

SymPy: Symbolic Computing in Python

Aaron Meurer¹, Christopher P. Smith², Mateusz Paprocki³, Ondřej Čertík⁴,
Sergey B. Kirpichev⁵, Matthew Rocklin⁶, AMiT Kumar⁷, Sergiu Ivanov⁸,
Jason K. Moore⁹, Sartaj Singh¹⁰, Thilina Rathnayake¹¹, Sean Vig¹², Brian
E. Granger¹³, Richard P. Muller¹⁴, Francesco Bonazzi¹⁵, Harsh Gupta¹⁶,
Shivam Vats¹⁷, Fredrik Johansson¹⁸, Fabian Pedregosa¹⁹, Matthew J.
Curry²⁰, Andy R. Terrel²¹, Štěpán Roučka²², Ashutosh Saboo²³, Isuru
Fernando²⁴, Sumith Kulal²⁵, Robert Cimrman²⁶, and Anthony Scopatz²⁷

¹University of South Carolina, Columbia, SC 29201 (asmeurer@gmail.com).

²Polar Semiconductor, Inc., Bloomington, MN 55425 (smichr@gmail.com).

³Continuum Analytics, Inc., Austin, TX 78701 (mattpap@gmail.com).

⁴Los Alamos National Laboratory, Los Alamos, NM 87545 (certik@lanl.gov). The Los Alamos National Laboratory is operated by Los Alamos National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy under Contract No. DE-AC52-06NA25396.

⁵Moscow State University, Faculty of Physics, Leninskie Gory, Moscow, 119991, Russia (skirpichev@gmail.com).

⁶Continuum Analytics, Inc., Austin, TX 78701 (mrocklin@gmail.com).

⁷Delhi Technological University, Shahbad Daultpur, Bawana Road, New Delhi 110042, India (dtu.amit@gmail.com).

⁸Université Paris Est Créteil, 61 av. Général de Gaulle, 94010 Créteil, France (sergiu.ivanov@u-pec.fr).

⁹University of California, Davis, Davis, CA 95616 (jkm@ucdavis.edu).

¹⁰Indian Institute of Technology (BHU), Varanasi, Uttar Pradesh 221005, India (singhsartaj94@gmail.com).

¹¹University of Moratuwa, Bandaranayake Mawatha, Katubedda, Moratuwa 10400, Sri Lanka (thilinarmtb.10@cse.mrt.ac.lk).

¹²University of Illinois at Urbana-Champaign, Urbana, IL 61801 (sean.v.775@gmail.com).

¹³California Polytechnic State University, San Luis Obispo, CA 93407 (ellisonbg@gmail.com).

¹⁴Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185 (rmuller@sandia.gov). 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.

¹⁵Max Planck Institute of Colloids and Interfaces, Department of Theory and Bio-Systems, Science Park Golm, 14424 Potsdam, Germany (francesco.bonazzi@mpikg.mpg.de).

¹⁶Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India (hargup@protonmail.com).

¹⁷Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India (shivamvats.iitkgp@gmail.com).

¹⁸INRIA Bordeaux-Sud-Ouest – LFANT project-team, 200 Avenue de la Vieille Tour, 33405 Talence, France (fredrik.johansson@gmail.com).

¹⁹INRIA – SIERRA project-team, 2 Rue Simone IFF, 75012 Paris, France (f@bianp.net).

²⁰Department of Physics and Astronomy, University of New Mexico, Albuquerque, NM 87131 (mattjcurry@gmail.com).

²¹Fashion Metric, Inc, Austin, TX 78681 (andy.terrel@gmail.com).

²²Faculty of Mathematics and Physics, Charles University in Prague, V Holešovičkách 2, 180 00 Praha, Czech Republic (stepan.roucka@mff.cuni.cz).

²³Birla Institute of Technology and Science, Pilani, K.K. Birla Goa Campus, NH 17B Bypass Road, Zuarinagar, Sancoale, Goa 403726, India (ashutosh.saboo96@gmail.com).

²⁴University of Moratuwa, Bandaranayake Mawatha, Katubedda, Moratuwa 10400, Sri Lanka (isuru.11@cse.mrt.ac.lk).
²⁵Indian Institute of Technology Bombay, Powai, Mumbai, Maharashtra 400076, India (sumith@cse.iitb.ac.in).
²⁶New Technologies – Research Centre, University of West Bohemia, Univerzitní 8, 306 14 Plzeň, Czech Republic (cimirman3@ntc.zcu.cz).
²⁷University of South Carolina, Columbia, SC 29201 (scopatz@cec.sc.edu).

ABSTRACT

SymPy is an open source computer algebra system written in pure Python. It is built with a focus on extensibility and ease of use, through both interactive and programmatic applications. These characteristics have led SymPy to become a popular symbolic library for the scientific Python ecosystem. This paper presents the architecture of SymPy, a description of its features, and a discussion of select domain specific submodules. The supplementary materials provide additional examples and further outline details of the architecture and features of SymPy.

Keywords: symbolic, Python, computer algebra system

1 INTRODUCTION

SymPy is a full featured computer algebra system (CAS) written in the Python [?] programming language. It is free and open source software, licensed under the 3-clause BSD license [?]. The SymPy project was started by Ondřej Čertík in 2005, and it has since grown to over 500 contributors. Currently, SymPy is developed on GitHub using a bazaar community model [?]. The accessibility of the codebase and the open community model allow SymPy to rapidly respond to the needs of users and developers.

Python is a dynamically typed programming language that has a focus on ease of use and readability.¹ Due in part to this focus, it has become a popular language for scientific computing and data science, with a broad ecosystem of libraries [?]. SymPy is itself used by many libraries and tools to support research within a variety of domains, such as SageMath [?] (pure and applied mathematics), yt [?] (astronomy and astrophysics), PyDy [?] (multibody dynamics), and SfePy [?] (finite elements).

Unlike many CAS's, SymPy does not invent its own programming language. Python itself is used both for the internal implementation and end user interaction. By using the operator overloading functionality of Python, SymPy follows the embedded domain specific language paradigm proposed by Hudak [?]. The exclusive usage of a single programming language makes it easier for people already familiar with that language to use or develop SymPy. Simultaneously, it enables developers to focus on mathematics, rather than language design. SymPy officially supports Python 2.6, 2.7 and 3.2–3.5.

SymPy is designed with a strong focus on usability as a library. Extensibility is important in its application program interface (API) design. Thus, SymPy makes no attempt to extend the Python language itself. The goal is for users of SymPy to be able to include SymPy alongside other Python libraries in their workflow, whether that be in an interactive environment or as a programmatic part in a larger system.

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

The remainder of this paper discusses key components of the SymPy library. Section ?? enumerates the features of SymPy and takes a closer look at some of the important ones. The section ?? looks at the numerical features of SymPy and its dependency library, mpmath.

¹This paper assumes a moderate familiarity with the Python programming language.

100 Section ?? looks at the domain specific physics submodules for performing symbolic and numerical
 101 calculations in classical mechanics and quantum mechanics. Section ?? discusses the architecture
 102 of SymPy. Conclusions and future directions for SymPy are given in section ?. All examples in
 103 this paper use SymPy version 1.0 and mpmath version 0.19.

104 The following statement imports all SymPy functions into the global Python namespace.²
 105 From here on, all examples in this paper assume that this statement has been executed:³

```
106 >>> from sympy import *
```

107 All examples could be tested on the SymPy Live instance, that is an online Python shell,
 108 which uses the Google App Engine to execute SymPy code.

109 2 OVERVIEW OF CAPABILITIES

110 This section gives a basic introduction of SymPy, and lists its features. A few features—
 111 assumptions, simplification, calculus, polynomials, printers, solvers, and matrices—are core
 112 components of SymPy and are discussed in depth. Many other features are discussed in depth in
 113 the supplementary material.

114 2.1 Basic Usage

115 Symbolic variables, called symbols, must be defined and assigned to Python variables before they
 116 can be used. This is typically done through the `symbols` function, which may create multiple
 117 symbols in a single function call. For instance,

```
118 >>> x, y, z = symbols('x y z')
```

119 creates three symbols representing variables named x , y , and z . In this particular instance, these
 120 symbols are all assigned to Python variables of the same name. However, the user is free to
 121 assign them to different Python variables, while representing the same symbol, such as `a`, `b`,
 122 `c = symbols('x y z')`. In order to minimize potential confusion, though, all examples in this
 123 paper will assume that the symbols x , y , and z have been assigned to Python variables identical
 124 to their symbolic names.

125 Expressions are created from symbols using Python’s mathematical syntax. For instance, the
 126 following Python code creates the expression $(x^2 - 2x + 3)/y$. Note that the expression remains
 127 unevaluated: it is represented symbolically.

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

130 Importantly, SymPy expressions are immutable. This simplifies the design of SymPy by
 131 allowing expression interning. It also enables expressions to be hashed, that is used to implement
 132 caching in SymPy.

133 2.2 List of Features

134 Although SymPy’s extensive feature set cannot be covered in-depth in this paper, calculus and
 135 other bedrock areas are discussed in their own subsections. Additionally, Table ?? gives a compact
 136 listing of all major capabilities present in the SymPy codebase. This grants a sampling from the
 137 breadth of topics and application domains that SymPy services. Unless stated otherwise, all
 138 features noted in Table ?? are symbolic in nature. Numeric features are discussed in Section ?.

Table 1. SymPy Features and Descriptions

²`import *` has been used here to aid the readability of the paper, but is best to avoid such wildcard import statements in production code, as they make it unclear which names are present in the namespace. Furthermore, imported names could clash with already existing imports from another package. For example, SymPy, the standard Python `math` library, and NumPy all define the `exp` function, but only the SymPy one will work with SymPy symbolic expressions.

³The three greater-than signs denote the user input for the Python interactive session, with the result, if there is one, shown on the next line.

Feature (submodules)	Description
Calculus (<code>sympy.core</code> , <code>sympy.series</code> , <code>sympy.integrals</code>)	Algorithms for computing derivatives, integrals, and limits.
Category Theory (<code>sympy.categories</code>)	Representation of objects, morphisms, and diagrams. Tools for drawing diagrams with Xy-pic.
Code Generation (<code>sympy.printing</code> , <code>sympy.codegen</code>)	Generation of compilable and executable code in a variety of different programming languages from expressions directly. Target languages include C, Fortran, Julia, JavaScript, Mathematica, MATLAB and Octave, Python, and Theano.
Combinatorics & Group Theory (<code>sympy.combinatorics</code>)	Permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [?], and Prufer sequences [?].
Concrete Math (<code>sympy.concrete</code>)	Summation, products, tools for determining whether summation and product expressions are convergent, absolutely convergent, hypergeometric, and for determining other properties; computation of Gosper's normal form [?] for two univariate polynomials.
Cryptography (<code>sympy.crypto</code>)	Block and stream ciphers, including shift, Affine, substitution, Vigenère's, Hill's, bifid, RSA, Kid RSA, linear-feedback shift registers, and Elgamal encryption.
Differential Geometry (<code>sympy.diffgeom</code>)	Representations of manifolds, metrics, tensor products, and coordinate systems in Riemannian and pseudo-Riemannian geometries [?].
Geometry (<code>sympy.geometry</code>)	Representations 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 objects.
Lie Algebras (<code>sympy.liealgebras</code>)	Representations of Lie algebras and root systems.
Logic (<code>sympy.logic</code>)	Boolean expressions, equivalence testing, satisfiability, and normal forms.
Matrices (<code>sympy.matrices</code>)	Tools for creating matrices of symbols and expressions. Both sparse and dense representations, as well as symbolic linear algebraic operations (e.g., inversion and factorization), are supported.
Matrix Expressions (<code>sympy.matrices.expressions</code>)	Matrices with symbolic dimensions (unspecified entries). Block matrices.
Number Theory (<code>sympy.ntheory</code>)	Prime number generation, primality testing, integer factorization, continued fractions, Egyptian fractions, modular arithmetic, quadratic residues, partitions, binomial and multinomial coefficients, prime number tools, hexadecimal digits of π , and integer factorization.
Plotting (<code>sympy.plotting</code>)	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 (<code>sympy.polys</code>)	Polynomial algebras over various coefficient domains. Functionality ranges from simple operations (e.g., polynomial division) to advanced computations (e.g., Gröbner bases [?] and multivariate factorization over algebraic number domains).
Printing (<code>sympy.printing</code>)	Functions for printing SymPy expressions in the terminal with ASCII or Unicode characters and converting SymPy expressions to L ^A T _E X and MathML.

Quantum Mechanics (<code>sympy.physics.quantum</code>)	Quantum states, bra-ket notation, operators, basis sets, representations, tensor products, inner products, outer products, commutators, anticommutators, and specific quantum system implementations.
Series (<code>sympy.series</code>)	Series expansion, sequences, and limits of sequences. This includes Taylor, Laurent, and Puiseux series as well as special series, such as Fourier and formal power series.
Sets (<code>sympy.sets</code>)	Representations of empty, finite, and infinite sets (including special sets such as the natural, integer, and complex numbers). Operations on sets such as union, intersection, Cartesian product, and building sets from other sets are supported.
Simplification (<code>sympy.simplify</code>)	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 (<code>sympy.solvers</code>)	Functions for symbolically solving equations, systems of equations, both linear and non-linear, inequalities, ordinary differential equations, partial differential equations, Diophantine equations, and recurrence relations.
Special Functions (<code>sympy.functions</code>)	Implementations of 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 (<code>sympy.stats</code>)	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 (<code>sympy.tensor</code>)	Symbolic manipulation of indexed objects.
Vectors (<code>sympy.vector</code>)	Basic operations on vectors and differential calculus with respect to 3D Cartesian coordinate systems.

2.3 Assumptions

SymPy performs logical inference through its assumptions system. The assumptions system allows users to specify that symbols have certain common mathematical properties, such as being positive, imaginary, or integral. SymPy is careful to never perform simplifications on an expression unless the assumptions allow them. For instance, the identity $\sqrt{t^2} = t$ holds if t is nonnegative ($t \geq 0$). However, for general complex t , no such identity holds.

By default, SymPy performs all calculations assuming that symbols are complex valued. This assumption makes it easier to treat mathematical problems in full generality.

```
>>> t = Symbol('t')
>>> sqrt(t**2)
sqrt(t**2)
```

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

Assumptions are set on `Symbol` objects when they are created. For instance `Symbol('t', positive=True)` will create a symbol named t that is assumed to be positive.

```

155 >>> t = Symbol('t', positive=True)
156 >>> sqrt(t**2)
157 t

```

Some of the common assumptions that SymPy allows are **positive**, **negative**, **real**, **nonpositive**, **integer**, **prime** and **commutative**.⁴ Assumptions on any object can be checked with the `is_assumption` attributes, like `t.is_positive`.

Assumptions are only needed to restrict a domain so that certain simplifications can be performed. They are not required to make the domain match the input of a function. For instance, one can create the object $\sum_{n=0}^m f(n)$ as `Sum(f(n), (n, 0, m))` without setting `integer=True` when creating the Symbol object `n`.

The assumptions system additionally has deductive capabilities. The assumptions use a three-valued logic using the Python built in objects `True`, `False`, and `None`. Note that `False` is returned if the SymPy object doesn't or can't have the assumption. For example, both `I.is_real` and `I.is_prime` return `False` for the imaginary unit `I`.

`None` represents the “unknown” case. This could mean that given assumptions do not unambiguously specify the truth of an attribute. For instance, `Symbol('x', real=True).is_positive` will give `None` because a real symbol might be positive or negative. The `None` could also mean that not enough is known or implemented to compute the given fact. For instance, `(pi + E).is_irrational` gives `None`—indeed, the rationality of $\pi + e$ is an open problem in mathematics [?].

Basic implications between the facts are used to deduce assumptions. For instance, the assumptions system knows that being an integer implies being rational.

```

177 >>> i = Symbol('i', integer=True)
178 >>> i.is_rational
179 True

```

Furthermore, expressions compute the assumptions on themselves based on the assumptions of their arguments. For instance, if `x` and `y` are both created with `positive=True`, then `(x + y).is_positive` will be `True` whereas `(x - y).is_positive` will be `None`.

2.4 Simplification

The generic way to simplify an expression is by calling the `simplify` function. It must be emphasized that simplification is not a rigorously defined mathematical operation [?]. The `simplify` function applies several simplification routines along with heuristics to make the output expression “simple”.⁵

It is often preferable to apply more directed simplification functions. These apply very specific rules to the input expression and are typically 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 common simplification functions.

Table 2. Some 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 [?]
<code>hyperexpand</code>	expand hypergeometric functions [? ?]

⁴SymPy assumes that two expressions A and B commute with each other multiplicatively, that is, $A \cdot B = B \cdot A$, unless they both have `commutative=False`. Many algorithms in SymPy require special consideration to work correctly with noncommutative products.

⁵The `measure` parameter of the `simplify` function lets the user specify the Python function used to determine how complex an expression is. The default measure function returns the total number of operations in the expression.

193 2.5 Calculus

194 SymPy provides all the basic operations of calculus, such as calculating limits, derivatives,
195 integrals, or summations.

196 Limits are computed with the `limit` function, using the Gruntz algorithm [?] for computing
197 symbolic limits and heuristics (a description of the Gruntz algorithm may be found in the
198 supplement). For example, the following computes $\lim_{x \rightarrow \infty} x \sin(\frac{1}{x}) = 1$. Note that SymPy denotes
199 ∞ as `oo`.

```
200 >>> limit(x*sin(1/x), x, oo)
201 1
```

As a more complex example, SymPy computes

$$\lim_{x \rightarrow 0} \left(2e^{\frac{1 - \cos(x)}{\sin(x)}} - 1 \right)^{\frac{\sinh(x)}{\operatorname{atan}^2(x)}} = e.$$

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

204 Derivatives are computed with the `diff` function, which recursively uses the various differen-
205 tiation rules.

```
206 >>> diff(sin(x)*exp(x), x)
207 exp(x)*sin(x) + exp(x)*cos(x)
```

Integrals are calculated with the `integrate` function. SymPy implements a combination of the Risch algorithm [?], table lookups, a reimplementa-
tion 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. The Meijer G-function algorithm and the Risch algorithm are respectively demonstrated below by the computation of

$$\int_0^\infty e^{-st} \log(t) dt = -\frac{\log(s) + \gamma}{s}$$

and

$$\int \frac{-2x^2(\log(x) + 1)e^{x^2} + (e^{x^2} + 1)^2}{x(e^{x^2} + 1)^2(\log(x) + 1)} dx = \log(\log(x) + 1) + \frac{1}{e^{x^2} + 1}.$$

```
208 >>> s, t = symbols('s t', positive=True)
209 >>> integrate(exp(-s*t)*log(t), (t, 0, oo)).simplify()
210 -(log(s) + EulerGamma)/s
211 >>> integrate((-2*x**2*(log(x) + 1)*exp(x**2) +
212 ... (exp(x**2) + 1)**2)/(x*(exp(x**2) + 1)**2*(log(x) + 1)), x)
213 log(log(x) + 1) + 1/(exp(x**2) + 1)
```

214 Summations are computed with `summation` using a combination of Gosper’s algorithm [?],
215 an algorithm that uses Meijer G-functions [? ?], and heuristics. Products are computed with
216 `product` function via a suite of heuristics.

```
217 >>> i, n = symbols('i n')
218 >>> summation(2**i, (i, 0, n - 1))
219 2**n - 1
220 >>> summation(i*factorial(i), (i, 1, n))
221 n*factorial(n) + factorial(n) - 1
```

222 Series expansions are computed with the `series` function. This example computes the power
223 series of $\sin(x)$ around $x = 0$ up to x^6 .

```

224 >>> series(sin(x), x, 0, 6)
225 x - x**3/6 + x**5/120 + O(x**6)

```

226 The supplementary material discusses series expansions methods in more depth.

227 Integrals, derivatives, summations, products, and limits that cannot be computed return
 228 unevaluated objects. These can also be created directly if the user chooses.

```

229 >>> integrate(x**x, x)
230 Integral(x**x, x)
231 >>> Sum(2**i, (i, 0, n - 1))
232 Sum(2**i, (i, 0, n - 1))

```

233 2.6 Polynomials

234 SymPy implements a suite of algorithms for polynomial manipulation, which ranges from
 235 relatively simple algorithms for doing arithmetic of polynomials, to advanced methods for
 236 factoring multivariate polynomials into irreducibles, symbolically determining real and complex
 237 root isolation intervals, or computing Gröbner bases.

238 Polynomial manipulation is useful in its own right. Within SymPy, though, it is mostly
 239 used indirectly as a tool in other areas of the library. In fact, many mathematical problems
 240 in symbolic computing are first expressed using entities from the symbolic core, preprocessed,
 241 and then transformed into a problem in the polynomial algebra, where generic and efficient
 242 algorithms are used to solve the problem. The solutions to the original problem are subsequently
 243 recovered from the results. This is a common scheme in symbolic integration or summation
 244 algorithms.

245 SymPy implements dense and sparse polynomial representations.⁶ Both are used in the uni-
 246 variate and multivariate cases. The dense representation is the default for univariate polynomials.
 247 For multivariate polynomials, the choice of representation is based on the application. The most
 248 common case for the sparse representation is algorithms for computing Gröbner bases (Buch-
 249 berger, F4, and F5) [? ? ?]. This is because different monomial orderings can be expressed easily
 250 in this representation. However, algorithms for computing multivariate GCDs or factorizations,
 251 at least those currently implemented in SymPy [?], are better expressed when the representation
 252 is dense. The dense multivariate representation is specifically a recursively-dense representation,
 253 where polynomials in $K[x_0, x_1, \dots, x_n]$ are viewed as a polynomials in $K[x_0][x_1] \dots [x_n]$. Note
 254 that despite this, the coefficient domain K , can be a multivariate polynomial domain as well.
 255 The dense recursive representation in Python gets inefficient as the number of variables increases.

256 Some examples for the `sympy.polys` submodule can be found in the supplement.

257 2.7 Printers

258 SymPy has a rich collection of expression printers. By default, an interactive Python session will
 259 render the `str` form of an expression, which has been used in all the examples in this paper so
 260 far. The `str` form of an expression is valid Python and roughly matches what a user would type
 261 to enter the expression.⁷

```

262 >>> phi0 = Symbol('phi0')
263 >>> str(Integral(sqrt(phi0), phi0))
264 'Integral(sqrt(phi0), phi0)'

```

265 A two-dimensional (2D) textual representation of the expression can be printed with
 266 monospace fonts via `pprint`. Unicode characters are used for rendering mathematical sym-
 267 bols such as integral signs, square roots, and parentheses. Greek letters and subscripts in symbol
 268 names that have Unicode code points associated are also rendered automatically.

⁶In a dense representation, the coefficients for all terms up to the degree of each variable are stored in memory. In a sparse representation, only the nonzero coefficients are stored.

⁷Many Python libraries distinguish the `str` form of an object, which is meant to be human-readable, and the `repr` form, which is meant to be valid Python that recreates the object. In SymPy, `str(expr) == repr(expr)`. In other words, the string representation of an expression is designed to be compact, human-readable, and valid Python code that could be used to recreate the expression. As it was noted in section ??, the `srepr` function prints the exact, verbose form of an expression.


```

269 >>> pprint(Integral(sqrt(phi0 + 1), phi0))
270
271 
$$\int \sqrt{\varphi_0 + 1} \, d(\varphi_0)$$


```

Alternately, the `use_unicode=False` flag can be set, which causes the expression to be printed using only ASCII characters.

```

272 >>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
273 /
274 |
275 | _____
276 | \ /  phi0 + 1  d(phi0)
277 |
278 /

```

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

```

280 >>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
281 \int \sqrt{\phi_0 + 1}\, d\phi_0

```

Users are encouraged to run the `init_printing` function at the beginning of interactive sessions, which automatically enables the best pretty printing supported by their environment. In the Jupyter Notebook or Qt Console [?], the \LaTeX printer is used to render expressions using MathJax or \LaTeX , if it is installed on the system. The 2D text representation is used otherwise.

Other printers such as MathML are also available. SymPy uses an extensible printer subsystem for customizing any given printer, and allows custom objects to define their printing behavior for any printer. The code generation functionality of SymPy relies on this subsystem to convert expressions into code in various target programming languages.

2.8 Solvers

SymPy has equation solvers that can handle ordinary differential equations, recurrence relationships, Diophantine equations⁸, and algebraic equations. There is also rudimentary support for simple partial differential equations.

There are two functions for solving algebraic equations in SymPy: `solve` and `solveset`. `solveset` has several design changes with respect to the older `solve` function. This distinction is present in order to resolve the usability issues with the previous `solve` function API while maintaining backward compatibility with earlier versions of SymPy. `solveset` only requires essential input information from the user. The function signatures of `solve` and `solveset` are

```

299 solve(f, *symbols, **flags)
300 solveset(f, symbol, domain=S.Complexes)

```

The `domain` parameter can be any set from the `sympy.sets` module (see the supplementary material for details on `sympy.sets`), but is typically either `S.Complexes` (the default) or `S.Reals`; the latter causes `solveset` to only return real solutions.

An important difference between the two functions is that the output API of `solve` varies with input (sometimes returning a Python list and sometimes a Python dictionary) whereas `solveset` always returns a SymPy set object.

Both functions implicitly assume that expressions are equal to 0. For instance, `solveset(x - 1, x)` solves $x - 1 = 0$ for x .

`solveset` is under active development as a planned replacement for `solve`. There are certain features which are implemented in `solve` that are not yet implemented in `solveset`, including multivariate systems, and some transcendental equations.

More examples of `solveset` and `solve` can be found in the supplement.

⁸See the supplementary material for an in depth discussion on the Diophantine submodule.

313 2.9 Matrices

314 Besides being an important feature in its own right, computations on matrices with symbolic
315 entries are important for many algorithms within SymPy. The following code shows some basic
316 usage of the `Matrix` class.

```
317 >>> A = Matrix([[x, x + y], [y, x]])
318 >>> A
319 Matrix([
320 [x, x + y],
321 [y, x]])
```

322 SymPy matrices support common symbolic linear algebra manipulations, including matrix
323 addition, multiplication, exponentiation, computing determinants, solving linear systems, and
324 computing inverses using LU decomposition, LDL decomposition, Gauss-Jordan elimination,
325 Cholesky decomposition, Moore-Penrose pseudoinverse, singular values, and adjugate matrix.

326 All operations are performed symbolically. For instance, eigenvalues are computed by
327 generating the characteristic polynomial using the Berkowitz algorithm and then solving it using
328 polynomial routines.

```
329 >>> A.eigenvals()
330 {x - sqrt(y*(x + y)): 1, x + sqrt(y*(x + y)): 1}
```

331 Internally these matrices store the elements as Lists of Lists (LIL), meaning the matrix is
332 stored as a list of lists of entries (effectively, the input format used to create the matrix `A` above),
333 making it a dense representation.⁹ For storing sparse matrices, the `SparseMatrix` class can be
334 used. Sparse matrices store their elements in Dictionary of Keys (DOK) format, meaning entries
335 are stored as `(row, column)` pairs mapping to the elements.

336 SymPy also supports matrices with symbolic dimension values. `MatrixSymbol` represents
337 a matrix with dimensions $m \times n$, where m and n can be symbolic. Matrix addition and
338 multiplication, scalar operations, matrix inverse, and transpose are stored symbolically as matrix
339 expressions.

340 Block matrices are also implemented in SymPy. `BlockMatrix` elements can be any matrix ex-
341 pression, including explicit matrices, matrix symbols, and other block matrices. All functionalities
342 of matrix expressions are also present in `BlockMatrix`.

343 When symbolic matrices are combined with the assumptions submodule for logical inference,
344 they provide powerful reasoning over invertibility, semi-definiteness, orthogonality, etc., which
345 are valuable in the construction of numerical linear algebra systems.

346 More examples for `Matrix` and `BlockMatrix` may be found in the supplement.

347 3 NUMERICS

348 While SymPy primarily focuses on symbolics, it is impossible to have a complete symbolic system
349 without the ability to numerically evaluate expressions. Many operations directly use numerical
350 evaluation, such as plotting a function, or solving an equation numerically. Beyond this, certain
351 purely symbolic operations require numerical evaluation to effectively compute. For instance,
352 determining the truth value of $e + 1 > \pi$ is most conveniently done by numerically evaluating
353 both sides of the inequality and checking which is larger.

354 3.1 Floating-Point Numbers

355 Floating-point numbers in SymPy are implemented by the `Float` class, which represents an
356 arbitrary-precision binary floating-point number by storing its value and precision (in bits).
357 This representation is distinct from the Python built-in `float` type, which is a wrapper around
358 machine `double` types and uses a fixed precision (53-bit).

359 Because Python `float` literals are limited in precision, strings should be used to input precise
360 decimal values:

⁹Similar to the polynomials submodule, dense here means that all entries are stored in memory, contrasted with a sparse representation where only nonzero entries are stored.

410 representing $\sum_{x=a}^b f(x)$ is represented in mpmath as `nsum(f, (a, b))`, where `f` is a numeric
 411 Python function.

412 The mpmath library supports special functions, root-finding, linear algebra, polynomial
 413 approximation, and numerical computation of limits, derivatives, integrals, infinite series, and
 414 solving ODEs. All features work in arbitrary precision and use algorithms that allow computing
 415 hundreds of digits rapidly (except in degenerate cases).

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

423 A wide array of higher mathematical functions is implemented with full support for complex
 424 values of all parameters and arguments, including complete and incomplete gamma functions,
 425 Bessel functions, orthogonal polynomials, elliptic functions and integrals, zeta and polylogarithm
 426 functions, the generalized hypergeometric function, and the Meijer G-function. The Meijer
 427 G-function instance $G_{1,3}^{3,0}(0; \frac{1}{2}, -1, -\frac{3}{2}|x)$ is a good test case [?]; past versions of both Maple and
 428 Mathematica produced incorrect numerical values for large $x > 0$. Here, mpmath automatically
 429 removes an internal singularity and compensates for cancellations (amounting to 656 bits of
 430 precision when $x = 10000$), giving correct values:

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

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

```
435 >>> meijerg([], [0], [[-S(1)/2, -1, -S(3)/2], []], 10000).evalf()
436 2.43925769071996e-94
```

437 Symbolic integration and summation often produce hypergeometric and Meijer G-function
 438 closed forms (see Subsection ??); numerical evaluation of such special functions is a useful
 439 complement to direct numerical integration and summation.

440 4 PHYSICS SUBMODULE

441 SymPy includes several submodules that allow users to solve domain specific problems. For
 442 example, a comprehensive physics submodule is included that is useful for solving problems
 443 in mechanics, optics, and quantum mechanics along with support for manipulating physical
 444 quantities with units.

445 4.1 Classical Mechanics

446 One of the core domains that SymPy supports is the physics of classical mechanics. This is in
 447 turn separated into two distinct components: vector algebra and mechanics.

448 4.1.1 Vector Algebra

449 The `sympy.physics.vector` submodule provides reference frame-, time-, and space-aware vector
 450 and dyadic objects that allow for three-dimensional operations such as addition, subtraction,
 451 scalar multiplication, inner and outer products, and cross products. The vector and dyadic
 452 objects can both be written in very compact notation that make it easy to express the vectors
 453 and dyadics in terms of multiple reference frames with arbitrarily defined relative orientations.
 454 The vectors are used to specify the positions, velocities, and accelerations of points; orientations,
 455 angular velocities, and angular accelerations of reference frames; and forces and torques. The
 456 dyadics are essentially reference frame-aware 3×3 tensors [?]. The vector and dyadic objects
 457 can be used for any one-, two-, or three-dimensional vector algebra, and they provide a strong
 458 framework for building physics and engineering tools.

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

```

464 >>> from sympy.physics.vector import ReferenceFrame
465 >>> A = ReferenceFrame('A')
466 >>> B = ReferenceFrame('B')
467 >>> C = ReferenceFrame('C')
468 >>> B.orient(A, 'body', (pi, pi/3, pi/4), 'zxz')
469 >>> C.orient(B, 'axis', (pi/2, B.x))
470 >>> v = 1*A.x + 2*B.z + 3*C.y
471 >>> v
472 A.x + 2*B.z + 3*C.y
473 >>> v.express(A)
474 A.x + 5*sqrt(3)/2*A.y + 5/2*A.z

```

4.1.2 Mechanics

The `sympy.physics.mechanics` submodule utilizes the `sympy.physics.vector` submodule to populate time-aware particle and rigid-body objects to fully describe the kinematics and kinetics of a rigid multi-body system. These objects store all of the information needed to derive the ordinary differential or differential algebraic equations that govern the motion of the system, i.e., the equations of motion. These equations of motion abide by Newton's laws of motion and can handle arbitrary kinematic constraints or complex loads. The submodule offers two automated methods for formulating the equations of motion based on Lagrangian Dynamics [?] and Kane's Method [?]. Lastly, there are automated linearization routines for constrained dynamical systems [?].

4.2 Quantum Mechanics

The `sympy.physics.quantum` submodule has extensive capabilities to solve problems in quantum mechanics, using 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, and anticommutators. 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 and defining the relevant class methods to provide system-specific logic.

Symbolic quantum operators and states may be defined, and one can perform a full range of operations with them.

```

495 >>> from sympy.physics.quantum import Commutator, Dagger, Operator
496 >>> from sympy.physics.quantum import Ket, qapply
497 >>> A = Operator('A')
498 >>> B = Operator('B')
499 >>> C = Operator('C')
500 >>> D = Operator('D')
501 >>> a = Ket('a')
502 >>> comm = Commutator(A, B)
503 >>> comm
504 [A,B]
505 >>> qapply(Dagger(comm*a)).doit()
506 -<a|*(Dagger(A)*Dagger(B) - Dagger(B)*Dagger(A))

```

Commutators can be expanded using common commutator identities:

```

508 >>> Commutator(C+B, A*D).expand(commutator=True)
509 -[A,B]*D - [A,C]*D + A*[B,D] + A*[C,D]

```

510 On top of this set of base objects, a number of specific quantum systems have been implemented
511 in a fully symbolic framework. These include:

- 512 • Many of the exactly solvable quantum systems, including simple harmonic oscillator states
513 and raising/lowering operators, infinite square well states, and 3D position and momentum
514 operators and states.
- 515 • Second quantized formalism of non-relativistic many-body quantum mechanics [?].
- 516 • Quantum angular momentum [?]. Spin operators and their eigenstates can be represented
517 in any basis and for any quantum numbers. A rotation operator representing the Wigner-D
518 matrix, which may be defined symbolically or numerically, is also implemented to rotate
519 spin eigenstates. Functionality for coupling and uncoupling of arbitrary spin eigenstates is
520 provided, including symbolic representations of Clebsch-Gordon coefficients and Wigner
521 symbols.
- 522 • Quantum information and computing [?]. Multidimensional qubit states, and a full
523 set of one- and two-qubit gates are provided and can be represented symbolically or as
524 matrices/vectors. With these building blocks, it is possible to implement a number of basic
525 quantum algorithms including the quantum Fourier transform, quantum error correction,
526 quantum teleportation, Grover's algorithm, dense coding, etc. In addition, any quantum
527 circuit may be plotted using the `circuit_plot` function (Figure ??).

528 Here are a few short examples of the quantum information and computing capabilities in
529 `sympy.physics.quantum`. Start with a simple four-qubit state and flip the second qubit from the
530 right using a Pauli-X gate:

```
531 >>> from sympy.physics.quantum.qubit import Qubit
532 >>> from sympy.physics.quantum.gate import XGate
533 >>> q = Qubit('0101')
534 >>> q
535 |0101>
536 >>> X = XGate(1)
537 >>> qapply(X*q)
538 |0111>
```

539 Qubit states can also be used in adjoint operations, tensor products, inner/outer products:

```
540 >>> Dagger(q)
541 <0101|
542 >>> ip = Dagger(q)*q
543 >>> ip
544 <0101|0101>
545 >>> ip.doit()
546 1
```

547 Quantum gates (unitary operators) can be applied to transform these states and then classical
548 measurements can be performed on the results:

```
549 >>> from sympy.physics.quantum.qubit import measure_all
550 >>> from sympy.physics.quantum.gate import H, X, Y, Z
551 >>> c = H(0)*H(1)*Qubit('00')
552 >>> c
553 H(0)*H(1)*|00>
554 >>> q = qapply(c)
555 >>> measure_all(q)
556 [(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]
```



Figure 1. The circuit diagram for a three-qubit quantum Fourier transform generated by SymPy.

557 Lastly, the following example demonstrates creating a three-qubit quantum Fourier transform,
 558 decomposing it into one- and two-qubit gates, and then generating a circuit plot for the sequence
 559 of gates (see Figure ??).

```
560 >>> from sympy.physics.quantum.qft import QFT
561 >>> from sympy.physics.quantum.circuitplot import circuit_plot
562 >>> fourier = QFT(0,3).decompose()
563 >>> fourier
564 SWAP(0,2)*H(0)*C((0),S(1))*H(1)*C((0),T(2))*C((1),S(2))*H(2)
565 >>> c = circuit_plot(fourier, nqubits=3)
```

566 5 ARCHITECTURE

567 Software architecture is of central importance in any large software project because it establishes
 568 predictable patterns of usage and development [?]. This section describes the essential structural
 569 components of SymPy, provides justifications for the design decisions that have been made, and
 570 gives example user-facing code as appropriate.

571 5.1 The Core

572 A computer algebra system stores mathematical expressions as data structures. For example,
 573 the mathematical expression $x + y$ is represented as a tree with three nodes, $+$, x , and y ,
 574 where x and y are ordered children of $+$. As users manipulate mathematical expressions
 575 with traditional mathematical syntax, the CAS manipulates the underlying data structures.
 576 Automated optimizations and computations such as integration, simplification, etc. are all
 577 functions that consume and produce expression trees.

578 In SymPy every symbolic expression is an instance of a Python `Basic` class,¹⁰ a superclass
 579 of all SymPy types providing common methods to all SymPy tree-elements, such as traversals.
 580 The children of a node in the tree are held in the `args` attribute. A terminal or leaf node in the
 581 expression tree has empty `args`.

582 For example, consider the expression $xy + 2$:

```
583 >>> x, y = symbols('x y')
584 >>> expr = x*y + 2
```

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

¹⁰Some internal classes, such as those used in the polynomial submodule, do not follow this rule for efficiency reasons.

```

587 >>> type(expr)
588 <class 'sympy.core.add.Add'>
589 >>> expr.args
590 (2, x*y)

```

Descending further down into the expression tree yields the full expression. For example, the next child node (given by `expr.args[0]`) is 2. Its class is `Integer`, and it has an empty `args` tuple, indicating that it is a leaf node.

```

594 >>> expr.args[0]
595 2
596 >>> type(expr.args[0])
597 <class 'sympy.core.numbers.Integer'>
598 >>> expr.args[0].args
599 ()

```

Symbols or symbolic constants, like e or π , are examples of leaf nodes.

```

601 >>> exp(1)
602 E
603 >>> exp(1).args
604 ()
605 >>> x.args
606 ()

```

A useful way to view an expression tree is using the `srepr` function, which returns a string representation of an expression as valid Python code¹¹ with all the nested class constructor calls to create the given expression.

```

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

```

Every SymPy expression satisfies a key identity invariant:

```

613 expr.func(*expr.args) == expr

```

This means that expressions are rebuildable from their `args`.¹² Note that in SymPy the `==` operator represents exact structural equality, not mathematical equality. This allows testing if any two expressions are equal to one another as expression trees. For example, even though $(x+1)^2$ and x^2+2x+1 are equal mathematically, SymPy gives

```

618 >>> (x + 1)**2 == x**2 + 2*x + 1
619 False

```

because they are different as expression trees (the former is a `Pow` object and the latter is an `Add` object).

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

¹¹ The `dotprint` function from the `sympy.printing.dot` submodule prints output to dot format, which can be rendered with Graphviz to visualize expression trees graphically.

¹²`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.

5.2 Extensibility

While the core of SymPy is relatively small, it has been extended to a wide variety of domains by a broad range of contributors. This is due, in part, to the fact that the same language, Python, is used both for the internal implementation and the external usage by users. All of the extensibility capabilities available to users are also utilized by SymPy itself. This eases the transition pathway from SymPy user to SymPy developer.

The typical way to create a custom SymPy object is to subclass an existing SymPy class, usually `Basic`, `Expr`, or `Function`. As it was stated before, all SymPy classes used for expression trees should be subclasses of the base class `Basic`. `Expr` is the `Basic` subclass for mathematical that can be added and multiplied together. The most commonly seen classes in SymPy are subclasses of `Expr`, including `Add`, `Mul`, and `Symbol`. Instances of `Expr` typically represent complex numbers, but may also include other “rings”, like matrix expressions. Not all SymPy classes are subclasses of `Expr`. For instance, logic expressions, such as `And(x, y)`, are subclasses of `Basic` but not of `Expr`.

The `Function` class is a subclass of `Expr` which makes it easier to define mathematical functions called with arguments. This includes named functions like $\sin(x)$ and $\log(x)$ as well as undefined functions like $f(x)$. Subclasses of `Function` should define a class method `eval`, which returns a canonical form of the function application (usually an instance of some other class, i.e., a `Number`) or `None`, if for given arguments that function should not be automatically evaluated.

Many SymPy functions perform various evaluations down the expression tree. Classes define their behavior in such functions by defining a relevant `_eval_*` method. For instance, an object can indicate to the `diff` function how to take the derivative of itself by defining the `_eval_derivative(self, x)` method, which may in turn call `diff` on its args. (Subclasses of `Function` should implement `fdiff` method instead, it returns the derivative of the function without considering the chain rule.) The most common `_eval_*` methods relate to the assumptions: `_eval_is_assumption` is used to deduce *assumption* on the object.

As an example of the notions presented in this section, Listing ?? presents a minimal version of the gamma function $\Gamma(x)$ from SymPy, which evaluates itself on positive integer arguments, has the positive and real assumptions defined, can be rewritten in terms of factorial with `gamma(x).rewrite(factorial)`, and can be differentiated. `self.func` is used throughout instead of referencing `gamma` explicitly so that potential subclasses of `gamma` can reuse the methods.

Listing 1. A minimal implementation of `sympy.gamma`.

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

class gamma(Function)
    @classmethod
    def eval(cls, arg):
        if isinstance(arg, Integer) and arg.is_positive:
            return factorial(arg - 1)

    def _eval_is_positive(self):
        x = self.args[0]
        if x.is_positive:
            return True
        elif x.is_noninteger:
            return floor(x).is_even

    def _eval_is_real(self):
        x = self.args[0]
        # noninteger means real and not integer
        if x.is_positive or x.is_noninteger:
            return True

    def _eval_rewrite_as_factorial(self, z):
```

```

682         return factorial(z - 1)
683
684     def fdiff(self, argindex=1):
685         from sympy.core.function import ArgumentIndexError
686         if argindex == 1:
687             return self.func(self.args[0])*polygamma(0, self.args[0])
688         else:
689             raise ArgumentIndexError(self, argindex)

```

690 The gamma function implemented in SymPy has many more capabilities than the above listing,
691 such as evaluation at rational points and series expansion.

692 5.3 Speed

693 Due to being written in pure Python, SymPy's speed is generally slower compared with its
694 commercial competitors. For many applications and uses of SymPy, that is not a problem, as
695 SymPy is able to return the answer quickly enough, but for some applications that require
696 handling of very long expressions and/or lots of small expressions, the speed becomes a problem.

697 For this reason, a new library called SymEngine [?] was started. It is a pure C++ library
698 with thin wrappers to other languages (Python, Ruby, Julia, ...) whose aim is to be the fastest
699 manipulation library. Preliminary benchmarks suggest that SymEngine is as fast or faster than
700 the commercial or open source competitors.

701 The development branch of SymPy recently started to use SymEngine as an optional backend,
702 initially in `sympy.physics.mechanics` only. The plan is to allow more algorithms in SymPy to
703 take advantage of the speed of SymEngine.

704 6 PROJECTS THAT DEPEND ON SYMPY

705 There are several projects that depend on SymPy as a library for implementing a part of their
706 functionality. A selection of these projects are listed in Table ??.

Table 3. Selected projects that depend on SymPy.

Project name	Domain	Description
SymPy Live		SymPy Live 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 . SymPy Live is maintained by the SymPy community.
SymPy Gamma		SymPy Gamma is a web application that executes and displays results for SymPy expressions, in a fashion similar to that of Wolfram Alpha. SymPy Gamma is maintained by the SymPy community. See the supplementary material for more information about SymPy Gamma.
Cadabra [?]		Cadabra is a CAS designed specifically for the resolution of problems encountered in field theory.
Octave Symbolic [?]		The OctSymPy 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.[<8;32;30m

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.
PyDy [?]	Multibody Dynamics with Python.
galgebra [?]	A module for geometric algebra (previously <code>sympy.galgebra</code>).
yt [?]	Python package for analyzing and visualizing volumetric data (<code>yt.units</code> uses SymPy).
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.
Quameon [?]	Quantum Monte Carlo in Python.
Lcapy [?]	Experimental Python package for teaching linear circuit analysis.

7 CONCLUSION AND FUTURE WORK

SymPy is a robust computer algebra system that provides a wide spectrum of features both in traditional computer algebra and in a plethora of scientific disciplines. This allows SymPy to be used in a first-class way with other Python projects, including the scientific Python stack. Unlike many other CAS's, SymPy is designed to be used in an extensible way: both as an end-user application and as a library.

SymPy expressions are immutable trees of Python objects. SymPy uses Python both as the internal language and the user language. This permits users to access to the same methods that the library implements in order to extend it for their needs. Additionally, SymPy has a powerful assumptions system for declaring and deducing mathematical properties of expressions.

SymPy supports a wide array of mathematical facilities. This includes functions for simplifying expressions, performing common calculus operations, pretty printing expressions, solving equations, and representing symbolic matrices. Other supported facilities include 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. This breadth of domains has been engendered by a strong and vibrant user community. Anecdotally, these users likely chose SymPy because of its ease of access.

Some of the planned future work for SymPy includes work on improving code generation, improvements to the speed of SymPy using SymEngine, improving the assumptions system, and improving the solvers submodule.

8 ACKNOWLEDGEMENTS

All authors thank the Google Summer of Code for its financial support of students who contributed to SymPy.

The author of this paper Ondřej Čertík thanks the Los Alamos National Laboratory for its financial support.

734 The author of this paper Richard P. Muller thanks Sandia National Laboratories for their
735 financial support.

736 The author of this paper Francesco Bonazzi thanks the Deutsche Forschungsgemeinschaft
737 (DFG) for its financial support via the International Research Training Group 1524 “Self-
738 Assembled Soft Matter Nano-Structures at Interfaces.”