

SYMPY: SYMBOLIC COMPUTING IN PYTHON

AARON MEURER*, CHRISTOPHER P. SMITH†, MATEUSZ PAPROCKI‡, ONDŘEJ
ČERTÍK§, SERGEY B. KIRPICHEV¶, MATTHEW ROCKLIN||, AMIT KUMAR#, SERGIU
IVANOV††, JASON K. MOORE‡‡, SARTAJ SINGH§§, THILINA RATHNAYAKE¶¶, SEAN VIG|||
, BRIAN E. GRANGER##, RICHARD P. MULLER¹, FRANCESCO BONAZZI², HARSH
GUPTA³, SHIVAM VATS⁴, FREDRIK JOHANSSON⁵, FABIAN PEDREGOSA⁶, MATTHEW
J. CURRY⁷, ANDY R. TERREL⁸, ASHUTOSH SABOO⁹, ISURU FERNANDO¹⁰, SUMITH
KULAL¹¹, ROBERT CIMRMAN¹², AND ANTHONY SCOPATZ¹³

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.

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

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

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

§Los Alamos National Laboratory, Los Alamos, NM 87545 (certik@lanl.gov).

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

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

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

††Université Paris Est Créteil, 61 av. Général de Gaulle, 94010 Créteil, France (sergiu.ivanov@upec.fr).

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

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

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

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

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

¹Center for Computing Research, Sandia National Laboratories, Albuquerque, NM 87185 (rmuller@sandia.gov).

²Max Planck Institute of Colloids and Interfaces, Department of Theory and Bio-Systems, Am Mühlenberg 1, 14424 Potsdam, Germany (francesco.bonazzi@mpikg.mpg.de).

³Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India (har-gup@protonmail.com).

⁴Indian Institute of Technology Kharagpur, Kharagpur, West Bengal 721302, India (shivamvats.iitkgp@gmail.com).

⁵INRIA Bordeaux-Sud-Ouest – LFANT project-team, 200 Avenue de la Vieille Tour, 33405 Talence, France (fredrik.johansson@gmail.com).

⁶INRIA – SIERRA project-team, 2 Rue Simone IFF, 75012 Paris, France (f@bianp.net).

⁷Department of Physics and Astronomy, University of New Mexico, Albuquerque, NM 87131 (mattjcurry@gmail.com).

⁸Fashion Metric, Inc, Austin, TX 78681 (andy.terrel@gmail.com).

⁹Birla Institute of Technology and Science, Pilani, K.K. Birla Goa Campus, NH 17B Bypass Road, Zuarinagar, Sancoale, Goa 403726, India (ashutosh.saboo96@gmail.com).

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

¹¹Indian Institute of Technology Bombay, Powai, Mumbai, Maharashtra 400076, India (sumith@cse.iitb.ac.in).

¹²New Technologies – Research Centre, University of West Bohemia, Univerzitní 8, 306 14 Plzeň, Czech Republic (cimrman3@ntc.zcu.cz).

¹³University of South Carolina, Columbia, SC 29201 (scopatz@cec.sc.edu).

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 L^AT_EX.

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

2. Architecture. Software architecture is of central importance in any large software project because it establishes predictable patterns of usage and development [39]. This section describes the essential structural components of SymPy, provides justifications for the design decisions that have been made, and gives example user-facing code as appropriate.

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.

```
>>> 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,

```
>>> 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 $(x^2 - 2x + 3)/y$.

```
>>> (x**2 - 2*x + 3)/y
(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 $xy + 2$:

```
>>> 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`.

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

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

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

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.

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

Every SymPy expression satisfies a key identity invariant:

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

This means that expressions are rebuildable from their `args`.¹ 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

```
>>> (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.

```
>>> 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.

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

Some of the common assumptions that SymPy allows are `positive`, `negative`, `real`, `nonpositive`, `integer`, `prime` and `commutative`.² 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

¹`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.

²If A and B are Symbols created with `commutative=False` then SymPy will keep $A \cdot B$ and $B \cdot A$ distinct.

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 trees³ 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 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

³Some internal classes, such as those used in the polynomial module, do not follow this rule for efficiency reasons.

subclasses of `gamma` can reuse the methods.

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

```

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

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

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

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

    def _eval_rewrite_as_factorial(self, z):
        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 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 [27], 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 [32] 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 [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 π , 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	Me-	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	Func-	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”.⁴

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

⁴The `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.

250 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 [34, 35]

251 **3.2. Calculus.** Integrals are calculated with the `integrate` function. SymPy im-
252 plements a combination of the Risch algorithm [6], table lookups, a reimplementa-
253 tion of Manuel Bronstein’s “Poor Man’s Integrator” [5], and an algorithm for computing
254 integrals based on Meijer G-functions [34, 35]. These allow SymPy to compute a wide
255 variety of indefinite and definite integrals. The Meijer G-function algorithm and the
256 Risch algorithm are respectively demonstrated below by the computation of

$$257 \quad \int_0^{\infty} e^{-st} \log(t) dt = -\frac{\log(s) + \gamma}{s}$$

258 and

$$259 \quad \int \frac{-2x^2 (\log(x) + 1) e^{x^2} + (e^{x^2} + 1)^2}{x(e^{x^2} + 1)^2 (\log(x) + 1)} dx = \log(\log(x) + 1) + \frac{1}{e^{x^2} + 1}.$$

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

266 Derivatives are computed with the `diff` function, which recursively uses the var-
267 ious differentiation rules.

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

270 Summations are computed with `summation` using a combination of Gosper’s algo-
271 rithm [17], an algorithm that uses Meijer G-functions [34, 35], and heuristics. Products
272 are computed with `product` function via a suite of heuristics.

```
273 >>> i, n = symbols('i n')
274 >>> summation(2**i, (i, 0, n - 1))
275 2**n - 1
276 >>> summation(i*factorial(i), (i, 1, n))
277 n*factorial(n) + factorial(n) - 1
```

278 Limits are computed with the `limit` function, using the Gruntz algorithm [18] for
279 computing symbolic limits and heuristics (a description of the Gruntz algorithm may
280 be found in the supplement). For example, the following computes $\lim_{x \rightarrow \infty} x \sin(\frac{1}{x}) = 1$.

281 Note that SymPy denotes ∞ as `oo`.

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

```
283 1
```

```
284 As a more complex example, SymPy computes
```

```
285
```

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

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

```
287 E
```

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

```
290 >>> integrate(x**x, x)
```

```
291 Integral(x**x, x)
```

```
292 >>> Sum(2**i, (i, 0, n - 1))
```

```
293 Sum(2**i, (i, 0, n - 1))
```

```
294
```

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.

```
299
```

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.

```
306
```

SymPy implements dense and sparse polynomial representations.⁵ 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 [29], 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.

```
319
```

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

```
320
```

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.

```
324 >>> phi0 = Symbol('phi0')
```

```
325 >>> str(Integral(sqrt(phi0), phi0))
```

⁵In 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.

```

326 'Integral(sqrt(phi0), phi0)'
327     Expressions can be printed in 2D with monospace fonts via pprint. Unicode
328 characters are used for rendering mathematical symbols such as integral signs, square
329 roots, and parentheses. Greek letters and subscripts in symbol names that have
330 Unicode code points associated are also rendered automatically.
    >>> pprint(Integral(sqrt(phi0 + 1), phi0))
    
$$\int \sqrt{\varphi_0 + 1} \, d(\varphi_0)$$

331
332 Alternately, the use_unicode=False flag can be set, which causes the expression to be
333 printed using only ASCII characters.
    >>> pprint(Integral(sqrt(phi0 + 1), phi0), use_unicode=False)
    /
    |
    | _____
    | \ / phi0 + 1  d(phi0)
    |
    /
340
341     The function latex returns a LATEX representation of an expression.
    >>> print(latex(Integral(sqrt(phi0 + 1), phi0)))
    \int \sqrt{\phi_0 + 1}\, d\phi_0
344
345     Users are encouraged to run the init_printing function at the beginning of in-
346 teractive sessions, which automatically enables the best pretty printing supported by
347 their environment. In the Jupyter Notebook or Qt Console [30], the LATEX printer is
348 used to render expressions using MathJax or LATEX, if it is installed on the system.
    The 2D text representation is used otherwise.
349
350     Other printers such as MathML are also available. SymPy uses an extensible
351 printer subsystem for customizing any given printer, and allows custom objects to
352 define their printing behavior for any printer. The code generation functionality of
353 SymPy relies on this subsystem to convert expressions into code in various target
    programming languages.
354
355 3.5. Solvers. SymPy has a module of equation solvers that can handle ordinary
356 differential equations, recurrence relationships, Diophantine equations, and algebraic
357 equations. There is also rudimentary support for simple partial differential equations.
358
359     There are two functions for solving algebraic equations in SymPy: solve and
360 solveset. solveset has several design changes with respect to the older solve func-
361 tion. This distinction is present in order to resolve the usability issues with the
362 previous solve function API while maintaining backward compatibility with earlier
363 versions of SymPy. solveset only requires essential input information from the user.
364
365     The function signatures of solve and solveset are
    solve(f, *symbols, **flags)
    solveset(f, symbol, domain=S.Complexes)
366
367     The domain parameter is typically either S.Complexes (the default) or S.Reals; the
368 latter causes solveset to only return real solutions.
369
370     An important difference between the two functions is that the output API of
371 solve varies with input (sometimes returning a Python list and sometimes a Python
    dictionary) whereas solveset always returns a SymPy set object.
372
373     Both functions implicitly assume that expressions are equal to 0. For instance,
    solveset(x - 1, x) solves  $x - 1 = 0$  for  $x$ .

```

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

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

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 `Matrix` class.

```
>>> A = Matrix(2, 2, [x, x + y, y, x])
>>> A
Matrix([
  [x, x + y],
  [y, x]])
```

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.

```
>>> A.eigenvals()
{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.⁶ 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:

```
>>> Float(1.1)
1.1000000000000000
```

⁶Similar 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.

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 Shanks transformation (Euler-Maclaurin summation can also be used) [3]. A function to evaluate oscillatory integrals by means of convergence acceleration is also available.

A wide array of higher mathematical functions are implemented with full support for complex values of all parameters and arguments, including complete and incomplete gamma functions, Bessel functions, orthogonal polynomials, elliptic functions and integrals, zeta and polylogarithm functions, the generalized hypergeometric function, and the Meijer G-function. The Meijer G-function instance $G_{1,3}^{3,0}(0; \frac{1}{2}, -1, -\frac{3}{2}|x)$ is a good test case [44]; past versions of both Maple and Mathematica produced incorrect numerical values for large $x > 0$. Here, mpmath automatically removes an internal singularity and compensates for cancellations (amounting to 656 bits of precision when $x = 10000$), giving correct values:

```
>>> 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:

```
>>> 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.

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 [42]. The vector and dyadic objects can be used for any one-, two-, or three-dimensional vector algebra, and they provide a strong framework for building physics and engineering tools.

The following Python code demonstrates how a vector is created using the orthogonal unit vectors of three reference frames that are oriented with respect to each

other, and the result of expressing the vector in the A frame. The B frame is oriented with respect to the A frame using Z-X-Z Euler Angles of magnitude π , $\frac{\pi}{2}$, and $\frac{\pi}{3}$ 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.

```

519 >>> from sympy.physics.vector import ReferenceFrame
520 >>> A = ReferenceFrame('A')
521 >>> B = ReferenceFrame('B')
522 >>> C = ReferenceFrame('C')
523 >>> B.orient(A, 'body', (pi, pi/3, pi/4), 'zxz')
524 >>> C.orient(B, 'axis', (pi/2, B.x))
525 >>> v = 1*A.x + 2*B.z + 3*C.y
526 >>> v
527 A.x + 2*B.z + 3*C.y
528 >>> v.express(A)
529 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 [23] and Kane's Method [22]. Lastly, there are automated linearization routines for constrained dynamical systems [31].

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 [38]: 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.

```

551 >>> from sympy.physics.quantum import Commutator, Dagger, Operator
552 >>> from sympy.physics.quantum import Ket, qapply
553 >>> A = Operator('A')
554 >>> B = Operator('B')
555 >>> C = Operator('C')
556 >>> D = Operator('D')
557 >>> a = Ket('a')
558 >>> comm = Commutator(A, B)
559 >>> comm
560 [A,B]
561 >>> qapply(Dagger(comm*a)).doit()
562 -<a|*(Dagger(A)*Dagger(B) - Dagger(B)*Dagger(A))
563 Commutators can be expanded using common commutator identities:

```



```

564 >>> Commutator(C+B, A*D).expand(commutator=True)
565 -[A,B]*D - [A,C]*D + A*[B,D] + A*[C,D]
566 On top of this set of base objects, a number of specific quantum systems have
567 been implemented in a fully symbolic framework. These include:
568 • Many of the exactly solvable quantum systems, including simple harmonic
569 oscillator states and raising/lowering operators, infinite square well states,
570 and 3D position and momentum operators and states.
571 • Second quantized formalism of non-relativistic many-body quantum mechan-
572 ics [13].
573 • Quantum angular momentum [46]. Spin operators and their eigenstates can
574 be represented in any basis and for any quantum numbers. A rotation opera-
575 tor representing the Wigner-D matrix, which may be defined symbolically or
576 numerically, is also implemented to rotate spin eigenstates. Functionality for
577 coupling and uncoupling of arbitrary spin eigenstates is provided, including
578 symbolic representations of Clebsch-Gordon coefficients and Wigner symbols.
579 • Quantum information and computing [26]. Multidimensional qubit states,
580 and a full set of one- and two-qubit gates are provided and can be represented
581 symbolically or as matrices/vectors. With these building blocks, it is possible
582 to implement a number of basic quantum algorithms including the quantum
583 Fourier transform, quantum error correction, quantum teleportation, Grover's
584 algorithm, dense coding, etc. In addition, any quantum circuit may be plotted
585 using the circuit_plot function (Figure 1).
586 Here are a few short examples of the quantum information and computing capa-
587 bilities in sympy.physics.quantum. Start with a simple four-qubit state and flip the
588 second qubit from the right using a Pauli-X gate:
589 >>> from sympy.physics.quantum.qubit import Qubit
590 >>> from sympy.physics.quantum.gate import XGate
591 >>> q = Qubit('0101')
592 >>> q
593 |0101>
594 >>> X = XGate(1)
595 >>> qapply(X*q)
596 |0111>
597 Qubit states can also be used in adjoint operations, tensor products, inner/outer
598 products:
599 >>> Dagger(q)
600 <0101|
601 >>> ip = Dagger(q)*q
602 >>> ip
603 <0101|0101>
604 >>> ip.doit()
605 1
606 Quantum gates (unitary operators) can be applied to transform these states and then
607 classical measurements can be performed on the results:
608 >>> from sympy.physics.quantum.qubit import measure_all
609 >>> from sympy.physics.quantum.gate import H, X, Y, Z
610 >>> c = H(0)*H(1)*Qubit('00')
611 >>> c
612 H(0)*H(1)*|00>
613 >>> q = qapply(c)

```

```

614 >>> measure_all(q)
615 [(|00>, 1/4), (|01>, 1/4), (|10>, 1/4), (|11>, 1/4)]

```

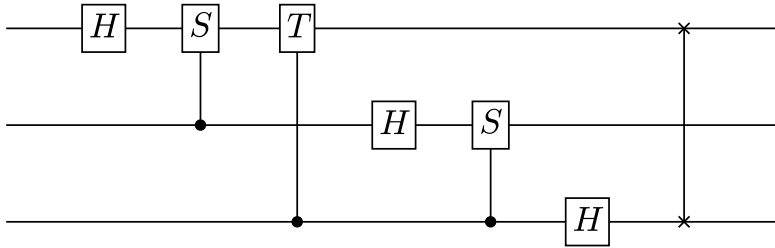


Fig. 1: The circuit diagram for a three-qubit quantum Fourier transform generated by SymPy.

```

616 Lastly, the following example demonstrates creating a three-qubit quantum Fourier
617 transform, decomposing it into one- and two-qubit gates, and then generating a circuit
618 plot for the sequence of gates (see Figure 1).
619 >>> from sympy.physics.quantum.qft import QFT
620 >>> from sympy.physics.quantum.circuitplot import circuit_plot
621 >>> fourier = QFT(0,3).decompose()
622 >>> fourier
623 SWAP(0,2)*H(0)*C((0),S(1))*H(1)*C((0),T(2))*C((1),S(2))*H(2)
624 >>> 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 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 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."

8. References.

REFERENCES

- [1] W. W. ADAMS AND P. LOUSTAUNAU, *An introduction to Gröbner bases*, no. 3, American Mathematical Society, 1994.
- [2] D. H. BAILEY, K. JEYABALAN, AND X. S. LI, *A comparison of three high-precision quadrature schemes*, *Experimental Mathematics*, 14 (2005), pp. 317–329.
- [3] C. M. BENDER AND S. A. ORSZAG, *Advanced Mathematical Methods for Scientists and Engineers*, Springer, 1st ed., October 1999.
- [4] N. BIGGS, E. K. LLOYD, AND R. J. WILSON, *Graph Theory, 1736-1936*, Oxford University Press, 1976.
- [5] M. BRONSTEIN, *Poor Man's Integrator*, <http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint>.
- [6] M. BRONSTEIN, *Symbolic Integration I: Transcendental Functions*, Springer-Verlag, New York, NY, USA, 2005.
- [7] B. BUCHBERGER, *Ein Algorithmus zum Auffinden der Basis Elemente des Restklassenrings nach einem nulldimensionalen Polynomideal*, PhD thesis, University of Innsbruck, Innsbruck, Austria, 1965.
- [8] J. CARETTE, *Understanding Expression Simplification*, in ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 2004, ACM Press, pp. 72–79, <http://dx.doi.org/10.1145/1005285.1005298>.
- [9] D. CERVONE, *Mathjax: a platform for mathematics on the web*, *Notices of the AMS*, 59 (2012), pp. 312–316.
- [10] R. CORMAN, *SfePy - write your own FE application*, in Proceedings of the 6th European Conference on Python in Science (EuroSciPy 2013), P. de Buyl and N. Varoquaux, eds., 2014, pp. 65–70. <http://arxiv.org/abs/1404.6391>.
- [11] J. C. FAUGÈRE, *A New Efficient Algorithm for Computing Gröbner Bases (F_4)*, *Journal of Pure and Applied Algebra*, 139 (1999), pp. 61–88, <http://www-calfor.lip6.fr/~jcf/Papers/F99a.pdf>.
- [12] J. C. FAUGÈRE, *A New Efficient Algorithm for Computing Gröbner Bases Without Reduction To Zero (F_5)*, in ISSAC '02: Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 2002, ACM Press, pp. 75–83, <http://dx.doi.org/10.1145/780506.780516>, <http://www-calfor.lip6.fr/~jcf/Papers/F02a.pdf>.
- [13] A. FETTER AND J. WALECKA, *Quantum Theory of Many-Particle Systems*, Dover Publications, 2003.
- [14] H. FU, X. ZHONG, AND Z. ZENG, *Automated and Readable Simplification of Trigonometric Expressions*, *Mathematical and Computer Modelling*, 55 (2006), pp. 1169–1177.
- [15] G. GEDE, D. L. PETERSON, A. S. NANJANGUD, J. K. MOORE, AND M. HUBBARD, *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, American Society of Mechanical Engineers, 2013,

- pp. V07BT10A051–V07BT10A051.
- [16] D. GOLDBERG, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), 23 (1991), pp. 5–48.
 - [17] R. W. GOSPER, *Decision procedure for indefinite hypergeometric summation*, Proceedings of the National Academy of Sciences, 75 (1978), pp. 40–42.
 - [18] D. GRUNTZ, *On Computing Limits in a Symbolic Manipulation System*, PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.
 - [19] C. V. HORSEN, *GMPY*. <https://pypi.python.org/pypi/gmpy2>, 2015.
 - [20] P. HUDAK, *Domain specific languages*, in Handbook of Programming Languages, Vol. III: Little Languages and Tools, MacMillan, Indianapolis, 1998, ch. 3, pp. 39–60.
 - [21] J. D. HUNTER, *Matplotlib: A 2d graphics environment*, Computing In Science & Engineering, 9 (2007), pp. 90–95.
 - [22] T. R. KANE AND D. A. LEVINSON, *Dynamics, Theory and Applications*, McGraw Hill, 1985.
 - [23] J. LAGRANGE, *Mécanique analytique*, no. v. 1 in Mécanique analytique, Ve Courcier, 1811.
 - [24] S. LANG, *Introduction to transcendental numbers*, Reading, Mass, (1966).
 - [25] M. LUTZ, *Learning Python*, O'Reilly Media, Inc., 2013.
 - [26] M. NIELSEN AND I. CHUANG, *Quantum Computation and Quantum Information*, Cambridge University Press, 2011.
 - [27] A. NIJENHUIS AND H. S. WILF, *Combinatorial Algorithms: For Computers and Calculators*, Academic Press, New York, NY, USA, second ed., 1978.
 - [28] T. E. OLIPHANT, *Python for scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 10–20.
 - [29] M. PAPROCKI, *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, 2010.
 - [30] F. PÉREZ AND B. E. GRANGER, *IPython: a system for interactive scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 21–29.
 - [31] D. L. PETERSON, G. GEDE, AND M. HUBBARD, *Symbolic linearization of equations of motion of constrained multibody systems*, Multibody System Dynamics, 33 (2014), pp. 143–161, <http://dx.doi.org/10.1007/s11044-014-9436-5>.
 - [32] M. PETKOVŠEK, H. S. WILF, AND D. ZEILBERGER, *A = BAK peters*, Wellesley, MA, (1996).
 - [33] E. RAYMOND, *The cathedral and the bazaar*, Knowledge, Technology & Policy, 12 (1999), pp. 23–49.
 - [34] K. ROACH, *Hypergeometric function representations*, in ISSAC '96: Proceedings of the 1996 International Symposium on Symbolic and Algebraic Computation, New York, NY, USA, 1996, ACM Press, pp. 301–308, <http://dx.doi.org/10.1145/236869.237088>, <http://www.planetquantum.com/TheMission/Papers/Issac96.pdf>.
 - [35] K. ROACH, *Meijer G function representations*, in ISSAC '97: Proceedings of the 1997 international symposium on Symbolic and algebraic computation, New York, NY, USA, 1997, ACM, pp. 205–211, <http://dx.doi.org/10.1145/258726.258784>.
 - [36] M. ROCKLIN AND A. R. TERREL, *Symbolic statistics with SymPy*, Computing in Science and Engineering, 14 (2012), <http://dx.doi.org/10.1109/MCSE.2012.56>.
 - [37] L. ROSEN, *Open source licensing*, vol. 692, Prentice Hall, 2005.
 - [38] J. SAKURAI AND J. NAPOLITANO, *Modern Quantum Mechanics*, Addison-Wesley, 2010.
 - [39] M. SHAW AND D. GARLAN, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996. Prentice Hall Ordering Information.
 - [40] W. STEIN AND D. JOYNER, *SAGE: System for Algebra and Geometry Experimentation*, Communications in Computer Algebra, 39 (2005).
 - [41] G. J. SUSSMAN AND J. WISDOM, *Functional Differential Geometry*, Massachusetts Institute of Technology Press, 2013.
 - [42] C.-T. TAI, *Generalized vector and dyadic analysis: applied mathematics in field theory*, vol. 9, Wiley-IEEE Press, 1997.
 - [43] H. TAKAHASI AND M. MORI, *Double exponential formulas for numerical integration*, Publications of the Research Institute for Mathematical Sciences, 9 (1974), pp. 721–741.
 - [44] V. T. TOTH, *Maple and Meijer's G-function: a numerical instability and a cure*. <http://www.vttoth.com/CMS/index.php/technical-notes/67>, 2007.
 - [45] M. J. TURK, B. D. SMITH, J. S. OISHI, S. SKORY, S. W. SKILLMAN, T. ABEL, AND M. L. NORMAN, *yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data*, The Astrophysical Journal Supplement Series, 192 (2011), pp. 9–+, <http://dx.doi.org/10.1088/0067-0049/192/1/9>, [arXiv:1011.3514](https://arxiv.org/abs/1011.3514).
 - [46] R. ZARE, *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics*, Wiley, 1991.