SYMPY: SYMBOLIC COMPUTING IN PYTHON

AARON MEURER[*], MATEUSZ PAPROCKI[†], ONDŘEJ ČERTÍK[‡], MATTHEW ROCKLIN[§], AMIT KUMAR[¶], SERGIU IVANOV[‖], JASON K. MOORE[#], SARTAJ SINGH[††], THILINA RATHNAYAKE[‡‡], SEAN VIG[§§], BRIAN E. GRANGER[¶¶], RICHARD P. MULLER[‖‖], FRANCESCO BONAZZI[##], HARSH GUPTA[1], SHIVAM VATS[2], FREDRIK JOHANSSON[3], FABIAN PEDREGOSA[4], MATTHEW J. CURRY[5], ASHUTOSH SABOO[6], ISURU FERNANDO[7], SUMITH[8], ROBERT CIMRMAN[9], AND ANTHONY SCOPATZ[10]

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 the standard 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.

1. Introduction.

SymPy is a full featured computer algebra system (CAS) written in the Python programming language [30]. It is free and open source software, being licensed under the 3-clause BSD license [46]. The SymPy project was started by Ondřej Čertík in 2005, and it has since grown to over 500 contributors. Currently,

---

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

[†] Continuum Analytics, Inc., Austin, TX 78701 (mattpap@gmail.com).

[‡] Los Alamos National Laboratory, Los Alamos, NM 87545 (certik@lanl.gov).

[§] Continuum Analytics, Inc., Austin, TX 78701 (mrocklin@gmail.com).

[¶] Delhi Technological University, Shahbad Daulatpur, Bawana Road, New Delhi 110042, India (dtu.amit@gmail.com).

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

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

[##] Technical University of Berlin, Physics Department, Hardenbergstr. 36, 10623 Berlin, Germany (francesco.bonazzi@gmail.com).

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

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

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

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

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

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

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

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

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

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

SymPy is developed on GitHub using a bazaar community model [42]. 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 [36]. SymPy is itself used by many libraries and tools to support research within a variety of domains, such as Sage [51] (pure mathematics), yt [55] (astronomy and astrophysics), PyDy [20] (multibody dynamics), and SfePy [12] (finite elements).

Unlike many CASs, SymPy does not invent its own programming language. Python itself is used both for the internal implementation and the end user interaction. 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 is in an interactive environment or for programmatic use as part of 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, including registering printers with Jupyter [39] frontends, including the Notebook and Qt Console, which will render SymPy expressions using MathJax [11] or LaTeX.

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. Following that, 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. Finally, Section 6 concludes the paper and discusses future work.

2. Architecture.

Software architecture is of central importance in any large software project because it establishes predictable patterns of usage and development [48]. 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 mathematical syntax. Note that in Python, exponentiation is represented by the `**` binary infix operator. 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

[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.

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 ($t \geq 0$). If t is real, the identity $\sqrt{t^2}=|t|$ holds. 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`, `nonnegative`, `real`, `integer`, 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`. `None` represents the "unknown" case. This could mean that the given assumption could be either true or false under the given information, for instance, `Symbol('x', real=True).is_positive` will give `None` because a real symbol might be positive or it might not. It could also mean not enough is implemented to compute

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

the given fact. For instance, `(pi + E).is_irrational` gives `None`, because SymPy does not know how to determine if $\pi + e$ is rational or irrational. Indeed, this case is an open problem in mathematics [29].

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`.

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

The typical way to create a custom SymPy object is to subclass an existing SymPy class, usually `Basic`, `Expr`, or `Function`. All SymPy classes used for expression trees[3] 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 values for which the function should be automatically evaluated, and `None` for arguments that 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`. The most common `_eval_*` methods relate to the assumptions. `_eval_is_`*`assumption`* defines the assumptions for *assumption*.

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. `fdiff` is a convenience method for subclasses of `Function`. `fdiff` returns the derivative of the function without considering the chain rule. `self.func` is used throughout instead of referencing `gamma` explicitly so that potential subclasses of `gamma` can reuse the methods.

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

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

```

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

```python
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. SymPy has an extensive feature set that is too encompassing for complete, in-depth coverage in this paper. However, calculus and other bedrock areas receive their own subsections here. 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 [35], and Prufer sequences [5].
Concrete Math	Summation, products, tools for determining whether summation and product expressions are convergent, absolutely convergent, hypergeometric, and other properties. May also compute Gosper's normal form [41] for two univariate polynomials.
Cryptography	Block and stream ciphers, including shift, Affine, substitution, Vigenere'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 [52].
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 two lines.
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, and integer factorization.
Plotting	Hooks for visualizing expressions via matplotlib [26] 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 [2] and multivariate factorization over algebraic number domains).
Printing	Functions for printing SymPy expressions in the terminal with ASCII or Unicode characters and converting SymPy expressions to LaTeX and MathML.
Quantum Mechanics	Quantum states, bra–ket notation, operators, basis sets, representations, tensor products, inner products, outer products, commutators, anticommutators, and specific quantum system implementations.
Series	Series expansion, sequences, and limits of sequences. This includes Taylor, Laurent, and Puiseux series as well as special series, such as Fourier and formal power series.

Sets	Representations of empty, finite, and infinite sets. This includes special sets such as for all natural, integer, and complex numbers. Operations on sets such as union, intersection, Cartesian product, and building sets from other sets are supported.
Simplification	Functions for manipulating and simplifying expressions. Includes algorithms for simplifying hypergeometric functions, trigonometric expressions, rational functions, combinatorial functions, square root denesting, and common subexpression elimination.
Solvers	Functions for symbolically solving equations, systems of equations, both linear and non-linear, inequalities, ordinary differential equations, partial differential equations, Diophantine equations, and recurrence relations.
Special Functions	Implementations of a number of well known special functions, including Dirac delta, Gamma, Beta, Gauss error functions, Fresnel integrals, Exponential integrals, Logarithmic integrals, Trigonometric integrals, Bessel, Hankel, Airy, B-spline, Riemann Zeta, Dirichlet eta, polylogarithm, Lerch transcendent, hypergeometric, elliptic integrals, Mathieu, Jacobi polynomials, Gegenbauer polynomial, Chebyshev polynomial, Legendre polynomial, Hermite polynomial, Laguerre polynomial, and spherical harmonic functions.
Statistics	Support for a random variable type as well as the ability to declare this variable from prebuilt distribution functions such as Normal, Exponential, Coin, Die, and other custom distributions [45].
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 an unambiguously defined mathematical operation [10]. The `simplify` function applies several simplification routines along with heuristics to make the output expression as "simple" as possible.

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

`expand`	expand the expression
`factor`	factor a polynomial into irreducibles
`collect`	collect polynomial coefficients
`cancel`	rewrite a rational function as p/q with common factors canceled
`apart`	compute the partial fraction decomposition of a rational function

`trigsimp`	simplify trigonometric expressions [18]

Substitutions are performed using the `.subs` method.

```
>>> (sin(x) + x**2 + 1).subs(x, y + 1)
(y + 1)**2 + sin(y + 1) + 1
```

3.2. Calculus. Integrals are calculated with the `integrate` function. SymPy implements a combination of the Risch algorithm [8], table lookups, a reimplementation of Manuel Bronstein's "Poor Man's Integrator" [7], and an algorithm for computing integrals based on Meijer G-functions [43, 44]. 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

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

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

Derivatives are computed recursively by the `diff` function which relies on various differentiation rules.

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

Summations and products are computed with `summation` and `product`, respectively. Summations are computed using a combination of Gosper's algorithm [22], an algorithm that uses Meijer G-functions [43, 44], and heuristics. Products are computed via a suite of heuristics.

Limits are computed with the `limit` function. The limit module implements the Gruntz algorithm [23] for computing symbolic limits (a description of the Gruntz algorithm may be found in the supplement). For example, the following computes $\lim_{x\to\infty} x\sin(\frac{1}{x}) = 1$. Note that SymPy denotes ∞ as `oo`.

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

As a more complex example, SymPy computes

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

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

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

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

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 on its own. 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.[4] 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) [9, 14, 15]. This is because different monomial orderings can be expressed easily in this representation. However, algorithms for computing multivariate GCDs or factorizations, at least those currently implemented in SymPy [37], are better expressed when the representation is dense. The dense multivariate representation is specifically a recursively dense representation, where polynomials in $K[x_0, x_1, \ldots, x_n]$ are viewed as a polynomials in $K[x_0][x_1]\ldots[x_n]$. Note that despite this, the coefficient domain K, can be a multivariate polynomial domain as well. The dense recursive representation in Python becomes inefficient when the number of variables gets high.

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

3.4. Printers. SymPy has a rich collection of expression printers for displaying expressions to the user. By default, an interactive Python session will render the `str` form of an expression; this effect 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 with 2D, 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))
⌠
⎮   ________
⎮ ╲╱ φ₀ + 1  d(φ₀)
⌡
```

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)
  /
```

[4] 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.

```
 |
 |    __________
 | \/ phi0 + 1  d(phi0)
 |
/
```

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

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

Users are encouraged to run the `init_printing` function at the beginning of interactive sessions, which automatically enables the best pretty printing supported by their environment. In the Jupyter Notebook or Qt Console [39], the LaTeX printer is used to render expressions using MathJax or LaTeX, if it is installed on the system. The 2D text representation is used otherwise.

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

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

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

The `domain` parameter is typically either `S.Complexes` (the default) or `S.Reals`; the latter causes `solveset` to only return real solutions. Both functions implicitly assume that expressions are equal to 0. For instance, `solveset(x - 1, x)` solves $x - 1 = 0$ for x.

It should be noted that `solve` has an inconsistent output API for different types of inputs. For instance, depending on the input, it may return a Python list or a Python dictionary. On the other hand, `solveset` has a canonical output API: it always returns a SymPy set object.

`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`. Notably, these include nonlinear multivariate system and 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.[5] 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)
1.10000000000000
>>> Float(1.1, 30)   # precision equivalent to 30 digits
1.10000000000000008881784197001
>>> Float("1.1", 30)
1.10000000000000000000000000000
```

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

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

`Float` numbers do not track their accuracy, and should be used with caution within symbolic expressions since familiar dangers of floating-point arithmetic apply [21]. A notorious case is that of catastrophic cancellation:

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

Applying the `evalf` method to the whole expression solves this problem. Internally, `evalf` estimates the number of accurate bits of the floating-point approximation for

[5] 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.

each sub-expression, and adaptively increases the working precision until the estimated accuracy of the final result matches the sought number of decimal digits:

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

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

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

```
>>> import mpmath
>>> mpmath.mp.dps = 30    # 30 digits of precision
>>> mpmath.mpf("0.1") + mpmath.exp(-50)
mpf('0.100000000000000000000192874984794')
>>> print(_)   # pretty-printed
0.100000000000000000000192874985
```

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

Like SymPy, mpmath is a pure Python library. Internally, mpmath represents a floating-point number $(-1)^s x \cdot 2^y$ by a tuple (s, x, y, b) where x and y are arbitrary-size Python integers and the redundant integer b stores the bit length of x for quick access. If GMPY [25] is installed, mpmath automatically uses the `gmpy.mpz` type for x, and GMPY methods for rounding-related operations, improving performance.

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

The double exponential (tanh-sinh) quadrature is used for numerical integration by default. For smooth integrands, this algorithm usually converges extremely rapidly, even when the integration interval is infinite or singularities are present at the endpoints [53, 3]. 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) [4]. 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 $G^{3,0}_{1,3}\left(0; \frac{1}{2}, -1, -\frac{3}{2} | x\right)$

is a good test case [54]; 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 suports 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 force and torques. The dyadics are essentially reference frame-aware 3×3 tensors. The vector and dyadic objects can be used for any one-, two-, or three-dimensional vector algebra, and they provide a strong framework for building physics and engineering tools.

The following Python code demonstrates how a vector is created using the orthogonal unit vectors of three reference frames that are oriented with respect to each other, and the result of expressing the vector in the A frame. The B frame is oriented with respect to the A frame using Z-X-Z Euler Angles of magnitude π, $\frac{\pi}{2}$, 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 [28] and Kane's Method [27]. Lastly, there are automated linearization routines for constrained dynamical systems [40].

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 [47]: 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
>>> A = Operator('A')
>>> B = Operator('B')
>>> C = Operator('C')
>>> D = Operator('D')
>>> a = Ket('a')
>>> comm = Commutator(A, B)
>>> comm
[A,B]
>>> qapply(Dagger(comm*a)).doit()
-<a|*(Dagger(A)*Dagger(B) - Dagger(B)*Dagger(A))
```

Commutators can be expanded using common commutator identities:

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

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

- Many of the exactly solvable quantum systems, including simple harmonic oscillator states and raising/lowering operators, infinite square well states, and 3D position and momentum operators and states.
- Second quantized formalism of non-relativistic many-body quantum mechanics [17].
- Quantum angular momentum [57]. Spin operators and their eigenstates can be represented in any basis and for any quantum numbers. A rotation operator representing the Wigner-D matrix, which may be defined symbolically or numerically, is also implemented to rotate spin eigenstates. Functionality for coupling and uncoupling of arbitrary spin eigenstates is provided, including symbolic representations of Clebsch-Gordon coefficients and Wigner symbols.

- Quantum information and computing [34]. Multidimensional qubit states, and a full set of one- and two-qubit gates are provided and can be represented symbolically or as matrices/vectors. With these building blocks, it is possible to implement a number of basic quantum algorithms including the quantum Fourier transform, quantum error correction, quantum teleportation, Grover's algorithm, dense coding, etc. In addition, any quantum circuit may be plotted using the `circuit_plot` function (Figure 1).

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

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

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

```
>>> Dagger(q)
<0101|
>>> ip = Dagger(q)*q
>>> ip
<0101|0101>
>>> ip.doit()
1
```

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

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

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).

```
>>> from sympy.physics.quantum.qft import QFT
>>> from sympy.physics.quantum.circuitplot import circuit_plot
>>> fourier = QFT(0,3).decompose()
>>> fourier
SWAP(0,2)*H(0)*C((0),S(1))*H(1)*C((0),T(2))*C((1),S(2))*H(2)
>>> 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

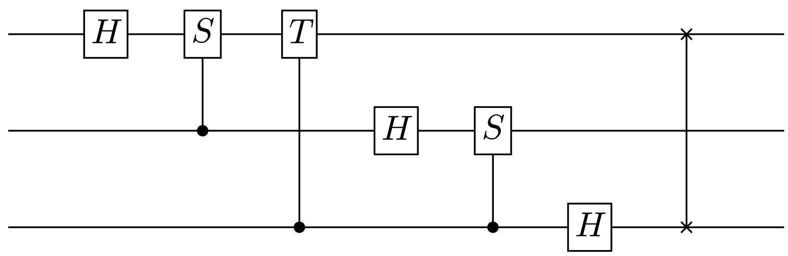

Fig. 1: The circuit diagram for a three-qubit quantum Fourier transform generated by SymPy.

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 on 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 where Google awards stipends to all students who successfully complete a requested free and open-source software coding project during the summer.

8. References.

REFERENCES

[1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Dover Publications, New York, NY, USA, ninth printing ed., 1964, http://www.math.ucla.edu/~cbm/aands/.

[2] W. W. Adams and P. Loustaunau, *An introduction to Gröbner bases*, no. 3, American Mathematical Soc., 1994.

[3] D. H. Bailey, K. Jeyabalan, and X. S. Li, *A comparison of three high-precision quadrature schemes*, Experimental Mathematics, 14 (2005), pp. 317–329.

[4] C. M. Bender and S. A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, Springer, 1st ed., October 1999.

[5] N. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory, 1736-1936*, Oxford University Press, 1976.

[6] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press, version 0.5.1 ed.

[7] M. Bronstein, *Poor Man's Integrator*, http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint.

[8] M. Bronstein, *Symbolic Integration I: Transcendental Functions*, Springer–Verlag, New York, NY, USA, 2005.

[9] B. Buchberger, *Ein Algorithmus zum Auffinden der Basis Elemente des Restklassenrings nach einem nulldimensionalen Polynomideal*, PhD thesis, University of Innsbruck, Innsbruck, Austria, 1965.

[10] 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/http://doi.acm.org/10.1145/1005285.1005298.

[11] D. Cervone, *Mathjax: a platform for mathematics on the web*, Notices of the AMS, 59 (2012), pp. 312–316.

[12] R. Cimrman, *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.

[13] R. J. Fateman, *A review of Mathematica*, Journal of Symbolic Computation, 13 (1992), pp. 545–579, http://dx.doi.org/DOI:10.1016/S0747-7171(10)80011-2.

[14] J. C. Faugère, *A New Efficient Algorithm for Computing Gröbner Bases (F4)*, Journal of Pure and Applied Algebra, 139 (1999), pp. 61–88, http://www-calfor.lip6.fr/~jcf/Papers/F99a.pdf.

[15] J. C. Faugère, *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, New York, NY, USA, 2002, ACM Press, pp. 75–83, http://dx.doi.org/http://doi.acm.org/10.1145/780506.780516, http://www-calfor.lip6.fr/~jcf/Papers/F02a.pdf.

[16] H. R. P. Ferguson, D. H. Bailey, and S. Arno, *Analysis of PSLQ, an integer relation finding algorithm*, Mathematics of Computation, 68 (1999), pp. 351–369.

[17] A. Fetter and J. Walecka, *Quantum Theory of Many-Particle Systems*, Dover Publications, 2003.

[18] H. Fu, X. Zhong, and Z. Zeng, *Automated and Readable Simplification of Trigonometric Expressions*, Mathematical and Computer Modelling, 55 (2006), pp. 1169–1177.

[19] Y. C. Fung, *A first course in continuum mechanics*, Pearson, third edition ed., 1993.

[20] 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.

[21] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), 23 (1991), pp. 5–48.

[22] R. W. Gosper, *Decision procedure for indefinite hypergeometric summation*, Proceedings of the National Academy of Sciences, 75 (1978), pp. 40–42.

[23] D. Gruntz, *On Computing Limits in a Symbolic Manipulation System*, PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.

[24] D. Gruntz and W. Koepf, *Formal power series*, (1993).

[25] C. V. Horsen, *GMPY*. https://pypi.python.org/pypi/gmpy2, 2015.

[26] J. D. Hunter, *Matplotlib: A 2d graphics environment*, Computing In Science & Engineering,

9 (2007), pp. 90–95.

[27] T. R. Kane and D. A. Levinson, *Dynamics, Theory and Applications*, McGraw Hill, 1985.

[28] J. Lagrange, *Mécanique analytique*, no. v. 1 in Mécanique analytique, Ve Courcier, 1811.

[29] S. Lang, *Introduction to transcendental numbers*, Reading, Mass, (1966).

[30] M. Lutz, *Learning python*, " O'Reilly Media, Inc.", 2013.

[31] L. R. U. Manssur, R. Portugal, and B. F. Svaiter, *Group-theoretic approach for symbolic tensor manipulation*, Int. J. Mod. Phys. C, 13 (2002), http://dx.doi.org/http://dx.doi.org/10.1142/S0129183102004571.

[32] J. Martín-García, *xact, efficient tensor computer algebra*, 2002-2016, http://metric.iem.csic.es/Martin-Garcia/xAct/.

[33] M. Moskewicz, C. Madigan, and S. Malik, *Method and system for efficient implementation of boolean satisfiability*, Aug. 26 2008, http://www.google.co.in/patents/US7418369. US Patent 7,418,369.

[34] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2011.

[35] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms: For Computers and Calculators*, Academic Press, New York, NY, USA, second ed., 1978.

[36] T. E. Oliphant, *Python for scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 10–20.

[37] 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.

[38] K. Peeters, *Cadabra: a field-theory motivated symbolic computer algebra system*, Computer Physics Communications, (2007).

[39] F. Pérez and B. E. Granger, *IPython: a system for interactive scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 21–29.

[40] 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.

[41] M. Petkovšek, H. S. Wilf, and D. Zeilberger, *A= bak peters*, Wellesley, MA, (1996).

[42] E. Raymond, *The cathedral and the bazaar*, Knowledge, Technology & Policy, 12 (1999), pp. 23–49.

[43] 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/http://doi.acm.org/10.1145/236869.237088, http://www.planetquantum.com/TheMission/Papers/Issac96.pdf.

[44] 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/http://doi.acm.org/10.1145/258726.258784.

[45] 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.

[46] L. Rosen, *Open source licensing*, vol. 692, Prentice Hall, 2005.

[47] J. Sakurai and J. Napolitano, *Modern Quantum Mechanics*, Addison-Wesley, 2010.

[48] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996. Prentice Hall Ordering Information.

[49] M. Sofroniou and G. Spaletta, *Precise numerical computation*, Journal of Logic and Algebraic Programming, 64 (2005), pp. 113–134.

[50] P. Solin, K. Segeth, and I. Dolezel, *Higher-Order Finite Element Methods*, Chapman & Hall / CRC Press, 2003.

[51] W. Stein and D. Joyner, *SAGE: System for Algebra and Geometry Experimentation*, Communications in Computer Algebra, 39 (2005).

[52] G. J. Sussman and J. Wisdom, *Functional Differential Geometry*, Massachusetts Institute of Technology Press, 2013.

[53] H. Takahasi and M. Mori, *Double exponential formulas for numerical integration*, Publications of the Research Institute for Mathematical Sciences, 9 (1974), pp. 721–741.

[54] 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.

[55] 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.

[56] S. Wolfram, *The Mathematica Book*, Wolfram Media, Champaign, IL, USA, fifth ed., 2003.

[57] R. Zare, *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics*, Wiley, 1991.

[58] O. Zienkiewicz, R. Taylor, and J. Zhu, *The Finite Element Method: Its Basis and Fundamentals*, Butterworth-Heinemann, seventh edition ed., 2013, http://dx.doi.org/http://dx.doi.org/10.1016/B978-1-85617-633-0.00019-8.

9. Supplement. As in the paper, all examples in the supplement assume the following has been run:

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
```

9.1. Limits: The Gruntz Algorithm. SymPy calculates limits using the Gruntz algorithm, as described in [23]. The basic idea is as follows: any limit can be converted to a limit $\lim_{x\to\infty} f(x)$ by substitutions like $x \to \frac{1}{x}$. Then the most varying subexpression ω (that converges to zero as $x \to \infty$ the fastest from all subexpressions) is identified in $f(x)$, and $f(x)$ is expanded into a series with respect to ω. Any positive powers of ω converge to zero. If there are negative powers of ω, then the limit is infinite. The constant term (independent of ω, but could depend on x) then determines the limit (one might need to recursively apply the Gruntz algorithm on this term to determine the limit).

To determine the most varying subexpression, the comparability classes must first be defined, by calculating L:

$$L \equiv \lim_{x\to\infty} \frac{\log |f(x)|}{\log |g(x)|} \tag{1}$$

The relations $<$, $>$, and $\sim$ are defined as follows: $f > g$ when $L = \pm\infty$ (it is said that f is more rapidly varying than g, i.e., f goes to ∞ or 0 faster than g), $f < g$ when $L = 0$ (f is less rapidly varying than g) and $f \sim g$ when $L \neq 0, \pm\infty$ (both f and g are bounded from above and below by suitable integral powers of the other). Note that if $f > g$, then $f > g^n$ for any n. Here are some examples of comparability classes:

$$2 < x < e^x < e^{x^2} < e^{e^x}$$

$$2 \sim 3 \sim -5$$

$$x \sim x^2 \sim x^3 \sim \frac{1}{x} \sim x^m \sim -x$$

$$e^x \sim e^{-x} \sim e^{2x} \sim e^{x+e^{-x}}$$

$$f(x) \sim \frac{1}{f(x)}$$

The Gruntz algorithm is now illustrated on the following example:

$$f(x) = e^{x+2e^{-x}} - e^x + \frac{1}{x}\,. \tag{2}$$

The goal is to calculate $\lim_{x\to\infty} f(x)$. First, the set of most rapidly varying subexpressions is determined—the so-called *mrv set*. For (2), the mrv set $\{e^x, e^{-x}, e^{x+2e^{-x}}\}$ is obtained. These are all subexpressions of (2) and they all belong to the same comparability class. This calculation can be done using SymPy as follows:

```
>>> from sympy.series.gruntz import mrv
>>> mrv(exp(x+2*exp(-x))-exp(x) + 1/x, x)[0].keys()
dict_keys([exp(x + 2*exp(-x)), exp(x), exp(-x)])
```

Next, any item ω is taken from mrv that converges to zero for $x \to \infty$. The item $\omega = e^{-x}$ is obtained. If such a term is not present in the mrv set (i.e., all terms converge to infinity instead of zero), the relation $f(x) \sim \frac{1}{f(x)}$ can be used.

The next step is to rewrite the mrv set in terms of ω: $\{\frac{1}{\omega}, \omega, \frac{1}{\omega}e^{2\omega}\}$. Then the original subexpressions are substituted back into $f(x)$ and expanded with respect to ω:

$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \tag{3}$$

Since ω is from the mrv set, then in the limit as $x \to \infty$, $\omega \to 0$, and so $2\omega + O(\omega^2) \to 0$ in (3):

$$f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2) \to 2 + \frac{1}{x} \tag{4}$$

Since the result $(2 + \frac{1}{x})$ still depends on x, the above procedure is iterated on the result until just a number (independent of x) is obtained, which is the final limit. In the above case the limit is 2, as can be verified by SymPy:

```
>>> limit(exp(x+2*exp(-x))-exp(x) + 1/x, x, oo)
2
```

In general, when $f(x)$ is expanded in terms of ω, it is obtained:

$$f(x) = \underbrace{O\left(\frac{1}{\omega^3}\right)}_{\infty} + \underbrace{\frac{C_{-2}(x)}{\omega^2}}_{\infty} + \underbrace{\frac{C_{-1}(x)}{\omega}}_{\infty} + C_0(x) + \underbrace{C_1(x)\omega}_{0} + \underbrace{O(\omega^2)}_{0} \tag{5}$$

The positive powers of ω are zero. If there are any negative powers of ω, then the result of the limit is infinity, otherwise the limit is equal to $\lim_{x\to\infty} C_0(x)$. The expression $C_0(x)$ is simpler than $f(x)$ and so the algorithm always converges. A proof of this and further details on the algorithm are given in Gruntz's PhD thesis [23].

9.2. Series.

9.2.1. Series Expansion. SymPy is able to calculate the symbolic series expansion of an arbitrary series or expression involving elementary and special functions and multiple variables. For this it has two different implementations: the `series` method and Ring Series.

The first approach stores a series as an instance of the `Expr` class. Each function has its specific implementation of its expansion, which is able to evaluate the Puiseux series expansion about a specified point. For example, consider a Taylor expansion about 0:

```
>>> from sympy import symbols, series
>>> x, y = symbols('x, y')
>>> series(sin(x+y) + cos(x*y), x, 0, 2)
1 + sin(y) + x*cos(y) + O(x**2)
```

The newer and much faster approach called Ring Series makes use of the observation that a truncated Taylor series is in fact a polynomial. Ring Series uses the efficient representation and operations of sparse polynomials. The choice of sparse polynomials is deliberate, as it performs well in a wider range of cases than a dense representation. Ring Series gives the user the freedom to choose the type of coefficients he wants to have in his series, allowing the use of faster operations on certain types.

For this, several low level methods for expansion of trigonometric, hyperbolic, and other elementary functions like inverse of a series, calculating nth root, etc, are implemented using variants of the Newton Method [6]. All these support Puiseux series expansion. The following example demonstrates the use of an elementary function that calculates the Taylor expansion of the sine of a series.

```
>>> from sympy import ring
>>> from sympy.polys.ring_series import rs_sin
>>> R, t = ring('t', QQ)
>>> rs_sin(t**2 + t, t, 5)
-1/2*t**4 - 1/6*t**3 + t**2 + t
```

The function `sympy.polys.rs_series` makes use of these elementary functions to expand an arbitrary SymPy expression. It does so by following a recursive strategy of expanding the lower most functions first and then composing them recursively to calculate the desired expansion. Currently, it only supports expansion about 0 and is under active development. Ring Series is several times faster than the default implementation with the speed difference increasing with the size of the series. The `sympy.polys.rs_series` takes as input any SymPy expression and hence there is no need to explicitly create a polynomial `ring`. An example demonstrating its use:

```
>>> from sympy.polys.ring_series import rs_series
>>> from sympy.abc import a, b
>>> from sympy import sin, cos
>>> rs_series(sin(a + b), a, 4)
-1/2*(sin(b))*a**2 + (sin(b)) - 1/6*a**3*(cos(b)) + a*(cos(b))
```

9.2.2. Formal Power Series. SymPy can be used for computing the Formal Power Series of a function. The implementation is based on the algorithm described in the paper on Formal Power Series [24]. The advantage of this approach is that an explicit formula for the coefficients of the series expansion is generated rather than just computing a few terms.

The following example shows how to use `fps`:

```
>>> f = fps(sin(x), x, x0=0)
>>> f.truncate(6)
x - x**3/6 + x**5/120 + O(x**6)
>>> f[15]
-x**15/1307674368000
```

9.2.3. Fourier Series. SymPy provides functionality to compute Fourier series of a function using the `fourier_series` function. Under the hood, this function computes a_0, a_n, and b_n coefficients using standard integration formulas.

Here's an example on how to compute Fourier series in SymPy:

```
>>> L = symbols('L')
>>> expr = 2 * (Heaviside(x/L) - Heaviside(x/L - 1)) - 1
>>> f = fourier_series(expr, (x, 0, 2*L))
>>> f.truncate(3)
4*sin(pi*x/L)/pi + 4*sin(3*pi*x/L)/(3*pi) + 4*sin(5*pi*x/L)/(5*pi)
```

9.3. Logic. SymPy supports construction and manipulation of boolean expressions through the `sympy.logic` module. SymPy symbols can be used as propositional variables and also be substituted as `True` or `False`. A good number of manipulation features for boolean expressions have been implemented in the `sympy.logic` module.

9.3.1. Constructing boolean expressions. A boolean variable can be declared as a SymPy `Symbol`. Python operators `&`, `|` and `~` are overridden when using SymPy objects to use the SymPy functionality for logical `And`, `Or`, and `Not`. Other logic functions are also integrated into SymPy, including `Xor` and `Implies`, which are constructed with `^` and `>>`, respectively. The above are just a shorthand, expressions can also be constructed by directly creating the relevant objects: `And()`, `Or()`, `Not()`, `Xor()`, `Implies()`, `Nand()`, `Nor()`, etc.

```
>>> e = (x & y) | z
>>> e.subs({x: True, y: True, z: False})
True
```

9.3.2. CNF and DNF. Any boolean expression can be converted to conjunctive normal form, disjunctive normal form, and negation normal form. The API also exposes methods to check if a boolean expression is in any of the above mentioned forms.

```
>>> from sympy.logic.boolalg import is_dnf, is_cnf
>>> to_cnf((x & y) | z)
And(Or(x, z), Or(y, z))
>>> to_dnf(x & (y | z))
Or(And(x, y), And(x, z))
>>> is_cnf((x | y) & z)
True
>>> is_dnf((x & y) | z)
True
```

9.3.3. Simplification and Equivalence. The module supports simplification of given boolean expression by making deductions from the expression. Equivalence of two logical expressions can also be checked. In the case of equivalence, it is possible to return the mapping of variables in two expressions so as to represent the same logical behavior.

```
>>> a, b, c = symbols('a b c')
>>> e = a & (~a | ~b) & (a | c)
>>> simplify(e)
And(Not(b), a)
>>> e1 = a & (b | c)
>>> e2 = (x & y) | (x & z)
>>> bool_map(e1, e2)
(And(Or(b, c), a), {a: x, b: y, c: z})
```

9.3.4. SAT solving. The module also supports satisfiability (SAT) checking of a given boolean expression. If satisfiable, it is possible to return a model for which the expression is satisfiable. The API also supports returning all possible models. The SAT solver has a clause learning DPLL algorithm implemented with a watch literal scheme and VSIDS heuristic [33].

```
>>> satisfiable(a & (~a | b) & (~b | c) & ~c)
False
>>> satisfiable(a & (~a | b) & (~b | c) & c)
{a: True, b: True, c: True}
```

9.4. Diophantine Equations. Diophantine equations play a central and an important role in number theory. A Diophantine equation has the form,

$$f(x_1, x_2, \ldots, x_n) = 0,$$

where $n \geq 2$ and $x_1, x_2, \ldots, x_n$ are integer variables. If we can find n integers $a_1, a_2, \ldots, a_n$ such that $x_1 = a_1, x_2 = a_2, \ldots, x_n = a_n$ satisfies the above equation, the equation is said to be solvable.

Currently, the following five types of Diophantine equations can be solved using SymPy's Diophantine module. $a_1, \ldots, a_{n+1}$, a, b, c, d, e, f, and k are explicitly given rational constants.

- Linear Diophantine equations: $a_1x_1 + a_2x_2 + \cdots + a_nx_n = b$
- General binary quadratic equation: $ax^2 + bxy + cy^2 + dx + ey + f = 0$
- Homogeneous ternary quadratic equation: $ax^2+by^2+cz^2+dxy+eyz+fzx = 0$
- Extended Pythagorean equation: $a_1x_1^2 + a_2x_2^2 + \cdots + a_nx_n^2 = a_{n+1}x_{n+1}^2$
- General sum of squares: $x_1^2 + x_2^2 + \cdots + x_n^2 = k$

When an equation is fed into the Diophantine module, it factors the equation (if possible) and solves each factor separately. Then, all the results are combined to create the final solution set. The following examples illustrate some of the basic functionalities of the Diophantine module.

```
>>> from sympy.solvers.diophantine import *
>>> diophantine(2*x + 3*y - 5)
set([(3*t_0 - 5, -2*t_0 + 5)])

>>> diophantine(2*x + 4*y - 3)
set()

>>> diophantine(x**2 - 4*x*y + 8*y**2 - 3*x + 7*y - 5)
set([(2, 1), (5, 1)])

>>> diophantine(x**2 - 4*x*y + 4*y**2 - 3*x + 7*y - 5)
set([(-2*t**2 - 7*t + 10, -t**2 - 3*t + 5)])

>>> diophantine(3*x**2 + 4*y**2 - 5*z**2 + 4*x*y - 7*y*z + 7*z*x)
set([(-16*p**2 + 28*p*q + 20*q**2,
3*p**2 + 38*p*q - 25*q**2,
4*p**2 - 24*p*q + 68*q**2)])

>>> x1, x2, x3, x4, x5, x6 = symbols('x1, x2, x3, x4, x5, x6')
>>> diophantine(9*x1**2 + 16*x2**2 + x3**2 + 49*x4**2 + 4*x5**2 - 25*x6**2)
set([(70*t1**2 + 70*t2**2 + 70*t3**2 + 70*t4**2 - 70*t5**2, 105*t1*t5,
420*t2*t5, 60*t3*t5, 210*t4*t5,
42*t1**2 + 42*t2**2 + 42*t3**2 + 42*t4**2 + 42*t5**2)])

>>> diophantine(x1**2 + x2**2 + x3**2 + x4**2 + x5**2 + x6**2 - 112)
set([(8, 4, 4, 4, 0, 0)])
```

9.5. Sets. SymPy supports representation of a wide variety of mathematical sets. This is achieved by first defining abstract representations of atomic set classes and then combining and transforming them using various set operations.

Each of the set classes inherits from the base class `Set` and defines methods to check membership and calculate unions, intersections, and set differences. When these methods are not able to evaluate to atomic set classes, they are represented as abstract unevaluated objects.

SymPy has the following atomic set classes:

- `EmptySet` represents the empty set $\emptyset$.
- `UniversalSet` is an abstract "universal set" for which everything is a member. The union of the universal set with any set gives the universal set and the intersection gives the other set itself.
- `FiniteSet` is functionally equivalent to Python's built in `set` object. Its members can be any SymPy object including other sets.
- `Integers` represents the set of integers $\mathbb{Z}$.
- `Naturals` represents the set of natural numbers $\mathbb{N}$, i.e., the set of positive integers.
- `Naturals0` represents the set of whole numbers $\mathbb{N}_0$, which are all the non-negative integers.
- `Range` represents a range of integers. A range is defined by specifying a start value, an end value, and a step size. The enumeration of a `Range` object is functionally equivalent to Python's `range` except it supports infinite endpoints, allowing the representation of infinite ranges.
- `Interval` represents an interval of real numbers. It is specified by giving the start and end point and specifying if it is open or closed in the respective ends.

Other than unevaluated classes of `Union`, `Intersection`, and `Complement` operations, SymPy has the following set classes.

- `ProductSet` defines the Cartesian product of two or more sets. The product set is useful when representing higher dimensional spaces. For example, to represent a three-dimensional space, SymPy uses take the Cartesian product of three real sets.
- `ImageSet` represents the image of a function when applied to a particular set. The image set of a function F with respect to a set S is $\{F(x)|x \in S\}$. SymPy uses image sets to represent sets of infinite solutions equations such as $\sin(x) = 0$.
- `ConditionSet` represents a subset of a set whose members satisfies a particular condition. The condition set of the set S with respect to the condition H is $\{x|H(x), x \in S\}$. SymPy uses condition sets to represent the set of solutions of equations and inequalities, where the equation or the inequality is the condition and the set is the domain being solved over.

A few other classes are implemented as special cases of the classes described above. The set of real numbers, `Reals`, is implemented as a special case of `Interval`. `ComplexRegion` is implemented as a special case of `ImageSet`. `ComplexRegion` supports both polar and rectangular representation of regions on the complex plane.

9.6. Category Theory. SymPy includes a module for dealing with categories—abstract mathematical objects representing classes of structures as classes of objects (points) and morphisms (arrows) between the objects. The module was designed with the following two goals in mind:

1. automatic typesetting of diagrams given by a collection of objects and of morphisms between them, and
2. specification and semi-automatic derivation of properties using commutative diagrams.

As of version 1.0, SymPy only implements the first goal, while a partially working draft of implementation of the second goal is available at https://github.com/scolobb/sympy/tree/ct4-commutativity.

In order to achieve the two goals, the module `sympy.categories` defines several classes representing some of the essential concepts: objects, morphisms, categories, and diagrams. In category theory, the inner structure of objects is often discarded in the favor of studying the properties of morphisms, so the class `Object` is essentially a synonym of the class `Symbol`. There are several morphism classes which do not have a particular internal structure either, though an exception is `CompositeMorphism`, which essentially stores a list of morphisms.

The class `Diagram` captures the properties of morphisms. This class stores a family of morphisms, the corresponding source and target objects, and, possibly, some properties of the morphisms. Generally, no restrictions are imposed on what the properties may be—for example, one might use strings of the form "forall", "exists", "unique", etc. Furthermore, the morphisms of a diagram are grouped into *premises* and *conclusions*, in order to be able to represent logical implications of the form "for a collection of morphisms P with properties $p : P \to \Omega$ (the premises), there exists a collection of morphisms C with properties $c : C \to \Omega$ (the conclusions)", where Ω is the universal collection of properties. Finally, the class `Category` includes a collection of diagrams which are deemed commutative and which therefore define the properties of this category.

Automatic typesetting of diagrams takes a `Diagram` and produces LaTeX code using the `Xy-pic` package. Typesetting is done in two stages: layout and generation of `Xy-pic` code. The layout stage is taken care of by the class `DiagramGrid`, which takes a `Diagram` and lays out the objects in a grid, trying to reduce the average length of the arrows in the final picture. By default, `DiagramGrid` uses a series of triangle-based heuristics to produce a rectangular grid. A linear layout can also be imposed. Furthermore, groups of objects can be given; in this case, the groups will be treated as atomic cells, and the member objects will be typeset independently of the other objects.

The second phase of diagram typesetting consists of actually drawing the picture and is carried out by the class `XypicDiagramDrawer`. An example of a diagram automatically typeset by `DiagramgGrid` and `XypicDiagramDrawer` in given in Figure 2.

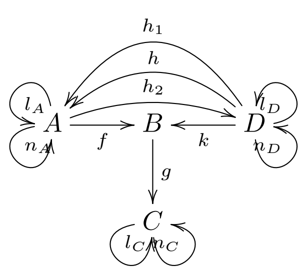

Fig. 2: An automatically typeset commutative diagram

As far as the second main goal of the module is concerned, the principal idea consists of automatically deciding whether a diagram is commutative or not, given a collection of "axioms": diagrams known to be commutative. The implementation is based on graph embeddings (injective maps): whenever an embedding of a commutative diagram into a given diagram is found, one concludes that the subdiagram is commutative. Deciding commutativity of the whole diagram is therefore based (theoretically) on finding a “cover” of the target diagram by embeddings of the axioms. The naïve implementation proved to be prohibitively slow; a better optimized version is therefore in order, as well as application of heuristics.

9.7. SymPy Gamma. SymPy Gamma is a simple web application that runs on Google App Engine. It executes and displays the results of SymPy expressions as well as additional related computations, in a fashion similar to that of Wolfram|Alpha. For instance, entering an integer will display its prime factors, digits in the base-10 expansion, and a factorization diagram. Entering a function will display its docstring; in general, entering an arbitrary expression will display its derivative, integral, series expansion, plot, and roots.

SymPy Gamma also has several additional features than just computing the results using SymPy.

- SymPy Gamma displays integration and differentiation steps in detail, which can be viewed in Figure 3:

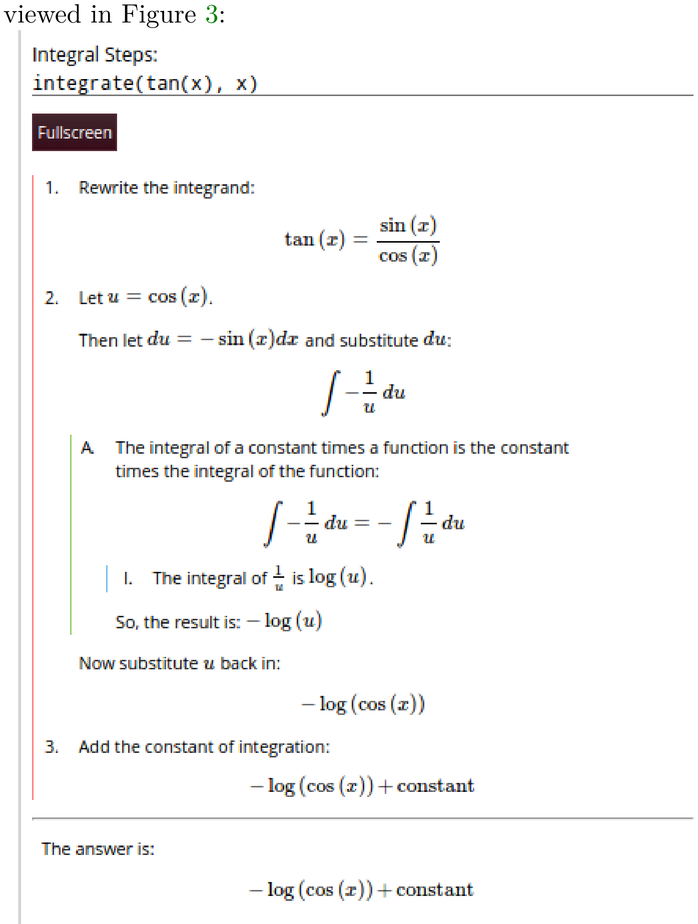

Fig. 3: Integral steps of $\tan(x)$

- SymPy Gamma displays the factor tree diagrams for different numbers.
- SymPy Gamma saves user search queries, and offers many such similar features for free, which Wolfram|Alpha only offers to its paid users.

Every input query from the user on SymPy Gamma is first parsed by its own parser, which handles several different forms of function names, which SymPy as a library does not support. For instance, SymPy Gamma supports queries like `sin x`, whereas SymPy does not support this, only `sin(x)`.

This parser converts the input query to the equivalent SymPy readable code, which is then eventually processed by SymPy, and the result is finally printed with the built-in LaTeX output and rendered on the SymPy Gamma web-application.

9.8. SymPy Live. SymPy Live is an online Python shell, which runs on the Google App Engine, that executes SymPy code. It is integrated in the SymPy documentation examples at http://docs.sympy.org.

This is accomplished by providing a HTML/JavaScript GUI for entering source code and visualization of output, and a server that evaluates the requested source code. It is an interactive AJAX shell that runs SymPy code using Python on the server.

9.9. Comparison with Mathematica. Wolfram Mathematica is a popular proprietary CAS. It features highly advanced algorithms. Mathematica has a core implemented in C++ [56] which interprets its own programming language, Wolfram Language.

Analogous to Lisp S-expressions, Mathematica uses its own style of M-expressions, which are arrays of either atoms or other M-expressions. The first element of the expression identifies the type of the expression and is indexed by zero, and the first argument is indexed starting with one. In SymPy, expression arguments are stored in a Python tuple (that is, an immutable array), while the expression type is identified by the type of the object storing the expression.

Mathematica can associate attributes to its atoms. Attributes may define mathematical properties and behavior of the nodes associated to the atom. In SymPy, the usage of static class fields is roughly similar to Mathematica's attributes, though other programming patterns may also be used the achieve an equivalent behavior, such as class inheritance.

Unlike SymPy, Mathematica expressions are mutable, that is, one can change parts of the expression tree without creating a new object. The mutability of Mathematica expressions allows for a lazy updating of any references to a given data structure.

Products in Mathematica are determined by some built in node types, such as `Times`, `Dot`, and others. `Times` is a representation of the `*` operator, and is always meant to represent a commutative product operator. The other notable product is `Dot`, which represents the `.` operator. This product represents matrix multiplication. It is not commutative. Unlike Mathematica, SymPy determines commutativity with respect to multiplication from the expression type of the factors. Mathematica puts the `Orderless` attribute on the expression type.

Regarding associative expressions, SymPy handles associativity of sums and products by automatically flattening them, Mathematica specifies the `Flat` attribute on the expression type.

Mathematica relies heavily on pattern matching—even the so-called equivalent of function declaration is in reality the definition of a pattern generating an expression tree transformation on input expressions. Mathematica's pattern matching is sensitive to associative, commutative, and one-identity properties of its expression tree nodes. SymPy has various ways to perform pattern matching. All of them play a lesser role in the CAS than in Mathematica and are basically available as a tool to rewrite expressions. The differential equation solver in SymPy somewhat relies on pattern matching to identify the kind of differential equation, but it is envisaged to replace that strategy with analysis of Lie symmetries in the future. Mathematica's real advantage is the ability to add new overloading to the expression builder at runtime, or for specific subnodes. Consider for example:

```
In[1]:= Unprotect[Plus]
Out[1]= {Plus}

In[2]:= Sin[x_]^2 + Cos[y_]^2 := 1

In[3]:= x + Sin[t]^2 + y + Cos[t]^2
Out[3]= 1 + x + y
```

This expression in Mathematica defines a substitution rule that overloads the functionality of the `Plus` node (the node for additions in Mathematica). The trailing underscore after a symbol means that it is to be considered a wildcard. This example may not be practical, as one may wish to keep this identity unevaluated. Nevertheless, it clearly illustrates the potential to define immediate transformation rules. In SymPy, the operations constructing the addition node in the expression tree are Python class constructors and cannot be modified at runtime.[6] The way SymPy deals with extending the missing runtime overloadability functionality is by subclassing the node types. Subclasses may redefine the class constructor to yield the proper extended functionality.

Unlike SymPy, Mathematica does not support type inheritance or polymorphism [13]. SymPy relies heavily on class inheritance, but for the most part, class inheritance is used to make sure that SymPy objects inherit the proper methods and implement the basic hashing system.

Matrices in SymPy are separate types from lists. In Mathematica, nested lists are interpreted as matrices whenever the sublists have the same length. The main difference to SymPy is that ordinary operators and functions do not get generalized the same way as used in traditional mathematics. Using the standard multiplication in Mathematica performs an element-wise product. This is compatible with Mathematica's convention of commutativity of `Times` nodes. Matrix products are expressed by the *dot* operator, or the `Dot` node. The same is true for the other operators, and even functions. Most notably, calling the exponential function `Exp` on a matrix returns an element-wise exponentiation of its elements. The usual matrix exponential is available through the `MatrixExp` function.

Unevaluated expressions in Mathematica can be achieved in various ways, most commonly with the `HoldForm` or `Hold` nodes, that block the evaluation of subnodes by the parser. Note that such a node cannot be expressed in Python, because of greedy evaluation. Whenever needed in SymPy, it is necessary to add the parameter `evaluate=False` to all subnodes.

In Mathematica, the operator `==` returns a boolean whenever it is able to immediately evaluate the truth of the equality, otherwise it returns an `Equal` expression. In SymPy, `==` means structural equality and is always guaranteed to return a boolean expression. To express a mathematical equality in SymPy it is necessary to explicitly construct an instance of the `Equality` class.

SymPy, in accordance with Python uses `**` to express the power operator, while Mathematica uses the more common `^`.

[6]In reality, Python supports monkey patching, nonetheless, it is a discouraged programming pattern.

SymPy's use of floating-point numbers is similar to that of most other CASs, including Maple and Maxima. By contrast, Mathematica uses a form of significance arithmetic [49] for approximate numbers. This offers further protection against numerical errors, although it comes with its own set of problems (for a critique of significance arithmetic, see Fateman [13]). Internally, SymPy's `evalf` method works similarly to Mathematica's significance arithmetic, but the semantics are isolated from the rest of the system.

9.10. Other Projects that use SymPy. There are several projects that use SymPy as a library for implementing a part of their project. Some of them are listed below:

- **Cadabra**: Cadabra is a CAS designed specifically for the solution of problems encountered in field theory.
- **Octave Symbolic**: The Octave-Forge Symbolic package adds symbolic calculation features to GNU Octave. These include common CAS tools such as algebraic operations, calculus, equation solving, Fourier and Laplace transforms, variable precision arithmetic, and other features.
- **SymPy.jl**: Provides a Julia interface to SymPy using PyCall.
- **Mathics**: Mathics is a free, general-purpose online CAS featuring Mathematica compatible syntax and functions. It is backed by highly extensible Python code, relying on SymPy for most mathematical tasks.
- **Mathpix**: An iOS App, that detects handwritten math as input, and uses SymPy Gamma to evaluate the math input and generate the relevant steps to solve the problem.
- **IKFast**: IKFast is a robot kinematics compiler provided by OpenRAVE. It analytically solves robot inverse kinematics equations and generates optimized C++ files. It uses SymPy for its internal symbolic mathematics.
- **Sage**: A CAS, visioned to be a viable free open source alternative to Magma, Maple, Mathematica and MATLAB. Sage includes many open source mathematical libraries, including SymPy.
- **SageMathCloud**: SageMathCloud is a web-based cloud computing and course management platform for computational mathematics.
- **PyDy**: Multibody Dynamics with Python.
- **galgebra**: Geometric algebra (previously `sympy.galgebra`).
- **yt**: Python package for analyzing and visualizing volumetric data (`yt.units` uses SymPy).
- **SfePy**: Simple finite elements in Python, see section 9.10.1.
- **Quameon**: Quantum Monte Carlo in Python.
- **Lcapy**: Experimental Python package for teaching linear circuit analysis.
- **Quantum Programming in Python**: Quantum 1D Simple Harmonic Oscillator and Quantum Mapping Gate.
- **LaTeX Expression project**: Easy LaTeX typesetting of algebraic expressions in symbolic form with automatic substitution and result computation.
- **Symbolic statistical modeling**: Adding statistical operations to complex physical models.

9.10.1. SfePy. **SfePy** (Simple finite elements in Python), cf. [12], is a Python package for solving partial differential equations (PDEs) in 1D, 2D and 3D by the finite element (FE) method [58]. SymPy is used within this package mostly for code generation and testing, namely:

- generation of the hierarchical FE basis module, involving generation and sym-

bolic differentiation of 1D Legendre and Lobatto polynomials, constructing the FE basis polynomials [50] and generating the C code;
- generation of symbolic conversion formulas for various groups of elastic constants [19]: provide any two of the Young's modulus, Poisson's ratio, bulk modulus, Lamé's first parameter, shear modulus (Lamé's second parameter) or longitudinal wave modulus and get the other ones;
- simple physical unit conversions, generation of consistent unit sets;
- testing FE solutions using method of manufactured (analytical) solutions: the differential operator of a PDE is symbolically applied and a symbolic right-hand side is created, evaluated in quadrature points, and subsequently used to obtain a numerical solution that is then compared to the analytical one;
- testing accuracy of 1D, 2D and 3D numerical quadrature formulas (cf. [1]) by generating polynomials of suitable orders, integrating them, and comparing the results with those obtained by the numerical quadrature.

9.11. Tensors. Ongoing work to provide the capabilities of tensor computer algebra has so far produced the `sympy.tensor` module. It is composed of three separate submodules, whose purposes are quite different: `sympy.tensor.indexed` and `sympy.tensor.indexed_methods` support indexed symbols, `sympy.tensor.array` contains facilities to operator on symbolic N-dimensional arrays, and finally `sympy.tensor.tensor` is used to define abstract tensors. The abstract tensors submodule is inspired by xAct [32] and Cadabra [38]. Canonicalization based on the Butler-Portugal [31] algorithm is supported in SymPy. It is currently limited to polynomial tensor expressions.

9.12. Numerical simplification. The `nsimplify` function in SymPy (a wrapper of `identify` in mpmath) attempts to find a simple symbolic expression that evaluates to the same numerical value as the given input. It works by applying a few simple transformations (including square roots, reciprocals, logarithms and exponentials) to the input and, for each transformed value, using the PSLQ algorithm [16] to search for a matching algebraic number or optionally a linear combination of user-provided base constants (such as π).

```
>>> t = 1 / (sin(pi/5)+sin(2*pi/5)+sin(3*pi/5)+sin(4*pi/5))**2
>>> nsimplify(t)
-2*sqrt(5)/5 + 1
>>> nsimplify(pi, tolerance=0.01)
22/7
>>> nsimplify(1.783919626661888, [pi], tolerance=1e-12)
pi/(-1/3 + 2*pi/3)
```

9.13. Examples.

9.13.1. Simplification.

- `expand`:
  ```
  >>> expand((x + y)**3)
  x**3 + 3*x**2*y + 3*x*y**2 + y**3
  ```
- `factor`:
  ```
  >>> factor(x**3 + 3*x**2*y + 3*x*y**2 + y**3)
  (x + y)**3
  ```
- `collect`:
  ```
  >>> collect(y*x**2 + 3*x**2 - x*y + x - 1, x)
  x**2*(y + 3) + x*(-y + 1) - 1
  ```
- `cancel`:

```
>>> cancel((x**2 + 2*x + 1)/(x**2 - 1))
(x + 1)/(x - 1)
```

- `apart`:

```
>>> apart((x**3 + 4*x - 1)/(x**2 - 1))
x + 3/(x + 1) + 2/(x - 1)
```

- `trigsimp`:

```
>>> trigsimp(cos(x)**2*tan(x) - sin(2*x))
-sin(2*x)/2
```

9.13.2. Polynomials.

- Factorization:

```
>>> t = symbols("t")
>>> f = (2115*x**4*y + 45*x**3*z**3*t**2 - 45*x**3*t**2 -
...      423*x*y**4 - 47*x*y**3 + 141*x*y*z**3 + 94*x*y*z*t -
...      9*y**3*z**3*t**2 + 9*y**3*t**2 - y**2*z**3*t**2 +
...      y**2*t**2 + 3*z**6*t**2 +  2*z**4*t**3 - 3*z**3*t**2 -
...      2*z*t**3)
>>> factor(f)
(t**2*z**3 - t**2 + 47*x*y)*(2*t*z + 45*x**3 - 9*y**3 - y**2 +
 3*z**3)
```

- Gröbner bases:

```
>>> x0, x1, x2 = symbols('x:3')
>>> I = [x0 + 2*x1 + 2*x2 - 1,
...      x0**2 + 2*x1**2 + 2*x2**2 - x0,
...      2*x0*x1 + 2*x1*x2 - x1]
>>> groebner(I, order='lex')
GroebnerBasis([7*x0 - 420*x2**3 + 158*x2**2 + 8*x2 - 7,
7*x1 + 210*x2**3 - 79*x2**2 + 3*x2,
84*x2**4 - 40*x2**3 + x2**2 + x2], x0, x1, x2, domain='ZZ',
order='lex')
```

- Root isolation:

```
>>> f = 7*z**4 - 19*z**3 + 20*z**2 + 17*z + 20
>>> intervals(f, all=True, eps=0.001)
([],
 [((-425/1024 - 625*I/1024, -1485/3584 - 2185*I/3584), 1),
  ((-425/1024 + 2185*I/3584, -1485/3584 + 625*I/1024), 1),
  ((3175/1792 - 2605*I/1792, 1815/1024 - 10415*I/7168), 1),
  ((3175/1792 + 10415*I/7168, 1815/1024 + 2605*I/1792), 1)])
```

9.13.3. Solvers.

- Single solution:

```
>>> solveset(x - 1, x)
{1}
```

- Finite solution set, quadratic equation:

```
>>> solveset(x**2 - pi**2, x)
{-pi, pi}
```

- No solution:

```
>>> solveset(1, x)
EmptySet()
```

- Interval solution:

```
>>> solveset(x**2 - 3 > 0, x, domain=S.Reals)
```

```
(-oo, -sqrt(3)) U (sqrt(3), oo)
```

- Infinitely many solutions:

```
>>> solveset(x - x, x, domain=S.Reals)
(-oo, oo)
>>> solveset(x - x, x, domain=S.Complexes)
S.Complexes
```

- Linear systems (`linsolve`)

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> b = Matrix([3, 6, 9])
>>> linsolve((A, b), x, y, z)
{(-1, 2, 0)}
>>> linsolve(Matrix(([1, 1, 1, 1], [1, 1, 2, 3])), (x, y, z))
{(-y - 1, y, 2)}
```

Below are examples of `solve` applied to problems not yet handled by `solveset`.

- Nonlinear (multivariate) system of equations (the intersection of a circle and a parabola):

```
>>> solve([x**2 + y**2 - 16, 4*x - y**2 + 6], x, y)
[(-2 + sqrt(14), -sqrt(-2 + 4*sqrt(14))),
 (-2 + sqrt(14), sqrt(-2 + 4*sqrt(14))),
 (-sqrt(14) - 2, -I*sqrt(2 + 4*sqrt(14))),
 (-sqrt(14) - 2, I*sqrt(2 + 4*sqrt(14)))]
```

- Transcendental equations:

```
>>> solve((x + log(x))**2 - 5*(x + log(x)) + 6, x)
[LambertW(exp(2)), LambertW(exp(3))]
>>> solve(x**3 + exp(x))
[-3*LambertW((-1)**(2/3)/3)]
```

9.13.4. Matrices.

- Matrix expressions

```
>>> m, n, p = symbols("m, n, p", integer=True)
>>> R = MatrixSymbol("R", m, n)
>>> S = MatrixSymbol("S", n, p)
>>> T = MatrixSymbol("T", m, p)
>>> U = R*S + 2*T
>>> U.shape
(m, p)
>>> U[0, 1]
2*T[0, 1] + Sum(R[0, _k]*S[_k, 1], (_k, 0, n - 1))
```

- Block Matrices

```
>>> n, m, l = symbols('n m l')
>>> X = MatrixSymbol('X', n, n)
>>> Y = MatrixSymbol('Y', m ,m)
>>> Z = MatrixSymbol('Z', n, m)
>>> B = BlockMatrix([[X, Z], [ZeroMatrix(m, n), Y]])
>>> B
Matrix([
[X, Z],
[0, Y]])
>>> B[0, 0]
X[0, 0]
```

```
>>> B.shape
(m + n, m + n)
```