

# SymPy: Symbolic Computing in Python

Aaron Meurer<sup>1</sup>, Christopher P. Smith<sup>2</sup>, Mateusz Paprocki<sup>3</sup>, Ondřej Čertík<sup>4</sup>, Sergey B. Kirpichev<sup>5</sup>, Matthew Rocklin<sup>6</sup>, AMiT Kumar<sup>7</sup>, Sergiu Ivanov<sup>8</sup>, Jason K. Moore<sup>9</sup>, Sartaj Singh<sup>10</sup>, Thilina Rathnayake<sup>11</sup>, Sean Vig<sup>12</sup>, Brian E. Granger<sup>13</sup>, Richard P. Muller<sup>14</sup>, Francesco Bonazzi<sup>15</sup>, Harsh Gupta<sup>16</sup>, Shivam Vats<sup>17</sup>, Fredrik Johansson<sup>18</sup>, Fabian Pedregosa<sup>19</sup>, Matthew J. Curry<sup>20</sup>, Andy R. Terrel<sup>21</sup>, Štěpán Roučka<sup>22</sup>, Ashutosh Saboo<sup>23</sup>, Isuru Fernando<sup>24</sup>, Sumith Kulal<sup>25</sup>, Robert Cimrman<sup>26</sup>, and Anthony Scopatz<sup>27</sup>

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

<sup>2</sup>Polar Semiconductor, Inc., Bloomington, MN 55425 (smichr@gmail.com).

<sup>3</sup>Continuum Analytics, Inc., Austin, TX 78701 (mattpap@gmail.com).

<sup>4</sup>Los Alamos National Laboratory, Los Alamos, NM 87545 (certik@lanl.gov).

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

<sup>6</sup>Continuum Analytics, Inc., Austin, TX 78701 (mrocklin@gmail.com).

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

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

<sup>9</sup>University of California, Davis, Davis, CA 95616 (jkm@ucdavis.edu).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

<sup>24</sup>University of Moratuwa, Bandaranayake Mawatha, Katubedda, Moratuwa 10400, Sri Lanka (isuru.11@cse.mrt.ac.lk).

<sup>25</sup>Indian Institute of Technology Bombay, Powai, Mumbai, Maharashtra 400076, India (sumith@cse.iitb.ac.in).

<sup>26</sup>New Technologies – Research Centre, University of West Bohemia, Univerzitní 8, 306

## 55 **ABSTRACT**

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

62 **Keywords:** symbolic, Python, computer algebra system

## 63 **1 INTRODUCTION**

64 SymPy is a full featured computer algebra system (CAS) written in the Python programming  
65 language [26]. It is free and open source software, licensed under the 3-clause BSD license [38].  
66 The SymPy project was started by Ondřej Čertík in 2005, and it has since grown to over 500  
67 contributors. Currently, SymPy is developed on GitHub using a bazaar community model [34].  
68 The accessibility of the codebase and the open community model allow SymPy to rapidly respond  
69 to the needs of users and developers.

70 Python is a dynamically typed programming language that has a focus on ease of use and  
71 readability. Due in part to this focus, it has become a popular language for scientific computing  
72 and data science, with a broad ecosystem of libraries [29]. SymPy is itself used by many libraries  
73 and tools to support research within a variety of domains, such as Sage [41] (pure mathematics),  
74 yt [46] (astronomy and astrophysics), PyDy [15] (multibody dynamics), and SfePy [10] (finite  
75 elements).

76 Unlike many CASs, SymPy does not invent its own programming language. Python itself  
77 is used both for the internal implementation and end user interaction. By using the operator  
78 overloading functionality of Python, SymPy follows the embedded domain specific language  
79 paradigm proposed by Hudak [20]. The exclusive usage of a single programming language makes  
80 it easier for people already familiar with that language to use or develop SymPy. Simultaneously,  
81 it enables developers to focus on mathematics, rather than language design.

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

87 As a library, SymPy does not have a built-in graphical user interface (GUI). However, SymPy  
88 exposes a rich interactive display system, and supports registering printers with Jupyter [31]  
89 frontends, including the Notebook and Qt Console, which will render SymPy expressions using  
90 MathJax [9] or L<sup>A</sup>T<sub>E</sub>X.

91 The remainder of this paper discusses key components of the SymPy software. Section 2  
92 discusses the architecture of SymPy. Section 3 enumerates the features of SymPy and takes  
93 a closer look at some of the important ones. The section 4 looks at the numerical features of  
94 SymPy and its dependency library, mpmath. Section 5 looks at the domain specific physics  
95 submodules for performing symbolic and numerical calculations in classical mechanics and  
96 quantum mechanics. Conclusions and future directions for SymPy are given in section 6.

## 97 **2 ARCHITECTURE**

98 Software architecture is of central importance in any large software project because it establishes  
99 predictable patterns of usage and development [40]. This section describes the essential structural

100 components of SymPy, provides justifications for the design decisions that have been made, and  
101 gives example user-facing code as appropriate.

## 102 2.1 Basic Usage

103 The following statement imports all SymPy functions into the global Python namespace.<sup>1</sup> From  
104 here on, all examples in this paper assume that this statement has been executed.<sup>2</sup>

```
105 >>> from sympy import *
```

106 Symbolic variables, called symbols, must be defined and assigned to Python variables before  
107 they 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, these  
111 symbols are all assigned to Python variables of the same name. However, the user is free to  
112 assign them to different Python variables, while representing the same symbol, such as `a`, `b`,  
113 `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  $(x^2 - 2x + 3)/y$ .

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

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

## 123 2.2 The Core

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

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

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

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

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

```
138 >>> type(expr)  
139 <class 'sympy.core.add.Add'>  
140 >>> expr.args  
141 (2, x*y)
```

---

<sup>1</sup>`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.

<sup>2</sup>All examples in this paper use SymPy version 1.0.

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.

```
145 >>> expr.args[0]
146 2
147 >>> type(expr.args[0])
148 <class 'sympy.core.numbers.Integer'>
149 >>> expr.args[0].args
150 ()
```

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<sup>3</sup> with all the nested class constructor calls to create the given expression.

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

Every SymPy expression satisfies a key identity invariant:

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

This means that expressions are rebuildable from their `args`.<sup>4</sup> Note that in SymPy the `==` operator represents exact structural equality, not mathematical equality. This allows testing if any two expressions are equal to one another as expression trees. For example, even though  $(x+1)^2$  and  $x^2+2x+1$  are equal mathematically, SymPy gives

```
162 >>> (x + 1)**2 == x**2 + 2*x + 1
163 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.

## 173 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.

```
181 >>> t = Symbol('t')
182 >>> sqrt(t**2)
183 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.

<sup>3</sup> The `dotprint` function from the `sympy.printing.dot` package prints output to dot format, which can be rendered with `Graphviz` to visualize expression trees graphically.

<sup>4</sup>`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.

```

189 >>> t = Symbol('t', positive=True)
190 >>> sqrt(t**2)
191 t

```

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`.<sup>5</sup> 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 [25].

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, 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 SymPy classes<sup>6</sup> 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.

<sup>5</sup>Special case is the `commutative` assumption: it can't be `None`. Unless a Symbol is created with `commutative=False`, it's assumed to commute with any other object with respect to the multiplication operation.

<sup>6</sup>Some internal classes, such as those used in the polynomial module, do not follow this rule for efficiency reasons.

As an example of the notions presented in this section, Listing 1 presents a minimal version of the gamma function  $\Gamma(x)$  from SymPy, which evaluates itself on positive integer arguments, has the positive and real assumptions defined, can be rewritten in terms of factorial with `gamma(x).rewrite(factorial)`, and can be differentiated. `self.func` is used throughout instead of referencing `gamma` explicitly so that potential subclasses of `gamma` can reuse the methods.

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

```

244 from sympy import Integer, Function, floor, factorial, polygamma
245
246 class gamma(Function)
247     @classmethod
248     def eval(cls, arg):
249         if isinstance(arg, Integer) and arg.is_positive:
250             return factorial(arg - 1)
251
252     def _eval_is_positive(self):
253         x = self.args[0]
254         if x.is_positive:
255             return True
256         elif x.is_noninteger:
257             return floor(x).is_even
258
259     def _eval_is_real(self):
260         x = self.args[0]
261         # noninteger means real and not integer
262         if x.is_positive or x.is_noninteger:
263             return True
264
265     def _eval_rewrite_as_factorial(self, z):
266         return factorial(z - 1)
267
268     def fdiff(self, argindex=1):
269         from sympy.core.function import ArgumentIndexError
270         if argindex == 1:
271             return self.func(self.args[0])*polygamma(0, self.args[0])
272         else:
273             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 1.** SymPy Features and Descriptions

Feature	Description
Calculus	Algorithms for computing derivatives, integrals, and limits.
Category Theory	Representation of objects, morphisms, and diagrams. Tools for drawing diagrams with Xy-pic.

Code Generation	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	Permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [28], and Prufer sequences [4].
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 [33] for two univariate polynomials.
Cryptography	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	Representations of manifolds, metrics, tensor products, and coordinate systems in Riemannian and pseudo-Riemannian geometries [42].
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 $\text{\LaTeX}$ 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 [37].
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”.<sup>7</sup>

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

<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 [14]
<code>hyperexpand</code>	expand hypergeometric functions [35, 36]

### 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  $\lim_{x \rightarrow \infty} x \sin(\frac{1}{x}) = 1$ . Note that SymPy denotes  $\infty$  as `oo`.

<sup>7</sup>The `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.



```

299 >>> limit(x*sin(1/x), x, oo)
300 1

```

As a more complex example, SymPy computes

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

```

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

```

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

```

303 >>> diff(sin(x)*exp(x), x)
304 exp(x)*sin(x) + exp(x)*cos(x)

```

Integrals are calculated with the `integrate` function. SymPy implements a combination of the Risch algorithm [6], table lookups, a reimplementation of Manuel Bronstein’s “Poor Man’s Integrator” [5], and an algorithm for computing integrals based on Meijer G-functions [35, 36]. 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

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

```

307 >>> s, t = symbols('s t', positive=True)
308 >>> integrate(exp(-s*t)*log(t), (t, 0, oo)).simplify()
309 -(log(s) + EulerGamma)/s
310 >>> integrate((-2*x**2*(log(x) + 1)*exp(x**2) +
311 ... (exp(x**2) + 1)**2)/(x*(exp(x**2) + 1)**2*(log(x) + 1)), x)
312 log(log(x) + 1) + 1/(exp(x**2) + 1)

```

Summations are computed with `summation` using a combination of Gosper’s algorithm [17], an algorithm that uses Meijer G-functions [35, 36], and heuristics. Products are computed with `product` function via a suite of heuristics.

```

316 >>> i, n = symbols('i n')
317 >>> summation(2**i, (i, 0, n - 1))
318 2**n - 1
319 >>> summation(i*factorial(i), (i, 1, n))
320 n*factorial(n) + factorial(n) - 1

```

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

```

323 >>> integrate(x**x, x)
324 Integral(x**x, x)
325 >>> Sum(2**i, (i, 0, n - 1))
326 Sum(2**i, (i, 0, n - 1))

```

### 3.3 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.<sup>8</sup> 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) [7, 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 [30], 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 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.

```
>>> 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.

```
>>> 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)
/
|
| _____
| \ / phi0 + 1 d(phi0)
|
/
```

<sup>8</sup>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.

373 The function `latex` returns a  $\text{\LaTeX}$  representation of an expression.

```
374 >>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
375 \int \sqrt{\phi_{0} + 1}\, d\phi_{0}
```

376 Users are encouraged to run the `init_printing` function at the beginning of interactive  
377 sessions, which automatically enables the best pretty printing supported by their environment.  
378 In the Jupyter Notebook or Qt Console [31], the  $\text{\LaTeX}$  printer is used to render expressions  
379 using MathJax or  $\text{\LaTeX}$ , if it is installed on the system. The 2D text representation is used  
380 otherwise.

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

### 385 3.5 Solvers

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

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

```
394 solve(f, *symbols, **flags)
395 solveset(f, symbol, domain=S.Complexes)
```

396 The `domain` parameter is typically either `S.Complexes` (the default) or `S.Reals`; the latter causes  
397 `solveset` to only return real solutions.

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

401 Both functions implicitly assume that expressions are equal to 0. For instance, `solveset(x -`  
402 `1, x)` solves  $x - 1 = 0$  for  $x$ .

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

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

### 407 3.6 Matrices

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

```
411 >>> A = Matrix(2, 2, [x, x + y, y, x])
412 >>> A
413 Matrix([
414 [x, x + y],
415 [y, x]])
```

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

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

```

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

```

Internally these matrices store the elements as lists of lists, making it a dense representation.<sup>9</sup> For storing sparse matrices, the `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:

```

446 >>> Float(1.1)
447 1.1000000000000000
448 >>> Float(1.1, 30) # precision equivalent to 30 digits
449 1.100000000000000008881784197001
450 >>> Float("1.1", 30)
451 1.10000000000000000000000000000000

```

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

```

454 >>> (pi + 1).evalf(25)
455 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 [16]. A notorious case is that of catastrophic cancellation:

```

459 >>> cos(exp(-100)).evalf(25) - 1
460 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:

```

465 >>> (cos(exp(-100)) - 1).evalf(25)
466 -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.

<sup>9</sup>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.

## 472

473  
474  
475  
476  
477

478  
479  
480  
481  
482  
483

484  
485  
486  
487  
488

489  
490  
491  
492

493  
494  
495  
496  
497  
498  
499

500  
501  
502  
503  
504  
505  
506  
507

508

509

510

511

512

513

514

515

516

## 517

518  
519  
520  
521

## 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  $3 \times 3$  tensors [43]. 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  $\pi$ ,  $\frac{\pi}{2}$ , and  $\frac{\pi}{3}$  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  $\frac{\pi}{2}$  rad.

```
>>> 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 [24] and Kane's Method [23]. Lastly, there are automated linearization routines for constrained dynamical systems [32].

## 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 [39]: 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.

```
>>> from sympy.physics.quantum import Commutator, Dagger, Operator
>>> from sympy.physics.quantum import Ket, qapply
```

```

574 >>> A = Operator('A')
575 >>> B = Operator('B')
576 >>> C = Operator('C')
577 >>> D = Operator('D')
578 >>> a = Ket('a')
579 >>> comm = Commutator(A, B)
580 >>> comm
581 [A,B]
582 >>> qapply(Dagger(comm*a)).doit()
583 -<a|*(Dagger(A)*Dagger(B) - Dagger(B)*Dagger(A))
584 Commutators can be expanded using common commutator identities:
585 >>> Commutator(C+B, A*D).expand(commutator=True)
586 -[A,B]*D - [A,C]*D + A*[B,D] + A*[C,D]
587     On top of this set of base objects, a number of specific quantum systems have been implemented
588     in a fully symbolic framework. These include:
589     • Many of the exactly solvable quantum systems, including simple harmonic oscillator states
590       and raising/lowering operators, infinite square well states, and 3D position and momentum
591       operators and states.
592     • Second quantized formalism of non-relativistic many-body quantum mechanics [13].
593     • Quantum angular momentum [47]. Spin operators and their eigenstates can be represented
594       in any basis and for any quantum numbers. A rotation operator representing the Wigner-D
595       matrix, which may be defined symbolically or numerically, is also implemented to rotate
596       spin eigenstates. Functionality for coupling and uncoupling of arbitrary spin eigenstates is
597       provided, including symbolic representations of Clebsch-Gordon coefficients and Wigner
598       symbols.
599     • Quantum information and computing [27]. Multidimensional qubit states, and a full
600       set of one- and two-qubit gates are provided and can be represented symbolically or as
601       matrices/vectors. With these building blocks, it is possible to implement a number of basic
602       quantum algorithms including the quantum Fourier transform, quantum error correction,
603       quantum teleportation, Grover's algorithm, dense coding, etc. In addition, any quantum
604       circuit may be plotted using the circuit_plot function (Figure 1).
605     Here are a few short examples of the quantum information and computing capabilities in
606     sympy.physics.quantum. Start with a simple four-qubit state and flip the second qubit from the
607     right using a Pauli-X gate:
608     >>> from sympy.physics.quantum.qubit import Qubit
609     >>> from sympy.physics.quantum.gate import XGate
610     >>> q = Qubit('0101')
611     >>> q
612     |0101>
613     >>> X = XGate(1)
614     >>> qapply(X*q)
615     |0111>
616     Qubit states can also be used in adjoint operations, tensor products, inner/outer products:
617     >>> Dagger(q)
618     <0101|
619     >>> ip = Dagger(q)*q
620     >>> ip
621     <0101|0101>
622     >>> ip.doit()
623     1

```

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

```

626 >>> from sympy.physics.quantum.qubit import measure_all
627 >>> from sympy.physics.quantum.gate import H, X, Y, Z
628 >>> c = H(0)*H(1)*Qubit('00')
629 >>> c
630 H(0)*H(1)*|00>
631 >>> q = qapply(c)
632 >>> measure_all(q)
633 [(|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).

```

637 >>> from sympy.physics.quantum.qft import QFT
638 >>> from sympy.physics.quantum.circuitplot import circuit_plot
639 >>> fourier = QFT(0,3).decompose()
640 >>> fourier
641 SWAP(0,2)*H(0)*C((0),S(1))*H(1)*C((0),T(2))*C((1),S(2))*H(2)
642 >>> c = circuit_plot(fourier, nqubits=3)

```

## 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 CASSs, 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 a 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

- [1] Adams, W. W. and Loustaunau, 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] Bronstein, M. (2005a). pmint—The Poor Man's Integrator. <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint>.
- [6] Bronstein, M. (2005b). *Symbolic Integration I: Transcendental Functions*. Springer-Verlag, New York, NY, USA.
- [7] Buchberger, B. (1965). *Ein Algorithmus zum Auffinden der Basis Elemente des Restklassenrings nach einem nulldimensionalen Polynomideal*. PhD thesis, University of Innsbruck, Innsbruck, Austria.
- [8] 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] Faugère, J. C. (1999). A New Efficient Algorithm for Computing Gröbner Bases (F4). *Journal of Pure and Applied Algebra*, 139(1-3):61–88.
- [12] 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.
- [13] Fetter, A. and Walecka, J. (2003). *Quantum Theory of Many-Particle Systems*. Dover Publications.
- [14] Fu, H., Zhong, X., and Zeng, Z. (2006). Automated and Readable Simplification of Trigonometric Expressions. *Mathematical and Computer Modelling*, 55(11-12):1169–1177.

- [15] 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.
- [16] Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48.
- [17] Gosper, R. W. (1978). Decision procedure for indefinite hypergeometric summation. *Proceedings of the National Academy of Sciences*, 75(1):40–42.
- [18] Gruntz, D. (1996). *On Computing Limits in a Symbolic Manipulation System*. PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland.
- [19] Horsen, C. V. (2015). GMPY. <https://pypi.python.org/pypi/gmpy2>.
- [20] Hudak, P. (1998). Domain specific languages. In *Handbook of Programming Languages, Vol. III: Little Languages and Tools*, chapter 3, pages 39–60. MacMillan, Indianapolis.
- [21] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95.
- [22] Johansson, F. et al. (2014). mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.19). <http://mpmath.org/>.
- [23] Kane, T. R. and Levinson, D. A. (1985). *Dynamics, Theory and Applications*. McGraw Hill.
- [24] Lagrange, J. (1811). *Mécanique analytique*. Number v. 1 in *Mécanique analytique*. Ve Courcier.
- [25] Lang, S. (1966). Introduction to transcendental numbers. *Reading, Mass.*
- [26] Lutz, M. (2013). *Learning Python*. O’Reilly Media, Inc.
- [27] Nielsen, M. and Chuang, I. (2011). *Quantum Computation and Quantum Information*. Cambridge University Press.
- [28] Nijenhuis, A. and Wilf, H. S. (1978). *Combinatorial Algorithms: For Computers and Calculators*. Academic Press, New York, NY, USA, second edition.
- [29] Oliphant, T. E. (2007). Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20.
- [30] 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.
- [31] Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29.
- [32] 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.
- [33] Petkovšek, M., Wilf, H. S., and Zeilberger, D. (1996). A=BAK peters. *Wellesley, MA*.
- [34] Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49.
- [35] 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.
- [36] 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.
- [37] Rocklin, M. and Terrel, A. R. (2012). Symbolic statistics with SymPy. *Computing in Science and Engineering*, 14.
- [38] Rosen, L. (2005). *Open source licensing*, volume 692. Prentice Hall.
- [39] Sakurai, J. and Napolitano, J. (2010). *Modern Quantum Mechanics*. Addison-Wesley.
- [40] Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. Prentice Hall Ordering Information.
- [41] Stein, W. and Joyner, D. (2005). SAGE: System for Algebra and Geometry Experimentation. *Communications in Computer Algebra*, 39(2).
- [42] Sussman, G. J. and Wisdom, J. (2013). *Functional Differential Geometry*. Massachusetts Institute of Technology Press.

- 766 [43] Tai, C.-T. (1997). *Generalized vector and dyadic analysis: applied mathematics in field*  
767 *theory*, volume 9. Wiley-IEEE Press.
- 768 [44] Takahasi, H. and Mori, M. (1974). Double exponential formulas for numerical integration.  
769 *Publications of the Research Institute for Mathematical Sciences*, 9(3):721–741.
- 770 [45] Toth, V. T. (2007). Maple and Meijer’s G-function: a numerical instability and a cure.  
771 <http://www.vttoth.com/CMS/index.php/technical-notes/67>.
- 772 [46] Turk, M. J., Smith, B. D., Oishi, J. S., Skory, S., Skillman, S. W., Abel, T., and Norman,  
773 M. L. (2011). yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The*  
774 *Astrophysical Journal Supplement Series*, 192:9–+.
- 775 [47] Zare, R. (1991). *Angular Momentum: Understanding Spatial Aspects in Chemistry and*  
776 *Physics*. Wiley.