SYMPY: SYMBOLIC COMPUTING IN PYTHON

ONDřEJ ČERTÍK[*], ISURU FERNANDO[†], AND ASHUTOSH SABOO[‡]

1. Introduction.

2. Architecture.

2.1. The Core. Every symbolic expression in SymPy 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, take 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.

```
>>> expr.args[0]
2
>>> type(expr.args[0])
<class 'sympy.core.numbers.Integer'>
>>> expr.args[0].args
()
```

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

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

Every SymPy expression satisfies a key invariant, namely, `expr.func(*expr.args) == expr`. 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

[*]Los Alamos National Laboratory (ondrej.certik@gmail.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).

[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.

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.

2.2. 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.

```
>>> 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`.

[2]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`.

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

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.

2.3. 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.

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 [4] 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` are typically complex numbers, but may also include other "rings" like matrix expressions. Not all SymPy classes are `Expr`. For instance, logic expressions, such as `And(x, y)` are `Basic` but not `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*.

Here is 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

[4] Some internal classes, such as those used in the polynomial module, do not follow this rule.

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])
        else:
            raise ArgumentIndexError(self, argindex)
```

The actual gamma function defined in SymPy has much more implemented than this, such as evaluation at rational points and series expansion.

3. Algorithms.

3.1. 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:

```
>>> Float(1.1)
1.10000000000000
>>> Float(1.1, 30)   # precision equivalent to 30 digits
1.10000000000000008881784197001
>>> Float("1.1", 30)
1.10000000000000000000000000000
```

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.

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):

```
>>> cos(exp(-100)).evalf(25) - 1
0
>>> (cos(exp(-100)) - 1).evalf(25)
-6.919482633683687653243407e-88
```

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

SymPy does not track the accuracy of approximate numbers outside of `evalf`. The familiar dangers of floating-point arithmetic apply [14], and symbolic expressions containing floating-point numbers should be treated with some caution. This approach is similar to Maple and Maxima.

By contrast, Mathematica uses a form of significance arithmetic [24] 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 [12]). 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.

3.1.1. Code generation. SymPy's `lambdify` can be used to convert a symbolic expression to a callable Python function for faster numerical evaluation. Various back ends are supported. The following example demonstrates creating a NumPy-based function from a SymPy expression, which automatically supports vectorized array evaluation [27]:

```
>>> f = lambdify((x, y), sin(x*y)**2, modules='numpy')
>>> from numpy import array
>>> f(array([1,2,3]), array([4,5,6]))
array([ 0.57275002,  0.29595897,  0.56398184])
```

SymPy can also generate C, C++, Fortran77, Fortran90 and Octave/Matlab source code, via the `codegen` function. [document this?]

3.1.2. The mpmath library. The implementation of arbitrary-precision floating-point arithmetic is supplied by the mpmath library, which originally was developed as a SymPy module but subsequently has been moved to a standalone Python package. The basic datatypes in mpmath are `mpf` and `mpc`, which respectively act as multi-precision substitutes for Python's `float` and `complex`. The floating-point precision is controlled by a global context:

```
>>> import mpmath
>>> mpmath.mp.dps = 30    # 30 digits of precision
>>> mpmath.mpf("0.1") + mpmath.exp(-50)
mpf('0.100000000000000000000192874984794')
>>> print(_)   # pretty-printed
0.100000000000000000000192874985
```

For pure numerical computing, it is convenient to use mpmath directly with `from mpmath import *` (it is best to avoid such an import statement when using SymPy simultaneously, since numerical functions such as `exp` will shadow the symbolic counterparts in SymPy).

Like SymPy, mpmath is a pure Python library. Internally, mpmath represents a floating-point number $(-1)^s x \cdot 2^y$ by a tuple (s, x, y, b) where x and y are arbitrary-size Python integers and the redundant integer b stores the bit length of x for quick access. If GMPY [17] is installed, mpmath automatically switches to using the `gmpy.mpz` type for x and using GMPY helper methods to perform rounding-related operations, improving performance.

The mpmath library includes support for special functions, root-finding, linear algebra, polynomial approximation, and numerical computation of limits, derivatives, integrals, infinite series, and ODE solutions. All features work in arbitrary precision and use algorithms that support computing hundreds of digits rapidly, except in degenerate cases.

The double exponential (tanh-sinh) quadrature is used for numerical integration by default. For smooth integrands, this algorithm usually converges extremely rapidly, even when the integration interval is infinite or singularities are present at the endpoints [25, 8]. However, for good performance, singularities in the middle of the interval must be specified by the user. To evaluate slowly converging limits and infinite series, mpmath automatically attempts to apply Richardson extrapolation and the Shanks transformation (Euler-Maclaurin summation can also be used) [9]. A function to evaluate oscillatory integrals by means of convergence acceleration is also available.

A wide array of higher mathematical functions are implemented with full support for complex values of all parameters and arguments, including complete and incomplete gamma functions, Bessel functions, orthogonal polynomials, elliptic functions and integrals, zeta and polylogarithm functions, the generalized hypergeometric function, and the Meijer G-function.

Most special functions are implemented as linear combinations of the generalized hypergeometric function ${}_pF_q$, which is computed by a combination of direct summation, argument transformations (for ${}_2F_1$, ${}_3F_2$, $\ldots$) and asymptotic expansions (for ${}_0F_1$, ${}_1F_1$, ${}_1F_2$, ${}_2F_2$, ${}_2F_3$) to cover the whole complex domain. Numerical integration and generic convergence acceleration are also used in a few special cases.

In general, linear combinations and argument transformations give rise to singularities that have to be removed for certain combinations of parameters. A typical example is the modified Bessel function of the second kind

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

where the limiting value $\lim_{\varepsilon\to 0} K_{n+\varepsilon}(z)$ 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 $G^{3,0}_{1,3}\left(0; \frac{1}{2}, -1, -\frac{3}{2} | x\right)$ is a good test case [26]; 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?]

3.1.3. 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 [13] 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)
```

3.2. Polynomials.

3.3. The Risch Algorithm.

3.4. The Gruntz Algorithm. The limit module implements the Gruntz algorithm [15].

Examples:

```
In [1]: limit(sin(x)/x, x, 0)
Out[1]: 1

In [2]: limit((2*E**((1-cos(x))/sin(x))-1)**(sinh(x)/atan(x)**2), x, 0)
Out[2]: E
```

3.4.1. Details. We first define comparability classes by calculating L:

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

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 ∞ 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:

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

$$2 \sim 3 \sim -5$$
$$x \sim x^2 \sim x^3 \sim \frac{1}{x} \sim x^m \sim -x$$
$$e^x \sim e^{-x} \sim e^{2x} \sim e^{x+e^{-x}}$$
$$f(x) \sim \frac{1}{f(x)}$$

The Gruntz algorithm, on an example:

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

$$\lim_{x\to\infty} f(x) = ?$$

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 ω 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 ω: $\{\frac{1}{\omega}, \omega, \frac{1}{\omega}e^{2\omega}\}$, substitute back into $f(x)$ and expand in ω:

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

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

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

We iterate until we get just a number, the final limit. Gruntz proved this algorithm always works and converges in his Ph.D. thesis [15].

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 ω. The limit is one of: 0, $\lim_{x\to\infty} C_0(x)$, ∞.

3.5. Logic.

3.6. Other.

4. Features.

SymPy has an extensive feature set that encompasses too much to cover in-depth here. Bedrock areas, such a 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 [23] 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 [21], 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. Basic Operations.

4.1.1. Expression manipulation.

4.1.2. Assumptions system. SymPy has two assumptions systems, referred to as new-style and old-style assumptions.

In the old-style assumptions system propositions are assigned to symbols upon class construction, for example, to declare the symbol i as positive integer, one would call

```
i = Symbol("i", integer=True, positive=True)
```

querying the assumptions is handled through attributes

```
i.is_positive
i.is_integer
```

These methods return either a boolean, indicating whether the preposition is true or false, or a None, when it is impossible to determine the truth value of the queried preposition.

Despite the fact that assumptions can only be declared on symbols, querying can happen on every expression.

```
In [1]: x,y = symbols('x y', positive=True)

In [2]: (x*y).is_positive
Out[2]: True

In [3]: z = symbols('z')

In [4]: (x*z).is_positive

In [5]: w = symbols('w', positive=False)

In [6]: (x*w).is_positive
Out[6]: False
```

The output 2 is true because SymPy's algorithms can deduce that the product of two positive numbers is positive, while there is no output for input 4, as the symbol z doesn't have any information about its sign, and the product $x \cdot z$ may be positive as well as negative. Finally, output 6 is false as the product of positive and negative numbers is negative.

The new-style assumptions are an assumptions system that exists alongside with the old-style, but is significantly different in the way predicates are used. Predicates in the new-style assumptions system are located under the Q namespace, they appear as `Q.positive`, `Q.integer` and so on.

Querying is provided through the `ask` functions. The previous example in the new-style assumptions can be written as

```
In [1]: ask(Q.positive(x*y), Q.positive(x) & Q.positive(y))
Out[1]: True

In [2]: ask(Q.positive(x*y), Q.positive(x))

In [3]: ask(Q.positive(x*y), Q.positive(x) & Q.negative(y))
Out[3]: False
```

That is, `ask` returns the truth value of its first parameter assuming that its latter argument is true.

Expressions like `Q.positive` are instances of the class `Predicate`, while the same expression with a parameter, such as `Q.positive(x)` is an instance of `AppliedPredicate`.

Logical connectors can be expressed through operator overloading, such as in `Q.positive(x) & Q.positive(y)`, or by directly constructing the identical expression through the logical connector class, in this case `And(Q.positive(x), Q.positive(y))`.

4.1.3. Calculus. Derivations can be computed with the `diff` function, or using the method with the same name on the expressions:

```
In [1]: diff(sin(x), x)
Out[1]: cos(x)

In [2]: sin(x).diff(x)
Out[2]: cos(x)
```

The class `Derivative` is a container for unevaluated derivatives

```
In [3]: expr = Derivative(sin(x), x)

In [4]: expr
Out[4]:
d
--(sin(x))
dx
```

To evaluate such a held expression, simply call the `doit` method:

```
In [5]: expr.doit()
Out[5]: cos(x)
```

Integrals can be analogously calculated either with the `integrate` function or with the method with the same name on expressions:

```
>>> integrate(sin(x), x)
-cos(x)
```

This expression returns an expression whose derivative is the original expression. Notice that integrals are defined up to an integration constant, for the sake of simplicity SymPy will not display the full generic expression.

Definite integration can be calculated with the same method, by specifying a range of the integration variable:

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

To express unevaluated integrals, the class `Integral` may help

```
Integral(sin(x), x)
```

as in the case of derivatives, the method `doit` will cause such an expression to be evaluated.

Limits:

```
In [9]: limit(sin(x)/x, x, 0)
Out[9]: 1
```

for unevaluated expressions, `Limit`.

TODO: Sums and products.

4.1.4. Expression outputs.

4.2. Calculus.

4.3. Sets.

SymPy supports representation of a wide variety of sets, this is achieved by first defining abstract representation for a smaller number of atomic set classes and then combining and transforming them using various set operations.

Each of the set classes inherits from the base set class and defines rules to check membership of a SymPy object in that set, to calculate union, intersection and set difference. In cases we are not able to evaluate these operations to atomic set classes they are represented as abstract unevaluated objects.

We have the following atomic set classes in SymPy.

- `EmptySet`: represents the empty set $\emptyset$.
- `UniversalSet`: Everything is a member of Universal Set. Union of Universal Set with any set gives Universal Set and intersection leads to the other set itself.
- `FiniteSet` is functionally equivalent to python's set object. Its members can be any SymPy object including other sets themselves.
- `Integers` represents set of Integers $\mathbb{Z}$.
- `Naturals` represents set of Natural numbers $\mathbb{N}$ i.e., set of positive integers.
- `Naturals0` represents the whole numbers which are all the non-negative integers, inclusive of zero.
- `Range` represents a range of integers and is defined by specifying a start value, an end value and a step size. Range is functionally equivalent to python's range except the fact that it accepts infinity at end points allowing us to represent infinite ranges.
- `RealInterval` is specified by giving the start and end point and specifying if it is open or closed in the respective ends. The set of real numbers is represented as a special case of a real interval where the start point is negative infinite and the end point is positive infinite.

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

- `ProductSet` abstractly defines the Cartesian product of two or more sets. 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 Image Set of a function F w.r.t a set S is $\{F(x)|x \in S\}$ In particular we use Image Set to represent the set of infinite solutions from trigonometric equations.
- `ConditionSet` represents subset of a set who's members satisfies a particular condition. In notation Condition Set of set S w.r.t to a condition H is $\{x|H(x), x \in S\}$. We use Condition Set to represent the set of solutions of an equation or an inequality where the equation or the inequality is the condition and the set is the domain in which we aim to find the solution.

A few other classes are implemented as special cases of the classes described above. The real number `Reals` is implemented as a special case of real interval where the start point is negative infinity and the end point is positive infinity. `ComplexRegion` is implemented as a special case of `ImageSet`, `ComplexRegion` supports both polar and rectangular representation of region on the complex plane.

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()
```

- Interval of solution

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

- Infinitely many solutions

```
>>> solveset(sin(x) - 1, x, domain=S.Reals)
ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers())
>>> solveset(x - x, x, domain=S.Reals)
(-oo, oo)
>>> solveset(x - x, x, domain=S.Complexes)
S.Complexes
```

- Linear system: finite and infinite solution for determined, under determined and over determined problems.

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), x, y, z)
{(-1,2,0)}
>>> linsolve(Matrix(([1, 1, 1, 1], [1, 1, 2, 3])), (x, y, z))
{(-y - 1, y, 2)}
```

The new solve i.e. **solveset** is under active development and is a planned replacement for **solve**, Hence there are some features which are implemented in solve and is not yet implemented in solveset. The table below show the current state of old and new solve functions.

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, $f(x_1, x_2, \ldots x_n) = 0$ where $n \geq 2$ and $x_1, x_2, \ldots x_n$ are integer variables. If we can find n integers $a_1, a_2, \ldots a_n$ such that $x_1 = a_1, x_2 = a_2, \ldots x_n = a_n$ 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: $a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$
- 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: $a_1x_1^2 + a_2x_2^2 + \cdots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
- General sum of squares: $x_1^2 + x_2^2 + \cdots + x_n^2 = k$

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)])

>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])

>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
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)])

>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)
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,

>>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)
set([(8, 4, 4, 4, 0, 0)])
```

4.5. Matrices. SymPy supports matrices with symbolic expressions as elements. There are two types of matrices, Mutable and Immutable. Mutable classes are the default in SymPy as mutability is important for performance, but it means that standard matrices can not interact well with the rest of SymPy. This is because the Basic object, from which most SymPy classes inherit, is immutable.

Immutable matrix classes inherit from Basic and can thus interact more naturally with the rest of SymPy.

```
In [1]: from sympy import Matrix, symbols, MatrixSymbol

In [2]: x, y = symbols('x y', positive=True)

In [3]: t = Matrix(2, 2, [x, x + y, y, x])

In [4]: t

Out[4]:
Matrix([
[    x, x + y],
[    y,     x]])

In [5]: t[0, 1] = y

In [6]: t
Out[6]:
Matrix([
[x, y],
[y, x]])
```

All SymPy matrix types can do linear algebra including matrix addition, multiplication, exponentiation, computing determinant, solving linear systems and computing inverses using LU decomposition, LDL decomposition, Gauss-Jordan elimination, Cholesky decomposition, Moore-Penrose pseudoinverse, adjugate matrix.

Eigenvalues are computed symbolically as well. Eigenvalues are computed by generating the characteristic polynomial using the Berkowitz algorithm and then solving it using polynomial routines. Diagonalizable matrices can be diagonalized first to compute the eigenvalues.

```
In [10]: t.eigenvals()
Out[10]: {x - y: 1, x + y: 1}
```

Internally these matrices store the elements as a list making it a dense representation. For storing sparse matrices, SparseMatrix and ImmutableSparseMatrix classes can be used. Sparse matrix classes store the elements in Dictionary of Keys (DoK) format.

SymPy also supports matrices with unknown dimension values. MatrixSymbol represents a matrix with dimensions `m, n` where `m` and `n` can be symbols or integers. Matrix addition and multiplication, scalar operations, matrix inverse and transpose are stored symbolically as matrix expressions. Mutable matrices are converted to corresponding immutable types before interacting with matrix expressions

```
In [11]: m, n, p = symbols("m, n, p", integer=True)

In [12]: r, s = MatrixSymbol("r", m, n), MatrixSymbol("s", n, p)

In [13]: u = r * s + 2*MatrixSymbol("t", m, p)

In [14]: u.shape
Out[14]: (m, p)

In [15]: u[0, 1]
Out[15]: 2*t[0, 1] + Sum(r[0, _k]*s[_k, 1], (_k, 0, n - 1))
```

Block matrices are also supported in SymPy. BlockMatrix elements can be any matrix expression which includes immutable matrices, matrix symbols and block matrices. All functionalities of matrix expressions are also present in BlockMatrix.

```
>>> from sympy import (MatrixSymbol, BlockMatrix, symbols,
...     Identity, ZeroMatrix, block_collapse)
>>> n, m, l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m,n), Y]])
>>> print(B)
Matrix([
[X, Z],
[0, Y]])
>>> print(B[0, 0])
X[0, 0]
```

4.6. Physics.

4.7. Series.

4.7.1. 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 nth 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
```

4.7.2. 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[16]. 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
```

4.7.3. Fourier Series. SymPy provides functionality to compute Fourier Series of a function using the `fourier_series` function. Under the hood it just computes $a0$, an, bn 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)
```

4.8. 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.

4.8.1. 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
```

4.8.2. 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
```

4.8.3. 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.

```
>>> from sympy import *
>>> a, b, c, x, y, z = symbols('a b c x y z')
>>> e = a & (~a | ~b) & (a | c)
>>> simplify(e)
And(Not(b), a)
>>> e1 = a & (b | c)
>>> e2 = (x & y) | (x & z)
>>> bool_map(e1, e2)
(And(Or(b, c), a), {b: y, a: x, c: z})
```

4.8.4. 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[20].

```
>>> from sympy import *
>>> a, b, c = symbols('a b c')
>>> satisfiable(a & (~a | b) & (~b | c) & ~c)
False
>>> satisfiable(a & (~a | b) & (~b | c) & c)
{b: True, a: True, c: True}
```

SymPy includes several packages that allow users to solve domain specific problems. For example, a comprehensive physics package is included that is useful for solving problems in classical mechanics, optics, and quantum mechanics along with support for manipuating physical quantities with units.

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

Listing 1 Python interpreter session showing how a vector is created using the orthogonal unit vectors of three reference frames that are oriented with respect to each other and the result of expressing the vector in the A frame. The B frame is oriented with respect to the A frame using Z-X-Z Euler Angles of magnitude π, $\frac{\pi}{2}$, and $\frac{\pi}{3}$rad, respectively whereas the C frame is oriented with respect to the B frame through a simple rotation about the B frame's X unit vector through $\frac{\pi}{2}$rad.

```
>>> from sympy import pi
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> C = ReferenceFrame('C')
>>> B.orient(A, 'body', (pi, pi / 3, pi / 4), 'zxz')
>>> C.orient(B, 'axis', (pi / 2, B.x))
>>> v = 1 * A.x + 2 * B.z + 3 * C.y
>>> v
A.x + 2*B.z + 3*C.y
>>> v.express(A)
A.x + 5*sqrt(3)/2*A.y + 5/2*A.z
```

4.10. Classical Mechanics. The `physics.mechanics` package utilizes the `physics.vector` 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 [19] and Kane's Method [18]. Lastly, there are automated linearization routines for constrained dynamical systems based on [22].

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

4.12. Optics. The `physics.optics` package provides Gaussian optics functions.

4.13. 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.

5. 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.
- **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.

5.1. 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:

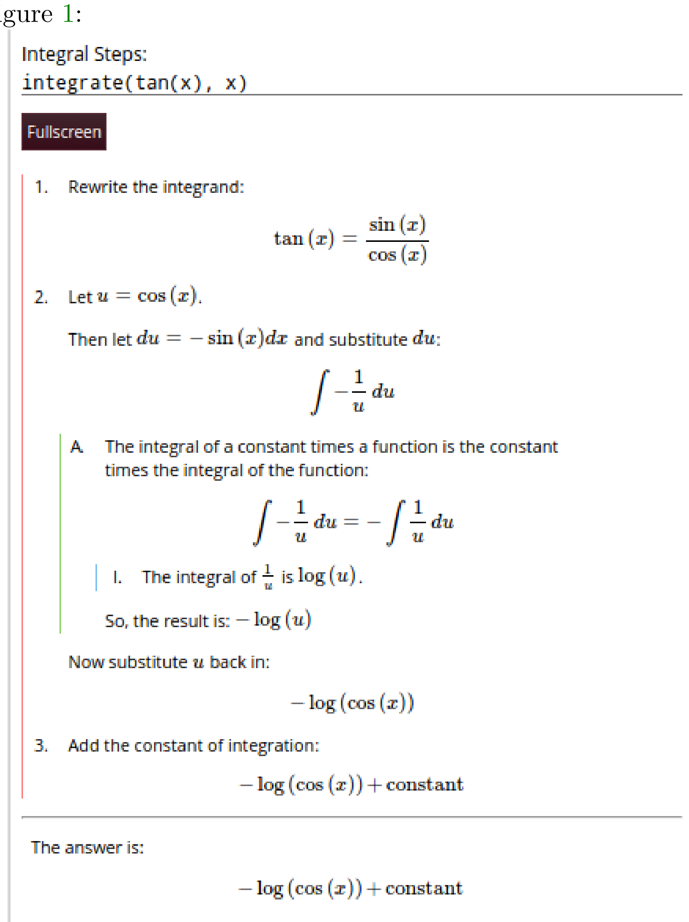

Fig. 1: Integral steps of $\tan(x)$

- 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.

5.2. 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.

6. Comparison with other CAS.

6.1. 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 (know 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 the 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`[2] 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]

Out[1]= {Plus}

In[2]:= Sin[x_]^2 + Cos[y_]^2 := 1

In[3]:= x + Sin[t]^2 + y + Cos[t]^2

Out[3]= 1 + x + y
```

This expression in Mathematica defines a substitution rule that overloads the functionality of the `Plus` node (the node for additions in Mathematica). The trailing underscore after a symbol means that it is to be considered a wildcard. This example may not be practical, one may wish to keep this identity unevaluated, nevertheless it clearly illustrates the potentiality to define one's own immediate transformation rules. In SymPy the operations constructing the addition node in the expression tree are Python class constructors, and cannot be modified at runtime[5] The way SymPy deals with extending the missing runtime overloadability functionality is by subclassing the node types. Subclasses may overload the class constructor to yield the proper extended functionality.

Unlike SymPy, Mathematica does not support type inheritance or polymorphism[12]. SymPy relies heavily on class inheritance, but for the most part, class inheritance is used to make sure that SymPy objects inherit the proper methods and implement the basic hashing system. Associativity of expressions can be achieved by inheriting the class `AssocOp`, which may appear a more cumbersome operation than Mathematica's attribute setting.

Matrices in SymPy are types on their own. In Mathematica, nested lists are interpreted as matrices whenever the sublists have the same length. The main difference to SymPy is that ordinary operators and functions do not get generalized the same way as used in traditional mathematics. Using the standard multiplication in Mathematica performs an elementwise product, this is compatible with Mathematica's convention of commutativity of `Times` nodes. Matrix product is expressed by the *dot* operator, or the `Dot` node. The same is true for the other operators, and even functions, most notably calling the exponential function `Exp` on a matrix returns an elementwise exponentiation of its elements. The real matrix exponentiationl is

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

available through the MatrixExp function.

Unevaluated expressions can be achieved in various ways, most commonly with the HoldForm or Hold nodes, that block the evaluation of subnodes by the parser. Note that such a node cannot be expressed in Python, because of greedy evaluation. Whenever needed in SymPy, it is necessary to add the parameter evaluate=False to all subnodes, or put the input expression in a string.

The operator == returns a boolean whenever it is able to immediately evaluate the truthness of the equality, otherwise it returns an Equal expression. In SymPy == means structural equality and is always guaranteed to return a boolean expression. To express an equality in SymPy it is necessary to explicitly construct the Equality class.

SymPy, in accordance with Python and unlike the usual programming convention, uses ** to express the power operator, while Mathematica uses the more common ^.

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] 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.

[13] 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.

[14] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), 23 (1991), pp. 5–48.

[15] D. Gruntz, *On Computing Limits in a Symbolic Manipulation System*, PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.

[16] D. Gruntz and W. Koepf, *Formal power series*, (1993).

[17] C. V. Horsen, *GMPY*. https://pypi.python.org/pypi/gmpy2, 2015.

[18] T. R. Kane and D. A. Levinson, *Dynamics, Theory and Applications*, McGraw Hill, 1985.

[19] J. Lagrange, *Mécanique analytique*, no. v. 1 in Mécanique analytique, Ve Courcier, 1811.

[20] 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.

[21] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms: For Computers and Calculators*, Academic Press, New York, NY, USA, second ed., 1978.

[22] 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.

[23] M. Petkovšek, H. S. Wilf, and D. Zeilberger, *A= bak peters*, Wellesley, MA, (1996).

[24] M. Sofroniou and G. Spaletta, *Precise numerical computation*, Journal of Logic and Alge-

braic Programming, 64 (2005), pp. 113–134.

[25] H. Takahasi and M. Mori, *Double exponential formulas for numerical integration*, Publications of the Research Institute for Mathematical Sciences, 9 (1974), pp. 721–741.

[26] 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.

[27] S. Van Der Walt, S. C. Colbert, and G. Varoquaux, *The NumPy array: a structure for efficient numerical computation*, Computing in Science & Engineering, 13 (2011), pp. 22–30.