

SYMPY: SYMBOLIC COMPUTING IN PYTHON

AARON MEURER*, ONDŘEJ ČERTÍK†, JASON K. MOORE‡, FRANCESCO BONAZZI§,
SHIVAM VATS¶, AMIT KUMAR||, HARSH GUPTA#, 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`.

*University of South Carolina (asmeurer@gmail.com).

†Los Alamos National Laboratory (ondrej.certik@gmail.com).

‡University of California, Davis (jkm@ucdavis.edu).

§?? (??).

¶Indian Institute of Technology Kharagpur (shivamvats.iitkgp@gmail.com).

||Delhi Technological University (dtu.amit@gmail.com).

#Indian Institute of Technology Kharagpur (hargup@protonmail.com).

††University of Moratuwa (isuru.11@cse.mrt.ac.lk).

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

```

37 >>> type(expr)
38 <class 'sympy.core.add.Add'>
39 >>> expr.args
40 (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.

```

44 >>> expr.args[0]
45 2
46 >>> type(expr.args[0])
47 <class 'sympy.core.numbers.Integer'>
48 >>> expr.args[0].args
49 ()

```

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

```

52 >>> srepr(expr)
53 "Add(Mul(Symbol('x'), Symbol('y')), Integer(2))"

```

Every SymPy expression satisfies a key invariant, namely, `expr.func(*expr.args) == expr`.¹ This means that expressions are rebuildable from their `args`.¹ 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`.² 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

¹`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.

²This 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`.

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.

```
>>> x = Symbol('x')
>>> sqrt(x**2)
sqrt(x**2)
```

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 $\sqrt{x^2}$.

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.

```
>>> x = Symbol('x', positive=True)
>>> sqrt(x**2)
x
```

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

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`.

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.

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`.

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.

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.

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

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 trees⁴ 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 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
```

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

```

176
177     def _eval_rewrite_as_factorial(self, z):
178         return factorial(z - 1)
179
180     def fdiff(self, argindex=1):
181         from sympy.core.function import ArgumentIndexError
182         if argindex == 1:
183             return self.func(self.args[0])*polygamma(0, self.args[0])
184         else:
185             raise ArgumentIndexError(self, argindex)
186
187     The actual gamma function defined in SymPy has many more capabilities, such
188     as evaluation at rational points and series expansion.

```

188 **3. Numerics.** The `Float` class holds an arbitrary-precision binary floating-point value and a precision in bits. An operation between two `Float` inputs is rounded to the larger of the two precisions. Since Python floating-point literals automatically evaluate to `double` (53-bit) precision, strings should be used to input precise decimal values:

```

193 >>> Float(1.1)
194 1.1000000000000000
195 >>> Float(1.1, 30) # precision equivalent to 30 digits
196 1.100000000000000008881784197001
197 >>> Float("1.1", 30)
198 1.10000000000000000000000000000000000000000000000000000

```

199 The preferred way to evaluate an expression numerically is with the `evalf` method, which internally estimates the number of accurate bits of the floating-point approximation for each sub-expression, and adaptively increases the working precision until the estimated accuracy of the final result matches the sought number of decimal digits.

203 The internal error tracking does not provide rigorous error bounds (in the sense of interval arithmetic) and cannot be used to track uncertainty in measurement data in any meaningful way; the sole purpose is to mitigate loss of accuracy that typically occurs when converting symbolic expressions to numerical values, for example due to catastrophic cancellation. This is illustrated by the following example (the input 25 specifies that 25 digits are sought):

```

209 >>> cos(exp(-100)).evalf(25) - 1
210 0
211 >>> (cos(exp(-100)) - 1).evalf(25)
212 -6.919482633683687653243407e-88

```

213 The `evalf` method works with complex numbers and supports more complicated expressions, such as special functions, infinite series and integrals.

215 SymPy does not track the accuracy of approximate numbers outside of `evalf`. The familiar dangers of floating-point arithmetic apply [18], and symbolic expressions containing floating-point numbers should be treated with some caution. This approach is similar to Maple and Maxima.

219 By contrast, Mathematica uses a form of significance arithmetic [32] for approximate numbers. This offers further protection against numerical errors, but leads to non-obvious semantics while still not being mathematically rigorous (for a critique of significance arithmetic, see Fateman [15]). SymPy's `evalf` internals are non-rigorous in the same sense, but have no bearing on the semantics of floating-point numbers in the rest of the system.

$$K_\nu(z) = \frac{1}{2} \left[\left(\frac{z}{2}\right)^{-\nu} \Gamma(\nu) {}_0F_1\left(1 - \nu, \frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^\nu \frac{\pi}{\nu \sin(\pi\nu) \Gamma(\nu)} {}_0F_1\left(\nu + 1, \frac{z^2}{4}\right) \right]$$

where the limiting value $\lim_{\varepsilon \rightarrow 0} K_{n+\varepsilon}(z)$ has to be computed when $\nu = n$ is an integer. A generic algorithm is used to evaluate hypergeometric-type linear combinations of the above type. This algorithm automatically detects cancellation problems, and computes limits numerically by perturbing parameters whenever internal singularities occur (the perturbation size is automatically decreased until the result is detected to converge numerically).

Due to this generic approach, particular combinations of hypergeometric functions can be specified easily. The implementation of the Meijer G-function takes only a few dozen lines of code, yet covers the whole input domain in a robust way. The Meijer G-function instance $G_{1,3}^{3,0}\left(0; \frac{1}{2}, -1, -\frac{3}{2} | x\right)$ is a good test case [34]; past versions of both Maple and Mathematica produced incorrect numerical values for large $x > 0$. Here, mpmath automatically removes the internal singularity and compensates for cancellations (amounting to 656 bits of precision when $x = 10000$), giving correct values:

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

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

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

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

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

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

4. Features. SymPy has an extensive feature set that encompasses too much to cover in-depth here. Bedrock areas, such as calculus, receive their own sub-sections below. Additionally, Table 1 describes other capabilities present in the SymPy code base. This gives a sampling from the breadth of topics and application domains that 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 [31] 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 [27], 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 polynomial, 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 2 lists some common simplification functions.

Table 2: SymPy 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()
-cos(x)
```

Definite integration can be calculated with the same method, by specifying a range of the integration variable. The following computes $\int_0^1 \sin(x) dx$.

```
>>> integrate(sin(x), (x, 0, 1))
-cos(1) + 1
```

SymPy implements a combination of the Risch algorithm [13], table lookups, a reimplement of Manuel Bronstein’s “Poor Man’s Integrator” [12], and an algorithm for computing integrals based on Meijer G-functions. These allow SymPy to compute a wide variety of indefinite and definite integrals.

Summations and products are also supported, via the evaluated `summation` and `product` and unevaluated `Sum` and `Product`, and use the same syntax as `integrate`. Summations are computed using a combination of Gosper’s algorithm and an algorithm that uses Meijer G-functions. Products are computed via some heuristics.

The limit module implements the Gruntz algorithm [19] for computing symbolic limits. For example, the following computes $\lim_{x \rightarrow \infty} x \sin(\frac{1}{x}) = 1$ (note that ∞ is `oo` in SymPy).

```
>>> limit(x*sin(1/x), x, oo)
1
```

As a more complicated example, SymPy computes $\lim_{x \rightarrow 0} \left(2e^{\frac{1-\cos(x)}{\sin(x)}} - 1 \right)^{\frac{\sinh(x)}{\operatorname{atan}^2(x)}} = e$.

```
>>> limit((2*E**((1-cos(x))/sin(x))-1)**(sinh(x)/atan(x)**2), x, 0)
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.

```
>>> phi0 = Symbol('phi0')
>>> str(Integral(sqrt(phi0), phi0))
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.

```
>>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
/
|
|  -----
|  \ /  phi0 + 1  d(phi0)
|
/
```

The function `latex` returns a \LaTeX representation of an expression.

```
>>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
\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` [29] the \LaTeX printer is used to render expressions using MathJax or \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. Solvers. SymPy has module of equation solvers for symbolic equations. There are two submodules to solve algebraic equations in SymPy, referred to as old solve function, `solve`, and new solve function, `solveset`. `Solveset` is introduced with several design changes with respect to old `solve` function to resolve the issues with old `solve` function, for example old `solve` function's input API has many flags which are not needed and they make it hard for the user and the developers to work on solvers. In contrast to old solve function, the `solveset` has a clean input API, It only asks for the much needed information from the user, following are the function signatures of old and new solve function:

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

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

- Single solution

```
>>> solveset(x - 1)
{1}
```

- Finite set of solution, quadratic equation

```
>>> solveset(x**2 - pi**2, x)
{-pi, pi}
```

- No Solution

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

426 • Interval of solution
427 >>> solveset(x**2 - 3 > 0, x, domain=S.Reals)
428 (-oo, -sqrt(3)) U (sqrt(3), oo)
429 • Infinitely many solutions
430 >>> solveset(sin(x) - 1, x, domain=S.Reals)
431 ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers())
432 >>> solveset(x - x, x, domain=S.Reals)
433 (-oo, oo)
434 >>> solveset(x - x, x, domain=S.Complexes)
435 S.Complexes
436 • Linear system: finite and infinite solution for determined, under determined
437 and over determined problems.
438 >>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
439 >>> b = Matrix([3, 6, 9])
440 >>> linsolve((A, b), x, y, z)
441 {(-1,2,0)}
442 >>> linsolve(Matrix([[1, 1, 1, 1], [1, 1, 2, 3]]), (x, y, z))
443 {(-y - 1, y, 2)}
444 The new solve i.e. **solveset** is under active development and is a planned replace-
445 ment for **solve**. Hence there are some features which are implemented in solve and is
446 not yet implemented in solveset. The table below show the current state of old and
447 new solve functions.

448

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

449

450

451

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

453 • Non Linear (multivariate) System of Equation: Intersection of a circle and a
454 parabola.

455 >>> solve([x**2 + y**2 - 16, 4*x - y**2 + 6], x, y)
456 [(-2 + sqrt(14), -sqrt(-2 + 4*sqrt(14))),
457 (-2 + sqrt(14), sqrt(-2 + 4*sqrt(14))),
458 (-sqrt(14) - 2, -I*sqrt(2 + 4*sqrt(14))),
459 (-sqrt(14) - 2, I*sqrt(2 + 4*sqrt(14)))]

460 • Transcendental Equation

461 >>> solve(x + log(x)**2 - 5*(x + log(x)) + 6, x)
462 [LambertW(exp(2)), LambertW(exp(3))]
463 >>> solve(x**3 + exp(x))
464 [-3*LambertW((-1)**(2/3)/3)]

465 **4.5. Matrices.** SymPy supports matrices with symbolic expressions as elements.■

466 >>> x, y = symbols('x y')
467 >>> A = Matrix(2, 2, [x, x + y, y, x])
468 >>> A

```

469 Matrix([
470 [ x, x + y],
471 [ y, x]])
472 All SymPy matrix types can do linear algebra including matrix addition, multipli-
473 cation, exponentiation, computing determinant, solving linear systems and comput-
474 ing inverses using LU decomposition, LDL decomposition, Gauss-Jordan elimination,
475 Cholesky decomposition, Moore-Penrose pseudoinverse, and adjugate matrix.
476 All operations are computed are computed symbolically. Eigenvalues are com-
477 puted by generating the characteristic polynomial using the Berkowitz algorithm and
478 then solving it using polynomial routines. Diagonalizable matrices can be diagonalized
479 first to compute the eigenvalues.
480 >>> A.eigenvals()
481 {x - sqrt(y*(x + y)): 1, x + sqrt(y*(x + y)): 1}
482 Internally these matrices store the elements as a list making it a dense repre-
483 sentation. For storing sparse matrices, the SparseMatrix class can be used. Sparse
484 matrices store the elements in a dictionary of keys (DoK) format.
485 SymPy also supports matrices with symbolic dimension values. MatrixSymbol
486 represents a matrix with dimensions  $m \times n$ , where  $m$  and  $n$  can be symbolic. Matrix
487 addition and multiplication, scalar operations, matrix inverse and transpose are stored
488 symbolically as matrix expressions.
489 >>> m, n, p = symbols("m, n, p", integer=True)
490 >>> R = MatrixSymbol("R", m, n)
491 >>> S = MatrixSymbol("S", n, p)
492 >>> T = MatrixSymbol("t", m, p)
493 >>> U = R*S + 2*T
494 >>> u.shape
495 (m, p)
496 >>> U[0, 1]
497 2*T[0, 1] + Sum(R[0, _k]*S[_k, 1], (_k, 0, n - 1))
498 Block matrices are also supported in SymPy. BlockMatrix elements can be any
499 matrix expression which includes explicit matrices, matrix symbols, and block matri-
500 ces. All functionalities of matrix expressions are also present in BlockMatrix.
501 >>> n, m, l = symbols('n m l')
502 >>> X = MatrixSymbol('X', n, n)
503 >>> Y = MatrixSymbol('Y', m, m)
504 >>> Z = MatrixSymbol('Z', n, m)
505 >>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
506 >>> B
507 Matrix([
508 [X, Z],
509 [0, Y]])
510 >>> B[0, 0]
511 X[0, 0]
512 >>> B.shape
513 (m + n, m + n)

```

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

518 units.

519 **5.1. Vector Algebra.** The `sympy.physics.vector` package provides reference
520 frame, time, and space aware vector and dyadic objects that allow for three dimensional operations such as addition, subtraction, scalar multiplication, inner and outer
521 products, cross products, etc. Both of these objects can be written in very compact
522 notation that make it easy to express the vectors and dyadics in terms of multiple
523 reference frames with arbitrarily defined relative orientations. The vectors are used
524 to specify the positions, velocities, and accelerations of points, orientations, angular
525 velocities, and angular accelerations of reference frames, and force and torques. The
526 dyadics are essentially reference frame aware 3×3 tensors. The vector and dyadic
527 objects can be used for any one-, two-, or three-dimensional vector algebra and they
528 provide a strong framework for building physics and engineering tools.

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

```
536 >>> from sympy import pi
537 >>> from sympy.physics.vector import ReferenceFrame
538 >>> A = ReferenceFrame('A')
539 >>> B = ReferenceFrame('B')
540 >>> C = ReferenceFrame('C')
541 >>> B.orient(A, 'body', (pi, pi / 3, pi / 4), 'zxz')
542 >>> C.orient(B, 'axis', (pi / 2, B.x))
543 >>> v = 1 * A.x + 2 * B.z + 3 * C.y
544 >>> v
545 A.x + 2*B.z + 3*C.y
546 >>> v.express(A)
547 A.x + 5*sqrt(3)/2*A.y + 5/2*A.z
```

548 **5.2. Classical Mechanics.** The `physics.mechanics` package utilizes the `physics.vector`
549 package to populate time aware particle and rigid body objects to fully describe the
550 kinematics and kinetics of a rigid multi-body system. These objects store all of the
551 information needed to derive the ordinary differential or differential algebraic equations
552 that govern the motion of the system, i.e., the equations of motion. These
553 equations of motion abide by Newton's laws of motion and can handle any arbitrary
554 kinematical constraints or complex loads. The package offers two automated methods
555 for formulating the equations of motion based on Lagrangian Dynamics [23] and
556 Kane's Method [22]. Lastly, there are automated linearization routines for constrained
557 dynamical systems based on [30].

558 **5.3. Quantum Mechanics.** The `sympy.physics.quantum` package provides
559 quantum functions, states, operators, and computation of standard quantum models.
560

561 **5.4. Optics.** The `physics.optics` package provides Gaussian optics functions.

562 **5.5. Units.** The `physics.units` module provides around two hundred predefined
563 prefixes and SI units that are commonly used in the sciences. Additionally, it
564 provides the `Unit` class which allows the user to define their own units. These
565 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.

5.6. Tensors. Ongoing work to provide the capabilities of tensor computer algebra has so far produced the `tensor` module. It is composed of three separated submodules, whose purposes are quite different: `tensor.indexed` and `tensor.indexed_methods` support indexed symbols, `tensor.array` contains facilities to operator on symbolic N -dimensional arrays and finally `tensor.tensor` is used to define abstract tensors. The abstract tensors subsection is inspired by `xAct`[25] and `Cadabra`[28]. Canonicalization based on the Butler-Portugal[24] algorithm is supported in SymPy. It is currently limited to polynomial tensor expressions.

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. HORSEN, *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] L. R. U. MANSSUR, R. PORTUGAL, AND B. F. SVAITER, *Group-theoretic approach for symbolic tensor manipulation*, *Int. J. Mod. Phys. C*, 13 (2002), <http://dx.doi.org/http://dx.doi.org/10.1142/S0129183102004571>.
- [25] J. MARTÍN-GARCÍA, *xact, efficient tensor computer algebra*, 2002-2016, <http://metric.iem.csic.es/Martin-Garcia/xAct/>.
- [26] 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.
- [27] A. NIJENHUIS AND H. S. WILF, *Combinatorial Algorithms: For Computers and Calculators*, Academic Press, New York, NY, USA, second ed., 1978.
- [28] K. PEETERS, *Cadabra: a field-theory motivated symbolic computer algebra system*, *Computer Physics Communications*, (2007).
- [29] F. PÉREZ AND B. E. GRANGER, *Ipython: a system for interactive scientific computing*, *Computing in Science & Engineering*, 9 (2007), pp. 21–29.
- [30] 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>.
- [31] M. PETKOVŠEK, H. S. WILF, AND D. ZEILBERGER, *A = bak peters*, Wellesley, MA, (1996).

- [32] M. SOFRONIOU AND G. SPALETTA, *Precise numerical computation*, Journal of Logic and Algebraic Programming, 64 (2005), pp. 113–134.
- [33] H. TAKAHASI AND M. MORI, *Double exponential formulas for numerical integration*, Publications of the Research Institute for Mathematical Sciences, 9 (1974), pp. 721–741.
- [34] 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. Limits: The Gruntz Algorithm. SymPy calculates limits using the Gruntz algorithm, as described in [19]. The basic idea is as follows: any limit can be converted to a limit $\lim_{x \rightarrow \infty} f(x)$ by substitutions like $x \rightarrow \frac{1}{x}$. Then the most varying subexpression ω (that converges to zero as $x \rightarrow \infty$ the fastest from all subexpressions) is identified in $f(x)$, and $f(x)$ is expanded into a series with respect to ω . Any positive powers of ω converge to zero. If there are negative powers of ω , then the limit is infinite. The constant term (independent of ω , but could depend on x) then determines the limit (one might need to recursively apply the Gruntz algorithm on this term to determine the limit).

To determine the most varying subexpression, the comparability classes must first be defined, by calculating L :

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

And then operations $<$, $>$ and \sim are defined as follows: $f > g$ when $L = \pm\infty$ (it is said that f is more rapidly varying than g , i.e., f goes to ∞ 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). Here are some examples of comparability classes:

$$2 < x < e^x < e^{x^2} < e^{e^x}$$

$$2 \sim 3 \sim -5$$

$$x \sim x^2 \sim x^3 \sim \frac{1}{x} \sim x^m \sim -x$$

$$e^x \sim e^{-x} \sim e^{2x} \sim e^{x+e^{-x}}$$

$$f(x) \sim \frac{1}{f(x)}$$

The Gruntz algorithm is now illustrated on the following example:

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

The goal is to calculate $\lim_{x \rightarrow \infty} f(x)$. First the set of most rapidly varying subexpressions is determined, the so called *mrv set*. For (2), the following mrv set $\{e^x, e^{-x}, e^{x+2e^{-x}}\}$ is obtained. These are all subexpressions of (2) and they all belong to the same comparability class. This calculation can be done using SymPy as follows:

```
>>> from sympy.series.gruntz import mrv
>>> mrv(exp(x+2*exp(-x))-exp(x) + 1/x, x)[0].keys()
dict_keys([exp(x + 2*exp(-x)), exp(x), exp(-x)])
```

Next any item ω is taken from mrv that converges to zero for $x \rightarrow \infty$. The item $\omega = e^{-x}$ is obtained. If such a term is not present in the mrv set (i.e., all terms converge to infinity instead of zero), the relation $f(x) \sim \frac{1}{f(x)}$ can be used.

Next step is to rewrite the mrv in terms of ω : $\{\frac{1}{\omega}, \omega, \frac{1}{\omega}e^{2\omega}\}$. Then the original subexpressions are substituted back into $f(x)$ and expanded with respect to ω :

$$(3) \quad f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2)$$

Since ω is from the mrv set, then in the limit $x \rightarrow \infty$ it is $\omega \rightarrow 0$ and so $2\omega + O(\omega^2) \rightarrow 0$ in (3):

$$(4) \quad f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \rightarrow 2 + \frac{1}{x}$$

Since the result $(2 + \frac{1}{x})$ still depends on x , the above procedure is iterated on the result until just a number (independent of x) is obtained, which is the final limit. In the above case the limit is 2, as can be verified by SymPy:

```
>>> limit(exp(x+2*exp(-x))-exp(x) + 1/x, x, oo)
2
```

In general, when $f(x)$ is expanded in terms of ω , it is obtained:

$$(5) \quad f(x) = \underbrace{O\left(\frac{1}{\omega^3}\right)}_{\infty} + \underbrace{\frac{C_{-2}(x)}{\omega^2}}_{\infty} + \underbrace{\frac{C_{-1}(x)}{\omega}}_{\infty} + C_0(x) + \underbrace{C_1(x)\omega}_0 + \underbrace{O(\omega^2)}_0$$

The positive powers of ω are zero. If there are any negative powers of ω , then the result of the limit is infinity, otherwise the limit is equal to $\lim_{x \rightarrow \infty} C_0(x)$. The expression $C_0(x)$ is simpler than $f(x)$ and so the algorithm always converges. A proof of this, as well as further details are given in Gruntz's Ph.D. thesis [19].

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

```

790 And, Or and negate. Several others like Xor, Implies can be constructed with ^, >>
791 respectively. The above are just a shorthand, expressions can also be constructed by
792 directly calling And(), Or(), Not(), Xor(), Nand(), Nor(), etc.
793 >>> from sympy import *
794 >>> x, y, z = symbols('x y z')
795 >>> e = (x & y) | z
796 >>> e.subs({x: True, y: True, z: False})
797 True

```

798 **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.

```

801 >>> from sympy import *
802 >>> x, y, z = symbols('x y z')
803 >>> to_cnf((x & y) | z)
804 And(Or(x, z), Or(y, z))
805 >>> to_dnf(x & (y | z))
806 Or(And(x, y), And(x, z))
807 >>> is_cnf((x | y) & z)
808 True
809 >>> is_dnf((x & y) | z)
810 True

```

811 **9.3.3. Simplification and Equivalence.** The module supports simplification of given boolean expression by making deductions on it. Equivalence of two expressions can also be checked. If so, it is possible to return the mapping of variables of two expressions so as to represent the same logical behaviour.

```

815 >>> from sympy import *
816 >>> a, b, c, x, y, z = symbols('a b c x y z')
817 >>> e = a & (~a | ~b) & (a | c)
818 >>> simplify(e)
819 And(Not(b), a)
820 >>> e1 = a & (b | c)
821 >>> e2 = (x & y) | (x & z)
822 >>> bool_map(e1, e2)
823 (And(Or(b, c), a), {b: y, a: x, c: z})

```

824 **9.3.4. SAT solving.** The module also supports satisfiability checking of a given boolean expression. If satisfiable, it is possible to return a model for which the expression is satisfiable. The API also supports returning all possible models. The SAT solver has a clause learning DPLL algorithm implemented with watch literal scheme and VSIDS heuristic[26].

```

829 >>> from sympy import *
830 >>> a, b, c = symbols('a b c')
831 >>> satisfiable(a & (~a | b) & (~b | c) & ~c)
832 False
833 >>> satisfiable(a & (~a | b) & (~b | c) & c)
834 {b: True, a: True, c: True}

```

835 **9.4. Diophantine Equations.** 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

838 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
 839 say that the equation is solvable.

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

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

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

```

851 >>> from sympy import symbols
852 >>> x, y, z = symbols("x, y, z", integer=True)
853
854 >>> diophantine(2*x + 3*y - 5)
855 set([(3*t_0 - 5, -2*t_0 + 5)])
856
857 >>> diophantine(2*x + 4*y - 3)
858 set()
859
860 >>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
861 set([(2, 1), (5, 1)])
862
863 >>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
864 set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])
865
866 >>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
867 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)])
868
869 >>> from sympy.abc import a, b, c, d, e, f
870 >>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
871 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)
872
873 >>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)
874 set([(8, 4, 4, 4, 0, 0)])

```

875 **9.5. Sets.** SymPy supports representation of a wide variety of mathematical
 876 sets. This is achieved by first defining abstract representations of atomic set classes
 877 and then combining and transforming them using various set operations.

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

882 SymPy has the following atomic set classes:

- 883 • `EmptySet` represents the empty set \emptyset .
- 884 • `UniversalSet` is an abstract “universal set” for which everything is a mem-
 885 ber. The union of the universal set with any set gives the universal set and
 886 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** supports both polar and rectangular representation of regions on the complex plane.

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

SymPy Gamma also has several additional features than just computing the results using SymPy.

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

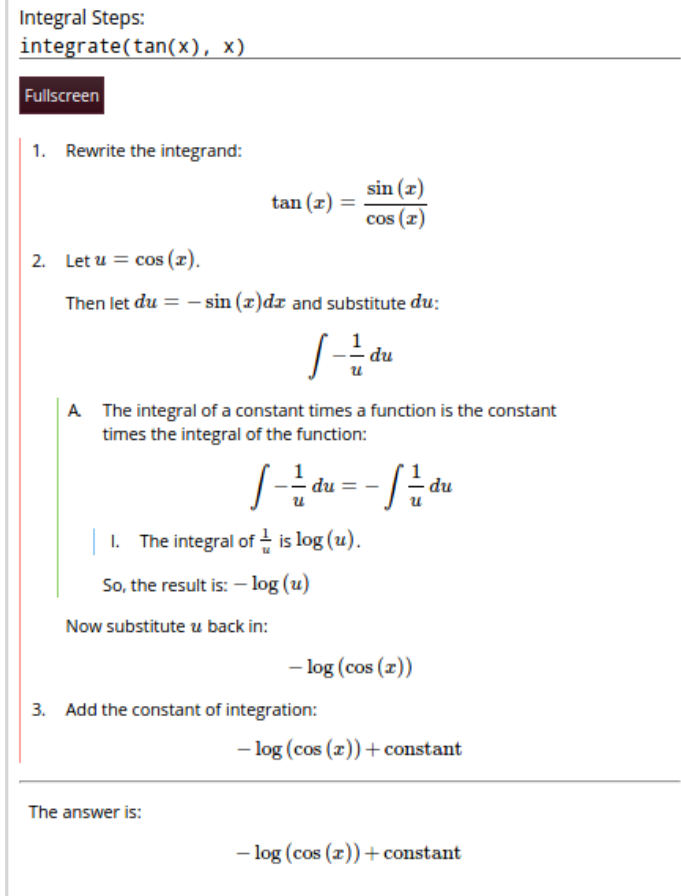


Fig. 1: Integral steps of $\tan(x)$

- 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.

9.7. 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.

9.8. 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]
```

```

999
1000 Out[1]= {Plus}
1001
1002 In[2]:= Sin[x_]^2 + Cos[y_]^2 := 1
1003
1004 In[3]:= x + Sin[t]^2 + y + Cos[t]^2
1005
1006 Out[3]= 1 + x + y
1007 This expression in Mathematica defines a substitution rule that overloads the func-
1008 tionality of the Plus node (the node for additions in Mathematica). The trailing
1009 underscore after a symbol means that it is to be considered a wildcard. This example
1010 may not be practical, one may wish to keep this identity unevaluated, nevertheless
1011 it clearly illustrates the potentiality to define one's own immediate transformation
1012 rules. In SymPy the operations constructing the addition node in the expression tree
1013 are Python class constructors, and cannot be modified at runtime5 The way SymPy
1014 deals with extending the missing runtime overloadability functionality is by subclass-
1015 ing the node types. Subclasses may overload the class constructor to yield the proper
1016 extended functionality.
1017 Unlike SymPy, Mathematica does not support type inheritance or polymorphism[15].
1018 SymPy relies heavily on class inheritance, but for the most part, class inheritance is
1019 used to make sure that SymPy objects inherit the proper methods and implement the
1020 basic hashing system. Associativity of expressions can be achieved by inheriting the
1021 class AssocOp, which may appear a more cumbersome operation than Mathematica's
1022 attribute setting.
1023 Matrices in SymPy are types on their own. In Mathematica, nested lists are
1024 interpreted as matrices whenever the sublists have the same length. The main differ-
1025 ence to SymPy is that ordinary operators and functions do not get generalized the
1026 same way as used in traditional mathematics. Using the standard multiplication in
1027 Mathematica performs an elementwise product, this is compatible with Mathemat-
1028 ica's convention of commutativity of Times nodes. Matrix product is expressed by
1029 the dot operator, or the Dot node. The same is true for the other operators, and
1030 even functions, most notably calling the exponential function Exp on a matrix returns
1031 an elementwise exponentiation of its elements. The real matrix exponentiationl is
1032 available through the MatrixExp function.
1033 Unevaluated expressions can be achieved in various ways, most commonly with
1034 the HoldForm or Hold nodes, that block the evaluation of subnodes by the parser.
1035 Note that such a node cannot be expressed in Python, because of greedy evaluation.
1036 Whenever needed in SymPy, it is necessary to add the parameter evaluate=False to
1037 all subnodes, or put the input expression in a string.
1038 The operator == returns a boolean whenever it is able to immediately evaluate
1039 the truthness of the equality, otherwise it returns an Equal expression. In SymPy ==
1040 means structural equality and is always guaranteed to return a boolean expression.
1041 To express an equality in SymPy it is necessary to explicitly construct the Equality
1042 class.
1043 SymPy, in accordance with Python and unlike the usual programming convention,
1044 uses ** to express the power operator, while Mathematica uses the more common ^.

```

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