An Efficient Computer Algebra System for Python

Pearu Peterson

pearu.peterson@gmail.com

Laboratory of Systems Biology, Institute of Cybernetics, Estonia

Simula Research Laboratory, Norway

- Introduction
- Design criteria
- Sympycore architecture
    - Implementation notes
    - Handling infinities
- Performance comparisons
- Conclusions

SIAM Conference on Computational Science and Engineering
March 2-6, 2009
Miami, Florida

1. Introduction

What is CAS?

Computer Algebra System (CAS) is a software program that facilitates symbolic mathematics.

The core functionality of a CAS is manipulation of mathematical expressions in symbolic form.

[Wikipedia]

Our aim — provide a package to manipulate mathematical expressions within a Python program.

Target applications — code generation for numerical applications, arbitrary precision computations, etc in Python.

Existing tools — Wikipedia lists more than 40 CAS-es:

- commercial/free,
- full-featured programs/problem specific libraries,
- in-house, C, C++, Haskell, Java, Lisp, etc programming languages.

from core functionality to a fully-featured system

Possible approaches

- wrap existing CAS libraries to Python:
  swiginac[GPL] (SWIG, GiNaC, 2008), PyGiNaC[GPL] (Boost.Python, GiNaC, 2006), SAGE[GPL] (NTL, Pari/GP, libSingular, etc.)
- create interfaces to CAS programs:
  Pythonica (Mathematica, 2004), SAGE[GPL] (Maxima, Axiom, Maple, Mathematica, MuPAD, etc.)
- write a CAS package from scratch:
  Sympy[BSD], Sympycore[BSD], Pymbolic[?] (2008), PySymbolic[LGPL] (2000), etc

2. Design criteria

Symbolic expressions in silica . . .

- memory efficient representation
- memory and CPU efficient manipulation
- support variety of mathematical concepts — algebraic approach
- extensibility is crucial — from core to fully-featured system
- separate core implementation details from library algorithms

Symbolic expressions

- atomic expressions — symbols, numbers
- composite expressions — unevaluated operations
- multiple representations possible

Representation of symbolic expressions consists of . . .

- Data structures to store operands
- Methods/functions to interpret data structures
- Classes to define algebraic properties

For example, `x * y` can be represented as

```
Ring(MUL, [x, y])
```

or as

```
CommutativeRing(BASE_EXP_DICT, {x: 1, y: 1})
```

3. Sympycore architecture

- Symbolic expressions are instances of **`Algebra`** subclasses.
- An **`Algebra`** instance holds pair of *head* and *data* parts:

  $\langle \textit{Algebra} \rangle$**`(`**$\langle \textit{head part} \rangle$**`,`** $\langle \textit{data part} \rangle$**`)`**

- The $\langle \textit{head} \rangle$ part holds operation methods.
- The $\langle \textit{data} \rangle$ part holds operands.
- The $\langle \textit{Algebra} \rangle$ class defines valid operation methods like **`__mul__`**, **`__add__`**, etc. that apply the corresponding operation methods (in $\langle \textit{head} \rangle$) to operands (in $\langle \textit{data} \rangle$).

3.1. Atomic heads

SYMBOL — data is arbitrary object (usually a string), ⟨*Algebra*⟩ instance represents any element of the corresponding algebraic structure:

```
x = Algebra(SYMBOL, 'x')
```

NUMBER — data is numeric object, ⟨*Algebra*⟩ instance represents a concrete element of the corresponding algebraic structure:

```
r = Algebra(NUMBER, 3.14)
```

3.2. Arithmetic heads

ADD — data is a list of operands to unevaluated addition operation: `Ring(ADD, [x, y]) -> x + y`

MUL — data is a list of operands to unevaluated multiplication operation: `Ring(MUL, [x, y]) -> x * y`

POW — data is a tuple of base and exponent: `Ring(POW, (x, y)) -> x ** y`

TERM_COEFF — data is a tuple of symbolic term and numeric coefficient: `Ring(TERM_COEFF, (x, 2)) -> 2 * x`

TERM_COEFF_DICT — data is a dictionary of term-coefficient pairs: `Ring(TERM_COEFF_DICT, {x: 2, y: 3}) -> 2*x + 3*y`

BASE_EXP_DICT — data is a dictionary of base-exponent pairs:

```
CommutativeRing(BASE_EXP_DICT, {x: 2, y: 3})
-> x**2 * y**3
```

EXP_COEFF_DICT — data contains polynomial symbols and a dictionary of exponents-coefficient pairs:

```
Ring(EXP_COEFF_DICT, Pair((x, y), {(2,0): 3, (5,6): 7}))
 -> 3*x**2 + 7*x**5*y**6
```

3.3. Other heads

NEG, POS, SUB, DIV, MOD — verbatim arithmetic heads:
`Ring(SUB, [x, y, z]) -> x - y - z`

INVERT, BOR, BXOR, BAND, LSHIFT, RSHIFT — binary heads

LT, LE, GT, GE, EQ, NE — relational heads:
`Logic(LT, (x, y)) -> x < y`

NOT, AND, OR, XOR, EQUIV, IMPLIES, IS, IN — logic heads: `Logic(OR, (x, y)) -> x or y`

APPLY, SUBSCRIPT, LAMBDA, ATTR, KWARG — functional heads: `Ring(Apply, (f, (x, y))) -> f(x, y)`

SPECIAL, CALLABLE — special heads

MATRIX — sparse matrix heads

UNION, INTERSECTION, SETMINUS — set heads

TUPLE, LIST, DICT — container heads

. . .

3.4. Algebra classes

```
Expr
  Algebra
    Verbatim
    Ring
      CommutativeRing
        Calculus
        Unit
      FunctionRing
      MatrixRing
    Logic
    Set
    ...
```

3.5. Examples

```
> from sympycore import *
> x,y,z=map(Calculus,'xyz')
> 3*x+y+x/2
Calculus('y + 7/2*x')
> (x+y)**2
Calculus('(y + x)**2')
> ((x+y)**2).expand()
Calculus('2*y*x + x**2 + y**2')
```

```
>>> from sympycore.physics import meter
>>> x*meter+2*meter
Unit(’(x + 2)*m’)

>>> f = Function(’f’)
>>> f+sin
FunctionRing_Calc_to_Calc(’Sin + f’)
>>> (f+sin)(x)
Calculus(’Sin(x) + f(x)’)

>>> m=Matrix([[1,2], [3,4]])
>>> print m.inv() * m
 1  0
 0  1
>>> print m.A * m
 1   4
 9  16

>>> Logic(’x>1 and a and x>1’)
Logic(’a and x>1’)
```

4. Implementation notes

Circular imports — modules implement initialization functions that are called when all subpackages are imported to initialize any module objects

Immutability of composites containing mutable types — ⟨*Expr instance*⟩`.is_writable` — `True` if hash is not computed yet.

```
hash(⟨dict⟩) = hash(frozenset(⟨dict⟩.items()))
hash(⟨list⟩) = hash(tuple(⟨list⟩))
```

Equality tests ⟨*Expr*⟩`.as_lowlevel()` — used in hash computations and in equality tests.

```
>>> Calculus(TERM_COEFF_DICT, {}).as_lowlevel()
0
>>> Calculus(TERM_COEFF_DICT, {x:1}).as_lowlevel()
Calculus(’x’)
>>> Calculus(TERM_COEFF_DICT, {x:1, y:1}).as_lowlevel()
(TERM_COEFF_DICT, {Calculus(’x’): 1, Calculus(’y’): 1})
```

5. Infinity problems

In most computer algebra systems handling infinities is inconsistent:

```
2*x*infinity -> x*infinity
```

but

```
x*infinity  + x*infinity -> 2*x*infinity.
```

```
expand((x + 2)*infinity) -> infinity + x*infinity
```

incorrect if `x=-1`.

5.1. Sympycore Infinity

Sympycore defines `Infinity` object to represent extended numbers such as directional infinities and undefined symbols in a consistent way.

Definition: `Infinity`$(d) = \lim_{r\to\infty}(r \times d),\ d \in \mathbb{C}$

Operations with finite numbers:

`Infinity`$(d) < op > n = \lim_{r\to\infty}(r \times d < op > n)$

Operations with infinite numbers:

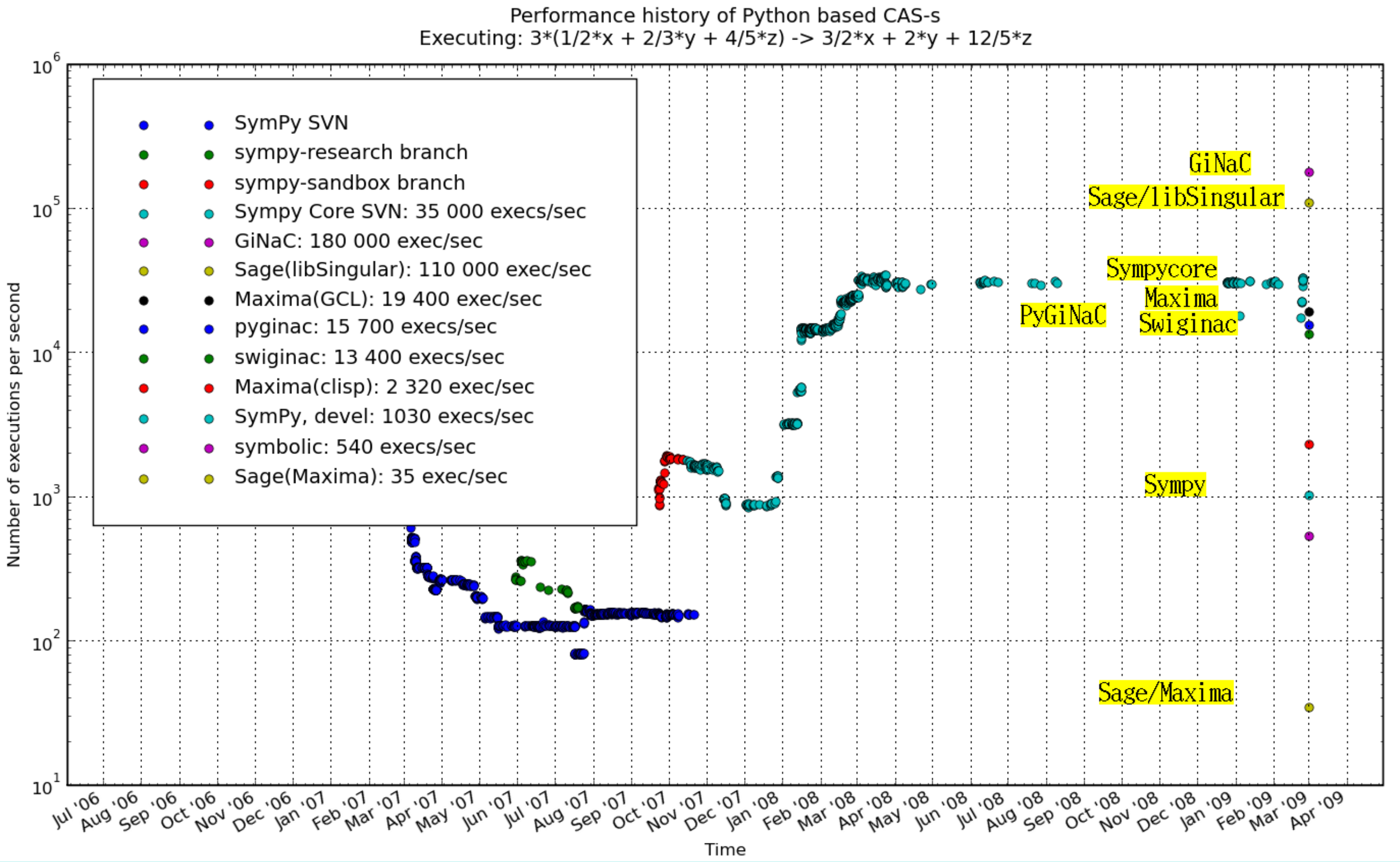
`Infinity`$(d_1) < op >$`Infinity`$(d_2) = \lim_{r_1\to\infty, r_2\to\infty}(r_1 \times d_1 < op > r_2 \times d_2)$

```
>>> oo = Infinity(1)
>>> x*oo - x*oo
Infinity(Calculus('EqualArg(x, -x)*x'))
```

always correctly evaluates to `undefined`=`Infinity`(0).

```
>>> x*oo + y
Infinity(Calculus('x*(1 + (EqualArg(x, y) - 1)*IsUnbounded(y))')
```

Performance comparisons



6. Conclusions

- Sympycore — a research project, its aim is to seek out new high-performance solutions to represent and manipulate symbolic expressions in Python language
- — fastest Python based CAS core implementation
- — uses algebraic approach, supporting various mathematical concepts is equally easy

http://sympycore.google.com
Pearu Peterson
Fredrik Johansson