SYMPY: SYMBOLIC COMPUTING IN PYTHON

ONDŘEJ ČERTÍK*, ISURU FERNANDO†, AND ASHUTOSH SABOO‡

1. Introduction. SymPy is a full featured computer algebra system (CAS) written in the Python programming language. It is a free and open source software, being licensed under the 3-clause BSD license. SymPy was started by Ondřej Čertík in 2005, and it has since grown into a large project with over 500 contributors. SymPy is developed on GitHub using a bazaar community model [40]. The accessibility of the codebase and the open community model allows SymPy to rapidly respond to the needs of the community of users, and has made the large contributor count possible.

Python is a popular dynamically typed programming language that has a focus on ease of use and readability. It also a very popular language for scientific computing and data science, with a wide range of useful libraries [35]. SymPy is itself used by many libraries and tools across many domains, such as Sage [44] (pure mathematics), yt [48] (astronomy and astrophysics), PyDy [23] (multibody dynamics), and SfePy [17] (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 one programming language makes it easier for people already familiar with it to use or develop SymPy and at the same time allows developers to focus on mathematics, rather than language design.

SymPy is designed with a strong focus on usability as a library. This means that extensibility is important in its application program interface (API) design. This is also one of the reasons SymPy makes no attempt to extend the Python language itself. The goal is for users of SymPy to be able to import SymPy alongside other Python libraries in their workflow, whether that is an interactive workflow or programmatic use as part of a larger system.

Being developed 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 [37] frontends, including the Notebook and Qt Console, which will pretty print SymPy expressions using MathJax [16] or LaTeX rendering.

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 doing classical mechanics and quantum mechanics. Finally, Section 6 concludes the paper and discusses future work.

2. Architecture.

2.1. Basic Usage. Being built on Python, SymPy requires that all variable names be defined before they can be used. The statement

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

will import all public SymPy functions into the global Python namespace. All the examples in this paper assume that this has been run.

*Los Alamos National Laboratory (ondrej.certik@gmail.com).
†University of Moratuwa (isuru.11@cse.mrt.ac.lk).
‡Birla Institute of Technology and Science, Pilani, K.K. Birla Goa Campus (ashutosh.saboo@gmail.com).

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 creates multiple symbols at once. For instance,

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

creates three symbols representing variables named x, y, and z, assigned to Python variables of the same name. The Python variable names that symbols are assigned to are immaterial—one could have just as well have written `a, b, c = symbols('x y z')`. All examples in this paper will assume that the symbols `x`, `y`, and `z` have been assigned to a variable identical to their names.

Expressions are created from symbols using Python syntax through operator overloading, which mirrors usual mathematical notation. Note that in Python, exponentiation is `**`. For instance, the following creates the expression $(x^2 - 2x + 3)/y$.

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

All SymPy expressions are immutable. This simplifies the design by allowing interning. It also allows expressions to be hashed and stored in Python dictionaries, thereby enabling caching and other features.

2.2. The Core. The core of a computer algebra system (CAS) refers to the module that is in charge of representing symbolic expressions and performing basic manipulations with them. In SymPy, every symbolic expression is an instance of properly designed Python classes, representing expression trees. The operators are represented by the type of an expression and the child nodes are stored in the `args` attribute. A leaf node in the expression tree has an empty `args`. The `args` attribute is provided by the class `Basic`, which is a superclass of all SymPy objects and provides common methods to all SymPy tree-elements. 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)
```

One can dig further into the expression tree to see the full expression. For example, the first child node, given by `expr.args[0]` is `2`. Its class is `Integer`, and it has empty `args`, 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 with 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 invariant, namely, `expr.func(*expr.args) == expr`. This means that expressions are rebuildable from their `args` [1]. Here, we note that in SymPy, the `==` operator represents exact structural equality, not mathematical equality. This allows one to test if any two expressions are equal to one another as expression trees.

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 builtin `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.

One must be careful in one particular instance: Python does not have a builtin rational literal type. Given a fraction of integers such as `1/2`, Python will perform floating point division and produce `0.5` [2]. Python uses eager evaluation, so expressions like `x + 1/2` will produce `x + 0.5`, because by the time any SymPy function sees the `1/2` it will have already been converted to `0.5` by Python. However, for a CAS like SymPy, one typically wants to work with exact rational numbers whenever possible. Working around this is simple, however: one can wrap one of the integers with `Integer`, like `x + Integer(1)/2`, or using `x + Rational(1, 2)`. SymPy provides a function `S` which can be used to convert objects to SymPy types with minimal typing, such as `x + S(1)/2`. This gotcha is a small downside to using Python directly instead of a custom domain specific language (DSL), and we consider it to be worth it for the advantages of using Python.

2.3. Assumptions. An important feature of the SymPy core is the 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)
```

[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.

[2] This is the behavior in Python 3. In Python 2, `1/2` will perform integer division and produce `0`, unless one uses `from __future__ import division`.

`t`

Some of the common assumptions that SymPy allows are `positive`, `negative`, `real`, `nonpositive`, `nonnegative`, `real`, `integer`, and `commutative` [3]. 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. It is 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 builtin 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 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, it is an open problem in mathematics.

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`.

SymPy also has an experimental assumptions system where facts are stored separate from objects, and deductions are made with a SAT solver. We will not discuss this system here.

2.4. Extensibility. Extensibility is an important feature for SymPy. Because the same language, Python, is used both for the internal implementation and the external usage by users, all the extensibility capabilities available to users are also used by functions that are part of SymPy.

The typical way to create a custom SymPy object is to subclass an existing SymPy class, generally one of `Basic`, `Expr`, or `Function`. All SymPy classes used for expression trees [4] 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.

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

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

Fig. 1: A stripped down version of `sympy.gamma`.

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

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

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

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

    def _eval_rewrite_as_factorial(self, z):
        return factorial(z - 1)

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

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, Figure 1 presents a stripped down 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. The actual gamma function defined in SymPy has many more capabilities, such as evaluation at rational points and series expansion.

3. Features. SymPy has an extensive feature set that encompasses too much to cover in-depth here. Bedrock areas, such as calculus, receive their own subsections below. Table 1 gives a compact listing of all major capabilities present in the SymPy codebase. This gives 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	Enables generation of compilable and executable code in a variety of different programming languages directly from expressions. Target languages include C, Fortran, Julia, JavaScript, Mathematica, Matlab and Octave, Python, and Theano.
Combinatorics & Group Theory	Implements permutations, combinations, partitions, subsets, various permutation groups (such as polyhedral, Rubik, symmetric, and others), Gray codes [34], and Prufer sequences [11].
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 [39] for two univariate polynomials.
Cryptography	Represents 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	Classes to represent manifolds, metrics, tensor products, and coordinate systems in Riemannian and pseudo-Riemannian geometries [45].
Geometry	Allows the creation 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	Represents Lie algebras and root systems.
Logic	boolean expression, equivalence testing, satisfiability, normal forms.
Matrices	Tools for creating matrices of symbols and expressions. This is capable of both sparse and dense representations and performing symbolic linear algebraic operations (e.g., inversion and factorization).
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, integer factorization.
Plotting	Hooks for visualizing expressions via matplotlib [**?**] or as text drawings when lacking a graphical back-end. 2D function plotting, 3D function plotting, and 2D implicit function plotting are supported.
Polynomials	Computes polynomial algebras over various coefficient domains. Functionality ranges from the simple (e.g., polynomial division) to the advanced (e.g., Gröbner bases [8] 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.
Series	Implements series expansion, sequences, and limit 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.
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 algebraically, systems of equations, both linear and non-linear, inequalities, ordinary differential equations, partial differential equations, Diophantine equations, and recurrence relations.
Special Functions	Implements 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.
Tensors	Symbolic manipulation of indexed objects.

Vectors	Provides basic vector math 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 unambigously defined mathematical operation [15]. The `simplify` function applies several simplification routines along with some 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 often 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 some common simplification functions.

Table 2: Some SymPy Simplification Functions

`expand`	expand the expression
	`>>> expand((x + y)**3)`
	`x**3 + 3*x**2*y + 3*x*y**2 + y**3`

`factor`	factor a polynomial into irreducibles
	`>>> factor(x**3 + 3*x**2*y + 3*x*y**2 + y**3)`
	`(x + y)**3`

`collect`	collect polynomial coefficients
	`>>> collect(y*x**2 + 3*x**2 - x*y + x - 1, x)`
	`x**2*(y + 3) + x*(-y + 1) - 1`

`cancel`	rewrite a rational function as p/q with common factors canceled
	`>>> cancel((x**2 + 2*x + 1)/(x**2 - 1))`
	`(x + 1)/(x - 1)`

`apart`	compute the partial fraction decomposition of a rational function
	`>>> apart((x**3 + 4*x - 1)/(x**2 - 1))`
	`x + 3/(x + 1) + 2/(x - 1)`

`trigsimp`	simplify trigonometric expressions [21]
	`>>> trigsimp(cos(x)**2*tan(x) - sin(2*x))`
	`-sin(2*x)/2`

Substitutions are performed through 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 [14], table lookups, a reimplementation of Manuel Bronstein's "Poor Man's Integrator" [13], and an algorithm for computing integrals based on Meijer G-functions. These allow SymPy to compute a wide variety of indefinite and definite integrals.

```
>>> integrate(sin(x), x)
-cos(x)
>>> integrate(sin(x), (x, 0, 1))
-cos(1) + 1
```

Derivatives are computed with the `diff` function. Derivatives are computed recursively using the 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, an algorithm that uses Meijer G-functions, and heuristics. Products are computed via some heuristics.

Limits are computed with the `limit` function. The limit module implements the Gruntz algorithm [25] for computing symbolic limits. For example, the following computes $\lim_{x\to\infty} x\sin(\frac{1}{x}) = 1$ (note that ∞ is `oo` in SymPy).

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

As a more complicated 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 can't 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 wide variety of algorithms for polynomial manipulation, which ranges from relatively simple algorithms for doing arithmetics 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, but in SymPy, it's 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 and in the end, solutions to original one are recovered. For example, this is a common scheme in symbolic integration or summation algorithms.

SymPy implements dense and sparse polynomial representations. Both are used in univariate and multivariate cases. 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 sparse representation is algorithms for computing Gröbner bases (Buchberger, F4 and F5), 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, are better expressed when the representation is dense. By dense multivariate representation we mean a recursively dense representation, where polynomial $K[x_0, x_1, \ldots, x_n]$ is viewed as a polynomial 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. Dense recursive representation in Python gets inefficient when the number of variables gets high.

Factorization:

```
>>> var("x,y,z,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)
(47*x*y + z**3*t**2 - t**2)*(45*x**3 - 9*y**3 - y**2 + 3*z**3 + 2*z*t)
```

Gröbner bases:

```
>>> var('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, oder='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:

```
>>> var('z')
>>> 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)])
```

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, which has been used in all the examples in this paper so far.

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

Expressions can be printed with 2D monospace text with `pprint`. This uses Unicode characters to render mathematical symbols such as integral signs, square roots, and parentheses. Greek letters and subscripts in symbol names are 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)
  /
 |
 |   __________
 | \/ 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 qtconsole [37] 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 which allows users to customize the printing for any given printer, and for custom objects to define their printing behavior for any printer. SymPy's code generation capabilities, which we will not discuss in-depth here, use this subsystem to convert expressions into code in various languages.

3.5. Solvers. SymPy has a module of equation solvers for symbolic equations. There are two submodules to solve algebraic equations in SymPy, referred to as old solve function, `solve`, and new solve function, `solveset`. Solveset is introduced with several design changes with respect to the old `solve` function to resolve the issues with old `solve` function, for example old `solve` function's input API has many flags which are not needed and they make it hard for the user and the developers to work on solvers. In contrast to the old solve function, the `solveset` has a clean input API, it only asks for the necessary information from the user. The function signatures of the old and new solve function:

```
solve(f, *symbols, **flags)  # old solve function
solveset(f, symbol, domain)  # new solve function
```

The old `solve` function has an inconsistent output API for various types of inputs, whereas the `solveset` has a canonical output API which is achieved using sets. It can consistently return various types of solutions.

- Single solution

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

- Finite set of solution, quadratic equation

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

- No Solution

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

- Interval of solution

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

- Infinitely many solutions

```
>>> solveset(sin(x) - 1, x, domain=S.Reals)
ImageSet(Lambda(_n, 2*_n*pi + pi/2), Integers())
>>> solveset(x - x, x, domain=S.Reals)
(-oo, oo)
>>> solveset(x - x, x, domain=S.Complexes)
S.Complexes
```

- Linear system: finite and infinite solution for determined, under determined and over determined problems.

```
>>> 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)}
```

The new solve i.e. **solveset** is under active development and is a planned replacement for **solve**, Hence there are some features which are implemented in solve and is not yet implemented in solveset. The table below show the current state of old and new solve functions.

Solveset vs Solve		
Feature	solve	solveset
Consistent Output API	No	Yes
Consistent Input API	No	Yes
Univariate	Yes	Yes
Linear System	Yes	Yes (linsolve)
Non Linear System	Yes	Not yet
Transcendental	Yes	Not yet

Below are some of the examples of old **solve** function:

- Non Linear (multivariate) System of Equation: 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 Equation

```
>>> 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)]
```

3.6. Matrices. SymPy supports matrices with symbolic expressions as elements.

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

All SymPy matrix types can do linear algebra including matrix addition, multiplication, exponentiation, computing determinant, 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 computed symbolically. Eigenvalues are computed by generating the characteristic polynomial using the Berkowitz algorithm and then solving it using polynomial routines. Diagonalizable matrices can be diagonalized first to compute the eigenvalues.

```
>>> A.eigenvals()
```

```
{x - sqrt(y*(x + y)): 1, x + sqrt(y*(x + y)): 1}
```

Internally these matrices store the elements as a list, making it a dense representation. For storing sparse matrices, the `SparseMatrix` class can be used. Sparse matrices store the elements in a dictionary of keys (DoK) format.

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.

```
>>> 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 are also supported in SymPy. `BlockMatrix` elements can be any matrix expression which includes explicit matrices, matrix symbols, and block matrices. All functionalities of matrix expressions are also present in `BlockMatrix`.

```
>>> 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)
```

4. Numerics. The `Float` class holds an arbitrary-precision binary floating-point value and a precision in bits. An operation between two `Float` inputs is rounded to the larger of the two precisions. Since Python floating-point literals automatically evaluate to `double` (53-bit) 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 preferred way to evaluate an expression numerically is with the `evalf` method, which internally estimates the number of accurate bits of the floating-point approximation for each sub-expression, and adaptively increases the working precision until the estimated accuracy of the final result matches the sought number of decimal digits.

The internal error tracking does not provide rigorous error bounds (in the sense of interval arithmetic) and cannot be used to track uncertainty in measurement data in any meaningful way; the sole purpose is to mitigate loss of accuracy that typically occurs when converting symbolic expressions to numerical values, for example due to catastrophic cancellation. This is illustrated by the following example (the input 25 specifies that 25 digits are sought):

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

SymPy does not track the accuracy of approximate numbers outside of `evalf`. The familiar dangers of floating-point arithmetic apply [24], and symbolic expressions containing floating-point numbers should be treated with some caution. This approach is similar to Maple and Maxima.

By contrast, Mathematica uses a form of significance arithmetic [42] for approximate numbers. This offers further protection against numerical errors, but leads to non-obvious semantics while still not being mathematically rigorous (for a critique of significance arithmetic, see Fateman [18]). SymPy's `evalf` internals are non-rigorous in the same sense, but have no bearing on the semantics of floating-point numbers in the rest of the system.

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 *` (it is best to avoid such an import statement when using SymPy simultaneously, since numerical functions such as `exp` will shadow the symbolic counterparts in SymPy).

Like SymPy, mpmath is a pure Python library. Internally, mpmath represents a floating-point number $(-1)^s x \cdot 2^y$ by a tuple (s, x, y, b) where x and y are arbitrary-size Python integers and the redundant integer b stores the bit length of x for quick access. If GMPY [27] is installed, mpmath automatically switches to using the `gmpy.mpz` type for x and using GMPY helper methods to perform rounding-related operations, improving performance.

The mpmath library includes support for 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 support 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 [46, 9]. 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 attempts to apply Richardson extrapolation and the Shanks transformation (Euler-Maclaurin summation can also be used) [10]. 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.

Most special functions are implemented as linear combinations of the generalized hypergeometric function ${}_pF_q$, which is computed by a combination of direct summation, argument transformations (for ${}_2F_1$, ${}_3F_2$, $\ldots$) and asymptotic expansions (for ${}_0F_1$, ${}_1F_1$, ${}_1F_2$, ${}_2F_2$, ${}_2F_3$) to cover the whole complex domain. Numerical integration and generic convergence acceleration are also used in a few special cases.

In general, linear combinations and argument transformations give rise to singularities that have to be removed for certain combinations of parameters. A typical example is the modified Bessel function of the second kind

$$K_\nu(z) = \frac{1}{2}\left[\left(\frac{z}{2}\right)^{-\nu}\Gamma(\nu){}_0F_1\left(1-\nu,\frac{z^2}{4}\right) - \left(\frac{z}{2}\right)^{\nu}\frac{\pi}{\nu\sin(\pi\nu)\Gamma(\nu)}{}_0F_1\left(\nu+1,\frac{z^2}{4}\right)\right]$$

where the limiting value $\lim_{\varepsilon\to 0} K_{n+\varepsilon}(z)$ has to be computed when $\nu = n$ is an integer. A generic algorithm is used to evaluate hypergeometric-type linear combinations of the above type. This algorithm automatically detects cancellation problems, and computes limits numerically by perturbing parameters whenever internal singularities occur (the perturbation size is automatically decreased until the result is detected to converge numerically).

Due to this generic approach, particular combinations of hypergeometric functions can be specified easily. The implementation of the Meijer G-function takes only a few dozen lines of code, yet covers the whole input domain in a robust way. 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 [47]; past versions of both Maple and Mathematica produced incorrect numerical values for large $x > 0$. Here, mpmath automatically removes the 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)
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
```

We highlight the generalized hypergeometric functions and the Meijer G-function, due to those functions' frequent appearance in closed forms for integrals and sums (see Section 3.2). Via mpmath, SymPy has relatively good support for evaluating sums and integrals numerically, using two complementary approaches: direct numerical evaluation, or first computing a symbolic closed form involving special functions.

4.2. 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 [19] 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)
```

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 classical mechanics, optics, and quantum mechanics along with support for manipuating physical quantities with units.

5.1. Classical 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, cross products, etc. 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 interpreter session showing 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 import pi
>>> 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 any arbitrary kinematical constraints or complex loads. The package offers two automated methods for formulating the equations of motion based on Lagrangian Dynamics [29] and Kane's Method [28]. Lastly, there are automated linearization routines for constrained dynamical systems based on [38].

5.2. Symbolic Quantum Mechanics. The `sympy.physics.quantum` package has extensive capabilities for symbolic quantum mechanics, with Python objects to represent the different mathematical objects relevant in quantum theory [41]: 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, anticommutators, etc. 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 to provide system specific logic.

For example, you can define symbolic quantum operators and states and 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. These include:

- Position/momentum operators and states, raising/lowering operators and states, simple harmonic oscillator, density matrices, hydrogen atom.
- Second quantized formalism of non-relativistic many-body quantum mechanics [20].
- Quantum angular momentum [49]. Spin operators and their eigenstates can be represented in any basis and for any quantum numbers. Facilities for Clebsch-Gordan Coefficients, Wigner Coefficients, rotations, and angular momentum coupling are also present in their symbolic and numerical forms.
- Quantum information and computing [33]. 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.

Here are a few short examples of the quantum information and computing capabilities in `sympy.physics.quantum`. We start with a simple 4 qubit state and flip one of the qubits:

```
>>> from sympy.physics.quantum.qubit import Qubit
>>> q = Qubit('0101')
>>> q
|0101>
>>> q.flip(1)
|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 Qubit, measure_all
>>> from sympy.physics.quantum.gate import H, X, Y, Z
>>> from sympy.physics.quantum.qapply import qapply
>>> 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)]
```

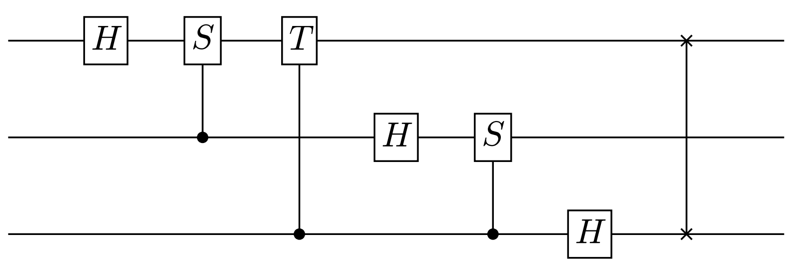

Fig. 2: The circuit diagram for a 3-qubit quantum fourier transform generated by SymPy.

Here is a final example of creating a 3-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 2).

```
>>> 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 CAS that provides a wide array of features. It is written in a general purpose programming language, Python, which allows it to be used in a first-class way with other Python projects, including the scientific Python stack. It is designed to be used in an extensible way. Unlike many other CASs, it is designed to be used both as a end-user application and as a library.

SymPy expressions are built from immutable trees of Python classes. It uses Python both as the internal language and the user language, meaning users can use the same methods that the library implements to extend it. SymPy has an assumptions system for declaring and deducing mathematical properties on expressions.

The numerics of SymPy are implemented in the mpmath library, which uses arbitrary precision floating point arithmetic implemented in pure Python. This allows expressions to be evaluated with concrete data as needed.

SymPy has submodules for many areas of mathematics. It has functions for simplifying expressions, doing common calculus operations, pretty printing expressions, solving equations, and 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.

Some of the planned future work for SymPy includes work on improving code generation, improvements to the speed of SymPy, and improving the solvers module.

7. Acknowledgements. 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.

8. References.

REFERENCES

[1] https://github.com/sympy/sympy/blob/master/doc/src/modules/polys/ringseries.rst.
[2] https://reference.wolfram.com/language/ref/Flat.html.
[3] https://reference.wolfram.com/language/ref/Orderless.html.
[4] https://reference.wolfram.com/language/ref/OneIdentity.html.
[5] https://reference.wolfram.com/language/tutorial/FlatAndOrderlessFunctions.html.
[6] *The software engineering of the wolfram system*, 2016, https://reference.wolfram.com/language/tutorial/TheSoftwareEngineeringOfTheWolframSystem.html.
[7] 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/.
[8] W. W. Adams and P. Loustaunau, *An introduction to Gröbner bases*, no. 3, American Mathematical Soc., 1994.

[9] D. H. Bailey, K. Jeyabalan, and X. S. Li, *A comparison of three high-precision quadrature schemes*, Experimental Mathematics, 14 (2005), pp. 317–329.
[10] C. M. Bender and S. A. Orszag, *Advanced Mathematical Methods for Scientists and Engineers*, Springer, 1st ed., October 1999.
[11] N. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory, 1736-1936*, Oxford University Press, 1976.
[12] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press, version 0.5.1 ed.
[13] M. Bronstein, *Poor Man's Integrator*, http://www-sop.inria.fr/cafe/Manuel.Bronstein/pmint.
[14] M. Bronstein, *Symbolic Integration I: Transcendental Functions*, Springer–Verlag, New York, NY, USA, 2005.
[15] 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.
[16] D. Cervone, *Mathjax: a platform for mathematics on the web*, Notices of the AMS, 59 (2012), pp. 312–316.
[17] 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.
[18] 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.
[19] 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.
[20] A. Fetter and J. Walecka, *Quantum Theory of Many-Particle Systems*, Dover Publications, 2003.
[21] H. Fu, X. Zhong, and Z. Zeng, *Automated and Readable Simplification of Trigonometric Expressions*, Mathematical and Computer Modelling, 55 (2006), pp. 1169–1177.
[22] Y. C. Fung, *A first course in continuum mechanics*, Pearson, third edition ed., 1993.
[23] 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.
[24] D. Goldberg, *What every computer scientist should know about floating-point arithmetic*, ACM Computing Surveys (CSUR), 23 (1991), pp. 5–48.
[25] D. Gruntz, *On Computing Limits in a Symbolic Manipulation System*, PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 1996.
[26] D. Gruntz and W. Koepf, *Formal power series*, (1993).
[27] C. V. Horsen, *GMPY*. https://pypi.python.org/pypi/gmpy2, 2015.
[28] T. R. Kane and D. A. Levinson, *Dynamics, Theory and Applications*, McGraw Hill, 1985.
[29] J. Lagrange, *Mécanique analytique*, no. v. 1 in Mécanique analytique, Ve Courcier, 1811.
[30] 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.
[31] J. Martín-García, *xact, efficient tensor computer algebra*, 2002-2016, http://metric.iem.csic.es/Martin-Garcia/xAct/.
[32] 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.
[33] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2011.
[34] A. Nijenhuis and H. S. Wilf, *Combinatorial Algorithms: For Computers and Calculators*, Academic Press, New York, NY, USA, second ed., 1978.
[35] T. E. Oliphant, *Python for scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 10–20.
[36] K. Peeters, *Cadabra: a field-theory motivated symbolic computer algebra system*, Computer Physics Communications, (2007).
[37] F. Pérez and B. E. Granger, *Ipython: a system for interactive scientific computing*, Computing in Science & Engineering, 9 (2007), pp. 21–29.
[38] 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.
[39] M. Petkovšek, H. S. Wilf, and D. Zeilberger, *A= bak peters*, Wellesley, MA, (1996).
[40] E. Raymond, *The cathedral and the bazaar*, Knowledge, Technology & Policy, 12 (1999), pp. 23–49.
[41] J. Sakurai and J. Napolitano, *Modern Quantum Mechanics*, Addison-Wesley, 2010.
[42] M. Sofroniou and G. Spaletta, *Precise numerical computation*, Journal of Logic and Algebraic Programming, 64 (2005), pp. 113–134.
[43] P. Solin, K. Segeth, and I. Dolezel, *Higher-Order Finite Element Methods*, Chapman & Hall / CRC Press, 2003.
[44] W. Stein and D. Joyner, *SAGE: System for Algebra and Geometry Experimentation*, 2005.
[45] G. J. Sussman and J. Wisdom, *Functional Differential Geometry*, Massachusetts Institute of Technology Press, 2013.
[46] H. Takahasi and M. Mori, *Double exponential formulas for numerical integration*, Publications of the Research Institute for Mathematical Sciences, 9 (1974), pp. 721–741.
[47] 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.
[48] 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.
[49] R. Zare, *Angular Momentum: Understanding Spatial Aspects in Chemistry and Physics*, Wiley, 1991.
[50] 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.

9.1. Limits: The Gruntz Algorithm. SymPy calculates limits using the Gruntz algorithm, as described in [25]. 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:

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

And then operations $<$, $>$ 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 is greater than any power of 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). 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 following 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.

Next step is to rewrite the mrv 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 $x \to \infty$ it is $\omega \to 0$ 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, as well as further details are given in Gruntz's Ph.D. thesis [25].

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 object of the `Basic` 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[1] 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[12] Method. 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:

```
>>> 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[26]. 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 it just computes $a0$, an, bn using standard integration formulas.

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

```
>>> L = symbols('L')
>>> f = fourier_series(2 * (Heaviside(x/L) - Heaviside(x/L - 1)) - 1, (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 `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 `logic` module.

9.3.1. Constructing boolean expressions. A boolean variable can be declared as a SymPy symbol. Python operators &, | and ~ are overloaded for logical `And`, `Or` and `negate`. Several others like `Xor`, `Implies` can be constructed with ˆ, » respectively. The above are just a shorthand, expressions can also be constructed by directly calling `And()`, `Or()`, `Not()`, `Xor()`, `Nand()`, `Nor()`, etc.

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> 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 permits to check if a boolean expression is in any of the above mentioned forms.

```
>>> from sympy.logic.boolalg import is_dnf, is_cnf
>>> x, y, z = symbols('x y z')
>>> 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 on it. Equivalence of two expressions can also be checked. If so, it is possible to return the mapping of variables of two expressions so as to represent the same logical behaviour.

```
>>> from sympy import *
>>> a, b, c, x, y, z = symbols('a b c x y z')
>>> 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 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 watch literal scheme and VSIDS heuristic[32].

```
>>> from sympy import *
>>> a, b, c = symbols('a b c')
>>> 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, $f(x_1, x_2, \dots x_n) = 0$ where $n \geq 2$ and $x_1, x_2, \dots x_n$ are integer variables. If we can find n integers $a_1, a_2, \dots a_n$ such that $x_1 = a_1, x_2 = a_2, \dots x_n = a_n$ satisfies the above equation, we say that the equation is solvable.

Currently, following five types of Diophantine equations can be solved using SymPy's Diophantine module.

- 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 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. Following examples illustrate some of the basic functionalities of the Diophantine module.

```
>>> from sympy import symbols
>>> x, y, z = symbols("x, y, z", integer=True)

>>> 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)])

>>> from sympy.abc import a, b, c, d, e, f
>>> diophantine(9*a**2 + 16*b**2 + c**2 + 49*d**2 + 4*e**2 - 25*f**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(a**2 + b**2 + c**2 + d**2 + e**2 + f**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 to 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 themselves.
- `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 whole numbers, 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. Range 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 Set Difference operations, we have 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 we simply take the Cartesian product of three real sets.
- `ImageSet` represents the image of a function when applied to a particular set. In notation, 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 subset of a set whose members satisfies a particular condition. In notation, 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`, $(-\infty, \infty)$. `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 basic version of the module for dealing with categories — abstract mathematical objects representing classes of structures as classes of objects (points) and morphisms (arrows) between the objects. This version of 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.

At the time of writing of this paper, the version in the `master` branch only implements the first goal, while a (very 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 `categories` defines several classes representing some of the essential concepts: objects, morphisms, categories, diagrams. Since in category theory the inner structure of its objects is often discarded (in the favour of studying the properties of morphisms), 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, except for `CompositeMorphism`, which essentially stores a list of morphisms. To capture the properties of morphisms, the class `Diagram` is expected to be used. 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 mophisms 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 in 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 3.

As far as the second main goal of the module is concerned, a (non-working) draft of an implementation is in https://github.com/scolobb/sympy/tree/ct4-commutativity. The principal idea consists in automatically deciding whether a diagram is commutative or not, given a collection of "axioms" — diagrams *known* to be commutative. The approach to implementation is based on graph embeddings (injective maps): whenever an embedding of a commutative diagram into a given diagram is found, one concludes that that 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

$$
\begin{tikzcd}
A \arrow[loop above, "l_A"] \arrow[loop below, "n_A"] \arrow[r, "f"'] \arrow[rr, bend left, "h_2"] & B \arrow[d, "g"] & D \arrow[l, "k"] \arrow[ll, bend right, "h"'] \arrow[ll, bend right=60, "h_1"'] \arrow[loop above, "l_D"] \arrow[loop below, "n_D"] \\
 & C \arrow[loop left, "l_C"] \arrow[loop right, "n_C"] &
\end{tikzcd}
$$

Fig. 3: An automatically typeset commutative diagram

better optimised version is therefore in order, as well as application of heuristics.

Contributions to automatic inference of commutativity of diagrams are welcome. The source code (both the one in master and in `ct4-commutativity`) is extensively documented. Even more extensive explanations (including some literary chatter) are given in https://scolobb.wordpress.com/.

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.

- It displays integration steps, differentiation steps in detail, which can be viewed in Figure 4:

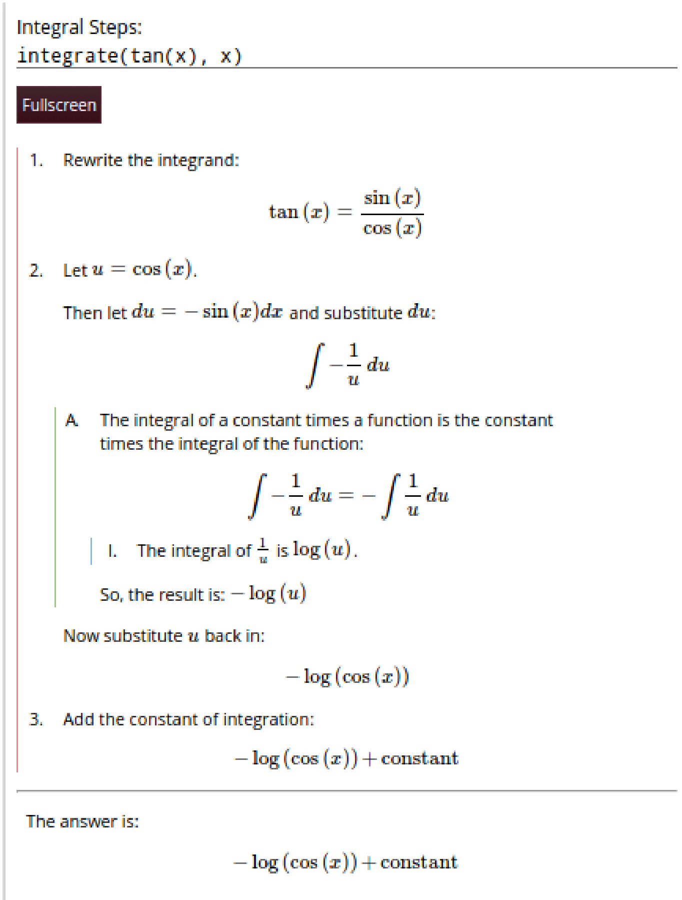

Fig. 4: Integral steps of $\tan(x)$

- It also displays the factor tree diagrams for different numbers.
- SymPy Gamma also 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 doesn't support. For instance, SymPy Gamma supports queries like `sin x`, whereas SymPy doesn't support this, and supports 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 formatted in LaTeX and displayed on the SymPy Gamma web-application.

9.8. SymPy Live. SymPy Live is an online Python shell, which runs on Google App Engine, that executes SymPy code. It is integrated in the SymPy documentation examples, located at this link.

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

Certain Features of SymPy Live:

- It supports the exact same syntax as SymPy, hence it can be used easily, to test for outputs of various SymPy expressions.
- It can be run as a standalone app or in an existing app as an admin-only handler, and can also be used for system administration tasks, as an interactive way to try out APIs, or as a debugging aid during development.
- It can also be used to plot figures (link), and execute all kinds of expressions that SymPy can evaluate.
- SymPy Live also formats the output in LaTeX for pretty-printing the output.

9.9. Comparison with Mathematica. Wolfram Mathematica is a popular proprietary CAS. It features highly advanced algorithms. Mathematica has a core implemented in C++ [6] which interprets its own programming language (know as Wolfram language).

Analogously to Lisp's S-expressions, Mathematica uses its own style of M-expressions, which are arrays of either atoms or other M-expression. The first element of the expression identifies the type of the expression and is indexed by zero, whereas the first argument is indexed by one. Notice that 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's expressions are mutable, that is one can change parts of the expression tree without the need of creating a new object. The reactivity of Mathematica allows for a lazy updating of any references to that data structure.

Products in Mathematica are determined by some builtin node types, such as `Times`, `Dot`, and others. `Times` is overloaded by the * operator, and is always meant to represent a commutative operator. The other notable product is `Dot`, overloaded by the `.` operator. This product represents matrix multiplication, it is not commutative. SymPy uses the same node for both scalar and matrix multiplication, the only exception being with abstract matrix symbols. Unlike Mathematica, SymPy determines commutativity with respect to multiplication from the factor's expression type. Mathematica puts the `Orderless` attribute on the expression type.

Regarding associative expressions, SymPy handles associativity by making associative expressions inherit the class `AssocOp`, while Mathematica specifies the `Flat`[2] 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 matching generating an expression tree transformation on input expressions. Mathematica's pattern matching is sensitive to associative[2], commutative[3], and one-identity[4] properties of its expression tree nodes[5]. 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, one may wish to keep this identity unevaluated, nevertheless it clearly illustrates the potentiality to define one's own 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.[5] The way SymPy deals with extending the missing runtime overloadability functionality is by subclassing the node types. Subclasses may overload the class constructor to yield the proper extended functionality.

Unlike SymPy, Mathematica does not support type inheritance or polymorphism [18]. 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. Associativity of expressions can be achieved by inheriting the class `AssocOp`, which may appear a more cumbersome operation than Mathematica's attribute setting.

Matrices in SymPy are types on their own. 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 elementwise product, this is compatible with Mathematica's convention of commutativity of `Times` nodes. Matrix product is 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 elementwise exponentiation of its elements. The real matrix exponentiationl is available through the `MatrixExp` function.

Unevaluated expressions 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, or put the input expression in a string.

The operator == returns a boolean whenever it is able to immediately evaluate the truthness 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 an equality in SymPy it is necessary to explicitly construct the `Equality` class.

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

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

9.10. Other Projects that use SymPy. There are several projects that use SymPy as a library for implementing a part of their project, or even as a part of back-end for their application as well.
Some of them are listed below:

- **Cadabra**: Cadabra is a symbolic computer algebra system (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 Computer Algebra System 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 uses Artificial Intelligence to detect 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.
- **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 9.11.1.
- **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.

9.11. Project Details. Below we provide particular examples of SymPy use in some of the projects listed above.

9.11.1. SfePy. **SfePy** (Simple finite elements in Python), cf. [17]. is a Python package for solving partial differential equations (PDEs) in 1D, 2D and 3D by the finite element (FE) method [50]. SymPy is used within this package mostly for code generation and testing, namely:

- generation of the hierarchical FE basis module, involving generation and symbolic differentiation of 1D Legendre and Lobatto polynomials, constructing the FE basis polynomials [43] and generating the C code;
- generation of symbolic conversion formulas for various groups of elastic constants [22] – 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. [7]) by generating polynomials of suitable orders, integrating them, and comparing the results with those obtained by the numerical quadrature.

9.12. Tensors. Ongoing work to provide the capabilities of tensor computer algebra has so far produced the `tensor` module. It is composed of three separated submodules, whose purposes are quite different: `tensor.indexed` and `tensor.indexed_methods` support indexed symbols, `tensor.array` contains facilities to operator on symbolic N-dimensional arrays and finally `tensor.tensor` is used to defineabstract tensors. The abstract tensors subsection is inspired by xAct[31] and Cadabra[36]. Canonicalization based on the Butler-Portugal[30] algorithm is supported in SymPy. It is currently limited to polynomial tensor expressions.