

SYMPY: SYMBOLIC COMPUTING IN PYTHON

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

1. Introduction. SymPy is a full featured computer algebra system (CAS) written in the Python programming language. It is open source, being licensed under the extremely permissive 3-clause BSD license. SymPy was started by Ondřej Čertík in 2005, and it has since grown into a large open source project, with over 500 contributors. SymPy is developed on GitHub using a bazaar community model [?]. The accessibility of the codebase and the open community model allows SymPy to rapidly respond to the needs of the community of users, and has made the large contributor count possible.

SymPy is written entirely in the Python programming language. Python is a popular dynamically typed programming language that has a focus on ease of use and readability. It also a very popular language for scientific computing and data science, with a wide range of useful libraries [?]. SymPy is itself used by many libraries and tools across many domains, such as Sage [?] (pure mathematics), yt [?] (astronomy and astrophysics), PyDy (multibody dynamics), and SfePy [?] (finite elements).

Unlike many CASs, SymPy does not invent its own programming language. Python is used both for the internal implementation and the user interaction. Exclusively using Python in this way makes it easier for people already familiar with the language to use or develop SymPy. It also lets the SymPy developers focus on mathematics, rather than language design.

SymPy is designed with a strong focus that it be usable as a library. This means that extensibility is important in its application program interface (API) design. This is also one of the reasons SymPy makes no attempt to extend the Python language itself. The goal is for users of SymPy to be able to import SymPy alongside other Python libraries in their workflow, whether that is an interactive workflow or programmatic use as part of a larger system.

Being developed as a library, SymPy does not have a built-in graphical user interface (GUI). However, SymPy exposes a rich interactive display system, including registering printers with Jupyter [?] frontends, including the Notebook and Qt Console, which will pretty print SymPy expressions using MathJax or L^AT_EX rendering.

Section ?? discusses the architecture of SymPy. Following that, Section ?? looks at the numerical features of SymPy and its dependency library, mpmath. Section ?? enumerates the features of SymPy and takes a closer look at some of the important ones. Section ?? looks at the domain specific physics submodules for doing classical mechanics and quantum mechanics. Finally, Section ?? concludes the paper and discusses future work.

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 *
```

*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).

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

The symbolic nature of SymPy comes from its implementation of symbolic variables, called symbols, which must be defined and 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 representing variables 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 `**`, as:

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

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

All SymPy expressions are immutable. This simplifies the design by allowing interning. It also allows expressions to be hashed and stored in a Python dictionary, which enables caching and other features.

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
```

By order of operations, the parent of the expression tree for `expr` is an addition, so it is of type `Add`. The child nodes of `expr` are 2 and $x*y$.

```
>>> 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` returns a string representation of the object as valid Python code, which contains all the nested class constructor calls to create the given expression.

```

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

```

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

98 Python allows classes to override mathematical operators. The Python interpreter
 99 translates the above `x*y + 2` to, roughly, `(x.__mul__(y)).__add__(2)`. Both `x` and `y`,
 100 returned from the `symbols` function, are `Symbol` instances. The `2` in the expression is
 101 processed by Python as a literal, and is stored as Python's builtin `int` type. When
 102 `2` is called by the `__add__` method of `Symbol`, it is converted to the SymPy type
 103 `Integer(2)`. In this way, SymPy expressions can be built in the natural way using
 104 Python operators and numeric literals.

105 One must be careful in one particular instance. Python does not have a builtin
 106 rational literal type. Given a fraction of integers such as `1/2`, Python will perform
 107 floating point division and produce `0.5` ². Python uses eager evaluation, so expres-
 108 sions like `x + 1/2` will produce `x + 0.5`, and by the time any SymPy function sees the
 109 `1/2` it has already been converted to `0.5` by Python. However, for a CAS like SymPy,
 110 one typically wants to work with exact rational numbers whenever possible. Work-
 111 ing around this is simple, however: one can wrap one of the integers with `Integer`,
 112 like `x + Integer(1)/2`, or using `x + Rational(1, 2)`. SymPy provides a function `S`
 113 which can be used to convert objects to SymPy types with minimal typing, such as
 114 `x + S(1)/2`. This gotcha is a small downside to using Python directly instead of a
 115 custom domain specific language (DSL), and we consider it to be worth it for the
 116 advantages listed above.

117 **2.3. Assumptions.** An important feature of the SymPy core is the assump-
 118 tions system. The assumptions system allows users to specify that symbols have
 119 certain common mathematical properties, such as being positive, imaginary, or inte-
 120 ger. SymPy is careful to never perform simplifications on an expression unless the
 121 assumptions allow them. For instance, the identity $\sqrt{x^2} = x$ holds if x is nonnegative
 122 ($x \geq 0$). If x is real, the identity $\sqrt{x^2} = |x|$ holds. However, for general complex x ,
 123 no such identity holds.

124 By default, SymPy performs all calculations assuming that variables are com-
 125 plex valued. This assumption makes it easier to treat mathematical problems in full
 126 generality.

```

127 >>> x = Symbol('x')
128 >>> sqrt(x**2)
129 sqrt(x**2)

```

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

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

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`.

```

136 >>> x = Symbol('x', positive=True)
137 >>> sqrt(x**2)
138 x
139 Some common assumptions that SymPy allows are positive, negative, real, nonpositive,
140 nonnegative, real, integer, and commutative 3. Assumptions on any object can be
141 checked with the is_assumption attributes, like x.is_positive.
142     Assumptions are only needed to restrict a domain so that certain simplifications
143     can be performed. It is not required to make the domain match the input of a function.
144     For instance, one can create the object  $\sum_{n=0}^m f(n)$  as Sum(f(n), (n, 0, m)) without
145     setting integer=True when creating the Symbol object n.
146     The assumptions system additionally has deductive capabilities. The assump-
147     tions use a three-valued logic using the Python builtin objects True, False, and
148     None. None represents the “unknown” case. This could mean that the given as-
149     sumption could be either true or false under the given information, for instance,
150     Symbol('x', real=True).is_positive will give None because a real symbol might be
151     positive or it might not. It could also mean not enough is implemented to compute
152     the given fact, for instance, (pi + E).is_irrational gives None, because SymPy does
153     not know how to determine if  $\pi + e$  is rational or irrational, indeed, it is an open
154     problem in mathematics.
155     Basic implications between the facts are used to deduce assumptions. For in-
156     stance, the assumptions system knows that being an integer implies being ratio-
157     nal, so Symbol('x', integer=True).is_rational returns True. Furthermore, expres-
158     sions compute the assumptions on themselves based on the assumptions of their
159     arguments. For instance, if x and y are both created with positive=True, then
160     (x + y).is_positive will be True.
161     SymPy also has an experimental assumptions system where facts are stored sep-
162     arate from objects, and deductions are made with a SAT solver. We will not discuss
163     this system here.

```

164 **2.4. Extensibility.** Extensibility is an important feature for SymPy. Because
165 the same language, Python, is used both for the internal implementation and the
166 external usage by users, all the extensibility capabilities available to users are also
167 used by functions that are part of SymPy.

168 The typical way to create a custom SymPy object is to subclass an existing SymPy
169 class, generally either `Basic`, `Expr`, or `Function`. All SymPy classes used for expression
170 trees ⁴ should be subclasses of the base class `Basic`, which defines some basic methods
171 for symbolic expression trees. `Expr` is the subclass for mathematical expressions that
172 can be added and multiplied together. Instances of `Expr` typically represent complex
173 numbers, but may also include other “rings” like matrix expressions. Not all SymPy
174 classes are subclasses of `Expr`. For instance, logic expressions, such as `And(x, y)` are
175 subclasses of `Basic` but not of `Expr`.

176 The `Function` class is a subclass of `Expr` which makes it easier to define mathe-
177 matical functions called with arguments. This includes named functions like $\sin(x)$
178 and $\log(x)$ as well as undefined functions like $f(x)$. Subclasses of `Function` should
179 define a class method `eval`, which returns values for which the function should be
180 automatically evaluated, and `None` for arguments that should not be automatically

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

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

181 evaluated.

182 Many SymPy functions require various evaluations down the expression tree. The
183 evaluation of such functions on of classes in SymPy is performed by defining a relevant
184 `_eval_*` method on the class. For instance, an object can signal to SymPy's `diff` func-
185 tion how to take the derivative of itself by defining the `_eval_derivative(self, x)`
186 method, which may in turn call `diff` on its `args`. The most common `_eval_*` methods
187 relate to the assumptions. `_eval_is_assumption` defines the assumptions for *assump-*
188 *tion*.

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

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

```
198
199 class gamma(Function)
200     @classmethod
201     def eval(cls, arg):
202         if isinstance(arg, Integer) and arg.is_positive:
203             return factorial(arg - 1)
204
205     def _eval_is_real(self):
206         x = self.args[0]
207         # noninteger means real and not integer
208         if x.is_positive or x.is_noninteger:
209             return True
210
211     def _eval_is_positive(self):
212         x = self.args[0]
213         if x.is_positive:
214             return True
215         elif x.is_noninteger:
216             return floor(x).is_even
217
218     def _eval_rewrite_as_factorial(self, z):
219         return factorial(z - 1)
220
221     def fdiff(self, argindex=1):
222         from sympy.core.function import ArgumentIndexError
223         if argindex == 1:
224             return self.func(self.args[0])*polygamma(0, self.args[0])
225         else:
226             raise ArgumentIndexError(self, argindex)
```

227 The actual gamma function defined in SymPy has many more capabilities, such as
228 evaluation at rational points and series expansion.

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 [?] 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 [?, ?]. 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) [?]. 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 ${}_2F_1$, ${}_3F_2$, ...) 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 $\lim_{\epsilon \rightarrow 0} K_{n+\epsilon}(z)$ 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 [?]; 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 (see Section ??). 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.

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 [?] 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 subsections below. Table ?? gives a compact listing of all major capabilities present in the SymPy codebase. This gives a sampling from the breadth of topics and application domains that SymPy services. Unless stated otherwise, all features noted in Table ?? are symbolic in nature. Numeric features are discussed in Section ??.

Table 1: SymPy Features and Descriptions

Feature	Description
Calculus	Algorithms for computing derivatives, integrals, and limits.
Category Theory	Representation of objects, morphisms, and diagrams. Tools for drawing diagrams with Xy-pic.
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.

Combinatorics Group Theory	&	Implements permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [?], and Prufer sequences [?].
Concrete Math		Summation, products, tools for determining whether summation and product expressions are convergent, absolutely convergent, hypergeometric, and other properties. May also compute Gosper's normal form [?] for two univariate polynomials.
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
Differential Geome- try Geometry		Classes to represent manifolds, metrics, tensor products, and coordinate systems. Allows the creation of 2D geometrical entities, such as lines and circles. Enables queries on these entities, such as asking the area of an ellipse, checking for collinearity of a set of points, or finding the intersection between two lines.
Lie Algebras Logic		Represents Lie algebras and root systems. boolean expression, equivalence testing, satisfiability, normal forms.
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).
Matrix Expressions		Matrices with symbolic dimensions (unspecified entries). Block matrices.
Number Theory		prime number generation, primality testing, integer factorization, continued fractions, Egyptian fractions, modular arithmetic, quadratic residues, partitions, binomial and multinomial coefficients, prime number tools, integer factorization.
Plotting		Hooks for visualizing expressions via matplotlib [?] or as text drawings when lacking a graphical back-end. 2D function plotting, 3D function plotting, and 2D implicit function plotting are supported.
Polynomials		Computes polynomial algebras over various coefficient domains. Functionality ranges from the simple (e.g., polynomial division) to the advanced (e.g., Gröbner bases [?] and multivariate factorization over algebraic number domains).
Printing		Functions for printing SymPy expressions in the terminal with ASCII or Unicode characters, and converting SymPy expressions to \LaTeX and MathML.
Series		Implements series expansion, sequences, and limit of sequences. This includes special series, such as Fourier and formal power series.

Sets	Representations of empty, finite, and infinite sets. This includes special sets such as for all natural, integer, and complex numbers. Operations on sets such as union, intersection, Cartesian product, and building sets from other sets.
Simplification	Functions for manipulating and simplifying expressions. Includes algorithms for simplifying hypergeometric functions, trigonometric expressions, rational functions, combinatorial functions, square root denesting, and common subexpression elimination.
Solvers	Functions for symbolically solving equations algebraically, systems of equations, both linear and non-linear, inequalities, ordinary differential equations, partial differential equations, Diophantine equations, and recurrence relations.
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.
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.
Tensors	Symbolic manipulation of indexed objects.
Vectors	Provides basic vector math and differential calculus with respect to 3D Cartesian coordinate systems.

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 [?]. 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 ?? 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 [?]

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 [?], table lookups, a reimplement of Manuel Bronstein’s “Poor Man’s Integrator” [?], 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 [?] 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.

```

414 >>> phi0 = Symbol('phi0')
415 >>> str(Integral(sqrt(phi0), phi0))
416 Integral(sqrt(phi0 + 1), x)
417     Expressions can be printed with 2D monospace text with pprint. This uses
418     Unicode characters to render mathematical symbols such as integral signs, square
419     roots, and parentheses. Greek letters and subscripts in symbol names are rendered
420     automatically.
421 >>> pprint(Integral(sqrt(phi0 + 1), phi0))
422 
$$\int \sqrt{\phi_0 + 1} \, d(\phi_0)$$

423 Alternately, the use_unicode=False flag can be set, which causes the expression to be
424 printed using only ASCII characters.
425 >>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
426 /
427 |
428 | _____
429 | \ / phi0 + 1  d(phi0)
430 |
431 /
432 The function latex returns a LATEX representation of an expression.
433 >>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
434 \int \sqrt{\phi_0 + 1}\, d\phi_0
435 Users are encouraged to run the init_printing function at the beginning of in-
436 teractive sessions, which automatically enables the best pretty printing supported by
437 their environment. In the Jupyter notebook or qtconsole [?] the LATEX printer is used
438 to render expressions using MathJax or LATEX if it is installed on the system. The 2D
439 text representation is used otherwise.
440 Other printers such as MathML are also available. SymPy uses an extensible
441 printer subsystem which allows users to customize the printing for any given printer,
442 and for custom objects to define their printing behavior for any printer. SymPy's
443 code generation capabilities, which we will not discuss in-depth here, use the same
444 printer model.
445
446 4.4. Solvers. SymPy has a module of equation solvers for symbolic equations.
447 There are two submodules to solve algebraic equations in SymPy, referred to as old
448 solve function, solve, and new solve function, solveset. Solveset is introduced with
449 several design changes with respect to the old solve function to resolve the issues
450 with old solve function, for example old solve function's input API has many flags
451 which are not needed and they make it hard for the user and the developers to work
452 on solvers. In contrast to the old solve function, the solveset has a clean input API,
453 it only asks for the necessary information from the user. The function signatures of
454 the old and new solve function:
455 solve(f, *symbols, **flags) # old solve function
456 solveset(f, symbol, domain) # new solve function
457 The old solve function has an inconsistent output API for various types of inputs,
458 whereas the solveset has a canonical output API which is achieved using sets. It can
459 consistently return various types of solutions.
460 • Single solution
461 >>> solveset(x - 1)

```

```

460 >>> {1}
461     • Finite set of solution, quadratic equation
462 >>> solveset(x**2 - pi**2, x)
463 {-pi, pi}
464     • No Solution
465 >>> solveset(1, x)
466 EmptySet()
467     • Interval of solution
468 >>> solveset(x**2 - 3 > 0, x, domain=S.Reals)
469 (-oo, -sqrt(3)) U (sqrt(3), oo)
470     • Infinitely many solutions
471 >>> solveset(sin(x) - 1, x, domain=S.Reals)
472 ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers())
473 >>> solveset(x - x, x, domain=S.Reals)
474 (-oo, oo)
475 >>> solveset(x - x, x, domain=S.Complexes)
476 S.Complexes
477     • Linear system: finite and infinite solution for determined, under determined
478       and over determined problems.
479 >>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
480 >>> b = Matrix([3, 6, 9])
481 >>> linsolve((A, b), x, y, z)
482 {(-1,2,0)}
483 >>> linsolve(Matrix(([1, 1, 1, 1], [1, 1, 2, 3])), (x, y, z))
484 {(-y - 1, y, 2)}
485 The new solve i.e. solveset is under active development and is a planned replace-
486 ment for solve, Hence there are some features which are implemented in solve and is
487 not yet implemented in solveset. The table below show the current state of old and
488 new solve functions.
489

```

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

```

492
493 Below are some of the examples of old solve function:
494     • Non Linear (multivariate) System of Equation: Intersection of a circle and a
495       parabola.
496 >>> solve([x**2 + y**2 - 16, 4*x - y**2 + 6], x, y)
497 [(-2 + sqrt(14), -sqrt(-2 + 4*sqrt(14))),
498  (-2 + sqrt(14), sqrt(-2 + 4*sqrt(14))),
499  (-sqrt(14) - 2, -I*sqrt(2 + 4*sqrt(14))),
500  (-sqrt(14) - 2, I*sqrt(2 + 4*sqrt(14)))]
501     • Transcendental Equation
502 >>> solve(x + log(x)**2 - 5*(x + log(x)) + 6, x)

```

```

503 [LambertW(exp(2)), LambertW(exp(3))]
504 >>> solve(x**3 + exp(x))
505 [-3*LambertW((-1)**(2/3)/3)]

```

506 **4.5. Matrices.** SymPy supports matrices with symbolic expressions as elements.■

```

507 >>> x, y = symbols('x y')
508 >>> A = Matrix(2, 2, [x, x + y, y, x])
509 >>> A
510 Matrix([
511 [ x, x + y],
512 [ y, x]])

```

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

517 All operations are computed symbolically. Eigenvalues are computed by gener-
518 ating the characteristic polynomial using the Berkowitz algorithm and then solving
519 it using polynomial routines. Diagonalizable matrices can be diagonalized first to
520 compute the eigenvalues.

```

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

```

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

526 SymPy also supports matrices with symbolic dimension values. `MatrixSymbol`
527 represents a matrix with dimensions $m \times n$, where m and n can be symbolic. Ma-
528 trix addition and multiplication, scalar operations, matrix inverse, and transpose are
529 stored symbolically as matrix expressions.

```

530 >>> m, n, p = symbols("m, n, p", integer=True)
531 >>> R = MatrixSymbol("R", m, n)
532 >>> S = MatrixSymbol("S", n, p)
533 >>> T = MatrixSymbol("t", m, p)
534 >>> U = R*S + 2*T
535 >>> u.shape
536 (m, p)
537 >>> U[0, 1]
538 2*T[0, 1] + Sum(R[0, _k]*S[_k, 1], (_k, 0, n - 1))

```

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

```

542 >>> n, m, l = symbols('n m l')
543 >>> X = MatrixSymbol('X', n, n)
544 >>> Y = MatrixSymbol('Y', m, m)
545 >>> Z = MatrixSymbol('Z', n, m)
546 >>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
547 >>> B
548 Matrix([
549 [X, Z],
550 [0, Y]])
551 >>> B[0, 0]

```

```

552 X[0, 0]
553 >>> B.shape
554 (m + n, m + n)

```

555 **5. Domain Specific Submodules.** SymPy includes several packages that al-
556 low users to solve domain specific problems. For example, a comprehensive physics
557 package is included that is useful for solving problems in classical mechanics, optics,
558 and quantum mechanics along with support for manipulating physical quantities with
559 units.

560 **5.1. Classical Mechanics.**

561 **5.1.1. Vector Algebra.** The `sympy.physics.vector` package provides reference
562 frame, time, and space aware vector and dyadic objects that allow for three dimen-
563 sional operations such as addition, subtraction, scalar multiplication, inner and outer
564 products, cross products, etc. Both of these objects can be written in very compact
565 notation that make it easy to express the vectors and dyadics in terms of multiple
566 reference frames with arbitrarily defined relative orientations. The vectors are used
567 to specify the positions, velocities, and accelerations of points, orientations, angular
568 velocities, and angular accelerations of reference frames, and force and torques. The
569 dyadics are essentially reference frame aware 3×3 tensors. The vector and dyadic
570 objects can be used for any one-, two-, or three-dimensional vector algebra and they
571 provide a strong framework for building physics and engineering tools.

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

```

578 >>> from sympy import pi
579 >>> from sympy.physics.vector import ReferenceFrame
580 >>> A = ReferenceFrame('A')
581 >>> B = ReferenceFrame('B')
582 >>> C = ReferenceFrame('C')
583 >>> B.orient(A, 'body', (pi, pi / 3, pi / 4), 'zxz')
584 >>> C.orient(B, 'axis', (pi / 2, B.x))
585 >>> v = 1 * A.x + 2 * B.z + 3 * C.y
586 >>> v
587 A.x + 2*B.z + 3*C.y
588 >>> v.express(A)
589 A.x + 5*sqrt(3)/2*A.y + 5/2*A.z

```

590 **5.1.2. Mechanics.** The `sympy.physics.mechanics` package utilizes the `sympy.`
591 `physics.vector` package to populate time aware particle and rigid body objects to
592 fully describe the kinematics and kinetics of a rigid multi-body system. These objects
593 store all of the information needed to derive the ordinary differential or differential al-
594 gebraic equations that govern the motion of the system, i.e., the equations of motion.
595 These equations of motion abide by Newton's laws of motion and can handle any ar-
596 bitrary kinematical constraints or complex loads. The package offers two automated
597 methods for formulating the equations of motion based on Lagrangian Dynamics [?]
598 and Kane's Method [?]. Lastly, there are automated linearization routines for con-
599 strained dynamical systems based on [?].

5.2. Symbolic Quantum Mechanics. The `sympy.physics.quantum` package has extensive capabilities for symbolic quantum mechanics, with Python objects to represent the different mathematical objects relevant in quantum theory [?]: states (bras and kets), operators (unitary, hermitian, etc.) and basis sets as well as operations on these objects such as representations, tensor products, inner products, outer products, commutators, anticommutators, etc. The base objects are designed in the most general way possible to enable any particular quantum system to be implemented by subclassing the base operators to provide system specific logic.

For example, you can define symbolic quantum operators and states and perform a full range of operations with them:

```
>>> from sympy.physics.quantum import Commutator, Dagger, Operator
>>> from sympy.physics.quantum import Ket, qapply
>>> A = Operator('A')
>>> B = Operator('B')
>>> C = Operator('C')
>>> D = Operator('D')
>>> a = Ket('a')
>>> comm = Commutator(A, B)
>>> comm
[A,B]
>>> qapply(Dagger(comm*a)).doit()
-<a|*(Dagger(A)*Dagger(B) - Dagger(B)*Dagger(A))
Commutators can be expanded using common commutator identities:
>>> Commutator(C+B, A*D).expand(commutator=True)
-[A,B]*D - [A,C]*D + A*[B,D] + A*[C,D]
```

On top of this set of base objects, a number of specific quantum systems have been implemented. These include:

- Position/momentum operators and states, raising/lowering operators and states, simple harmonic oscillator, density matrices, hydrogen atom.
- Second quantized formalism of non-relativistic many-body quantum mechanics [?].
- Quantum angular momentum [?]. Spin operators and their eigenstates can be represented in any basis and for any quantum numbers. Facilities for Clebsch-Gordan Coefficients, Wigner Coefficients, rotations, and angular momentum coupling are also present in their symbolic and numerical forms.
- Quantum information and computing [?]. Multidimensional qubit states, and a full set of one- and two-qubit gates are provided and can be represented symbolically or as matrices/vectors. With these building blocks it is possible to implement a number of basic quantum algorithms including the quantum Fourier transform, quantum error correction, quantum teleportation, Grover's algorithm, dense coding, etc.

Here are a few short examples of the quantum information and computing capabilities in `sympy.physics.quantum`. We start with a simple 4 qubit state and flip one of the qubits:

```
>>> from sympy.physics.quantum.qubit import Qubit
>>> q = Qubit('0101')
>>> q
|0101>
>>> q.flip(1)
|0111>
```



```

650 Qubit states can also be used in adjoint operations, tensor products, inner/outer
651 products:
652 >>> Dagger(q)
653 <0101|
654 >>> ip = Dagger(q)*q
655 >>> ip
656 <0101|0101>
657 >>> ip.doit()
658 Quantum gates (unitary operators) can be applied to transform these states and then
659 classical measurements can be performed on the results:
660 >>> from sympy.physics.quantum.qubit import Qubit, measure_all
661 >>> from sympy.physics.quantum.gate import H, X, Y, Z
662 >>> from sympy.physics.quantum.qapply import qapply
663 >>> c = H(0)*H(1)*Qubit('00')
664 >>> c
665 H(0)*H(1)*|00>
666 >>> q = qapply(c)
667 >>> measure_all(q)
668 [(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]

```

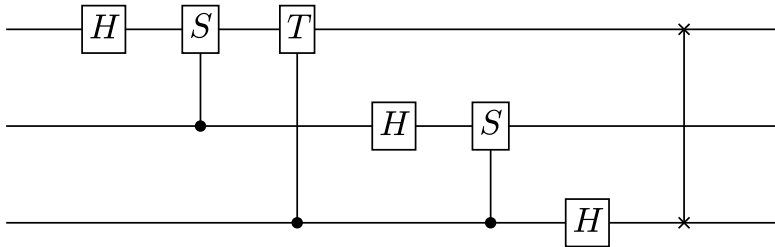


Fig. 1: The circuit diagram for a 3-qubit quantum fourier transform generated by SymPy.

```

669 Here is a final example of creating a 3-qubit quantum fourier transform, decomposing
670 it into one- and two-qubit gates, and then generating a circuit plot for the sequence
671 of gates (see Figure ??).
672 >>> from sympy.physics.quantum.qft import QFT
673 >>> from sympy.physics.quantum.circuitplot import circuit_plot
674 >>> fourier = QFT(0,3).decompose()
675 >>> fourier
676 SWAP(0,2)*H(0)*C((0),S(1))*H(1)*C((0),T(2))*C((1),S(2))*H(2)
677 >>> circuit_plot(fourier, nqubits=3)

```

678 **6. Conclusion and future work.** SymPy is a robust CAS that provides a wide
679 array of features. It is written in a general purpose programming language, Python,
680 which allows it to be used in a first-class way with other Python projects, including
681 the scientific Python stack. It is designed to be used in an extensible way. Unlike

many other CASs, it is designed to be used both as a end-user application and as a library.

SymPy expressions are built from immutable trees of Python classes. It uses Python both as the internal language and the user language, meaning users can use the same methods that the library implements to extend it. SymPy has an assumptions system for declaring and deducing mathematical properties on expressions.

The numerics of SymPy are implemented in the mpmath library, which uses arbitrary precision floating point arithmetic implemented in pure Python. This allows expressions to be evaluated with concrete data as needed.

SymPy has submodules for many areas of mathematics. It has functions for simplifying expressions, doing common calculus operations, pretty printing expressions, solving equations, and symbolic matrices. Other areas also included are discrete math, concrete math, plotting, geometry, statistics, polynomials, sets, series, vectors, combinatorics, group theory, code generation, tensors, Lie algebras, cryptography, and special functions. Additionally, SymPy contains submodules targeting certain specific domains, such as classical mechanics and quantum mechanics.

Some of the planned future work for SymPy includes work on improving code generation, improvements to the speed of SymPy, and improving the solvers module.

7. Acknowledgements. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

8. References.

9. Supplement.

9.1. Limits: The Gruntz Algorithm. SymPy calculates limits using the Gruntz algorithm, as described in [?]. 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:

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

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

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

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

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

718 The Gruntz algorithm is now illustrated on the following example:

719 (2)
$$f(x) = e^{x+2e^{-x}} - e^x + \frac{1}{x}.$$

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

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

727 Next any item ω is taken from mrsv that converges to zero for $x \rightarrow \infty$. The item
 728 $\omega = e^{-x}$ is obtained. If such a term is not present in the mrsv set (i.e., all terms
 729 converge to infinity instead of zero), the relation $f(x) \sim \frac{1}{f(x)}$ can be used.

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

732 (3)
$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2)$$

733 Since ω is from the mrsv set, then in the limit $x \rightarrow \infty$ it is $\omega \rightarrow 0$ and so
 734 $2\omega + O(\omega^2) \rightarrow 0$ in (??):

735 (4)
$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \rightarrow 2 + \frac{1}{x}$$

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

739 `>>> limit(exp(x+2*exp(-x))-exp(x) + 1/x, x, oo)`
 740 `2`

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

742 (5)
$$f(x) = \underbrace{O\left(\frac{1}{\omega^3}\right)}_{\infty} + \underbrace{\frac{C_{-2}(x)}{\omega^2}}_{\infty} + \underbrace{\frac{C_{-1}(x)}{\omega}}_{\infty} + C_0(x) + \underbrace{C_1(x)\omega}_0 + \underbrace{O(\omega^2)}_0$$

743 The positive powers of ω are zero. If there are any negative powers of ω , then the
 744 result of the limit is infinity, otherwise the limit is equal to $\lim_{x \rightarrow \infty} C_0(x)$. The expression
 745 $C_0(x)$ is simpler than $f(x)$ and so the algorithm always converges. A proof of this, as
 746 well as further details are given in Gruntz's Ph.D. thesis [?].

747 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[?] 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[?] 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[?]. 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)
```

```

797 >>> f.truncate(6)
798 x - x**3/6 + x**5/120 + 0(x**6)
799 >>> f[15]
800 -x**15/1307674368000

```

801 **9.2.3. Fourier Series.** SymPy provides functionality to compute Fourier Series
802 of a function using the `fourier_series` function. Under the hood it just computes a_0 ,
803 a_n , b_n using standard integration formulas.

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

```

805 >>> L = symbols('L')
806 >>> f = fourier_series(2 * (Heaviside(x/L) - Heaviside(x/L - 1)) - 1, (x, 0, 2*L))
807 >>> f.truncate(3)
808 4*sin(pi*x/L)/pi + 4*sin(3*pi*x/L)/(3*pi) + 4*sin(5*pi*x/L)/(5*pi)

```

809 **9.3. Logic.** SymPy supports construction and manipulation of boolean expres-
810 sions through the `logic` module. SymPy symbols can be used as propositional vari-
811 ables and also be substituted as `True` or `False`. A good number of manipulation
812 features for boolean expressions have been implemented in the `logic` module.

813 **9.3.1. Constructing boolean expressions.** A boolean variable can be de-
814 clared as a SymPy symbol. Python operators `&`, `|` and `~` are overloaded for logical
815 `And`, `Or` and `negate`. Several others like `Xor`, `Implies` can be constructed with `^`, `»`
816 respectively. The above are just a shorthand, expressions can also be constructed by
817 directly calling `And()`, `Or()`, `Not()`, `Xor()`, `Nand()`, `Nor()`, etc.

```

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

```

823 **9.3.2. CNF and DNF.** Any boolean expression can be converted to conjunc-
824 tive normal form, disjunctive normal form and negation normal form. The API also
825 permits to check if a boolean expression is in any of the above mentioned forms.

```

826 >>> from sympy import *
827 >>> x, y, z = symbols('x y z')
828 >>> to_cnf((x & y) | z)
829 And(Or(x, z), Or(y, z))
830 >>> to_dnf(x & (y | z))
831 Or(And(x, y), And(x, z))
832 >>> is_cnf((x | y) & z)
833 True
834 >>> is_dnf((x & y) | z)
835 True

```

836 **9.3.3. Simplification and Equivalence.** The module supports simplification
837 of given boolean expression by making deductions on it. Equivalence of two expres-
838 sions can also be checked. If so, it is possible to return the mapping of variables of
839 two expressions so as to represent the same logical behaviour.

```

840 >>> from sympy import *
841 >>> a, b, c, x, y, z = symbols('a b c x y z')
842 >>> e = a & (~a | ~b) & (a | c)
843 >>> simplify(e)

```

```

844 And(Not(b), a)
845 >>> e1 = a & (b | c)
846 >>> e2 = (x & y) | (x & z)
847 >>> bool_map(e1, e2)
848 (And(Or(b, c), a), {b: y, a: x, c: z})

```

849 **9.3.4. SAT solving.** The module also supports satisfiability checking of a given
850 boolean expression. If satisfiable, it is possible to return a model for which the ex-
851 pression is satisfiable. The API also supports returning all possible models. The SAT
852 solver has a clause learning DPLL algorithm implemented with watch literal scheme
853 and VSIDS heuristic[?].

```

854 >>> from sympy import *
855 >>> a, b, c = symbols('a b c')
856 >>> satisfiable(a & (~a | b) & (~b | c) & ~c)
857 False
858 >>> satisfiable(a & (~a | b) & (~b | c) & c)
859 {b: True, a: True, c: True}

```

860 **9.4. Diophantine Equations.** Diophantine equations play a central and an im-
861 portant role in number theory. A Diophantine equation has the form, $f(x_1, x_2, \dots, x_n) =$
862 0 where $n \geq 2$ and x_1, x_2, \dots, x_n are integer variables. If we can find n integers
863 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
864 say that the equation is solvable.

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

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

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

```

876 >>> from sympy import symbols
877 >>> x, y, z = symbols("x, y, z", integer=True)
878
879 >>> diophantine(2*x + 3*y - 5)
880 set([(3*t_0 - 5, -2*t_0 + 5)])
881
882 >>> diophantine(2*x + 4*y - 3)
883 set()
884
885 >>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
886 set([(2, 1), (5, 1)])
887
888 >>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
889 set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])
890
891 >>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
892 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)])

```

893

894 `>>> from sympy.abc import a, b, c, d, e, f`

895 `>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**2)`

896 `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, 42*t1**2 +`

897

898 `>>> diophantine(a**2 + b**2 + c**2 + d**2 + e**2 + f**2 - 112)`

899 `set([(8, 4, 4, 4, 0, 0)])`

900 **9.5. Sets.** SymPy supports representation of a wide variety of mathematical
 901 sets. This is achieved by first defining abstract representations of atomic set classes
 902 and then combining and transforming them using various set operations.

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

907 SymPy has the following atomic set classes:

- 908 • `EmptySet` represents the empty set \emptyset .
- 909 • `UniversalSet` is an abstract “universal set” for which everything is a member.
 910 The union of the universal set with any set gives the universal set and the
 911 intersection gives to the other set itself.
- 912 • `FiniteSet` is functionally equivalent to Python’s built `inset` object. Its mem-
 913 bers can be any SymPy object including other sets themselves.
- 914 • `Integers` represents the set of Integers \mathbb{Z} .
- 915 • `Naturals` represents the set of Natural numbers \mathbb{N} , i.e., the set of positive
 916 integers.
- 917 • `Naturals0` represents the whole numbers, which are all the non-negative in-
 918 tegers.
- 919 • `Range` represents a range of integers. A range is defined by specifying a start
 920 value, an end value, and a step size. Range is functionally equivalent to
 921 Python’s `range` except it supports infinite endpoints, allowing the represen-
 922 tation of infinite ranges.
- 923 • `Interval` represents an interval of real numbers. It is specified by giving the
 924 start and end point and specifying if it is open or closed in the respective
 925 ends.

926 Other than unevaluated classes of Union, Intersection and Set Difference opera-
 927 tions, we have following set classes.

- 928 • `ProductSet` defines the Cartesian product of two or more sets. The product
 929 set is useful when representing higher dimensional spaces. For example to
 930 represent a three-dimensional space we simply take the Cartesian product of
 931 three real sets.
- 932 • `ImageSet` represents the image of a function when applied to a particular
 933 set. In notation, the image set of a function F with respect to a set S is
 934 $\{F(x)|x \in S\}$. SymPy uses image sets to represent sets of infinite solutions
 935 equations such as $\sin(x) = 0$.
- 936 • `ConditionSet` represents subset of a set whose members satisfies a particular
 937 condition. In notation, the condition set of the set S with respect to the
 938 condition H is $\{x|H(x), x \in S\}$. SymPy uses condition sets to represent
 939 the set of solutions of equations and inequalities, where the equation or the
 940 inequality is the condition and the set is the domain being solved over.

941 A few other classes are implemented as special cases of the classes described above.

942 The set of real numbers, `Reals` is implemented as a special case of `Interval`, $(-\infty, \infty)$.
 943 `ComplexRegion` is implemented as a special case of `ImageSet`. `ComplexRegion` supports
 944 both polar and rectangular representation of regions on the complex plane.

945 **9.6. SymPy Gamma.** SymPy Gamma is a simple web application that runs
 946 on Google App Engine. It executes and displays the results of SymPy expressions as
 947 well as additional related computations, in a fashion similar to that of Wolfram|Alpha.
 948 For instance, entering an integer will display its prime factors, digits in the base-10
 949 expansion, and a factorization diagram. Entering a function will display its docstring;
 950 in general, entering an arbitrary expression will display its derivative, integral, series
 951 expansion, plot, and roots.

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

- 954 • It displays integration steps, differentiation steps in detail, which can be
 955 viewed in Figure ??:

956

Integral Steps:
`integrate(tan(x), x)`

Fullscreen

1. Rewrite the integrand:

$$\tan(x) = \frac{\sin(x)}{\cos(x)}$$
2. Let $u = \cos(x)$.
 Then let $du = -\sin(x)dx$ and substitute du :

$$\int -\frac{1}{u} du$$

A. The integral of a constant times a function is the constant times the integral of the function:

$$\int -\frac{1}{u} du = - \int \frac{1}{u} du$$

I. The integral of $\frac{1}{u}$ is $\log(u)$.

So, the result is: $-\log(u)$

Now substitute u back in:

$$-\log(\cos(x))$$
3. Add the constant of integration:

$$-\log(\cos(x)) + \text{constant}$$

The answer is:

$$-\log(\cos(x)) + \text{constant}$$

Fig. 2: Integral steps of $\tan(x)$

- 957 • It also displays the factor tree diagrams for different numbers.
 - 958 • SymPy Gamma also saves user search queries, and offers many such similar
 959 features for free, which Wolfram|Alpha only offers to its paid users.
- 960 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++ [?] 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 asso-

ciative 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[?], commutative[?], and one-identity[?] properties of its expression tree nodes[?]. 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 [?]. 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 exponentiation is

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

1057 available through the `MatrixExp` function.

1058 Unevaluated expressions can be achieved in various ways, most commonly with

1059 the `HoldForm` or `Hold` nodes, that block the evaluation of subnodes by the parser.

1060 Note that such a node cannot be expressed in Python, because of greedy evaluation.

1061 Whenever needed in SymPy, it is necessary to add the parameter `evaluate=False` to

1062 all subnodes, or put the input expression in a string.

1063 The operator `==` returns a boolean whenever it is able to immediately evaluate

1064 the truthness of the equality, otherwise it returns an `Equal` expression. In SymPy `==`

1065 means structural equality and is always guaranteed to return a boolean expression.

1066 To express an equality in SymPy it is necessary to explicitly construct the `Equality`

1067 class.

1068 SymPy, in accordance with Python and unlike the usual programming convention,

1069 uses `**` to express the power operator, while Mathematica uses the more common `^`.

1070 **9.9. Other Projects that use SymPy.** There are several projects that use

1071 SymPy as a library for implementing a part of their project, or even as a part of

1072 back-end for their application as well.

1073 Some of them are listed below:

- 1074 • **Cadabra**: Cadabra is a symbolic computer algebra system (CAS) designed
- 1075 specifically for the solution of problems encountered in field theory.
- 1076 • **Octave Symbolic**: The Octave-Forge Symbolic package adds symbolic cal-
- 1077 culation features to GNU Octave. These include common Computer Algebra
- 1078 System tools such as algebraic operations, calculus, equation solving, Fourier
- 1079 and Laplace transforms, variable precision arithmetic and other features.
- 1080 • **SymPy.jl**: Provides a Julia interface to SymPy using PyCall.
- 1081 • **Mathics**: Mathics is a free, general-purpose online CAS featuring Mathe-
- 1082 matica compatible syntax and functions. It is backed by highly extensible
- 1083 Python code, relying on SymPy for most mathematical tasks.
- 1084 • **Mathpix**: An iOS App, that uses Artificial Intelligence to detect handwritten
- 1085 math as input, and uses SymPy Gamma, to evaluate the math input and
- 1086 generate the relevant steps to solve the problem.
- 1087 • **IKFast**: IKFast is a robot kinematics compiler provided by **OpenRAVE**.
- 1088 It analytically solves robot inverse kinematics equations and generates opti-
- 1089 mized C++ files. It uses SymPy for its internal symbolic mathematics.
- 1090 • **Sage**: A CAS, visioned to be a viable free open source alternative to Magma,
- 1091 Maple, Mathematica and Matlab.
- 1092 • **SageMathCloud**: SageMathCloud is a web-based cloud computing and
- 1093 course management platform for computational mathematics.
- 1094 • **PyDy**: Multibody Dynamics with Python.
- 1095 • **galgebra**: Geometric algebra (previously `sympy.galgebra`).
- 1096 • **yt**: Python package for analyzing and visualizing volumetric data (`yt.units`
- 1097 uses SymPy).
- 1098 • **SfePy**: Simple finite elements in Python, see Section ??.
- 1099 • **Quameon**: Quantum Monte Carlo in Python.
- 1100 • **Lcapy**: Experimental Python package for teaching linear circuit analysis.
- 1101 • **Quantum Programming in Python**: Quantum 1D Simple Harmonic Os-
- 1102 cillator and Quantum Mapping Gate.
- 1103 • **LaTeX Expression project**: Easy LaTeX typesetting of algebraic expres-
- 1104 sions in symbolic form with automatic substitution and result computation.

- **Symbolic statistical modeling:** Adding statistical operations to complex physical models.

9.10. Project Details. Below we provide particular examples of SymPy use in some of the projects listed above.

9.10.1. SfePy. **SfePy** (Simple finite elements in Python), cf. [?]. is a Python package for solving partial differential equations (PDEs) in 1D, 2D and 3D by the finite element (FE) method [?]. SymPy is used within this package mostly for code generation and testing, namely:

- generation of the hierarchical FE basis module, involving generation and symbolic differentiation of 1D Legendre and Lobatto polynomials, constructing the FE basis polynomials [?] and generating the C code;
- generation of symbolic conversion formulas for various groups of elastic constants [?] – provide any two of the Young’s modulus, Poisson’s ratio, bulk modulus, Lamé’s first parameter, shear modulus (Lamé’s second parameter) or longitudinal wave modulus and get the other ones;
- simple physical unit conversions, generation of consistent unit sets;
- testing FE solutions using method of manufactured (analytical) solutions – the differential operator of a PDE is symbolically applied and a symbolic right-hand side is created, evaluated in quadrature points, and subsequently used to obtain a numerical solution that is then compared to the analytical one;
- testing accuracy of 1D, 2D and 3D numerical quadrature formulas (cf. [?]) by generating polynomials of suitable orders, integrating them, and comparing the results with those obtained by the numerical quadrature.

9.11. Tensors. Ongoing work to provide the capabilities of tensor computer algebra has so far produced the `tensor` module. It is composed of three separated sub-modules, 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[?] and Cadabra[?]. Canonicalization based on the Butler-Portugal[?] algorithm is supported in SymPy. It is currently limited to polynomial tensor expressions.

9.12. Symbolic probability. SymPy has a module for symbolic probability calculations, `sympy.stats`[?].