

1 SymPy: Symbolic Computing in Python

2 Supplementary material

3 The supplementary material take a deeper look at certain topics in SymPy for which there
4 was not enough room to discuss in the paper. Section 1 discusses the Gruntz algorithm, used to
5 calculate limits in the SymPy. Sections 2–8 discuss in depth some selected submodules. Section 9
6 discusses numerical simplification. Section 10 provides additional examples for topics, discussed
7 in the main paper. In section 11 the SymPy Gamma and the SymPy Live projects are introduced.
8 Finally, section 12 has a brief comparison of SymPy with Wolfram Mathematica.

9 As in the paper, all examples in the supplement assume that the following has been run:

```
10 >>> from sympy import *  
11 >>> x, y, z = symbols('x y z')
```

12 1 LIMITS: THE GRUNTZ ALGORITHM

13 SymPy calculates limits using the Gruntz algorithm, as described in [4]. The basic idea is as
14 follows: any limit can be converted to a limit $\lim_{x \rightarrow \infty} f(x)$ by substitutions like $x \rightarrow \frac{1}{x}$. Then
15 the subexpression ω (that converges to zero as $x \rightarrow \infty$ faster than all other subexpressions) is
16 identified in $f(x)$, and $f(x)$ is expanded into a series with respect to ω . Any positive powers
17 of ω converge to zero (while negative powers indicate an infinite limit) and any constant term
18 independent of ω determines the limit. When a constant term still depends on x the Gruntz
19 algorithm is applied again until a final numerical value is obtained as the limit.

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

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

The relations $<$, $>$, 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), $f < g$ 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). Note that if $f > g$, then $f > g^n$ for any n . Here are some examples of comparability classes:

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

The Gruntz algorithm is now illustrated with the following example:

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

20 First, the set of most rapidly varying subexpressions is determined—the so-called *mrsv set*. For (2),
21 the mrsv set $\{e^x, e^{-x}, e^{x+2e^{-x}}\}$ is obtained. These are all subexpressions of (2) and they all
22 belong to the same comparability class. This calculation can be done using SymPy as follows:

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

Next, an arbitrary item ω is taken from mrv set that converges to zero for $x \rightarrow \infty$ and doesn't have subexpressions in the given mrv set. If such a term is not present in the mrv set (i.e., all terms converge to infinity instead of zero), the relation $f(x) \sim \frac{1}{f(x)}$ can be used. In the considered case, only item $\omega = e^{-x}$ can be accepted.

The next step is to rewrite the mrv set in terms of $\omega = g(x)$. Every element $f(x)$ of the mrv set is rewritten as $A\omega^c$, where

$$c = \lim_{x \rightarrow \infty} \frac{\log f(x)}{\log g(x)}, \quad A = e^{\log f - c \log g} \quad (3)$$

Note that this step includes calculation of more simple limits, for instance

$$\lim_{x \rightarrow \infty} \frac{\log e^{x+2e^{-x}}}{\log e^{-x}} = \lim_{x \rightarrow \infty} \frac{x+2e^{-x}}{-x} = -1 \quad (4)$$

In this example we obtain the rewritten mrv set: $\{\frac{1}{\omega}, \omega, \frac{1}{\omega}e^{2\omega}\}$. This can be done in SymPy with

```
>>> from sympy.series.gruntz import mrv, rewrite
>>> m = mrv(exp(x+2*exp(-x))-exp(x) + 1/x, x)
>>> w = Symbol('w')
>>> rewrite(m[1], m[0], x, w)[0]
1/x + exp(2*w)/w - 1/w
```

Then the rewritten subexpressions are substituted back into $f(x)$ in (2) and the result is expanded with respect to ω :

$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \quad (5)$$

Since ω is from the mrv set, then in the limit as $x \rightarrow \infty$, $\omega \rightarrow 0$, and so $2\omega + O(\omega^2) \rightarrow 0$ in (5):

$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \rightarrow 2 + \frac{1}{x} \quad (6)$$

In this example the result $(2 + \frac{1}{x})$ still depends on x , so the above procedure is repeated until just a value independent of x is obtained. This is the final limit. In the above case the limit is 2, as can be verified by SymPy:¹

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

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

$$f(x) = \underbrace{O\left(\frac{1}{\omega^3}\right)}_{\infty} + \underbrace{\frac{C_{-2}(x)}{\omega^2}}_{\infty} + \underbrace{\frac{C_{-1}(x)}{\omega}}_{\infty} + C_0(x) + \underbrace{C_1(x)\omega}_0 + \underbrace{O(\omega^2)}_0 \quad (7)$$

The positive powers of ω are zero. If there are any negative powers of ω , then the result of the limit is infinity, otherwise the limit is equal to $\lim_{x \rightarrow \infty} C_0(x)$. The expression $C_0(x)$ is always simpler than original $f(x)$, same is true for limits, arising in the rewrite stage (3), so the algorithm converges. A proof of this and further details on the algorithm are given in Gruntz's PhD thesis [4].

¹To see intermediate steps, discussed above, interested readers can switch on debugging output by setting the environment variable `SYMPY_DEBUG=True`, before importing anything from the SymPy namespace.

2 SERIES

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 instance of the `Expr` 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:

```
>>> 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 fact that a truncated Taylor series is simply a polynomial. Correspondingly, they may be represented by sparse polynomials which perform well in a wide range of cases. Ring Series also gives the user the freedom to choose the type of coefficients to use, resulting in faster operations on certain types.

For this, several low-level methods for expansion of trigonometric, hyperbolic and other elementary operations (like series inversion, calculating the n th root, etc.) are implemented using variants of the Newton Method [Brent and Zimmermann]. 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.polys.ring_series import rs_sin
>>> R, t = ring('t', QQ)
>>> rs_sin(t**2 + t, t, 5)
-1/2*t**4 - 1/6*t**3 + t**2 + t
```

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 lowermost 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 demonstrating its use:

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

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 [5]. 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
```

2.3 Fourier Series

SymPy provides functionality to compute Fourier series of a function using the `fourier_series` function:

```
95 >>> L = symbols('L')
96 >>> expr = 2 * (Heaviside(x/L) - Heaviside(x/L - 1)) - 1
97 >>> f = fourier_series(expr, (x, 0, 2*L))
98 >>> f.truncate(3)
99 4*sin(pi*x/L)/pi + 4*sin(3*pi*x/L)/(3*pi) + 4*sin(5*pi*x/L)/(5*pi)
```

3 LOGIC

SymPy supports construction and manipulation of boolean expressions through the `sympy.logic` submodule. SymPy symbols can be used as propositional variables and subsequently be replaced with `True` or `False` values. Many functions for manipulating boolean expressions have been implemented in the `sympy.logic` submodule.

3.1 Constructing Boolean Expressions

A boolean variable can be declared as a SymPy `Symbol`. Python operators `&`, `|` and `~` are overridden when using SymPy objects to use the SymPy functionality for logical `And`, `Or`, and `Not`. Other logic functions are also integrated into SymPy, including `Xor` and `Implies`, which are constructed with `^` and `>>`, respectively. Expressions can therefore be constructed either by using the shortcut operator notation or by directly creating the relevant objects: `And()`, `Or()`, `Not()`, `Xor()`, `Implies()`, `Nand()`, `Nor()`, etc.:

```
112 >>> e = (x & y) | z
113 >>> e.subs({x: True, y: True, z: False})
114 True
```

3.2 CNF and DNF

Any boolean expression can be converted to conjunctive normal form, disjunctive normal form, or negation normal form. The API also exposes methods to check if a boolean expression is in any of the aforementioned forms.

```
119 >>> from sympy.logic.boolalg import is_dnf, is_cnf
120 >>> to_cnf((x & y) | z)
121 And(Or(x, z), Or(y, z))
122 >>> to_dnf(x & (y | z))
123 Or(And(x, y), And(x, z))
124 >>> is_cnf((x | y) & z)
125 True
126 >>> is_dnf((x & y) | z)
127 True
```

3.3 Simplification and Equivalence

The `sympy.logic` submodule supports simplification of given boolean expression by making deductions from the expression. Equivalence of two logical expressions can also be checked. In the case of equivalence, the function `bool_map` can be used to show which variables of the first expression correspond to which variables of the second one.

```
133 >>> a, b, c = symbols('a b c')
134 >>> e = a & (~a | ~b) & (a | c)
135 >>> simplify(e)
136 And(Not(b), a)
137 >>> e1 = a & (b | c)
138 >>> e2 = (x & y) | (x & z)
139 >>> bool_map(e1, e2)
140 (And(Or(b, c), a), {a: x, b: y, c: z})
```

141 3.4 SAT Solving

142 The submodule also supports satisfiability (SAT) checking of a given boolean expression. If an
143 expression is satisfiable, it is possible to return a variable assignment which satisfies it. The
144 API also supports listing all possible assignments. The SAT solver has a clause learning DPLL
145 algorithm implemented with a watch literal scheme and VSIDS heuristic [8].

```
146 >>> satisfiable(a & (~a | b) & (~b | c) & ~c)
147 False
148 >>> satisfiable(a & (~a | b) & (~b | c) & c)
149 {a: True, b: True, c: True}
```

150 4 DIOPHANTINE EQUATIONS

151 Diophantine equations play a central role in number theory. A Diophantine equation has the
152 form, $f(x_1, x_2, \dots, x_n) = 0$ where $n \geq 2$ and x_1, x_2, \dots, x_n are integer variables. If there are n
153 integers a_1, a_2, \dots, a_n such that $x_1 = a_1, x_2 = a_2, \dots, x_n = a_n$ satisfies the above equation, the
154 equation is said to be solvable.

155 Currently, the following five types of Diophantine equations can be solved using SymPy's
156 Diophantine submodule (a_1, \dots, a_{n+1} , a , b , c , d , e , f , and k are explicitly given rational constants):

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

162 The `diophantine` function factors the equation it is given (if possible), solves each factor
163 separately, and combines the results to give a final solution set. The following examples illustrate
164 some of the basic functionalities of the Diophantine submodule.

```
165 >>> from sympy.solvers.diophantine import *
166 >>> diophantine(2*x + 3*y - 5)
167 set([(3*t_0 - 5, -2*t_0 + 5)])
168
169 >>> diophantine(2*x + 4*y - 3)
170 set()
171
172 >>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
173 set([(2, 1), (5, 1)])
174
175 >>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
176 set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])
177
178 >>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
179 set([(-16*p**2 + 28*p*q + 20*q**2,
180 3*p**2 + 38*p*q - 25*q**2,
181 4*p**2 - 24*p*q + 68*q**2)])
182
183 >>> x1, x2, x3, x4, x5, x6 = symbols('x1 x2 x3 x4 x5 x6')
184 >>> diophantine(9*x1**2 + 16*x2**2 + x3**2 + 49*x4**2 + 4*x5**2 - 25*x6**2)
185 set([(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5,
186 420*t2*t5, 60*t3*t5, 210*t4*t5,
187 42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)])
188
```

```

189 >>> a, b, c, d = symbols('a b c d')
190 >>> diophantine(a**2 + b**2 + c**2 + d**2 - 23)
191 set([(2, 3, 3, 1)])

```

192 5 SETS

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

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

199 SymPy has the following atomic set classes:

- 200 • `EmptySet` represents the empty set \emptyset .
- 201 • `UniversalSet` is an abstract “universal set” of which everything is a member. The union of
 202 the universal set with any set gives the universal set and the intersection gives the other
 203 set itself.
- 204 • `FiniteSet` is functionally equivalent to Python’s built in `set` object. Its members can be
 205 any SymPy object including other sets.
- 206 • `Integers` represents the set of integers \mathbb{Z} .
- 207 • `Naturals` represents the set of natural numbers \mathbb{N} , i.e., the set of positive integers.
- 208 • `Naturals0` represents the set of whole numbers \mathbb{N}_0 , which are all the non-negative integers.
- 209 • `Range` represents a range of integers. A range is defined by specifying a start value, an end
 210 value, and a step size. The enumeration of a `Range` object is functionally equivalent to
 211 Python’s `range` except it supports infinite endpoints, allowing the representation of infinite
 212 ranges.
- 213 • `Interval` represents an interval of real numbers. It is defined by giving the start and the
 214 end points and by specifying if the interval is open or closed on the respective ends.

215 Other than unevaluated classes of `Union`, `Intersection`, and `Complement` operations, SymPy
 216 has the following set classes.

- 217 • `ProductSet` defines the Cartesian product of two or more sets. The product set is useful
 218 when representing higher dimensional spaces. For example, to represent a three-dimensional
 219 space, SymPy uses the Cartesian product of three real sets.
- 220 • `ImageSet` represents the image of a function when applied to a particular set. The image
 221 set of a function F with respect to a set S is $\{F(x) \mid x \in S\}$. SymPy uses image sets to
 222 represent sets of infinite solutions of equations such as $\sin(x) = 0$.
- 223 • `ConditionSet` represents a subset of a set whose members satisfy a particular condition.
 224 The subset of set S given by the condition H is $\{x \mid H(x), x \in S\}$. SymPy uses condition
 225 sets to represent the set of solutions of equations and inequalities, where the equation or
 226 the inequality is the condition and the set is the domain over which it is being solved.

227 A few other classes are implemented as special cases of the classes described above. The set of
 228 real numbers, `Reals`, is implemented as a special case of `Interval`. `ComplexRegion` is implemented
 229 as a special case of `ImageSet`. `ComplexRegion` supports both polar and rectangular representation
 230 of regions on the complex plane.

6 STATISTICS

The `sympy.stats` submodule provides random variable types and methods for computing of statistical properties of expressions involving random variables, which can be either continuous or discrete, the latter ones being further divided into finite and infinite. The variables are associated with probability densities on corresponding domains and internally defined in terms of probability spaces. Apart from the possibility of defining the random variables from user supplied density distribution, SymPy provides definitions of most common distributions, including `Uniform`, `Poisson`, `Normal`, `Binomial`, `Bernoulli`, and many others.

Properties of random expressions can be calculated using, e.g., `expectation` (abbreviated `E`) and `variance` to calculate expectation and variance. Internally, these functions generate integrals and summations, which are automatically evaluated. The evaluation can be suppressed using `evaluate=False` keyword argument.

Conditions on random variables can be defined with inequalities, equalities, and logical operators and their overall probabilities are obtained using `P`. The features can be illustrated on a model of two dice throws:

```
>>> from sympy.stats import Die, P, E
>>> X, Y = Die("X"), Die("Y")
>>> P(Eq(X, 6) & Eq(Y, 6))
1/36
>>> P(X>Y)
5/12
```

The conditions can also be supplied as a second parameter to `E`, `P`, and other methods to calculate the property given the condition:

```
>>> E(X, X+Y<5)
5/3
```

Using the facilities of the `sympy.stats` submodule, one can, for example, calculate the well known properties of Maxwellian velocity distribution

```
>>> from sympy.stats import Maxwell, density
>>> kT, m, t = symbols("kT m t", positive=True)
>>> v = Maxwell("v", sqrt(kT/m))
>>> E(v) # mean velocity
2*sqrt(2)*sqrt(kT)/(sqrt(pi)*sqrt(m))
>>> E(v, evaluate=False) # unevaluated mean velocity
Integral(sqrt(2)*m**(3/2)*v**3*exp(-m*v**2/(2*kT))/(sqrt(pi)*kT**(3/2)),
(v, 0, oo))
>>> E(m*v**2/2) # mean energy
3*kT/2
>>> solve(density(v)(t).diff(t), t)[0] # most probable velocity
sqrt(2)*sqrt(kT)/sqrt(m)
```

More information on the `sympy.stats` submodule can be found in [10].

7 CATEGORY THEORY

SymPy includes a submodule for dealing with categories—abstract mathematical objects representing classes of structures as classes of objects (points) and morphisms (arrows) between the objects. It was designed with the following two goals in mind:

1. automatic typesetting of diagrams given by a collection of objects and of morphisms between them, and
2. specification and semi-automatic derivation of properties using commutative diagrams.

As of version 1.0, SymPy only implements the first goal, while a partially working draft of implementation of the second goal is available at <https://github.com/scolobb/sympy/tree/ct4-commutativity>.

In order to achieve the two goals, the submodule `sympy.categories` defines several classes representing some of the essential concepts: objects, morphisms, categories, and diagrams. In category theory, the inner structure of objects is often discarded in the favor of studying the properties of morphisms, so the class `Object` is essentially a synonym of the class `Symbol`. There are several morphism classes which do not have a particular internal structure either, though an exception is `CompositeMorphism`, which essentially stores a list of morphisms.

The class `Diagram` captures the properties of morphisms. This class stores a family of morphisms, the corresponding source and target objects, and, possibly, some properties of the morphisms. Generally, no restrictions are imposed on what the properties may be—for example, one might use strings of the form “forall”, “exists”, “unique”, etc. Furthermore, the morphisms of a diagram are grouped into *premises* and *conclusions* in order to be able to represent logical implications of the form “for a collection of morphisms P with properties $p : P \rightarrow \Omega$ (the premises), there exists a collection of morphisms C with properties $c : C \rightarrow \Omega$ (the conclusions)”, where Ω is the universal collection of properties. Finally, the class `Category` includes a collection of diagrams which are deemed commutative and which therefore define the properties of this category.

Automatic typesetting of diagrams takes a `Diagram` and produces L^AT_EX code using the `Xy-pic` package. Typesetting is done in two stages: layout and generation of `Xy-pic` code. The layout stage is taken care of by the class `DiagramGrid`, which takes a `Diagram` and lays out the objects in a grid, trying to reduce the average length of the arrows in the final picture. By default, `DiagramGrid` uses a series of triangle-based heuristics to produce a rectangular grid. A linear layout can also be imposed. Furthermore, groups of objects can be given; in this case, the groups will be treated as atomic cells, and the member objects will be typeset independently of the other objects.

The second phase of diagram typesetting consists in actually drawing the picture and is carried out by the class `XypicDiagramDrawer`. An example of a diagram automatically typeset by `DiagramGrid` and `XypicDiagramDrawer` is given in Figure 1.

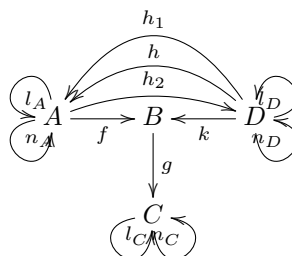


Figure 1. An automatically typeset commutative diagram

As far as the second main goal of `sympy.categories` is concerned, the principal idea consists in automatically deciding whether a diagram is commutative or not, given a collection of “axioms”: diagrams known to be commutative. The implementation is based on graph embeddings (injective maps): whenever an embedding of a commutative diagram into a given diagram is found, one concludes that the subdiagram is commutative. Deciding commutativity of the whole diagram is therefore based (theoretically) on finding a “cover” of the target diagram by embeddings of the axioms. The naïve implementation proved to be prohibitively slow; a better optimized version is therefore in order, as well as application of heuristics.

8 TENSORS

Ongoing work to provide the capabilities of tensor computer algebra has so far produced the `sympy.tensor` submodule. It comprises three submodules whose purposes are quite different: `sympy.tensor.indexed` and `sympy.tensor.indexed_methods` support indexed symbols, `sympy.tensor.array` contains facilities to operate on symbolic N -dimensional arrays, and finally `sympy.`

320 `tensor.tensor` is used to define abstract tensors. The abstract tensors submodule is inspired
 321 by `xAct` [7] and `Cadabra` [9]. Canonicalization based on the Butler-Portugal [6] algorithm
 322 is supported in `SymPy`. Tensor support in `SymPy` is currently limited to polynomial tensor
 323 expressions.

324 9 NUMERICAL SIMPLIFICATION

325 The `nsimplify` function in `SymPy` (a wrapper of `identify` in `mpmath`) attempts to find a simple
 326 symbolic expression that evaluates to the same numerical value as the given input. It works
 327 by applying a few simple transformations (including square roots, reciprocals, logarithms and
 328 exponentials) to the input and, for each transformed value, using the PSLQ algorithm [3] to
 329 search for a matching algebraic number or optionally a linear combination of user-provided base
 330 constants (such as π).

```
331 >>> t = 1 / (sin(pi/5)+sin(2*pi/5)+sin(3*pi/5)+sin(4*pi/5))**2
332 >>> nsimplify(t)
333 -2*sqrt(5)/5 + 1
334 >>> nsimplify(pi, tolerance=0.01)
335 22/7
336 >>> nsimplify(1.783919626661888, [pi], tolerance=1e-12)
337 pi/(-1/3 + 2*pi/3)
```

338 10 EXAMPLES

339 10.1 Simplification

- 340 • `expand`:

```
341 >>> expand((x + y)**3)
342 x**3 + 3*x**2*y + 3*x*y**2 + y**3
```

- 343 • `factor`:

```
344 >>> factor(x**3 + 3*x**2*y + 3*x*y**2 + y**3)
345 (x + y)**3
```

- 346 • `collect`:

```
347 >>> collect(y*x**2 + 3*x**2 - x*y + x - 1, x)
348 x**2*(y + 3) + x*(-y + 1) - 1
```

- 349 • `cancel`:

```
350 >>> cancel((x**2 + 2*x + 1)/(x**2 - 1))
351 (x + 1)/(x - 1)
```

- 352 • `apart`:

```
353 >>> apart((x**3 + 4*x - 1)/(x**2 - 1))
354 x + 3/(x + 1) + 2/(x - 1)
```

- 355 • `trigsimp`:

```
356 >>> trigsimp(cos(x)**2*tan(x) - sin(2*x))
357 -sin(2*x)/2
```

358 10.2 Polynomials

- 359 • Factorization:

```
360 >>> t = symbols('t')
361 >>> f = (2115*x**4*y + 45*x**3*z**3*t**2 - 45*x**3*t**2 -
362 ...      423*x*y**4 - 47*x*y**3 + 141*x*y*z**3 + 94*x*y*z*t -
363 ...      9*y**3*z**3*t**2 + 9*y**3*t**2 - y**2*z**3*t**2 +
364 ...      y**2*t**2 + 3*z**6*t**2 + 2*z**4*t**3 - 3*z**3*t**2 -
365 ...      2*z*t**3)
366 >>> factor(f)
367 (t**2*z**3 - t**2 + 47*x*y)*(2*t*z + 45*x**3 - 9*y**3 - y**2 +
368 3*z**3)
```

- 369 • Gröbner bases:

```
370 >>> x0, x1, x2 = symbols('x:3')
371 >>> I = [x0 + 2*x1 + 2*x2 - 1,
372 ...      x0**2 + 2*x1**2 + 2*x2**2 - x0,
373 ...      2*x0*x1 + 2*x1*x2 - x1]
374 >>> groebner(I, order='lex')
375 GroebnerBasis([7*x0 - 420*x2**3 + 158*x2**2 + 8*x2 - 7,
376 7*x1 + 210*x2**3 - 79*x2**2 + 3*x2,
377 84*x2**4 - 40*x2**3 + x2**2 + x2], x0, x1, x2, domain='ZZ',
378 order='lex')
```

- 379 • Root isolation:

```
380 >>> f = 7*z**4 - 19*z**3 + 20*z**2 + 17*z + 20
381 >>> intervals(f, all=True, eps=0.001)
382 ([],
383  [((-425/1024 - 625*I/1024, -1485/3584 - 2185*I/3584), 1),
384  ((-425/1024 + 2185*I/3584, -1485/3584 + 625*I/1024), 1),
385  ((3175/1792 - 2605*I/1792, 1815/1024 - 10415*I/7168), 1),
386  ((3175/1792 + 10415*I/7168, 1815/1024 + 2605*I/1792), 1)])
```

387 10.3 Solvers

- 388 • Single solution:

```
389 >>> solveset(x - 1, x)
390 {1}
```

- 391 • Finite solution set, quadratic equation:

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

- 394 • No solution:

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

- 397 • Interval solution:

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

- 400 • Infinitely many solutions:

```
401 >>> solveset(x - x, x, domain=S.Reals)
402 (-oo, oo)
403 >>> solveset(x - x, x, domain=S.Complexes)
404 S.Complexes
```

- 405 • Linear systems (linsolve)

```
406 >>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
407 >>> b = Matrix([3, 6, 9])
408 >>> linsolve((A, b), x, y, z)
409 {(-1, 2, 0)}
410 >>> linsolve(Matrix((([1, 1, 1, 1], [1, 1, 2, 3])), (x, y, z))
411 {(-y - 1, y, 2)}
```

412 Below are examples of `solve` applied to problems not yet handled by `solveset`.

- 413 • Nonlinear (multivariate) system of equations (the intersection of a circle and a parabola):

```
414 >>> solve([x**2 + y**2 - 16, 4*x - y**2 + 6], x, y)
415 [(-2 + sqrt(14), -sqrt(-2 + 4*sqrt(14))),
416  (-2 + sqrt(14), sqrt(-2 + 4*sqrt(14))),
417  (-sqrt(14) - 2, -I*sqrt(2 + 4*sqrt(14))),
418  (-sqrt(14) - 2, I*sqrt(2 + 4*sqrt(14)))]
```

- 419 • Transcendental equations:

```
420 >>> solve((x + log(x))**2 - 5*(x + log(x)) + 6, x)
421 [LambertW(exp(2)), LambertW(exp(3))]
422 >>> solve(x**3 + exp(x))
423 [-3*LambertW((-1)**(2/3)/3)]
```

424 10.4 Matrices

- 425 • Matrix expressions

```
426 >>> m, n, p = symbols('m n p', integer=True)
427 >>> R = MatrixSymbol('R', m, n)
428 >>> S = MatrixSymbol('S', n, p)
429 >>> T = MatrixSymbol('T', m, p)
430 >>> U = R*S + 2*T
431 >>> U.shape
432 (m, p)
433 >>> U[0, 1]
434 2*T[0, 1] + Sum(R[0, _k]*S[_k, 1], (_k, 0, n - 1))
```

- 435 • Block Matrices

```
436 >>> n, m, l = symbols('n m l')
437 >>> X = MatrixSymbol('X', n, n)
438 >>> Y = MatrixSymbol('Y', m, m)
439 >>> Z = MatrixSymbol('Z', n, m)
440 >>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
441 >>> B
442 Matrix([
443 [X, Z],
```

```

444     [0, Y]])
445     >>> B[0, 0]
446     X[0, 0]
447     >>> B.shape
448     (m + n, m + n)

```

449 11 SYMPY GAMMA

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

456 SymPy Gamma also has several features beyond just computing the results using SymPy.

- 457 • SymPy Gamma displays integration and differentiation steps in detail, which can be viewed
 458 in Figure 2:

Integral steps:

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:

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

459

460 **Figure 2.** Integral steps of $\tan(x)$

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

464 Every input query from the user on SymPy Gamma is first parsed by its own parser capable of
 465 handling several different forms of function names which SymPy as a library does not support.

466 For instance, SymPy Gamma supports queries like `sin x`, whereas SymPy will only recognise
467 `sin(x)`.

468 This parser converts the input query to the equivalent SymPy readable code, which is then
469 processed by SymPy, and the result is finally printed with the built-in \LaTeX output and rendered
470 by the SymPy Gamma web application.

471 12 COMPARISON WITH MATHEMATICA

472 Wolfram Mathematica is a popular proprietary CAS that features highly advanced algorithms,
473 has a core written in C++ [12], and interprets its own programming language, Wolfram Language.

474 Analogous to Lisp S-expressions, Mathematica uses its own style of M-expressions, which
475 are arrays of either atoms or other M-expressions. The first element of the expression identifies
476 the type of the expression and is indexed by zero, and the first argument is indexed starting
477 with one. In SymPy, expression arguments are stored in a Python tuple (that is, an immutable
478 array), while the expression type is identified by the type of the object storing the expression.

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

483 Unlike SymPy, Mathematica's expressions are mutable: one can change parts of the expression
484 tree without the need of creating a new object. The mutability of Mathematica expressions
485 allows for a lazy updating of any references to a given data structure.

486 Products in Mathematica are determined by some built in node types, such as `Times`, `Dot`,
487 and others. `Times` is a representation of the `*` operator, and is always meant to represent a
488 commutative product operator. The other notable product is `Dot`, which represents the `.` operator.
489 This product represents matrix multiplication. It is not commutative. Unlike Mathematica,
490 SymPy determines commutativity with respect to multiplication from the expression type of the
491 factors. Mathematica puts the `Orderless` attribute on the expression type.

492 Regarding associative expressions, SymPy handles associativity of sums and products by
493 automatically flattening them, Mathematica specifies the `Flat` attribute on the expression type.

494 Mathematica relies heavily on pattern matching—even the so-called equivalent of function
495 declaration is in reality the definition of a pattern generating an expression tree transformation
496 on input expressions. Mathematica's pattern matching is sensitive to associative, commutative,
497 and one-identity properties of its expression tree nodes. SymPy has various ways to perform
498 pattern matching. All of them play a lesser role in the CAS than in Mathematica and are
499 basically available as a tool to rewrite expressions. The differential equation solver in SymPy
500 somewhat relies on pattern matching to identify differential equation types, but it is envisaged to
501 replace that strategy with analysis of Lie symmetries in the future. Mathematica's real advantage
502 is the ability to add (at runtime) new overloading to the expression builder or specific subnodes.
503 Consider for example:

```
504 In[1]:= Unprotect[Plus]
505 Out[1]= {Plus}
506
507 In[2]:= Sin[x_]^2 + Cos[y_]^2 := 1
508
509 In[3]:= x + Sin[t]^2 + y + Cos[t]^2
510 Out[3]= 1 + x + y
```

511 This expression in Mathematica defines a substitution rule that overloads the functionality of
512 the `Plus` node (the node for additions in Mathematica). A symbol with a trailing underscore is
513 treated as a wildcard. Although one may wish to keep this identity unevaluated, this example
514 clearly illustrates the potential to define one's own immediate transformation rules. In SymPy,
515 the operations constructing the addition node in the expression tree are Python class constructors
516 and cannot be modified at runtime.² The way SymPy deals with extending the missing runtime

²Nonetheless, Python supports monkey patching but it is a discouraged programming pattern.

overloadability functionality is by subclassing the node types: subclasses may redefine the class constructor to yield the proper extended functionality.

Unlike SymPy, Mathematica does not support type inheritance or polymorphism [2]. 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.

While Mathematica interprets nested lists as matrices whenever the sublists have the same length, matrices in SymPy are a type in their own right, allowing ordinary operators and functions (like multiplication and exponentiation) to be used as they traditionally are in mathematics.

```
>>> a, b = symbols('a b')
>>> exp(Matrix([[1, 1], [0, 2]])) * Matrix([a, b])
Matrix([
[E*a + b*(-E + exp(2))],
[
b*exp(2)]])
```

Using the standard multiplication in Mathematica performs an element-wise product and calling the exponential function `Exp` on a matrix returns an element-wise exponentiation of its elements.

Unevaluated expressions in Mathematica can be achieved in various ways, most commonly with the `HoldForm` or `Hold` nodes, that block the evaluation of subnodes by the parser. 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.

In Mathematica, the operator `==` returns a boolean whenever it is able to immediately evaluate the truth 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 a mathematical equality in SymPy it is necessary to explicitly construct an instance of 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 `^`.

SymPy's use of floating-point numbers is similar to that of most other CAS's, including Maple and Maxima. By contrast, Mathematica uses a form of significance arithmetic [11] for approximate numbers. This offers further protection against numerical errors, although it comes with its own set of problems (for a critique of significance arithmetic, see Fateman [2]). Internally, SymPy's `evalf` method works similarly to Mathematica's significance arithmetic, but the semantics are isolated from the rest of the system.

REFERENCES

- [Brent and Zimmermann] Brent, R. P. and Zimmermann, P. *Modern Computer Arithmetic*. Cambridge University Press, version 0.5.1 edition.
- [2] Fateman, R. J. (1992). A review of Mathematica. *Journal of Symbolic Computation*, 13(5):545–579.
- [3] Ferguson, H. R. P., Bailey, D. H., and Arno, S. (1999). Analysis of PSLQ, an integer relation finding algorithm. *Mathematics of Computation*, 68(225):351–369.
- [4] Gruntz, D. (1996). *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland.
- [5] Gruntz, D. and Koepf, W. (1993). Formal power series.
- [6] Manssur, L. R. U., Portugal, R., and Svaite, B. F. (2002). Group-theoretic approach for symbolic tensor manipulation. *International Journal of Modern Physics C*, 13.
- [7] Martín-García, J. (2002-2016). xAct, efficient tensor computer algebra.
- [8] Moskewicz, M., Madigan, C., and Malik, S. (2008). Method and system for efficient implementation of boolean satisfiability. US Patent 7,418,369.
- [9] Peeters, K. (2007). Cadabra: a field-theory motivated symbolic computer algebra system. *Computer Physics Communications*.
- [10] Rocklin, M. and Terrel, A. R. (2012). Symbolic statistics with SymPy. *Computing in Science and Engineering*, 14.
- [11] Sofroniou, M. and Spaletta, G. (2005). Precise numerical computation. *Journal of Logic and Algebraic Programming*, 64(1):113–134.

⁵⁷⁰ [12] Wolfram, S. (2003). *The Mathematica Book*. Wolfram Media, Champaign, IL, USA, fifth
⁵⁷¹ edition.