SymPy - Python library for symbolic mathematics

Ondřej Čertík

Institute of Physics, Academy of Sciences of the Czech Republic

February 29, 2008

Introduction

Contens of this talk:

- Review of the current state:
  - History
  - Different approaches to symbolic manipulation we tried
  - Symbolic limits
  - Integration with SAGE
- Future
  - where to go from here
  - our priorities and principles

SymPy

- A Python library for symbolic mathematics
- http://code.google.com/p/sympy/

```
>>> from sympy import Symbol, limit, sin, oo
>>> x=Symbol("x")
>>> limit(sin(x)/x, x, 0)
1
>>> integrate(x+sinh(x), x)
>>> (1/2)*x**2 + cosh(x)
```

SymPy

What SymPy can do

- basics (expansion, complex numbers, differentiation, taylor (laurent) series, substitution, arbitrary precision integers, rationals and floats, pattern matching)
- noncommutative symbols
- limits and some integrals
- polynomials (division, gcd, square free decomposition, groebner bases, factorization)
- symbolic matrices (determinants, LU decomposition...)
- solvers (some algebraic and differential equations)
- 2D geometry module
- plotting (2D and 3D)

Motivation

Why?

Motivation

Why?

- There are people who want to develop it, so it will be developed. :)

Motivation

Why?

- There are people who want to develop it, so it will be developed. :)
- BSD licensed (like SciPy and NumPy) $\rightarrow$ use it how you want
- small, pure python $\rightarrow$ easily include it your own projects
- It's in Debian, Ubuntu, Gentoo, Arch, Sage, ...

SymPy

Other symbolic manipulation software: GiNaC, Giac, Qalculate, Yacas, Eigenmath, Axiom, PARI, Maxima, SAGE, Singular, Mathomatic, Octave, ...

Problems:

- all use their own language (except GiNaC, Giac and SAGE)
- GiNaC and Giac still too complicated (C++), difficult to extend

What we want

- Python library and that's it (no environment, no new language, nothing)
- Rich funcionality
- Pure Python (non Python modules could be optional) – works on Linux, Windows, Mac out of the box

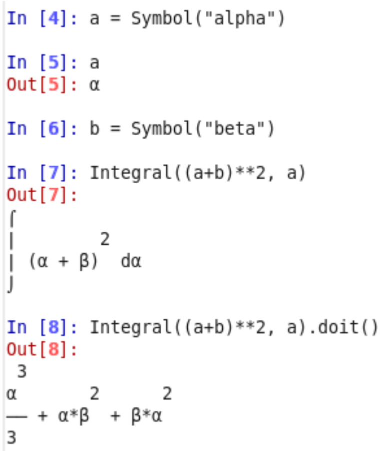
SymPy console

Acutally, I didn't tell the full truth, we have one nice thing – isympy:

```
$ bin/isympy
Python 2.4.4 console for SymPy 0.5.6-hg. These commands wer
>>> from __future__ import division
>>> from sympy import *
>>> x, y, z = symbols('xyz')
>>> k, m, n = symbols('kmn', integer=True)
```

```
In [1]: integrate(ln(x), x)
Out[1]: -x + x*log(x)
```

Unicode prettyprinting



```
In [4]: a = Symbol("alpha")

In [5]: a
Out[5]: α

In [6]: b = Symbol("beta")

In [7]: Integral((a+b)**2, a)
Out[7]:
⌠
⎮        2
⎮ (α + β)  dα
⌡

In [8]: Integral((a+b)**2, a).doit()
Out[8]:
 3
α        2       2
── + α*β  + β*α
3
```

Recent changes in isympy:

- pretty printing by default
- use unicode printing if available

SAGE

- aims to glue together every useful open source mathematics software package and provide a transparent interface to all of them
- http://www.sagemath.org/
- More on relationship between SAGE and SymPy later

```
sage: limit(sin(x)/x, x=0)
1
sage: integrate(x+sinh(x), x)
cosh(x) + x^2/2

In [1]: limit(sin(x)/x, x, 0)
Out[1]: 1

In [2]: integrate(x+sinh(x), x)
Out[2]: (1/2)*x**2 + cosh(x)
```

History I

In 2005, I wanted to use symbolic mathematics in Python

- pyginac used boost-python, very slow compilation (30s per file),
- I wrote swiginac together with Ola Skavhaug in SWIG, it works, but too difficult to extend the GiNaC core behind it
- Is it really that difficult to have a system, that can calculate all I need and still be easy to extend?

Let's reinvent the wheel for the 35th time.

History II

- end of summer 2005: I implemented my first code, mostly translating ideas from GiNaC to Python.
- spring 2006: I discovered the Gruntz algorithm for limits
- end of summer 2006: I implemented limits in SymPy
- February 2007: Fabian Seoane joined and this was the boost to SymPy's development
- Google Summer of Code, SymPy is under the umbrella of Python Software Foundation, the Space Telescope Science Institute and Portland State University

Contributions

- Fabian: everything, without him, SymPy wouldn't be here
- Mateusz (GSoC): concrete math, symbolic integration, many bugfixes
- Jason (GSoC): geometry, a lot of bugfixes
- Robert (GSoC): polynomials (groebner basis et al.)
- Brian (GSoC): plotting
- Chris (GSoC): linear algebra
- Pearu: new core (10x to 100x speedup)
- Fredrik: fast floating point arithmetics in pure Python (faster than Decimal)
- Jurjen: pretty printing
- Kirill: unicode printing, a lot of bugfixes
- others: bug reports, bug fixes

Approaches we tried I

GiNaC ".eval()" approach, without their "ex" class:

- classes: Basic, Add, Mul, Pow, Rational, Funcion (sin, cos, exp, log)

Example:

- $x + y + x \longrightarrow$ Add(Add(Symbol("x"), Symbol("y")), Symbol("x"))

  ```
  e = Add(Add(x, y), x)
  e.eval()
  ```

  "e" becomes Add(Mul(2, x), y)

Disadvantages:

- User has to call ".eval()" by hand
- Wasteful construction of instances

Approaches we tried II

Automatic evaluation of ".eval()":

- classes: Basic, Add, Mul, Pow, Rational, Funcion (sin, cos, exp, log)

Example:

- $x + y + x \rightarrow$ Add(Add(Symbol("x"), Symbol("y")), Symbol("x"))

  ```
  e = Add(Add(x, y), x)
  ```

  "e" becomes Add(Mul(2, x), y) automatically

Disadvantages:

- Wasteful construction of instances

Approaches we tried III

Not using ".eval()" at all, simplify immediatelly in "__new__"

- classes: Basic, Add, Mul, Pow, Rational, Funcion (sin, cos, exp, log)

Example:

- $x + y + x \rightarrow$ Add(Add(Symbol("x"), Symbol("y")), Symbol("x"))

  ```
  e = Add(Add(x, y), x)
  ```

  "e" becomes Add(Mul(2, x), y) immediatelly, no intermediate classes constructed

Approaches we tried IV

How to deal with functions:

- Sin, ApplySin, Cos, ApplyCos, ...
  - one class to represent a function (sin)
  - another class to represent "applied" function (sin(x))
  - SAGE way
- sin, cos, ...
  - just one class to represent a function (sin)
  - instance of this class to represent "applied" function (sin(x))
  - SymPy way

We decided to use the second option. Why?

- all logic is in one class, easy to extend and understand
- the less classes, the better

the Schwarzschild solution in the General Relativity

spherically symmetric metric ($diag(-e^{\nu(r)}, e^{\lambda(r)}, r^2, r^2 \sin^2\theta)$) $\rightarrow$ Christoffel symbols $\rightarrow$ Riemann tensor $\rightarrow$ Einstein equations $\rightarrow$ solver

```
ondra@pc232:~/sympy/examples$ time python relativity.py
...
[SKIP]
...
----------------------------------------
metric:
-C1 - C2/r 0 0 0
0 1/(C1 + C2/r) 0 0
0 0 r**2 0
0 0 0 r**2*sin(\theta)**2

real 0m1.092s
user 0m1.024s
sys 0m0.068s
```

Symbolic limits

- Gruntz algorithm
- the algorithm is so simple that everyone should know how it works :)

Comparability classes

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

We define $<$, $>$, $\sim$:

- $f > g$ when $L = \pm\infty$
  - f is greater than any power of g
  - f is more rapidly varying than g
  - f goes to ∞ or 0 faster than g
- $f < g$ when $L = 0$
  - f is lower than any power of g
  - ...
- $f \sim g$ when $L \neq 0, \pm\infty$
  - both f and g are bounded from above and below by suitable integral powers of the other

Examples:

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

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

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

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

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

The Gruntz algorithm I

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

$$\lim_{x\to\infty} f(x) = ?$$

Strategy:

- mrv set: the set of most rapidly varying subexpressions
  - $\{e^{x}, e^{-x}, e^{x+2e^{-x}}\}$
  - the same comparability class
- take an item ω converging to 0 at infinity
  - $\omega = e^{-x}$
  - if not present in the mrv set, use the relation $f(x) \sim \frac{1}{f(x)}$
- rewrite the mrv set using ω
  - $\{\frac{1}{\omega}, \omega, \frac{1}{\omega}e^{2\omega}\}$
- substitute back in $f(x)$ and expand in ω:
  - $f(x) = \frac{1}{x} - \frac{1}{\omega} + \frac{1}{\omega}e^{2\omega} = 2 + \frac{1}{x} + 2\omega + O(\omega^2)$

The Gruntz algorithm II

Crucial observation: ω is from the mrv set, so

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

- We iterate until we get just a number, the final limit
- Gruntz proved this always works and converges in his Ph.D. thesis

Generally:

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

- we look at the lowest power of ω
- the limit is one of: 0, $\lim_{x\to\infty} C_0(x)$, ∞

Integration with SAGE

From SymPy to SAGE:

- using "_sage_()" methods:

From SAGE to SymPy:

- using "_sympy_()" methods:

Why SymPy in SAGE? Isn't Maxima good enough?

- pure Python
- easily extensible (the main reason I started SymPy), at least we try :)
- small, people can easily use it without SAGE (which is big)
- options are always good

The question of speed

- Being pure Python has many advantages
- speed is good enough for many purposes
- sympycore project tries to speed SymPy even more
- later, when internals of SymPy settle some more, use C++, C or maybe Cython.

Current state and future

Now what?

- Fix bugs (there are still too many)
- Try to make most of the common tasks easy to do:
    - Playing with defined and undefined functions (diff(f(x), x))
    - most of the integrals, limits, differential/algebraic equations should work
- Collaborate with SAGE, implement only things, that are needed

Principles

Linus

Talk is cheap. Show me the code.

- Have something now, not tomorrow
- Strictly following the Zen of Python ("import this" in Python)
- Every single feature in SymPy must have tests
- Main hg version always needs to pass all tests