

# 1 SymPy: Symbolic Computing in Python

## 2 Supplementary material

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

```
4 >>> from sympy import *  
5 >>> x, y, z = symbols('x y z')
```

## 6 1 LIMITS: THE GRUNTZ ALGORITHM

7 SymPy calculates limits using the Gruntz algorithm, as described in [7]. The basic idea is as  
8 follows: any limit can be converted to a limit  $\lim_{x \rightarrow \infty} f(x)$  by substitutions like  $x \rightarrow \frac{1}{x}$ . Then  
9 the subexpression  $\omega$  (that converges to zero as  $x \rightarrow \infty$  faster than all other subexpressions) is  
10 identified in  $f(x)$ , and  $f(x)$  is expanded into a series with respect to  $\omega$ . Any positive powers  
11 of  $\omega$  converge to zero (while negative powers indicate an infinite limit) and any constant term  
12 independent of  $\omega$  determines the limit. When a constant term still depends on  $x$  the Gruntz  
13 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  $\infty$  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:

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

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

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

20 Next, an arbitrary item  $\omega$  is taken from mrsv set that converges to zero for  $x \rightarrow \infty$  and doesn't  
21 have subexpressions in the given mrsv set. If such a term is not present in the mrsv set (i.e.,  
22 all terms converge to infinity instead of zero), the relation  $f(x) \sim \frac{1}{f(x)}$  can be used. In the  
23 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

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

24 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

```
25 >>> from sympy.series.gruntz import mrv, rewrite
26 >>> m = mrv(exp(x+2*exp(-x))-exp(x) + 1/x, x)
27 >>> w = Symbol('w')
28 >>> rewrite(m[1], m[0], x, w)[0]
29 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  $\omega$ :

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

Since  $\omega$  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):

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

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

```
33 >>> limit(exp(x+2*exp(-x))-exp(x) + 1/x, x, oo)
34 2
```

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

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

35 The positive powers of  $\omega$  are zero. If there are any negative powers of  $\omega$ , then the result of the  
36 limit is infinity, otherwise the limit is equal to  $\lim_{x \rightarrow \infty} C_0(x)$ . The expression  $C_0(x)$  is always simpler  
37 than original  $f(x)$ , same is true for limits, arising in the rewrite stage (3), so the algorithm  
38 converges. A proof of this and further details on the algorithm are given in Gruntz's PhD  
39 thesis [7].

## 20 2 SERIES

### 41 2.1 Series Expansion

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

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

<sup>1</sup>To 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.

```

48 >>> series(sin(x+y) + cos(x*y), x, 0, 2)
49 1 + sin(y) + x*cos(y) + 0(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.

```

60 >>> from sympy.polys.ring_series import rs_sin
61 >>> R, t = ring('t', QQ)
62 >>> rs_sin(t**2 + t, t, 5)
63 -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:

```

71 >>> from sympy.polys.ring_series import rs_series
72 >>> from sympy.abc import a, b
73 >>> rs_series(sin(a + b), a, 4)
74 -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 [8]. 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`:

```

81 >>> f = fps(sin(x), x, x0=0)
82 >>> f.truncate(6)
83 x - x**3/6 + x**5/120 + 0(x**6)
84 >>> f[15]
85 -x**15/1307674368000

```

## 2.3 Fourier Series

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

```

89 >>> L = symbols('L')
90 >>> expr = 2 * (Heaviside(x/L) - Heaviside(x/L - 1)) - 1
91 >>> f = fourier_series(expr, (x, 0, 2*L))
92 >>> f.truncate(3)
93 4*sin(pi*x/L)/pi + 4*sin(3*pi*x/L)/(3*pi) + 4*sin(5*pi*x/L)/(5*pi)

```

### 94 3 LOGIC

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

#### 99 3.1 Constructing boolean expressions

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

```
106 >>> e = (x & y) | z
107 >>> e.subs({x: True, y: True, z: False})
108 True
```

#### 109 3.2 CNF and DNF

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

```
113 >>> from sympy.logic.boolalg import is_dnf, is_cnf
114 >>> to_cnf((x & y) | z)
115 And(Or(x, z), Or(y, z))
116 >>> to_dnf(x & (y | z))
117 Or(And(x, y), And(x, z))
118 >>> is_cnf((x | y) & z)
119 True
120 >>> is_dnf((x & y) | z)
121 True
```

#### 122 3.3 Simplification and Equivalence

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

```
127 >>> a, b, c = symbols('a b c')
128 >>> e = a & (~a | ~b) & (a | c)
129 >>> simplify(e)
130 And(Not(b), a)
131 >>> e1 = a & (b | c)
132 >>> e2 = (x & y) | (x & z)
133 >>> bool_map(e1, e2)
134 (And(Or(b, c), a), {a: x, b: y, c: z})
```

#### 135 3.4 SAT solving

136 The submodule also supports satisfiability (SAT) checking of a given boolean expression. If an  
137 expression is satisfiable, it is possible to return a variable assignment which satisfies it. The  
138 API also supports listing all possible assignments. The SAT solver has a clause learning DPLL  
139 algorithm implemented with a watch literal scheme and VSIDS heuristic [11].

```
140 >>> satisfiable(a & (~a | b) & (~b | c) & ~c)
141 False
142 >>> satisfiable(a & (~a | b) & (~b | c) & c)
143 {a: True, b: True, c: True}
```

## 4 DIOPHANTINE EQUATIONS

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

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

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

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

```
>>> from sympy.solvers.diophantine import *
>>> diophantine(2*x + 3*y - 5)
set([(3*t_0 - 5, -2*t_0 + 5)])

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

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

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

>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
set([(-16*p**2 + 28*p*q + 20*q**2,
3*p**2 + 38*p*q - 25*q**2,
4*p**2 - 24*p*q + 68*q**2)])

>>> x1, x2, x3, x4, x5, x6 = symbols('x1, x2, x3, x4, x5, x6')
>>> diophantine(9*x1**2 + 16*x2**2 + x3**2 + 49*x4**2 + 4*x5**2 - 25*x6**2)
set([(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5,
420*t2*t5, 60*t3*t5, 210*t4*t5,
42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)])

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

## 5 SETS

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

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

SymPy has the following atomic set classes:

194     • **EmptySet** represents the empty set  $\emptyset$ .

195     • **UniversalSet** is an abstract “universal set” of which everything is a member. The union of  
196       the universal set with any set gives the universal set and the intersection gives the other  
197       set itself.

198     • **FiniteSet** is functionally equivalent to Python’s built in **set** object. Its members can be  
199       any SymPy object including other sets.

200     • **Integers** represents the set of integers  $\mathbb{Z}$ .

201     • **Naturals** represents the set of natural numbers  $\mathbb{N}$ , i.e., the set of positive integers.

202     • **Naturals0** represents the set of whole numbers  $\mathbb{N}_0$ , which are all the non-negative integers.

203     • **Range** represents a range of integers. A range is defined by specifying a start value, an end  
204       value, and a step size. The enumeration of a **Range** object is functionally equivalent to  
205       Python’s **range** except it supports infinite endpoints, allowing the representation of infinite  
206       ranges.

207     • **Interval** represents an interval of real numbers. It is defined by giving the start and the  
208       end points and by specifying if the interval is open or closed on the respective ends.

209     Other than unevaluated classes of **Union**, **Intersection**, and **Complement** operations, SymPy  
210     has the following set classes.

211     • **ProductSet** defines the Cartesian product of two or more sets. The product set is useful  
212       when representing higher dimensional spaces. For example, to represent a three-dimensional  
213       space, SymPy uses the Cartesian product of three real sets.

214     • **ImageSet** represents the image of a function when applied to a particular set. The image  
215       set of a function  $F$  with respect to a set  $S$  is  $\{F(x) \mid x \in S\}$ . SymPy uses image sets to  
216       represent sets of infinite solutions of equations such as  $\sin(x) = 0$ .

217     • **ConditionSet** represents a subset of a set whose members satisfy a particular condition.  
218       The subset of set  $S$  given by the condition  $H$  is  $\{x \mid H(x), x \in S\}$ . SymPy uses condition  
219       sets to represent the set of solutions of equations and inequalities, where the equation or  
220       the inequality is the condition and the set is the domain over which it is being solved.

221     A few other classes are implemented as special cases of the classes described above. The set of  
222     real numbers, **Reals**, is implemented as a special case of **Interval**. **ComplexRegion** is implemented  
223     as a special case of **ImageSet**. **ComplexRegion** supports both polar and rectangular representation  
224     of regions on the complex plane.

## 225   6 STATISTICS

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

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

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

```

240 >>> from sympy.stats import Die, P, E
241 >>> X, Y = Die("X"), Die("Y")
242 >>> P(Eq(X, 6) & Eq(Y, 6))
243 1/36
244 >>> P(X>Y)
245 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:

```

248 >>> E(X, X+Y<5)
249 5/3

```

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

```

252 >>> from sympy.stats import Maxwell, density
253 >>> kT, m, x = symbols("kT m x", positive=True)
254 >>> v = Maxwell("v", sqrt(kT/m))
255 >>> E(v) # mean velocity
256 2*sqrt(2)*sqrt(kT)/(sqrt(pi)*sqrt(m))
257 >>> E(v, evaluate=False) # unevaluated mean velocity
258 Integral(sqrt(2)*m**(3/2)*v**3*exp(-m*v**2/(2*kT))/(sqrt(pi)*kT**(3/2)),
259 (v, 0, oo))
260 >>> E(m*v**2/2) # mean energy
261 3*kT/2
262 >>> solve(density(v)(x).diff(x), x)[0] # most probable velocity
263 sqrt(2)*sqrt(kT)/sqrt(m)

```

More information on the `sympy.stats` submodule can be found in [13].

## 265 7 CATEGORY THEORY

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

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

272 As of version 1.0, SymPy only implements the first goal, while a partially working draft  
 273 of implementation of the second goal is available at [https://github.com/scolobb/sympy/tree/](https://github.com/scolobb/sympy/tree/ct4-commutativity)  
 274 `ct4-commutativity`.

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

281 The class `Diagram` captures the properties of morphisms. This class stores a family of  
 282 morphisms, the corresponding source and target objects, and, possibly, some properties of the  
 283 morphisms. Generally, no restrictions are imposed on what the properties may be—for example,  
 284 one might use strings of the form “forall”, “exists”, “unique”, etc. Furthermore, the morphisms  
 285 of a diagram are grouped into *premises* and *conclusions* in order to be able to represent logical  
 286 implications of the form “for a collection of morphisms  $P$  with properties  $p : P \rightarrow \Omega$  (the premises),  
 287 there exists a collection of morphisms  $C$  with properties  $c : C \rightarrow \Omega$  (the conclusions)”, where  $\Omega$  is

288 the universal collection of properties. Finally, the class `Category` includes a collection of diagrams  
 289 which are deemed commutative and which therefore define the properties of this category.

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

298 The second phase of diagram typesetting consists in actually drawing the picture and is  
 299 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

300  
 301 As far as the second main goal of `sympy.categories` is concerned, the principal idea consists in  
 302 automatically deciding whether a diagram is commutative or not, given a collection of “axioms”:  
 303 diagrams known to be commutative. The implementation is based on graph embeddings (injective  
 304 maps): whenever an embedding of a commutative diagram into a given diagram is found, one  
 305 concludes that the subdiagram is commutative. Deciding commutativity of the whole diagram is  
 306 therefore based (theoretically) on finding a “cover” of the target diagram by embeddings of the  
 307 axioms. The naïve implementation proved to be prohibitively slow; a better optimized version is  
 308 therefore in order, as well as application of heuristics.

## 309 8 SYMPY GAMMA

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

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

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



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:

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

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

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

Every input query from the user on SymPy Gamma is first parsed by its own parser capable of handling several different forms of function names which SymPy as a library does not support. For instance, SymPy Gamma supports queries like `sin x`, whereas SymPy will only recognise `sin(x)`.

This parser converts the input query to the equivalent SymPy readable code, which is then processed by SymPy, and the result is finally printed with the built-in L<sup>A</sup>T<sub>E</sub>X output and rendered by the SymPy Gamma web application.

## 9 SYMPY LIVE

SymPy Live is an online Python shell, which uses the Google App Engine to executes SymPy code. It is integrated in the SymPy documentation examples at <http://docs.sympy.org>.

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

## 10 COMPARISON WITH MATHEMATICA

Wolfram Mathematica is a popular proprietary CAS that features highly advanced algorithms, has a core written in C++ [16], and interprets its own programming language, Wolfram Language.

Analogous to Lisp S-expressions, Mathematica uses its own style of M-expressions, which are arrays of either atoms or other M-expressions. The first element of the expression identifies the type of the expression and is indexed by zero, and the first argument is indexed starting with one. In 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: one can change parts of the expression tree without the need of creating a new object. The mutability of Mathematica expressions allows for a lazy updating of any references to a given data structure.

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

Regarding associative expressions, SymPy handles associativity of sums and products by automatically flattening them, 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 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 differential equation types, 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 (at runtime) new overloading to the expression builder or 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). A symbol with a trailing underscore is treated as a wildcard. Although one may wish to keep this identity unevaluated, this example clearly illustrates the potential 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.<sup>2</sup> The way SymPy deals with extending the missing runtime 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 [4]. 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.

---

<sup>2</sup>Nonetheless, Python supports monkey patching but it is a discouraged programming pattern.

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.

```
>>> 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 CASs, including Maple and Maxima. By contrast, Mathematica uses a form of significance arithmetic [14] 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 [4]). Internally, SymPy's `evalf` method works similarly to Mathematica's significance arithmetic, but the semantics are isolated from the rest of the system.

## 11 OTHER PROJECTS THAT USE SYMPY

There are several projects that use SymPy as a library for implementing a part of their functionality. Some of them are listed below:

- **Cadabra:** Cadabra is a CAS designed specifically for the resolution of problems encountered in field theory.
- **Octave Symbolic:** The Octave-Forge Symbolic package adds symbolic calculation features to GNU Octave. These include common CAS tools such as algebraic operations, calculus, equation solving, Fourier and Laplace transforms, variable precision arithmetic, and other features.
- **SymPy.jl:** Provides a Julia interface to SymPy using PyCall.
- **Mathics:** Mathics is a free, general-purpose online CAS featuring Mathematica compatible syntax and functions. It is backed by highly extensible Python code, relying on SymPy for most mathematical tasks.
- **Mathpix:** An iOS App, that detects handwritten math as input, and uses SymPy Gamma to evaluate the math input and generate the relevant steps to solve the problem.
- **IKFast:** IKFast is a robot kinematics compiler provided by OpenRAVE. It analytically solves robot inverse kinematics equations and generates optimized C++ files. It uses SymPy for its internal symbolic mathematics.
- **Sage:** A CAS, visioned to be a viable free open source alternative to Magma, Maple, Mathematica and MATLAB. Sage includes many open source mathematical libraries, including SymPy.

- 435 • **SageMathCloud:** SageMathCloud is a web-based cloud computing and course manage-  
436 ment platform for computational mathematics.
- 437 • **PyDy:** Multibody Dynamics with Python.
- 438 • **galgebra:** Geometric algebra (previously `sympy.galgebra`).
- 439 • **yt:** Python package for analyzing and visualizing volumetric data (`yt.units` uses SymPy).
- 440 • **SfePy:** Simple finite elements in Python, see section 11.1.
- 441 • **Quameon:** Quantum Monte Carlo in Python.
- 442 • **Lcapy:** Experimental Python package for teaching linear circuit analysis.
- 443 • **Quantum Programming in Python:** Quantum 1D Simple Harmonic Oscillator and  
444 Quantum Mapping Gate.
- 445 • **LaTeX Expression project:** Easy  $\text{\LaTeX}$  typesetting of algebraic expressions in symbolic  
446 form with automatic substitution and result computation.
- 447 • **Symbolic statistical modeling:** Adding statistical operations to complex physical  
448 models.

### 449 11.1 SfePy

450 **SfePy** (Simple finite elements in Python), cf. [3], is a Python package for solving partial  
451 differential equations (PDEs) in 1D, 2D and 3D by the finite element (FE) method [17]. SymPy  
452 is used within this package mostly for code generation and testing, namely:

- 453 • generation of the hierarchical FE basis module, involving generation and symbolic differenti-  
454 ation of 1D Legendre and Lobatto polynomials, constructing the FE basis polynomials [15]  
455 and generating the C code;
- 456 • generation of symbolic conversion formulas for various groups of elastic constants [6]:  
457 provide any two of the Young’s modulus, Poisson’s ratio, bulk modulus, Lamé’s first  
458 parameter, shear modulus (Lamé’s second parameter) or longitudinal wave modulus and  
459 get the other ones;
- 460 • simple physical unit conversions, generation of consistent unit sets;
- 461 • testing FE solutions using method of manufactured (analytical) solutions: the differential  
462 operator of a PDE is symbolically applied and a symbolic right-hand side is created,  
463 evaluated in quadrature points, and subsequently used to obtain a numerical solution that  
464 is then compared to the analytical one;
- 465 • testing accuracy of 1D, 2D and 3D numerical quadrature formulas (cf. [1]) by generating  
466 polynomials of suitable orders, integrating them, and comparing the results with those  
467 obtained by the numerical quadrature.

## 468 12 TENSORS

469 Ongoing work to provide the capabilities of tensor computer algebra has so far produced  
470 the `sympy.tensor` submodule. It comprises three submodules whose purposes are quite differ-  
471 ent: `sympy.tensor.indexed` and `sympy.tensor.indexed_methods` support indexed symbols, `sympy.`  
472 `tensor.array` contains facilities to operate on symbolic  $N$ -dimensional arrays, and finally `sympy.`  
473 `tensor.tensor` is used to define abstract tensors. The abstract tensors submodule is inspired  
474 by xAct [10] and Cadabra [12]. Canonicalization based on the Butler-Portugal [9] algorithm  
475 is supported in SymPy. Tensor support in SymPy is currently limited to polynomial tensor  
476 expressions.

## 477 13 NUMERICAL SIMPLIFICATION

478 The `nsimplify` function in SymPy (a wrapper of `identify` in `mpmath`) attempts to find a simple  
479 symbolic expression that evaluates to the same numerical value as the given input. It works  
480 by applying a few simple transformations (including square roots, reciprocals, logarithms and  
481 exponentials) to the input and, for each transformed value, using the PSLQ algorithm [5] to  
482 search for a matching algebraic number or optionally a linear combination of user-provided base  
483 constants (such as  $\pi$ ).

```
484 >>> t = 1 / (sin(pi/5)+sin(2*pi/5)+sin(3*pi/5)+sin(4*pi/5))**2
485 >>> nsimplify(t)
486 -2*sqrt(5)/5 + 1
487 >>> nsimplify(pi, tolerance=0.01)
488 22/7
489 >>> nsimplify(1.783919626661888, [pi], tolerance=1e-12)
490 pi/(-1/3 + 2*pi/3)
```

## 491 14 EXAMPLES

### 492 14.1 Simplification

- 493 • `expand`:

```
494 >>> expand((x + y)**3)
495 x**3 + 3*x**2*y + 3*x*y**2 + y**3
```

- 496 • `factor`:

```
497 >>> factor(x**3 + 3*x**2*y + 3*x*y**2 + y**3)
498 (x + y)**3
```

- 499 • `collect`:

```
500 >>> collect(y*x**2 + 3*x**2 - x*y + x - 1, x)
501 x**2*(y + 3) + x*(-y + 1) - 1
```

- 502 • `cancel`:

```
503 >>> cancel((x**2 + 2*x + 1)/(x**2 - 1))
504 (x + 1)/(x - 1)
```

- 505 • `apart`:

```
506 >>> apart((x**3 + 4*x - 1)/(x**2 - 1))
507 x + 3/(x + 1) + 2/(x - 1)
```

- 508 • `trigsimp`:

```
509 >>> trigsimp(cos(x)**2*tan(x) - sin(2*x))
510 -sin(2*x)/2
```

### 511 14.2 Polynomials

- 512 • `Factorization`:

```
513 >>> t = symbols('t')
514 >>> f = (2115*x**4*y + 45*x**3*z**3*t**2 - 45*x**3*t**2 -
515 ...      423*x*y**4 - 47*x*y**3 + 141*x*y*z**3 + 94*x*y*z*t -
516 ...      9*y**3*z**3*t**2 + 9*y**3*t**2 - y**2*z**3*t**2 +
```

```

517     ...      y**2*t**2 + 3*z**6*t**2 + 2*z**4*t**3 - 3*z**3*t**2 -
518     ...      2*z*t**3)
519 >>> factor(f)
520 (t**2*z**3 - t**2 + 47*x*y)*(2*t*z + 45*x**3 - 9*y**3 - y**2 +
521     3*z**3)

```

- Gröbner bases:

```

523 >>> x0, x1, x2 = symbols('x:3')
524 >>> I = [x0 + 2*x1 + 2*x2 - 1,
525     ...      x0**2 + 2*x1**2 + 2*x2**2 - x0,
526     ...      2*x0*x1 + 2*x1*x2 - x1]
527 >>> groebner(I, order='lex')
528 GroebnerBasis([7*x0 - 420*x2**3 + 158*x2**2 + 8*x2 - 7,
529     7*x1 + 210*x2**3 - 79*x2**2 + 3*x2,
530     84*x2**4 - 40*x2**3 + x2**2 + x2], x0, x1, x2, domain='ZZ',
531     order='lex')

```

- Root isolation:

```

533 >>> f = 7*z**4 - 19*z**3 + 20*z**2 + 17*z + 20
534 >>> intervals(f, all=True, eps=0.001)
535 ([],
536     [((-425/1024 - 625*I/1024, -1485/3584 - 2185*I/3584), 1),
537     ((-425/1024 + 2185*I/3584, -1485/3584 + 625*I/1024), 1),
538     ((3175/1792 - 2605*I/1792, 1815/1024 - 10415*I/7168), 1),
539     ((3175/1792 + 10415*I/7168, 1815/1024 + 2605*I/1792), 1)])

```

### 14.3 Solvers

- Single solution:

```

542 >>> solveset(x - 1, x)
543 {1}

```

- Finite solution set, quadratic equation:

```

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

```

- No solution:

```

548 >>> solveset(1, x)
549 EmptySet()

```

- Interval solution:

```

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

```

- Infinitely many solutions:

```

554 >>> solveset(x - x, x, domain=S.Reals)
555 (-oo, oo)
556 >>> solveset(x - x, x, domain=S.Complexes)
557 S.Complexes

```

- Linear systems (linsolve)

```

558
559     >>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
560     >>> b = Matrix([3, 6, 9])
561     >>> linsolve((A, b), x, y, z)
562     {(-1, 2, 0)}
563     >>> linsolve(Matrix([[1, 1, 1, 1], [1, 1, 2, 3]]), (x, y, z))
564     {(-y - 1, y, 2)}

```

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

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

```

567     >>> solve([x**2 + y**2 - 16, 4*x - y**2 + 6], x, y)
568     [(-2 + sqrt(14), -sqrt(-2 + 4*sqrt(14))),
569      (-2 + sqrt(14), sqrt(-2 + 4*sqrt(14))),
570      (-sqrt(14) - 2, -I*sqrt(2 + 4*sqrt(14))),
571      (-sqrt(14) - 2, I*sqrt(2 + 4*sqrt(14)))]

```

- Transcendental equations:

```

573     >>> solve((x + log(x))**2 - 5*(x + log(x)) + 6, x)
574     [LambertW(exp(2)), LambertW(exp(3))]
575     >>> solve(x**3 + exp(x))
576     [-3*LambertW((-1)**(2/3)/3)]

```

## 14.4 Matrices

- Matrix expressions

```

579     >>> m, n, p = symbols('m n p', integer=True)
580     >>> R = MatrixSymbol('R', m, n)
581     >>> S = MatrixSymbol('S', n, p)
582     >>> T = MatrixSymbol('T', m, p)
583     >>> U = R*S + 2*T
584     >>> U.shape
585     (m, p)
586     >>> U[0, 1]
587     2*T[0, 1] + Sum(R[0, _k]*S[_k, 1], (_k, 0, n - 1))

```

- Block Matrices

```

589     >>> n, m, l = symbols('n m l')
590     >>> X = MatrixSymbol('X', n, n)
591     >>> Y = MatrixSymbol('Y', m, m)
592     >>> Z = MatrixSymbol('Z', n, m)
593     >>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
594     >>> B
595     Matrix([
596     [X, Z],
597     [0, Y]])
598     >>> B[0, 0]
599     X[0, 0]
600     >>> B.shape
601     (m + n, m + n)

```

## 15 REFERENCES

### REFERENCES

- [1] Abramowitz, M. and Stegun, I. A. (1964). *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover Publications, New York, NY, USA, ninth printing edition.
- [Brent and Zimmermann] Brent, R. P. and Zimmermann, P. *Modern Computer Arithmetic*. Cambridge University Press, version 0.5.1 edition.
- [3] Cimrman, R. (2014). SfePy - write your own FE application. In de Buyl, P. and Varoquaux, N., editors, *Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013)*, pages 65–70. <http://arxiv.org/abs/1404.6391>.
- [4] Fateman, R. J. (1992). A review of Mathematica. *Journal of Symbolic Computation*, 13(5):545–579.
- [5] 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.
- [6] Fung, Y. C. (1993). *A first course in continuum mechanics*. Pearson, third edition edition.
- [7] Gruntz, D. (1996). *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland.
- [8] Gruntz, D. and Koepf, W. (1993). Formal power series.
- [9] Manssur, L. R. U., Portugal, R., and Svaiter, B. F. (2002). Group-theoretic approach for symbolic tensor manipulation. *International Journal of Modern Physics C*, 13.
- [10] Martín-García, J. (2002-2016). xAct, efficient tensor computer algebra.
- [11] Moskewicz, M., Madigan, C., and Malik, S. (2008). Method and system for efficient implementation of boolean satisfiability. US Patent 7,418,369.
- [12] Peeters, K. (2007). Cadabra: a field-theory motivated symbolic computer algebra system. *Computer Physics Communications*.
- [13] Rocklin, M. and Terrel, A. R. (2012). Symbolic statistics with SymPy. *Computing in Science and Engineering*, 14.
- [14] Sofroniou, M. and Spaletta, G. (2005). Precise numerical computation. *Journal of Logic and Algebraic Programming*, 64(1):113–134.
- [15] Solin, P., Segeth, K., and Dolezel, I. (2003). *Higher-Order Finite Element Methods*. Chapman & Hall / CRC Press.
- [16] Wolfram, S. (2003). *The Mathematica Book*. Wolfram Media, Champaign, IL, USA, fifth edition.
- [17] Zienkiewicz, O., Taylor, R., and Zhu, J. (2013). *The Finite Element Method: Its Basis and Fundamentals*. Butterworth-Heinemann, seventh edition edition.