Open Source Computer Algebra Systems

SymPy

David Joyner*

2011-10-31

This survey[1] will look at SymPy, a free and open source computer algebra system written entirely in Python, available from `http://sympy.org`. SymPy is licensed under the "modified BSD" license, as is its beautiful logo designed by Fredrik Johansson.

SymPy is also used in other mathematics software systems. For example, SfePy (`http://sfepy.org/`) includes SymPy. SfePy is a software for solving systems of coupled partial differential equations (PDEs) by the finite element

*Address: Mathematics Department US Naval Academy, Annapolis, MD 21402, USA, wdj@usna.edu

[1]This paper is licensed under the Creative Commons attribution license cc-by-sa `http://creativecommons.org/licenses/by/3.0/us/`, or the Free Software Foundations GFDL `http://www.fsf.org/licensing/licenses/fdl.html` (your choice).

method in 2D and 3D. The software Sage (`http://www.sagemath.org/`) also includes SymPy.

SymPy's latest release (as of October 2011) is 0.7.1.

1 SymPy's history and language

> Some people say, that GPL had its place in history. Well, maybe it had, maybe not, it's hard to say - let's leave it for historians. But today in 2011, I think that there is no advantage of GPL over BSD. The only thing that matters is to have a solid community, so that lots of people contribute to the project. Also it seems that these days, there are a lot of people who come to open-source, who don't really feel strongly about licensing, they just want their code to be used (no matter how), and thus use BSD.
>
> - *Ondřej Čertík*

1.1 History

SymPy was started by Ondřej Čertík in 2005 while an physics undergraduate at Charles University in Prague. He wrote some code during the summer, then he wrote some more code during the summer 2006. In February 2007, Fabian Pedregosa joined the project and helped fixed many things, contributed documentation and made it alive again.

Many students improved SymPy incredibly as part of the Google Summer of Code (GSoC):

- GSoC2007: Mateusz Paprocki, Brian Jorgensen, Jason Gedge, Robert Schwarz and Chris Wu,
- GSoC2008: Fredrik Johansson,
- GSoC2009: Priit Laes, Freddie Witherden, Aaron Meurer, Fabian Pedregosa, Dale Peterson,
- GSoC2010: Addison Cugini, Christian Muise, Aaron Meurer, Matthew Curry, Øyvind Jensen,

- GSoC2011: Tom Bachmann, Gilbert Gede, Tomo Lazovich, Saptarshi Mandal, Sherjil Ozair, Vladimir Perić, Matthew Rocklin, Sean Vig, Jeremias Yehdegho.

Of course, SymPy is a team project and it was developed by a lot of people, not just GSoC students. For example, Pearu Peterson joined the development during the summer 2007 and he has made SymPy much more competitive by rewriting the core from scratch, that has made it from 10x to 100x faster.

On Jan 4, 2011 the project leadership was passed to Aaron Meurer.

1.2 Active developers

SymPy is a project with an active community of over 100 developers contributing to it. However, the most active as of this writing (October 2011) are Aaron Meurer, Ondřej Čertík , Brian Granger, Matuesz Paprocki, Ronan Lamy, and Chris Smith.

1.3 Language

> To me, there are three main advantages of SymPy. First, is that it is very portable, since it is written in pure Python with no dependencies. Second is that it is built on top of a very strong and easy to use language, Python, which is a huge improvement over every other computer algebra system, especially the ones that have invented their own language. Third, since it's just a Python module, it can very easily be used as a library.
>
> - *Aaron Meurer*

There are even versions of SymPy for smartphones! In addition, besides the fact that it is free, one of the best advantages to SymPy is how *easy* it is to install. From a teacher's perspective, this is important.

SymPy runs inside a normal Python interpreter, such as the one that comes with your system's Python or IPython. It is started by executing the Python script `isympy` found within SymPy's bin subdirectory. This starts a normal Python shell (IPython shell if you have the IPython package installed), that (pre)executes the following commands:

Python

```
>>> from __future__ import division
```

```
>>> from sympy import *
>>> x, y, z, t = symbols('x y z t')
>>> k, m, n = symbols('k m n', integer=True)
>>> f, g, h = symbols('f g h', cls=Function)
```

So starting `isympy` is equivalent to starting Python (or IPython) and executing the above commands by hand. For example, the following simple Python commands were run from the command line after starting `isympy`.

SymPy

```
>>> L = [i^2 for i in range(10) if i%2==0]
>>> L; len(L)
[2, 0, 6, 4, 10]
5
```

The following example was taken from Mateusz Paprocki's master's degree thesis [P].

Example 1 Suppose we are in possession of American coins: pennies, nickels, dimes and quarters. We would like to compose a certain quantity out of those coins, say 117, such that the number of coins used is minimal. Let's forget about the minimality criterion for a moment. In this scenario it is not a big problem to compose the requested value. We can simply take 117 pennies and we are done, as long as we have so many of them. Alternatively we can take 10 dimes, 3 nickels and 2 pennies, or 2 quarters, 3 dimes, 5 nickels and 12 pennies, etc. There are quite a few combinations that can be generated to get the desired value. But which of those combinations leads to the minimal number of necessary coins? To answer this question we will take advantage of Gröbner bases computed with respect to a total degree ordering of monomials.

Our problem is a minimisation problem, so we take advantage of total degree ordering. This is a correct choice because total degree ordering takes monomials with smaller sums of exponents first and we can observe that the smaller the sum of exponents in a solution to our coins problem will be, the less coins will be needed. So, the chosen ordering of monomials encodes the cost function of our problem.

How to get the minimal number of required coins? Suppose we take any admissible solution to the studied problem. This can be the trivial solution in which we take 117 pennies or any other such that the total value of coins

is equal to 117. We encode the chosen solution as a binomial with numbers of particular coins as exponents of p, n, d and q, and we reduce this binomial with respect to the Gröbner basis G utilizing graded lexicographic ordering of monomials.

SymPy

```
>>> var('p, n, d, q')
(p, n, d, q)
>>> F = [p**5 - n, p**10 - d, p**25 - q]
>>> G = groebner(F, order='grlex')
>>> reduced(p**117, G, order='grlex')[1]
     2  4
d n p  q
```

The answer, that we were able to compute with SymPy, is 1 dime, 1 nickel, 2 pennies, and 4 quarters. which altogether give the requested value of 117. This is also the minimal solution to our problem.

2 Documentation and capabilities

2.1 Basics

A great deal of information about SymPy is available on its website.

- Internet:
  Website :
  `http://sympy.org/`
- Documentation:
  On-line reference manual:
  `http://docs.sympy.org/`
  `http://docs.sympy.org/0.7.1/tutorial.html`
  `https://github.com/sympy/sympy/wiki`
- *Talks, blogs, papers, and lecture notes*:
  `http://planet.sympy.org/`
  `https://github.com/sympy/sympy/wiki/SymPy-Papers`
- Interfaces:
  Command line

There are very useful command-line history, tab-completion, and command-line help features available.
Online version:
`http://live.sympy.org/`
The Sage notebook interface (available in any Sage installation, available from `http://www.sagemath.org/`, or on-line at: `http://www.sagenb.org/` (select `sympy` in the drop-down menu).

- Availability:
  Source code and binaries: `http://sympy.org/download.html`

  SymPy is easy to install on all operating systems that support Python (e.g., MS Windows, Mac OSs, All flavors of Linux, and so on).

  There is also a version for the iPhone which is available separately from the iTunes online store (look for "Python Math" or go to `http://sabonrai.com/wp/pythonmath/`).

  For the smartphone OS "Maemo", there is a Sympy app described at `http://www.robertocolistete.net/Integral/`.

- *Support*:
  There is an email list for SymPy support and development:
  `http://groups.google.com/group/sympy?pli=1`
  `sympy@googlegroups.com`

- *License*:
  Modified BSD.

2.2 Capabilities

Currently, SymPy core has extensive computational capabilities including:

- basic arithmetics $*, /, +, -, **$,
- basic simplification (like $a*b*b+2*b*a*b \mapsto 3*a*b^2$),
- expansion (like $(a+b)^2 \mapsto a^2+2*a*b+b^2$),
- functions ($\exp, \ln, \ldots$),
- complex numbers (like exp(I*x).expand(complex=True) $\mapsto \cos(x)+I*\sin(x)$),

- differentiation,
- taylor (laurent) series,
- substitution (like $x \mapsto \ln(x)$, or $\sin \mapsto \cos$),
- arbitrary precision integers, rationals and floats,
- noncommutative symbols,
- pattern matching .

Then there are many SymPy modules, such as (for example) for these tasks:

- more functions ($\sin, \cos, \tan$, atan, asin, acos, factorial, zeta, legendre),
- limits (like $\text{limit}(x * \log(x), x, 0) \to 0$),
- integration using extended Risch-Norman heuristic,
- polynomials (division, gcd, square free decomposition, groebner bases, factorization),
- solvers (algebraic, difference and differential equations, and systems of equations),
- symbolic matrices (determinants, LU decomposition...),
- Pauli and Dirac algebra,
- geometry module,
- plotting (2D and 3D) .

2.2.1 Quantum physics

Thanks mostly to work done by physicist Brian Granger and students he has directed, SymPy has a lot of functionality in quantum physics implemented. I quote (by permission) from an email he sent to the author:

> Many problems in quantum mechanics require extensive symbolic manipulations. These manipulations use the various mathematical entities of quantum mechanics: hilbert spaces, operators, inner/outer/tensor products, commutators/anticommutators, states, and so on. To make these symbolic manipulations possibly in SymPy, we have implemented these entities in a completely general and abstract manner. By using subclassing, we can treat particular quantum mechanical systems.
>
> The most important work we have done on this front is to create classes for simulating quantum computers and other quantum information problems. We are now using that framework to build optimization algorithms for quantum computers. Being able to handle everything symbolically is massively important for us, as the alternative requires constructing and multiplying extremely large matrices.
>
> It turns out that the object-oriented nature of SymPy/Python has been really powerful for us and is a very nice way of abstracting a general theory (using base classes) and particular realizations of that theory in physical systems (using subclasses).
>
> We have also implemented a number of other physical systems as well, but none as significant as the quantum computing stuff.
>
> - *Brian Granger*

For further details, see for example the papers by Cugini [Cug] and Curry [Cur], or blog posts on quantum mechanics on Planet SymPy (`http://planet.sympy.org/`).

2.2.2 Packages

To plot a function in SymPy you need to install a separate plotting package, such as matplotlib or pyglet. To install them into your local Python install, follow the instructions on their website (that is, `http://www.pyglet.org/` or `http://matplotlib.sourceforge.net/`).

2.2.3 Examples

This section merely presents a few examples to illustrate some of SymPy's functionality.

To plot a function in 2d, such as $y = \sin(x)$, use the following SymPy commands.

SymPy

```
>>> x = Symbol("x")
>>> Plot(sin(x), [x, -3, 3])
[0]: sin(x), 'mode=cartesian'
```

A separate window will pop-up with your (color) plot in it. See Figure 1.

To plot a surface in 3d, such as $z = xy^3 - yx^3$, use the following SymPy commands.

SymPy

```
>>> var('x y z')
>>> Plot(x*y**3-y*x**3)
```

A separate window will pop-up with your (color) plot in it. See Figure 1.

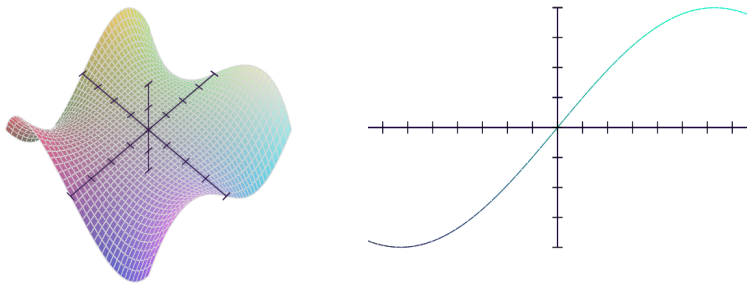

Figure 1: 2D and 3D plots using SymPy

SymPy has good functionality with linear algebra. The method `rref` returns the row-reduced echelon form of a matrix, as well as a list of the pivot columns.

SymPy

```
>>> M = Matrix(3,3,lambda i,j: i+j)
>>> print M
[0, 1, 2]
[1, 2, 3]
[2, 3, 4]
```

```
>>> print M.rref()
([1, 0, -1]
[0, 1,  2]
[0, 0,  0], [0, 1])
```

Indeed, there is a pivot in the first two columns but not the third, so SymPy returns only $[0, 1]$, as it should.

SymPy

```
>>> M = Matrix([[1,1,2],[1,2,1],[2,1,1]])
>>> print M
[1, 1, 2]
[1, 2, 1]
[2, 1, 1]
>>> print M.eigenvals()
{1: 1, -1: 1, 4: 1}
>>> print M.eigenvects()
[(1, 1, [[ 1]
[-2]
[ 1]]), (-1, 1, [[-1]
[ 0]
[ 1]]), (4, 1, [[1]
[1]
[1]])]
>>> print M.charpoly(x)
Poly(x**3 - 4*x**2 - x + 4, x, domain='ZZ')

>>> lam = M.eigenvals()
>>> lam.keys()
[1, -1, 4]
>>> lamb = lam.keys()
>>> expand((x-lamb[0])*(x-lamb[1])*(x-lamb[2]))
 3      2
x  - 4*x  - x + 4
```

In the last command, we merely verify that the characteristic polynomial is indeed a polynomial whose roots are the eigenvalues.

Thanks mostly to the work of Aaron Meurer, SymPy can solve higher order linear ordinary differential equations. See Meuer [M] for details of the implementation. To solve the 3-rd order constant coefficient ordinary differential equation,

$$y''' - 3y'' + 3y' - y = 10e^x,$$

use the following SymPy commands.

SymPy

```
>>> x = Symbol("x")
>>> y = Function('y')
>>> de = y(x).diff(x,3)-3*y(x).diff(x,2)+3*y(x).diff(x)-y(x)-6*exp(x)
>>> soln = dsolve(de, y(x))
>>> print soln
y(x) == (C1 + C2*x + C3*x**2 + x**3)*exp(x)
```

Surprisingly, it seems Maxima cannot do this at the present time.

SymPy handles easily a wide range of calculus computations, as the following commands illustrate.

SymPy

```
>>> print legendre(8, x).diff(x)
6435*x**7/16 - 9009*x**5/16 + 3465*x**3/16 - 315*x/16
>>> print integrate(legendre(8, x),x)
715*x**9/128 - 429*x**7/32 + 693*x**5/64 - 105*x**3/32 + 35*x/128
>>> j0 = lambda x: besselj(0,x)
>>> limit(j0(x),x,0)
besselj(0, 0)
>>> N(limit(j0(x),x,0))
1.00000000000000
```

SymPy handles Taylor series expansions with no problem.

SymPy

```
>>> x = Symbol('x')
>>> f = 1/cos(x)
>>> print f.series(x, 0, 10)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 + O(x**10)
```

Did you know SymPy can solve a linear recurrence sequence? For example, consider

$$u_{n+2} = 2u_{n+1} + 8u_n,$$

where we know $u_0 = 2$ and $u_1 = 7$.

SymPy

```
>>> n = Symbol('n', integer=True)
>>> u = Function('u')
>>> f=u(n+2)-2*u(n+1)-8*u(n)
>>> print rsolve(f, u(n), {u(0):2, u(1):7})
(-2)**n/6 + 11*4**n/6
```

In short, if you need to use a free mathematics program in teaching or research, and one which is easily installed on various platforms, SymPy is a great place to start. If you get stuck, there is plenty of information online, or you can join the SymPy googlegroups and email your question to `sympy@googlegroups.com`.

Acknowledgements: I thank Ondřej Čertík , Aaron Meurer, Mateusz Paprocki, Vladimir Perić and Brian Granger for helpful comments and corrections. (Any remaining errors are of course my responsibility). Some of the material above is taken with only slight modification from Paprocki's paper [P], the tutorial [PM], or the documentation at the official SymPy website [S].

References

[Cug] Addison Cugini, *Quantum Mechanics, Quantum Computation, and the Density Operator in SymPy*, preprint date 06/12/2011
`http://digitalcommons.calpoly.edu/physsp/38`

[Cur] Matthew Curry, *Symbolic Quantum Circuit Simplification in SymPy* preprint date June 6, 2011.
`http://digitalcommons.calpoly.edu/physsp/39`

[M] A. Meurer, *Variation of Parameters and More*, blog post available at
`http://asmeurersympy.wordpress.com/2009/08/01/variation-of-parameters-and-more/`.

[P] Mateusz Paprocki, *Design and Implementation Issues of a Computer Algebra System in an Interpreted, Dynamically Typed Programming Language*, Master's Thesis, 2010, University of Technology of Wroclaw, Poland.
`https://github.com/mattpap/masters-thesis`
`http://mattpap.github.com/masters-thesis/html/index.html`

[PM] —— and Aaron Meurer, *Guide to symbolic mathematics with SymPy*, SciPy conference 2011 presentation,
`http://mattpap.github.com/scipy-2011-tutorial/html/mathematics.html`

[S] SymPy website
http://www.sympy.org/