered a wildcard. This example
966 may not be practical, one may wish to keep this identity unevaluated, nevertheless
967 it clearly illustrates the potentiality to define one's own immediate transformation
968 rules. In SymPy the operations constructing the addition node in the expression tree
969 are Python class constructors, and cannot be modified at runtime5 The way SymPy
970 deals with extending the missing runtime overloadability functionality is by subclass-
971 ing the node types. Subclasses may overload the class constructor to yield the proper
972 extended functionality.
973 Unlike SymPy, Mathematica does not support type inheritance or polymorphism[15].
974 SymPy relies heavily on class inheritance, but for the most part, class inheritance is
975 used to make sure that SymPy objects inherit the proper methods and implement the
976 basic hashing system. Associativity of expressions can be achieved by inheriting the
977 class AssocOp, which may appear a more cumbersome operation than Mathematica's
978 attribute setting.
979 Matrices in SymPy are types on their own. In Mathematica, nested lists are
980 interpreted as matrices whenever the sublists have the same length. The main differ-
981 ence to SymPy is that ordinary operators and functions do not get generalized the
982 same way as used in traditional mathematics. Using the standard multiplication in
983 Mathematica performs an elementwise product, this is compatible with Mathemat-
984 ica's convention of commutativity of Times nodes. Matrix product is expressed by
985 the dot operator, or the Dot node. The same is true for the other operators, and
986 even functions, most notably calling the exponential function Exp on a matrix returns
987 an elementwise exponentiation of its elements. The real matrix exponentiationl is
988 available through the MatrixExp function.
989 Unevaluated expressions can be achieved in various ways, most commonly with
990 the HoldForm or Hold nodes, that block the evaluation of subnodes by the parser.
991 Note that such a node cannot be expressed in Python, because of greedy evaluation.
992 Whenever needed in SymPy, it is necessary to add the parameter evaluate=False to
993 all subnodes, or put the input expression in a string.
994 The operator == returns a boolean whenever it is able to immediately evaluate
995 the truthness of the equality, otherwise it returns an Equal expression. In SymPy ==
996 means structural equality and is always guaranteed to return a boolean expression.
997 To express an equality in SymPy it is necessary to explicitly construct the Equality
998 class.
999 SymPy, in accordance with Python and unlike the usual programming convention,
1000 uses ** to express the power operator, while Mathematica uses the more common ^.

```

---

<sup>5</sup>In reality, Python supports monkey patching, nonetheless it is a discouraged programming pattern.