

1 SymPy: ~~Symbolic Computing~~ symbolic 2 computing in Python

3 Aaron Meurer¹, Christopher P. Smith², Mateusz Paprocki³, Ondřej Čertík⁴,
4 Sergey B. Kirpichev⁵, Matthew Rocklin³, AMiT Kumar⁶, Sergiu Ivanov⁷,
5 Jason K. Moore⁸, Sartaj Singh⁹, Thilina Rathnayake¹⁰, Sean Vig¹¹, Brian
6 E. Granger¹², Richard P. Muller¹³, Francesco Bonazzi¹⁴, Harsh Gupta¹⁵,
7 Shivam Vats¹⁵, Fredrik Johansson¹⁶, Fabian Pedregosa¹⁷, Matthew J.
8 Curry^{18, 19, 20}, Andy R. Terrel^{21, 22}, Štěpán Roučka²³, Ashutosh Saboo²⁴,
9 Isuru Fernando¹⁰, Sumith Kulal²⁵, Robert Cimrman²⁶, and Anthony
10 Scopatz¹

11 ¹Department of Mechanical Engineering, University of South Carolina, Columbia, South
12 Carolina, United States

13 ²Polar Semiconductor, Inc., Bloomington, Minnesota, United States

14 ³Continuum Analytics, Inc., Austin, Texas, United States

15 ⁴Los Alamos National Laboratory, Los Alamos, New Mexico, United States

16 ⁵Faculty of Physics, Moscow State University, Moscow, Russia

17 ⁶Department of Applied Mathematics, Delhi Technological University, New Delhi, India

18 ⁷Université Paris Est Créteil, Créteil, France

19 ⁸Mechanical and Aerospace Engineering, University of California, Davis, Davis,
20 California, United States

21 ⁹Mathematical Sciences, Indian Institute of Technology (BHU), Varanasi, Uttar Pradesh,
22 India

23 ¹⁰Department of Computer Science and Engineering, University of Moratuwa,
24 Katubedda, Moratuwa, Sri Lanka

25 ¹¹University of Illinois at Urbana-Champaign, Urbana, Illinois, United States

26 ¹²California Polytechnic State University, San Luis Obispo, California, United States

27 ¹³Center for Computing Research, Sandia National Laboratories, Albuquerque, New
28 Mexico, United States

29 ¹⁴Department of Theory and Bio-Systems, Max Planck Institute of Colloids and
30 Interfaces, Potsdam, Germany

31 ¹⁵Indian Institute of Technology Kharagpur, Kharagpur, West Bengal, India

32 ¹⁶INRIA Bordeaux-Sud-Ouest – LFANT project-team, Talence, France

33 ¹⁷INRIA – SIERRA project-team, Paris, France

34 ¹⁸Department of Physics and Astronomy, University of New Mexico, Albuquerque, New
35 Mexico, United States

36 ¹⁹Center for Quantum Information and Control, University of New Mexico, Albuquerque,
37 New Mexico, United States

38 ²⁰Sandia National Laboratories, Albuquerque, New Mexico, United States

39 ²¹Fashion Metric, Inc, Austin, Texas, United States

40 ²²NumFOCUS, Austin, TX, United States

41 ²³Department of Surface and Plasma Science, Faculty of Mathematics and Physics,
42 Charles University in Prague, Praha, Czech Republic

43 ²⁴Birla Institute of Technology and Science, Pilani, K.K. Birla Goa Campus, Sancoale,
44 Goa, India

45 ²⁵Indian Institute of Technology Bombay, Mumbai, Maharashtra, India

46 ²⁶New Technologies – Research Centre, University of West Bohemia, Plzeň, Czech
47 Republic

48 **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 submodules. The supplementary material provide additional examples and further outline details of the architecture and features of SymPy.

Keywords: Python, computer algebra system, symbolics

1 INTRODUCTION

SymPy is a full featured computer algebra system (CAS) written in the Python programming language [32]–[32] programming language. It is free and open source software, licensed under the 3-clause BSD license [49]. 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 [43]. 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 [37]. SymPy is itself used as a dependency by many libraries and tools to support research within a variety of domains, such as Sage [?] (pure SageMath [58] (pure and applied mathematics), yt [64] (astronomy and astrophysics), PyDy [19] (multibody dynamics), and SfePy [10] (finite elements).

Unlike many CASs, 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 [24]. 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 Being a library, SymPy does not have a built-in graphical user interface (GUI). However, SymPy exposes a rich interactive display system, and supports registering printers–display formatters with Jupyter [40]–[29] frontends, including the Notebook and Qt Console, which will render SymPy expressions using MathJax [9] or L^AT_EX.

The remainder of this paper discusses key components of the SymPy software. Section 5 discusses the architecture of SymPy library. Section 3 enumerates the features of SymPy and takes a closer look at some of the important ones. The section 3 looks at the numerical features of SymPy and its dependency library, mpmath. Section 4 looks at the domain specific physics submodules for performing symbolic and numerical calculations in classical mechanics and quantum mechanics. Section 5 discusses the architecture of SymPy. Section 6 looks at a selection of packages that depend on SymPy. Conclusions and future directions for SymPy are given in section 7. All examples in this paper use SymPy version 1.0 and mpmath version 0.19.

The following statement imports all SymPy functions into the global Python namespace.²

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

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.

94 [From here on, all examples in this paper assume that this statement has been executed.](#)³

95 `>>> from sympy import *`

96 [All the examples in this paper can be tested on SymPy Live, an online Python shell that](#)
97 [uses the Google App Engine \[11\] to execute SymPy code. SymPy Live is also integrated into](#)
98 [the SymPy documentation at http://docs.sympy.org.](#)

99 **2 ARCHITECTURE OVERVIEW OF CAPABILITIES**

100 **3 FEATURES**

101 [This section gives a basic introduction of SymPy, and lists its features. A few features—assumptions,](#)
102 [simplification, calculus, polynomials, printers, solvers, and matrices—are core components of](#)
103 [SymPy and are discussed in depth. Many other features are discussed in depth in the supplementary](#)
104 [material.](#)

105 **2.1 Basic Usage**

106 [Symbolic variables, called symbols, must be defined and assigned to Python variables before they](#)
107 [can be used. This is typically done through the `symbols` function, which may create multiple](#)
108 [symbols in a single function call. For instance,](#)

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

110 [creates three symbols representing variables named \$x\$, \$y\$, and \$z\$. In this particular instance,](#)
111 [these symbols are all assigned to Python variables of the same name. However, the user is free](#)
112 [to assign them to different Python variables, while representing the same symbol, such as `a`, `b`,](#)
113 [and `c = symbols\('x y z'\)`. In order to minimize potential confusion, though, all examples in this](#)
114 [paper will assume that the symbols \$x\$, \$y\$, and \$z\$ have been assigned to Python variables identical](#)
115 [to their symbolic names.](#)

116 [Expressions are created from symbols using Python's mathematical syntax. For instance, the](#)
117 [following Python code creates the expression \$\(x^2 - 2x + 3\)/y\$. Note that the expression remains](#)
118 [unevaluated: it is represented symbolically.](#)

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

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

121 **2.2 List of Features**

122 [Although SymPy's extensive feature set cannot be covered ~~in-depth in depth~~ in this paper,](#)
123 [calculus and other bedrock areas](#) ~~bedrock areas, that is, those areas that are used throughout~~
124 [the library](#), are discussed in their own subsections [below](#). Additionally, Table 1 gives a compact
125 listing of all major capabilities present in the SymPy codebase. This grants a sampling from the
126 breadth of topics and application domains that SymPy services. Unless stated otherwise, all
127 features noted in Table 1 are symbolic in nature. Numeric features are discussed in Section 3.

Table 1. SymPy Features and Descriptions

Feature (submodules)	Description
Calculus (sympy.core , sympy.calculus , sympy.integrals , sympy.series)	Algorithms for computing derivatives, integrals, and limits.
Category Theory (sympy.categories)	Representation of objects, morphisms, and diagrams. Tools for drawing diagrams with Xy-pic [48] .

³ [The 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.](#)

Code Generation (<u>sympy.printing</u> , <u>sympy.codegen</u>)	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 (<u>sympy.combinatorics</u>)	Permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [36], and Prufer sequences [4].
Concrete Math (<u>sympy.concrete</u>)	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 [42] for two univariate polynomials.
Cryptography (<u>sympy.crypto</u>)	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 (<u>sympy.diffgeom</u>)	Representations of manifolds, metrics, tensor products, and coordinate systems in Riemannian and pseudo-Riemannian geometries [52].
Geometry (<u>sympy.geometry</u>)	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 (<u>sympy.liealgebras</u>)	Representations of Lie algebras and root systems.
Logic (<u>sympy.logic</u>)	Boolean expressions, equivalence testing, satisfiability, and normal forms.
Matrices (<u>sympy.matrices</u>)	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 (<u>sympy.matrices.expressions</u>)	Matrices with symbolic dimensions (unspecified entries). Block matrices.
Number Theory (<u>sympy.ntheory</u>)	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 (<u>sympy.plotting</u>)	Hooks for visualizing expressions via matplotlib [25] or as text drawings when lacking a graphical back-end. 2D function plotting, 3D function plotting, and 2D implicit function plotting are supported.
Polynomials (<u>sympy.polys</u>)	Polynomial algebras over various coefficient domains. Functionality ranges from simple operations (e.g., polynomial division) to advanced computations (e.g., Gröbner bases [1] and multivariate factorization over algebraic number domains).
Printing (<u>sympy.printing</u>)	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 (<u>sympy.physics.quantum</u>)	Quantum states, bra-ket notation, operators, basis sets, representations, tensor products, inner products, outer products, commutators, anticommutators, and specific quantum system implementations.

Series (sympy.series)	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 (sympy.sets)	Representations of empty, finite, and infinite sets. This includes (including special sets such as for all the natural, integer, and complex numbers). Operations on sets such as union, intersection, Cartesian product, and building sets from other sets are supported.
Simplification (sympy.simplify)	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 (sympy.solvers)	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 (sympy.functions)	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 (sympy.stats)	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 [47].
Tensors (sympy.tensor)	Symbolic manipulation of indexed objects.
Vectors (sympy.vector)	Basic operations on vectors and differential calculus with respect to 3D Cartesian coordinate systems.

2.3 Assumptions

The assumptions system allows users to specify that symbols have certain common mathematical properties, such as being positive, imaginary, or integer. SymPy is careful to never perform simplifications on an expression unless the assumptions allow them. For instance, the identity $\sqrt{t^2} = t$ 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.

```
>>> t = Symbol('t', positive=True)
>>> sqrt(t**2)
t
```

Some of the common assumptions are `negative`, `real`, `nonpositive`, `integer`, `prime` and `commutative`.⁴ Assumptions on any SymPy 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_SUBSCRIPTNBreal` and `I.is_SUBSCRIPTNBprime` 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_SUBSCRIPTNBpositive` will give `None` because a real symbol might be positive or negative. `None` could also mean that not enough is known or implemented to compute the given fact. For instance, `(pi + E).is_SUBSCRIPTNBirrational` gives `None`—indeed, the rationality of $\pi + e$ is an open problem in mathematics [31].

Basic implications between the facts are used to deduce assumptions. Deductions are made using the Rete algorithm [13].⁵ For instance, the assumptions system knows that being an integer implies being rational.

```
>>> i = Symbol('i', integer=True)
>>> i.is_rational
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_SUBSCRIPTNBpositive` will be `True` (whereas `(x - y).is_SUBSCRIPTNBpositive` 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 [2]–[34]. The `simplify` function applies several simplification routines along with heuristics to make the output expression “simple”.⁶

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 2 lists common simplification functions.

Examples for these simplification functions can be found in the supplement.

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.

⁵For historical reasons, this algorithm is distinct from the `sympy.logic` submodule, which is discussed in the supplementary material. SymPy also has an experimental assumptions system which stores facts separate from objects, and uses `sympy.logic` and a SAT solver for deduction. We will not discuss this system here.

⁶The 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.

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 [18]
<code>hyperexpand</code>	expand hypergeometric functions [44, 45]

2.5 Calculus

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

Limits are computed with the `limit` function, using the Gruntz algorithm [22] for computing symbolic limits and heuristics (a description of the Gruntz algorithm may be found in the supplement). For example, the following computes $\lim_{x \rightarrow \infty} x \sin(\frac{1}{x}) = 1$. Note that SymPy denotes ∞ as `oo`.

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

As a more complex example, SymPy computes

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

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

Derivatives are computed with the `diff` function, which recursively uses the various differentiation rules.

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

Integrals are calculated with the `integrate` function. SymPy implements a combination of the Risch algorithm [7], table lookups, a reimplement of Manuel Bronstein’s “Poor Man’s Integrator” [6], and an algorithm for computing integrals based on Meijer G-functions [44, 45]. 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

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

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


Summations are computed with [the summation using function, which uses](#) a combination of Gosper's algorithm [21], an algorithm that uses Meijer G-functions [44, 45], and heuristics. Products are computed with `product` function via a suite of heuristics.

```
>>> i, n = symbols('i n')
>>> summation(2**i, (i, 0, n - 1))
2**n - 1
>>> summation(i*factorial(i), (i, 1, n))
n*factorial(n) + factorial(n) - 1
```

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

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

[The supplementary material discusses series expansions methods in more depth.](#)

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

```
>>> integrate(x**x, x)
Integral(x**x, x)
>>> Sum(2**i, (i, 0, n - 1))
Sum(2**i, (i, 0, n - 1))
```

2.6 Polynomials

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

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

SymPy implements dense and sparse polynomial representations.⁷ Both are used in the univariate and multivariate cases. The dense representation is the default for univariate polynomials. For multivariate polynomials, the choice of representation is based on the application. The most common case for the sparse representation is algorithms for computing Gröbner bases (Buchberger, F4, and F5) [8, 14, 15]. This is because different monomial orderings can be expressed easily in this representation. However, algorithms for computing multivariate GCDs or factorizations, at least those currently implemented in SymPy [38], are better expressed when the representation is dense. The dense multivariate representation is specifically a recursively-dense representation, 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 that despite this, the coefficient domain K , can be a multivariate polynomial domain as well. The dense recursive representation in Python gets inefficient as the number of variables increases.

Some examples for the `sympy.polys` [module-submodule](#) can be found in the supplement.

2.7 Printers

SymPy has a rich collection of expression printers. By default, an interactive Python session will render the `str` form of an expression, which has been used in all the examples in this paper so

⁷In 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.

far. The `str` form of an expression is valid Python and roughly matches what a user would type to enter the expression.⁸

```
>>> phi0 = Symbol('phi0')
>>> str(Integral(sqrt(phi0), phi0))
'Integral(sqrt(phi0), phi0)'
```

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

```
>>> pprint(Integral(sqrt(phi0 + 1), phi0))
```

$$\int \sqrt{\varphi_0 + 1} \, d(\varphi_0)$$

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

```
>>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
```

$$\int \sqrt{\varphi_0 + 1} \, d(\varphi_0)$$

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

```
>>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
\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 [40], 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, which allows extending any given printer, and also 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 a module of equation solvers that can handle ordinary differential equations, recurrence relationships, Diophantine equations⁹, 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

⁸Many 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 noted in section 5.1, the `srepr` function prints the exact, verbose form of an expression.

⁹See the supplementary material for an in depth discussion on the Diophantine submodule.

```

296 solve(f, *symbols, **flags)
297 solveset(f, symbol, domain=S.Complexes)

```

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

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

304 Both functions implicitly assume that expressions are equal to 0. For instance, solveset(x -
 305 1, x) solves $x - 1 = 0$ for x .

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

309 ~~More examples of~~ [Some examples for](#) solveset and solve can be found in the supplement.

310 2.9 Matrices

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

```

314 >>> A = Matrix([[x, x + y], [y, x]])
315 >>> A
316 Matrix([
317 [x, x + y],
318 [y, x]])

```

319 SymPy matrices support common symbolic linear algebra manipulations, including matrix
 320 addition, multiplication, exponentiation, computing determinants, solving linear systems, [singular](#)
 321 [values](#), and computing inverses using LU decomposition, LDL decomposition, Gauss-Jordan
 322 elimination, Cholesky decomposition, Moore-Penrose pseudoinverse, ~~and adjugate matrix~~ [or](#)
 323 [adjugate matrices](#).

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

```

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

```

329 Internally these matrices store the elements as ~~lists of lists~~ [Lists of Lists \(LIL\)](#) [27], meaning
 330 [the matrix is stored as a list of lists of entries \(effectively, the input format used to create](#)
 331 [the matrix A above\)](#), making it a dense representation.¹⁰ For storing sparse matrices, the
 332 SparseMatrix class can be used. Sparse matrices store their elements ~~as a dictionary of keys in~~
 333 [Dictionary of Keys \(DOK\) format, meaning entries are stored as a dict of \(row, column\) pairs](#)
 334 [mapping to the elements](#).

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

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

342 When symbolic matrices are combined with the assumptions ~~module~~ [submodule](#) for logical
 343 inference, they provide powerful reasoning over invertibility, semi-definiteness, orthogonality, etc.,
 344 which are valuable in the construction of numerical linear algebra systems [46].

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

¹⁰Similar to the polynomials ~~module~~ [submodule](#), dense here means that all entries are stored in memory, contrasted with a sparse representation where only nonzero entries are stored.

3 NUMERICS

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

3.1 Floating-Point Numbers

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

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

```
>>> Float(1.1)
1.1000000000000000
>>> Float(1.1, 30) # precision equivalent to 30 digits
1.100000000000000008881784197001
>>> Float("1.1", 30)
1.100000000000000000000000000000
```

The `evalf` method converts a constant symbolic expression to a `Float` with the specified precision, here 25 digits:

```
>>> (pi + 1).evalf(25)
4.141592653589793238462643
```

`Float` numbers do not track their accuracy, and should be used with caution within symbolic expressions since familiar dangers of floating-point arithmetic apply [20]. A notorious case is that of catastrophic cancellation:

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

Applying the `evalf` method to the whole expression solves this problem. Internally, `evalf` estimates the number of accurate bits of the floating-point approximation for each sub-expression, and adaptively increases the working precision until the estimated accuracy of the final result matches the sought number of decimal digits:

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

The `evalf` method works with complex numbers and supports more complicated expressions, such as special functions, infinite series, and integrals. The internal error tracking does not provide rigorous error bounds (in the sense of interval arithmetic) and cannot be used to accurately track uncertainty in measurement data; the sole purpose is to mitigate loss of accuracy that typically occurs when converting symbolic expressions to numerical values.

3.2 The mpmath library

The implementation of arbitrary-precision floating-point arithmetic is supplied by the `mpmath` library [26]. Originally, it was developed as a SymPy ~~module~~ `submodule` but has subsequently been moved to a standalone pure-Python package. The basic datatypes in `mpmath` are `mpf` and `mpc`, which respectively act as multiprecision substitutes for Python's `float` and `complex`. The floating-point precision is controlled by a global context: ¹¹

¹¹ All examples in this section use `mpmath` version 0.19.

```

392 >>> import mpmath
393 >>> mpmath.mp.dps = 30      # 30 digits of precision
394 >>> mpmath.mpf("0.1") + mpmath.exp(-50)
395 mpf('0.1000000000000000000000000000192874984794')
396 >>> print(_)    # pretty-printed
397 0.1000000000000000000000000000192874985

```

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

Like SymPy, mpmath is a pure Python library. A design decision of SymPy is to keep it and its required dependencies pure Python. This is a primary advantage of mpmath over other multiple precision libraries such as GNU MPFR [17], which is faster. Like SymPy, mpmath is ~~a pure Python library~~, also BSD licensed (GNU MPFR is licensed under the GNU Lesser General Public License [49]).

Internally, `mpmath` represents a floating-point number $(-1)^s x \cdot 2^y$ by a tuple (s, x, y, b) where x and y are arbitrary-size Python integers and the redundant integer b stores the bit length of x for quick access. If GMPY [23] is installed, `mpmath` automatically uses the `gmpy.mpz` type for x , and GMPY methods for rounding-related operations, improving performance.

Most `mpmath` and `SymPy` functions use the same naming scheme, although this is not true in every case. For example, the symbolic `SymPy` summation expression `Sum(f(x), (x, a, b))` representing $\sum_{x=a}^b f(x)$ is represented in `mpmath` as `nsum(f, (a, b))`, where `f` is a numeric Python function.

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

The double exponential (tanh-sinh) quadrature is used for numerical integration by default. For smooth integrands, this algorithm usually converges extremely rapidly, even when the integration interval is infinite or singularities are present at the endpoints [54, 2]. However, for good performance, singularities in the middle of the interval must be specified by the user. To evaluate slowly converging limits and infinite series, mpmath automatically tries Richardson extrapolation and the Shanks transformation (Euler-Maclaurin summation can also be used) [3]. A function to evaluate oscillatory integrals by means of convergence acceleration is also available.

A wide array of higher mathematical functions ~~are~~is implemented with full support for complex values of all parameters and arguments, including complete and incomplete gamma functions, Bessel functions, orthogonal polynomials, elliptic functions and integrals, zeta and polylogarithm functions, the generalized hypergeometric function, and the Meijer G-function. The Meijer G-function instance $G_{1,3}^{3,0}(0; \frac{1}{2}, -1, -\frac{3}{2}|x)$ is a good test case [63]; past versions of both Maple and Mathematica produced incorrect numerical values for large $x > 0$. Here, mpmath automatically removes an internal singularity and compensates for cancellations (amounting to 656 bits of precision when $x = 10000$), giving correct values:

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

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

```
438 >>> meijerg([[[]],[0]], [[-S(1)/2,-1,-S(3)/2],[[]], 10000).evalf()
439 2.43925769071996e-94
```

Symbolic integration and summation often ~~produces~~produce hypergeometric and Meijer G-function closed forms (see ~~Subsection~~section 2.5); numerical evaluation of such special functions is a useful complement to direct numerical integration and summation.

4 DOMAIN-SPECIFIC SUBMODULES **PHYSICS SUBMODULE**

SymPy includes several [packages-submodules](#) that allow users to solve domain specific [physics](#) problems. For example, a comprehensive physics [package-submodule](#) is included that is useful for solving problems in mechanics, optics, and quantum mechanics along with support for manipulating physical quantities with units.

4.1 Classical Mechanics

One of the core domains that SymPy supports is the physics of classical mechanics. This is in turn separated into two distinct components: vector algebra [symbolics](#) and mechanics.

4.1.1 Vector Algebra

The `sympy.physics.vector` [package-submodule](#) provides reference frame-, time-, and space-aware vector and dyadic objects that allow for [three-dimensional-three-dimensional](#) operations such as addition, subtraction, scalar multiplication, inner and outer products, and cross products. ~~Both of these objects~~ [The vector and dyadic objects both](#) can be written in very compact notation that make it easy to express the vectors and dyadics in terms of multiple reference frames with arbitrarily defined relative orientations. The vectors are used to specify the positions, velocities, and accelerations of points; orientations, angular velocities, and angular accelerations of reference frames; and forces and torques. The dyadics are essentially reference frame-aware 3×3 tensors [53]. The vector and dyadic objects can be used for any one-, two-, or three-dimensional vector algebra, and they provide a strong framework for building physics and engineering tools.

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 $\frac{\pi}{2}$.

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

4.1.2 Mechanics

The `sympy.physics.mechanics` [package-submodule](#) utilizes the `sympy.physics.vector` [package-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 [package-submodule](#) offers two automated methods for formulating the equations of motion based on Lagrangian Dynamics [30] and Kane's Method [28]. Lastly, there are automated linearization routines for constrained dynamical systems [41].

4.2 Quantum Mechanics

The `sympy.physics.quantum` [package-submodule](#) has extensive capabilities ~~for performing symbolic~~ [to solve problems in](#) quantum mechanics, using Python objects to represent the different mathematical objects relevant in quantum theory [50]: 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.

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

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

506 Commutators can be expanded using common commutator identities:

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

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

- 511 • Many of the exactly solvable quantum systems, including simple harmonic oscillator states
512 and raising/lowering operators, infinite square well states, and 3D position and momentum
513 operators and states.
- 514 • Second quantized formalism of non-relativistic many-body quantum mechanics [16].
- 515 • Quantum angular momentum [65]. Spin operators and their eigenstates can be repre-
516 sented in any basis and for any quantum numbers. A rotation operator representing the
517 ~~Wigner-D-matrix~~Wigner D-matrix, which may be defined symbolically or numerically, is
518 also implemented to rotate spin eigenstates. Functionality for coupling and uncoupling of
519 arbitrary spin eigenstates is provided, including symbolic representations of Clebsch-Gordon
520 coefficients and Wigner symbols.
- 521 • Quantum information and computing [35]. Multidimensional qubit states, and a full
522 set of one- and two-qubit gates are provided and can be represented symbolically or as
523 matrices/vectors. With these building blocks, it is possible to implement a number of basic
524 quantum algorithms including the quantum Fourier transform, quantum error correction,
525 quantum teleportation, Grover's algorithm, dense coding, etc. In addition, any quantum
526 circuit may be plotted using the `circuit_plot` function (Figure 1).

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

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

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

```
539 >>> Dagger(q)
540 <0101|
541 >>> ip = Dagger(q)*q
```

```

542 >>> ip
543 <0101|0101>
544 >>> ip.doit()
545 1

```

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

```

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

```

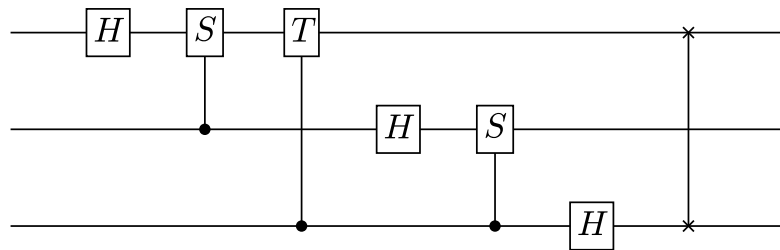


Figure 1. The circuit diagram for a three-qubit quantum Fourier transform generated by SymPy.

Lastly, the following example demonstrates creating a three-qubit quantum Fourier transform, decomposing it into one- and two-qubit gates, and then generating a circuit plot for the sequence of gates (see Figure 1).

```

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

```

5 CONCLUSION AND FUTURE WORK ARCHITECTURE

Software architecture is of central importance in any large software project because it establishes predictable patterns of usage and development [51]. This section describes the essential structural components of SymPy, provides justifications for the design decisions that have been made, and gives example user-facing code as appropriate.

5.1 Basic Usage

The following statement imports all SymPy functions into the global Python namespace. From here on, all examples in this paper assume that this statement has been executed.¹¹

¹¹ All examples in this paper use SymPy version 1.0.

Symbolic variables, called symbols, must be defined and assigned to Python variables before they can be used. This is typically done through the `symbols` function, which may create multiple symbols in a single function call. For instance, `creates three symbols representing variables named x , y , and z . In this particular instance, these symbols are all assigned to Python variables of the same name. However, the user is free to assign them to different Python variables, while representing the same symbol, such as a, b, c = symbols('x y z'). In order to minimize potential confusion, though, all examples in this paper will assume that the symbols, and have been assigned to Python variables identical to their symbolic names.`

Expressions are created from symbols using Python's mathematical syntax. For instance, the following Python code creates the expression $(x^2 - 2x + 3)/y$.

Importantly, SymPy expressions are immutable. This simplifies the design of SymPy by allowing expression interning. It also enables expressions to be hashed and stored in Python dictionaries, thereby permitting features such as caching.

5.1 The Core

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

In SymPy every symbolic expression is an instance of a Python Basic class, a the class Basic,¹¹ the superclass of all SymPy types providing common methods to all SymPy tree elements, such as traversals. The children of a node in the tree are held in the `args` attribute. A terminal or leaf node in the expression tree has empty `args`.

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

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

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

```
>>> type(expr)
<class 'sympy.core.add.Add'>
>>> expr.args
(2, x*y)
```

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.

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

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

```
>>> exp(1)
E
>>> exp(1).args
()
>>> x.args
()
```

¹¹Some internal classes, such as those used in the polynomial submodule, do not follow this rule for efficiency reasons.

622 A useful way to view an expression tree is using the `srepr` function, which returns a string
623 representation of an expression as valid Python code¹² with all the nested class constructor calls
624 to create the given expression.

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

627 Every SymPy expression satisfies a key identity invariant:

```
628 expr.func(*expr.args) == expr
```

629 This means that expressions are rebuildable from their `args`.¹³ Note that in SymPy the `==`
630 operator represents exact structural equality, not mathematical equality. This allows testing if
631 any two expressions are equal to one another as expression trees. For example, even though
632 $(x+1)^2$ and x^2+2x+1 are equal mathematically, SymPy gives

```
633 >>> (x + 1)**2 == x**2 + 2*x + 1
634 False
```

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

637 Another important property of SymPy expressions is that they are immutable. This simplifies
638 the design of SymPy, and enables expression interning. It also enables expressions to be hashed,
639 which allows expressions to be used as keys in Python dictionaries, and is used to implement
640 caching in SymPy.

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

648 5.2 Assumptions

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

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

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

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

661 ~~Some of the common assumptions that SymPy allows are `positive`, `negative`, `real`, `nonpositive`,~~
662 ~~`integer`, `prime` and `commutative`.¹⁴ Assumptions on any object can be checked with the `assumption`~~
663 ~~attributes, like —~~

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

¹² The `dotprint` function from the `sympy.printing.dot` submodule prints output to dot format, which can be
668 rendered with `Graphviz` to visualize expression trees graphically.

¹³ `expr.func` is used instead of `type(expr)` to allow the function of an expression to be distinct from its actual
669 Python class. In most cases the two are the same.

¹⁴ ~~If `A` and `B` are `Symbols` created with `commutative=False` then SymPy will keep `A-B` and `B-A` distinct.~~

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.isSUBSCRIPTNbreal` and `I.isSUBSCRIPTNBprime` 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).isSUBSCRIPTNBpositive` 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).isSUBSCRIPTNBirrational` gives `None`, because determining whether $\pi + e$ is rational or irrational is an open problem in mathematics [31].

Basic implications between the facts are used to deduce assumptions. For instance, the assumptions system knows that being an integer implies being rational, so `Symbol('x', integer=True).isSUBSCRIPTNBirrational` returns `True`. Furthermore, expressions compute the assumptions on themselves based on the assumptions of their arguments. For instance, if `x` and `y` are both created with `positive=True`, then `(x + y).isSUBSCRIPTNBpositive` will be `True` whereas `(x - y).isSUBSCRIPTNBpositive` will be `None`.

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 because, 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`. All As it was stated before, all SymPy classes used for expression trees¹⁴ should be subclasses of the base class `Basic`, which defines some basic methods for symbolic expression trees. `Expr` is the `Basic` subclass for mathematical expressions objects 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`.¹⁴

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 an evaluated value for the function application (usually an instance of some other class, i.e.e.g., a `Number`), or `None`, if for given arguments that function if for the given arguments it 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 the `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 1 presents a minimal an example of this extensibility. It gives a stripped down version of the `gamma` function $\Gamma(x)$ from SymPy, which evaluates. The methods defined allow it to evaluate itself on positive integer arguments, has the positive and real assumptions defined, can define the real assumption, allow it to be rewritten in terms of factorial (with `gamma(x).rewrite(factorial)`, and can), and allow it to be differentiated. `self.func` is used throughout instead of referencing `gamma` explicitly so that potential subclasses of `gamma` can reuse the methods.

¹⁴Some internal classes, such as those used in the polynomial module, do not follow this rule for efficiency reasons.

¹⁴See the supplement for more information on the `sympy.logic` submodule.

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

```

719 from sympy import \DIFdelbegin \DIFdel{Integer, Function, floor}\DIFdelend \DIFaddbegin \DIFadd{Fun
720
721 class gamma(Function)\DIFaddbegin \DIFadd{:
722     }\DIFaddend @classmethod
723     def eval(cls, arg):
724         if isinstance(arg, Integer) and arg.is_positive:
725             return factorial(arg - 1)
726
727     def _eval_is\DIFdelbegin \DIFdel{_positive(self):
728         x = self.args}\DIFDELCMD < [%%
729 \DIFdel{0}%DIFDELCMD < ]
730 %DIFDELCMD <      %%
731 \DIFdel{if x.is_positive:
732     return True
733     elif x.is_noninteger:
734         return floor(x).is_even
735 }\DIFDELCMD <
736
737 %DIFDELCMD <      %%
738 \DIFdel{def _eval_is}\DIFdelend _real(self):
739     x = self.args[0]
740     # noninteger means real and not integer
741     if x.is_positive or x.is_noninteger:
742         return True
743
744     def _eval_rewrite_as_factorial(self, z):
745         return factorial(z - 1)
746
747     def fdiff(self, argindex=1):
748         from sympy.core.function import ArgumentIndexError
749         if argindex == 1:
750             return self.func(self.args[0])*polygamma(0, self.args[0])
751         else:
752             raise ArgumentIndexError(self, argindex)

```

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

755 5.3 Performance

756 Due to being written in pure Python without the use of extension modules, SymPy's performance
757 characteristics are generally poorer than its commercial competitors. For many applications,
758 the performance of SymPy, as measured by clock cycles, memory usage, and memory layout, is
759 sufficient. However, the boundaries for when SymPy's pure Python strategy becomes insufficient
760 are when the user requires handling of very long expressions or many small expressions. Where
761 this boundray lies depends on the system at hand, but tends to be within the range of 10^4 – 10^6
762 symbols for modern computers.

763 For this reason, a new project called SymEngine [60] has been started. The aim of this
764 project is to develop a library with better performance characteristics for symbolic manipulation.
765 SymEngine is a pure C++ library, which allows it fine-grained control over the memory layout
766 of expressions. SymEngine has thin wrappers to other languages (Python, Ruby, Julia, etc.).
767 Its aim is to be the fastest symbolic manipulation library. Preliminary benchmarks suggest that
768 SymEngine is as performant as its commercial and open source competitors.

769 The development version of SymPy has recently started to use SymEngine as an optional
770 backend, initially in `sympy.physics.mechanics` only. Future work will involve allowing more
771 algorithms in SymPy to use SymEngine as a backend.

6 ACKNOWLEDGEMENTS PROJECTS THAT DEPEND ON SYMPY

There are several projects that depend on SymPy as a library for implementing a part of their functionality. A selection of these projects are listed in Table 3.

Table 3. Selected projects that depend on SymPy.

Project name	Description
SymPy Gamma	An open source analog of Wolfram Alpha that uses SymPy [61]. There is more information about SymPy Gamma supplementary material.
Cadabra	A CAS designed specifically for the resolution of problems encountered in field theory [39].
GNU Octave Symbolic Package	An implementation of a symbolic toolbox for Octave using SymPy [59].
SymPy.jl	A Julia interface to SymPy, provided using PyCall [62].
Mathics	A free, online CAS featuring Mathematica compatible syntax and functions [56].
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 [33].
IKFast	A robot kinematics compiler provided by OpenRAVE [12].
SageMath	A free open-source mathematics software system, which builds on top of many existing open-source packages, including SymPy [58].
PyDy	Multibody Dynamics with Python [19].
galgebra	A Python package for geometric algebra (previously sympy.galgebra) [5].
yt	A Python package for analyzing and visualizing volumetric data [64].
SfePy	A Python package for solving partial differential equations (PDEs) in 1D, 2D, and 3D by the finite element (FE) method [66, 10].
Quameon	Quantum Monte Carlo in Python [57].
Lcapy	An experimental Python package for teaching linear circuit analysis [55].

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 CASs, 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 has submodules for many areas of mathematics. This includes functions for supports a wide array of mathematical facilities. These include functions for assuming and deducing common mathematical facts, simplifying expressions, performing common calculus operations, manipulating polynomials, pretty printing expressions, solving equations, and representing symbolic matrices. Other included areas are 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. SymPy has strong support for arbitrary precision numerics, backed by the mpmath package. Additionally, SymPy contains submodules targeting certain specific physics 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 many of these users~~ chose SymPy because of its ease of access. SymPy is a dependency of many external projects across a wide spectrum of domains.

SymPy expressions are immutable trees of Python objects. 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 uses Python both as the internal language and the user language. This permits users to access the same methods used by the library itself in order to extend it for their needs.

Some of the planned future work for SymPy includes work on improving code generation, improvements to the speed of SymPy ~~(one area of work in this direction is , a C++ symbolic manipulation library that is planned to be usable as an alternative core for SymPy) using~~ SymEngine, improving the assumptions system, and improving the solvers ~~module~~submodule.

REFERENCES

- [1] Adams, W. W. and Loustau, P. (1994). *An introduction to Gröbner bases*. Number 3. American Mathematical Society.
- [2] Bailey, D. H., Jeyabalan, K., and Li, X. S. (2005). A comparison of three high-precision quadrature schemes. *Experimental Mathematics*, 14(3):317–329.
- [3] Bender, C. M. and Orszag, S. A. (1999). *Advanced Mathematical Methods for Scientists and Engineers*. Springer, 1st edition.
- [4] Biggs, N., Lloyd, E. K., and Wilson, R. J. (1976). *Graph Theory, 1736-1936*. Oxford University Press.
- [5] Bromborsky, A. (2016). *geometric algebra/calculus modules for SymPy*. <https://github.com/brombo/galgebra>.
- [6] Bronstein, M. (2005a). pmint—The Poor Man’s Integrator. <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint>.
- [7] Bronstein, M. (2005b). *Symbolic Integration I: Transcendental Functions*. Springer-Verlag, New York, NY, USA.
- [8] Buchberger, B. (1965). *Ein Algorithmus zum Auffinden der Basis Elemente des Restklassenrings nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, Innsbruck, Austria.
- ~~[9] Carette, J. (2004). Understanding Expression Simplification. In ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, pages 72–79, New York, NY, USA. ACM Press.~~
- [9] Cervone, D. (2012). Mathjax: a platform for mathematics on the web. *Notices of the AMS*, 59(2):312–316.
- [10] 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>.
- [11] Ciurana, E. (2009). Google app engine. *Developing with Google App Engine*, pages 1–10.
- [12] Diankov, R. (2010). Ikfast: The robot kinematics compiler. http://openrave.org/docs/latest_stable/openravepy/ikfast/.
- [13] Doorenbos, R. B. (1995). *Production matching for large learning systems*. PhD thesis, University of Southern California.
- [14] Faugère, J. C. (1999). A New Efficient Algorithm for Computing Gröbner Bases (F4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88.
- [15] Faugère, J. C. (2002). A New Efficient Algorithm for Computing Gröbner Bases Without Reduction To Zero (F5). In *ISSAC '02: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation*, pages 75–83, New York, NY, USA. ACM Press.
- [16] Fetter, A. and Walecka, J. (2003). *Quantum Theory of Many-Particle Systems*. Dover Publications.
- [17] Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., and Zimmermann, P. (2007). Mpf: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2).

- [18] Fu, H., Zhong, X., and Zeng, Z. (2006). Automated and Readable Simplification of Trigonometric Expressions. *Mathematical and Computer Modelling*, 55(11-12):1169–1177.
- [19] Gede, G., Peterson, D. L., Nanjangud, A. S., Moore, J. K., and Hubbard, M. (2013). Constrained multibody dynamics with Python: From symbolic equation generation to publication. In *ASME 2013 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages V07BT10A051–V07BT10A051. American Society of Mechanical Engineers.
- [20] Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48.
- [21] Gosper, R. W. (1978). Decision procedure for indefinite hypergeometric summation. *Proceedings of the National Academy of Sciences*, 75(1):40–42.
- [22] Gruntz, D. (1996). *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland.
- [23] Horsen, C. V. (2015). GMPY. <https://pypi.python.org/pypi/gmpy2>.
- [24] Hudak, P. (1998). Domain specific languages. In [Salas, P. H., editor, Handbook of Programming Languages, Vol. III: Little Languages and Tools](#), chapter 3, pages 39–60. MacMillan, Indianapolis.
- [25] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.
- [26] [Johansson, F. et al. \(2014\). mpmath: a Python library for arbitrary-precision floating-point arithmetic \(version 0.19\). <http://mpmath.org/>.](#)
- [27] [Jones, E., Oliphant, T., Peterson, P., et al. \(2001–\). SciPy: Open source scientific tools for Python. \[Online; accessed 2016-09-28\].](#)
- [28] Kane, T. R. and Levinson, D. A. (1985). *Dynamics, Theory and Applications*. McGraw Hill.
- [29] [Khuyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., et al. \(2016\). Jupyter notebooks—a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas: Proceedings of the 20th International Conference on Electronic Publishing*, page 87. IOS Press.](#)
- [30] Lagrange, J. (1811). *Mécanique analytique*. Number v. 1 in *Mécanique analytique*. Ve Courcier.
- [31] Lang, S. (1966). Introduction to transcendental numbers. *Reading, Mass.*
- [32] Lutz, M. (2013). *Learning Python*. O’Reilly Media, Inc.
- [33] [Mathpix, Inc. \(2016\). Mathpix — Solve and graph math using pictures. <http://mathpix.com/>.](#)
- [34] [Moses, J. \(1971\). Algebraic Simplification: A Guide for the Perplexed. In *SYMSAC ’71: Proceedings of the second ACM Symposium on Symbolic and Algebraic Computation*, pages 282–304. New York, NY, USA. ACM Press.](#)
- [35] Nielsen, M. and Chuang, I. (2011). *Quantum Computation and Quantum Information*. Cambridge University Press.
- [36] Nijenhuis, A. and Wilf, H. S. (1978). *Combinatorial Algorithms: For Computers and Calculators*. Academic Press, New York, NY, USA, second edition.
- [37] Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20.
- [38] Paprocki, M. (2010). Design and implementation issues of a computer algebra system in an interpreted, dynamically typed programming language. Master’s thesis, University of Technology of Wrocław, Poland.
- [39] [Peeters, K. \(2007\). Cadabra: a field-theory motivated symbolic computer algebra system. *Computer Physics Communications*.](#)
- [40] Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29.
- [41] Peterson, D. L., Gede, G., and Hubbard, M. (2014). Symbolic linearization of equations of motion of constrained multibody systems. *Multibody System Dynamics*, 33(2):143–161.
- [42] Petkovšek, M., Wilf, H. S., and Zeilberger, D. (1996). *A=BAK peters*. Wellesley, MA.

- [43] Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49.
- [44] Roach, K. (1996). Hypergeometric function representations. In *ISSAC '96: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation*, pages 301–308, New York, NY, USA. ACM Press.
- [45] Roach, K. (1997). Meijer G function representations. In *ISSAC '97: Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 205–211, New York, NY, USA. ACM.
- [46] Rocklin, M. (2013). *Mathematically informed linear algebra codes through term rewriting*. PhD thesis, PhD Thesis, August.
- [47] Rocklin, M. and Terrel, A. R. (2012). Symbolic statistics with SymPy. *Computing in Science and Engineering*, 14.
- [48] Rose, K. H. et al. (1999). *Xy-pic user's guide*.
- [49] Rosen, L. (2005). *Open source licensing*, volume 692. Prentice Hall.
- [50] Sakurai, J. and Napolitano, J. (2010). *Modern Quantum Mechanics*. Addison-Wesley.
- [51] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. Prentice Hall Ordering Information.
- ~~[51] Stein, W. and Joyner, D. (2005). *SAGE: System for Algebra and Geometry Experimentation*. Communications in Computer Algebra, 39(2).~~
- [52] Sussman, G. J. and Wisdom, J. (2013). *Functional Differential Geometry*. Massachusetts Institute of Technology Press.
- [53] Tai, C.-T. (1997). *Generalized vector and dyadic analysis: applied mathematics in field theory*, volume 9. Wiley-IEEE Press.
- [54] Takahasi, H. and Mori, M. (1974). Double exponential formulas for numerical integration. *Publications of the Research Institute for Mathematical Sciences*, 9(3):721–741.
- [55] The Lcapy Developers (2016). *Lcapy, a Python package for linear circuit analysis*. <http://lcapy.elec.canterbury.ac.nz/>.
- [56] The Mathics Developers (2016). *Mathics, a free, general-purpose online computer algebra system featuring Mathematica-compatible syntax and functions*. <https://mathics.github.io/>.
- [57] The Quameon Developers (2016). *Quameon, Quantum Monte Carlo in Python*. <http://quameon.sourceforge.net/>.
- [58] The Sage Developers (2016). *SageMath, the Sage Mathematics Software System*. <http://www.sagemath.org>.
- [59] The Symbolic Package Developers (2016). *The Symbolic Package for GNU Octave*. <http://octave.sourceforge.net/symbolic>.
- [60] The SymPy Developers (2016a). *SymEngine, a fast symbolic manipulation library, written in C++*. <https://github.com/symengine/symengine>.
- [61] The SymPy Developers (2016b). *SymPy Gamma*. <http://www.sympygamma.com/>.
- [62] The SymPy.jl Developers (2016). *SymPy.jl, a package to bring Python's SymPy functionality into Julia via PyCall*. <https://github.com/JuliaPy/SymPy.jl>.
- [63] Toth, V. T. (2007). Maple and Meijer's G-function: a numerical instability and a cure. <http://www.vttoth.com/CMS/index.php/technical-notes/67>.
- [64] Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., and Norman, M. L. (2011). yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement Series*, 192:9–9.
- [65] Zare, R. (1991). *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics*. Wiley.
- [66] Zienkiewicz, O., Taylor, R., and Zhu, J. (2013). *The Finite Element Method: Its Basis and Fundamentals*. Butterworth-Heinemann, seventh edition edition.