A peer-reviewed version of this preprint was published in PeerJ on 2 January 2017.

View the peer-reviewed version (peerj.com/articles/cs-103), which is the preferred citable publication unless you specifically need to cite this preprint.

https://doi.org/10.7717/peerj-cs.103
SymPy: Symbolic Computing in Python

Aaron Meurer1, Christopher P. Smith2, Mateusz Paprocki3, Ondrej 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).
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 Daluapth, Bawana Road, New Delhi 110042, India (dtu.amit@gmail.com).
8University 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 (singhsarta94@gmail.com).
11University of Moratuwa, Bandaranayake Mawatha, Katubedda, Moratuwa 10400, Sri Lanka (thilinanrmtb.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).
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...
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 [25]. It is free and open source software, licensed under the 3-clause BSD license [37]. 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 [33]. 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. Due in part to this focus, it has become a popular language for scientific computing and data science, with a broad ecosystem of libraries [28]. SymPy is itself used by many libraries and tools to support research within a variety of domains, such as Sage [40] (pure mathematics), yt [45] (astronomy and astrophysics), PyDy [15] (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 [20]. 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 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 printers with Jupyter [30] frontends, including the Notebook and Qt Console, which will render SymPy expressions using MathJax [9] or LaTeX.
components of SymPy, provides justifications for the design decisions that have been made, and
gives example user-facing code as appropriate.

2.1 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.

```python
>>> from sympy import *
```

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,

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

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 `x`, `y`, and `z` 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 \((x^2 - 2x + 3)/y\).

```python
>>> (x**2 - 2*x + 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.

2.2 The Core
A computer algebra system (CAS) represents 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 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 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 \(xy + 2\):

```python
>>> expr = x*y + 2
```

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

```python
>>> type(expr)
<class 'sympy.core.add.Add'>
```

```python
>>> 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.

---

1 All examples in this paper use SymPy version 1.0.
>>> expr.args[0]
2
>>> type(expr.args[0])
<class 'sympy.core.numbers.Integer'>

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

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

Every SymPy expression satisfies a key identity invariant:

```python
eval(expr) == expr
```

This means that expressions are rebuildable from their `args` 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 

```python
x + 1)^2 and x^2 + 2x + 1 are equal mathematically, SymPy gives
```

```python
>>> (x + 1)**2 == x**2 + 2*x + 1
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.

### 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.

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

By assuming the most general case, that symbols are 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.

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

`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.
Some of the common assumptions that SymPy allows are positive, negative, real, nonpositive,
integer, prime and commutative. 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, because determining whether \( \pi + e \) is rational or irrational is an open problem in mathematics [24].

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).is_rational` 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 \( (x + y).is_positive \) will be True whereas \( (x - y).is_positive \) will be None.

### 2.4 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 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 SymPy classes used for expression trees should be
subclasses of the base class Basic, which defines some basic methods for symbolic expression trees.
Expr is the subclass for mathematical expressions that can be added and multiplied together.
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 4 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
\( \text{gamma}(x).\text{rewrite}(\text{factorial}) \), and can be differentiated. `self.func` is used throughout instead of
referencing `gamma` explicitly so that potential subclasses of `gamma` can reuse the methods.

---

### Footnotes

1. If \( A \) and \( B \) are Symbols created with `commutative=False` then SymPy will keep \( A \cdot B \) and \( B \cdot A \) distinct.
2. Some internal classes, such as those used in the polynomial module, do not follow this rule for efficiency reasons.
Listing 1. A minimal implementation of `sympy.gamma`.

```python
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):
        return factorial(z - 1)

    def fdiff(self, argindex=1):
        from sympy.core.function import ArgumentIndexError
        if argindex == 1:
            return self.func(self.args[0]) * polygamma(0, self.args[0])
        else:
            raise ArgumentIndexError(self, argindex)
```

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

3 FEATURES

Although SymPy’s extensive feature set cannot be covered in-depth in this paper, calculus and other bedrock areas are discussed in their own subsections. Additionally, Table 1 gives a compact listing of all major capabilities present in the SymPy codebase. This grants a sampling from the breadth of topics and application domains that SymPy services. Unless stated otherwise, all features noted in Table 1 are symbolic in nature. Numeric features are discussed in Section 4.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculus</td>
<td>Algorithms for computing derivatives, integrals, and limits.</td>
</tr>
<tr>
<td>Category Theory</td>
<td>Representation of objects, morphisms, and diagrams. Tools for drawing diagrams with Xy-pic.</td>
</tr>
<tr>
<td>Code Generation</td>
<td>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.</td>
</tr>
<tr>
<td>Combinatorics &amp; Group Theory</td>
<td>Permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [27], and Prufer sequences [4].</td>
</tr>
</tbody>
</table>
Concrete Math

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 \[32\] for two univariate polynomials.

Cryptography

Block and stream ciphers, including shift, Affine, substitution, Vigenère's, Hill's, bfoo, RSA, Kid RSA, linear-feedback shift registers, and Elgamal encryption.

Differential Geometry

Representations of manifolds, metrics, tensor products, and coordinate systems in Riemannian and pseudo-Riemannian geometries \[41\].

Geometry

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

Representations of Lie algebras and root systems.

Logic

Boolean expressions, equivalence testing, satisfiability, and normal forms.

Matrices

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

Matrices with symbolic dimensions (unspecified entries). Block matrices.

Number Theory

Prime number generation, primality testing, integer factorization, continued fractions, Egyptian fractions, modular arithmetic, quadratic residues, partitions, binomial and multinomial coefficients, prime number tools, hexadecimal digits of \(\pi\), and integer factorization.

Plotting

Hooks for visualizing expressions via matplotlib \[21\] or as text drawings when lacking a graphical back-end. 2D function plotting, 3D function plotting, and 2D implicit function plotting are supported.

Polynomials

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

Functions for printing SymPy expressions in the terminal with ASCII or Unicode characters and converting SymPy expressions to \(\LaTeX\) and MathML.

Quantum Mechanics

Quantum states, bra–ket notation, operators, basis sets, representations, tensor products, inner products, outer products, commutators, anticommutators, and specific quantum system implementations.

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

Representations of empty, finite, and infinite sets. This includes special sets such as for all natural, integer, and complex numbers. Operations on sets such as union, intersection, Cartesian product, and building sets from other sets are supported.

Simplification

Functions for manipulating and simplifying expressions. Includes algorithms for simplifying hypergeometric functions, trigonometric expressions, rational functions, combinatorial functions, square root denesting, and common subexpression elimination.

Solvers

Functions for symbolically solving equations, systems of equations, both linear and non-linear, inequalities, ordinary differential equations, partial differential equations, Diophantine equations, and recurrence relations.
Special 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 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 [36].

Tensors Symbolic manipulation of indexed objects.
Vectors Basic operations on vectors and differential calculus with respect to
3D Cartesian coordinate systems.

3.1 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 [8]. 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 2 lists common simplification
functions.

Table 2. Some SymPy Simplification Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expand</td>
<td>expand the expression</td>
</tr>
<tr>
<td>factor</td>
<td>factor a polynomial into irreducibles</td>
</tr>
<tr>
<td>collect</td>
<td>collect polynomial coefficients</td>
</tr>
<tr>
<td>cancel</td>
<td>rewrite a rational function as p/q with common factors canceled</td>
</tr>
<tr>
<td>apart</td>
<td>compute the partial fraction decomposition of a rational function</td>
</tr>
<tr>
<td>trigsimp</td>
<td>simplify trigonometric expressions [14]</td>
</tr>
<tr>
<td>hyperexpand</td>
<td>expand hypergeometric functions [34] [35]</td>
</tr>
</tbody>
</table>

3.2 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 [18] for computing
symbolic limits and heuristics (a description of the Gruntz algorithm may be found in the
supplement). For example, the following computes \( \lim_{x \to \infty} x \sin \left( \frac{1}{x} \right) = 1 \). Note that SymPy denotes
\( \infty \) as oo.

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

As a more complex example, SymPy computes

\[
\lim_{x \to 0} \left( 2e^{\frac{1-\cos (x)}{\sin (x)}} - 1 \right) \frac{\sinh(x)}{\sqrt{\tan^{2}(x)}} = e.
\]

[5]The measure parameter of the simplify function lets 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.
Derivatives are computed with the \texttt{diff} function, which recursively uses the various differentiation rules.

\begin{verbatim}
>>> diff(sin(x)*exp(x), x)
exp(x)*sin(x) + exp(x)*cos(x)
\end{verbatim}

Integrals are calculated with the \texttt{integrate} function. SymPy implements a combination of the Risch algorithm \cite{risch}, table lookups, a reimplementation of Manuel Bronstein’s “Poor Man’s Integrator” \cite{bronstein}, and an algorithm for computing integrals based on Meijer G-functions \cite{meijer, g-function}. 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

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

Summations are computed with \texttt{summation} using a combination of Gosper’s algorithm \cite{gosper}, an algorithm that uses Meijer G-functions \cite{meijer, g-function}, and heuristics. Products are computed with \texttt{product} function via a suite of heuristics.

\begin{verbatim}
>>> 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)
\end{verbatim}

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. Both are used in the univar-
iate 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) [11, 12]. 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 [29], are better expressed when the representation
is dense. The dense multivariate representation is specifically a recursively-dense representa-
tion, where polynomials in $K[x_0, x_1, ..., x_n]$ are viewed as a polynomials in $K[x_0][x_1][...][x_n]$. 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 symy.polys module can be found in the supplement.

3.4 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
far. The str form of an expression is valid Python and roughly matches what a user would type
to enter the expression.

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

Expressions 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.

```python
>>> pprint(Integral(sqrt(phi0 + 1), phi0))
⌠
⎮ ________
⎮ ╲╱ φ₀ + 1 d(φ₀)
⌡
```

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

```python
>>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
/  
|  
| \sqrt{phi0 + 1} d(phi0)
|  
/  
```

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

```python
>>> 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 [30], the \LaTeX printer is used to render expressions

\footnote{In 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.}
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.

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

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

\begin{verbatim}
solve(f, *symbols, **flags)
solveset(f, symbol, domain=S.Complexes)
\end{verbatim}

The \texttt{domain} parameter is typically either \texttt{S.Complexes} (the default) or \texttt{S.Reals}; the latter causes \texttt{solveset} to only return real solutions.

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

Both functions implicitly assume that expressions are equal to 0. For instance, \texttt{solveset(x - 1, x)} solves \(x - 1 = 0\) for \(x\).

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

More examples of \texttt{solveset} and \texttt{solve} can be found in the supplement.

3.6 Matrices
Besides being an important feature in its own right, computations on matrices with symbolic entries are important for many algorithms within SymPy. The following code shows some basic usage of the \texttt{Matrix} class.

\begin{verbatim}
>>> A = Matrix(2, 2, [x, x + y, y, x])
>>> A
Matrix([[x, x + y],
        [y, x]])
\end{verbatim}

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

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

\begin{verbatim}
>>> A.eigenvals()
\{x - sqrt(y*(x + y)): 1, x + sqrt(y*(x + y)): 1\}
\end{verbatim}

Internally these matrices store the elements as lists of lists, making it a dense representation.\footnote{Similar to the polynomials module, dense here means that all entries are stored in memory, contrasted with a sparse representation where only nonzero entries are stored.}

For storing sparse matrices, the \texttt{SparseMatrix} class can be used. Sparse matrices store their elements as a dictionary of keys.
SymPy also supports matrices with symbolic dimension values. `MatrixSymbol` represents a matrix with dimensions $m \times n$, where $m$ and $n$ can be symbolic. Matrix addition and multiplication, scalar operations, matrix inverse, and transpose are stored symbolically as matrix expressions.

Block matrices are also implemented in SymPy. `BlockMatrix` elements can be any matrix expression, including explicit matrices, matrix symbols, and other block matrices. All functionalities of matrix expressions are also present in `BlockMatrix`.

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

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

4 NUMERICS

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:

```python
>>> Float(1.1)
1.10000000000000
>>> Float(1.1, 30) # precision equivalent to 30 digits
1.10000000000000000000000000000
>>> Float("1.1", 30)
1.10000000000000000000000000000
```

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

```python
>>> (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 [10]. A notorious case is that of catastrophic cancellation:

```python
>>> 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:

```python
>>> (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.

4.1 The mpmath library

The implementation of arbitrary-precision floating-point arithmetic is supplied by the mpmath library. Originally, it was developed as a SymPy module 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:

```python
>>> import mpmath
>>> mpmath.mp.dps = 30  # 30 digits of precision
>>> mpmath.mpf("0.1") + mpmath.exp(-50)
mpf('0.100000000000000000000192874984794')
```

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. 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 [19] is installed, mpmath automatically uses the `gmpy.mpz` type for $x$, and GMPY methods for rounding-related operations, improving performance.

The mpmath library supports special functions, root-finding, linear algebra, polynomial approximation, and numerical computation of limits, derivatives, integrals, infinite series, and ODE solutions. All features work in arbitrary precision and use algorithms that 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 [43, 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 Shank's 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 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 $G_{4,3}^{3,0}(0; \frac{1}{2},-1,-\frac{3}{2}|x)$ is a good test case [43]. Past versions of both Maple and Mathemtica 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:

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

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

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

Symbolic integration and summation often produces hypergeometric and Meijer G-function closed forms (see Subsection 3.2); numerical evaluation of such special functions is a useful complement to direct numerical integration and summation.

### 5 DOMAIN SPECIFIC SUBMODULES

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

---

8 All examples in this section use mpmath version 0.19.
5.1 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.

5.1.1 Vector Algebra

The `sympy.physics.vector` package provides reference frame-, time-, and space-aware vector and dyadic objects that allow for three-dimensional operations such as addition, subtraction, scalar multiplication, inner and outer products, and cross products. Both of these objects 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. 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, π/2, and π/3 rad, 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 π/2 rad.

```python
>>> from sympy.physics.vector import ReferenceFrame
>>> A = ReferenceFrame('A')
>>> B = ReferenceFrame('B')
>>> C = ReferenceFrame('C')
>>> 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
```

5.1.2 Mechanics

The `sympy.physics.mechanics` package utilizes the `sympy.physics.vector` package 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 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.

5.2 Quantum Mechanics

The `sympy.physics.quantum` package has extensive capabilities for performing symbolic 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.

```python
>>> from sympy.physics.quantum import Commutator, Dagger, Operator
>>> from sympy.physics.quantum import Ket, qapply
```
>>> A = Operator('A')
>>> B = Operator('B')
>>> C = Operator('C')
>>> D = Operator('D')
>>> a = Ket('a')

```
>>> comm = Commutator(A, B)
>>> comm
[A,B]
```

```
>>> qapply(Dagger(comm*a)).doit()
-<a|*(Dagger(A)*Dagger(B) - Dagger(B)*Dagger(A))
```

Commutators can be expanded using common commutator identities:

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

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

- Many of the exactly solvable quantum systems, including simple harmonic oscillator states and raising/lowering operators, infinite square well states, and 3D position and momentum operators and states.

- Second quantized formalism of non-relativistic many-body quantum mechanics [13].

- Quantum angular momentum [46]. Spin operators and their eigenstates can be represented in any basis and for any quantum numbers. A rotation operator representing the Wigner-D matrix, which may be defined symbolically or numerically, is also implemented to rotate spin eigenstates. Functionality for coupling and uncoupling of arbitrary spin eigenstates is provided, including symbolic representations of Clebsch-Gordon coefficients and Wigner symbols.

- Quantum information and computing [26]. Multidimensional qubit states, and a full set of one- and two-qubit gates are provided and can be represented symbolically or as matrices/vectors. With these building blocks, it is possible to implement a number of basic quantum algorithms including the quantum Fourier transform, quantum error correction, quantum teleportation, Grover’s algorithm, dense coding, etc. In addition, any quantum circuit may be plotted using the `circuit_plot` function (Figure 1).

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

```
>>> from sympy.physics.quantum.qubit import Qubit
>>> from sympy.physics.quantum.gate import XGate
>>> q = Qubit('0101')
>>> q
|0101>
>>> X = XGate(1)
>>> qapply(X*q)
|0111>
```

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

```
>>> Dagger(q)
<0101|
>>> ip = Dagger(q)*q
>>> ip
<0101|0101>
>>> ip.doit()
1
```
Quantum gates (unitary operators) can be applied to transform these states and then classical measurements can be performed on the results:

```python
>>> from sympy.physics.quantum.qubit import measure_all
>>> from sympy.physics.quantum.gate import H, X, Y, Z
>>> c = H(0)*H(1)*Qubit('00')
>>> c
H(0)*H(1)*|00>
>>> q = qapply(c)
>>> measure_all(q)
[(|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:

6 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 simplifying expressions, performing common calculus operations, pretty printing expressions, solving equations, and representing symbolic matrices. Other included areas are discrete math, concrete math, plotting, geometry, statistics, polynomials, sets, series, vectors, combinatorics, group theory, code generation, tensors, Lie algebras, cryptography, and special functions. Additionally, SymPy contains submodules targeting certain specific domains, such as classical mechanics and
quantum mechanics. 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 (one area of work in this direction is SymEngine, a C++ symbolic manipulation library that is planned to be usable as an alternative core for SymPy), improving the assumptions system, and improving the solvers module.

7 ACKNOWLEDGEMENTS

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.

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.

Google Summer of Code is an international annual program in which Google awards stipends to all students who successfully complete a requested free and open-source software coding project during the summer.

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

REFERENCES


