On providing a CAS for Python

Pearu Peterson
pearu.peterson@gmail.com

Centre for Nonlinear Studies, Estonia
Simula Research Laboratory, Norway

- Introduction — What is CAS? Why another CAS?
- Sympycore — development, comparisons, secret notes
- Conclusions

Abstract

During the last ten years there has been many attempts to provide a Computer Algebra System (CAS) for Python that have important applications in code generation tools, for example. In most cases, one of the following approaches has been proposed: wrap existing CAS libraries to Python, create Python interfaces to existing CAS programs, or implement pure Python CAS from scratch. In this talk I will discuss pros and cons of these approaches as well as try to give an overview of what is the current state with CAS-s for Python. Finally, a pure Python package, sympycore, will be introduced as sufficiently efficient and robust implementation of a CAS for Python. For example, the sympycore speed is comparable with the speed of many CAS-s that are implemented using a compiled language.

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]

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.

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.

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

It is almost trivial to implement a simple and efficient Python program for manipulating symbolic expressions but highly non-trivial to generalize it to a full-featured and sufficiently efficient CAS

Some minimal set of CAS features

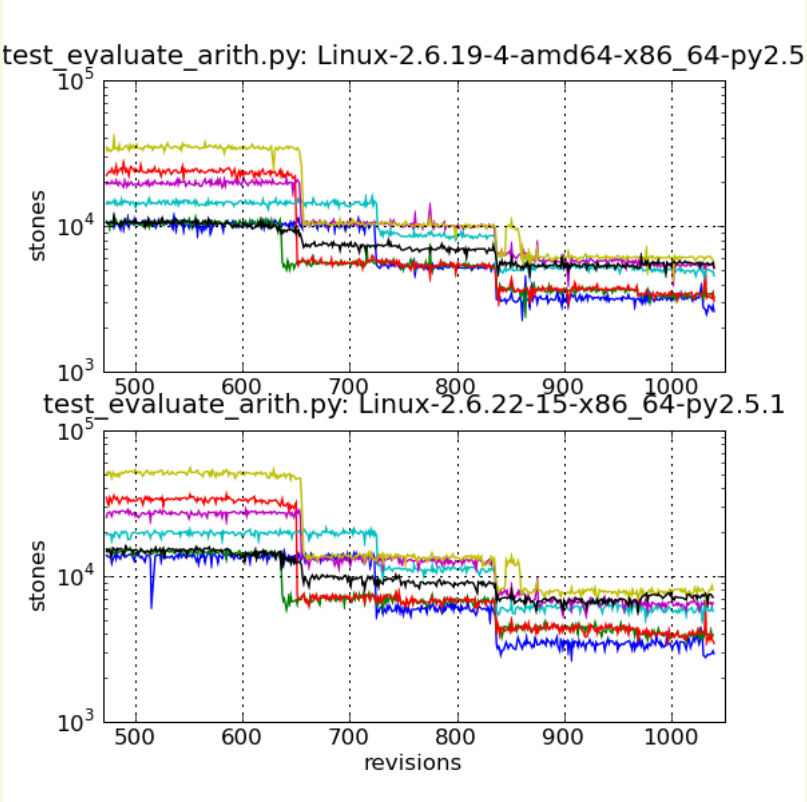
- Symbolic expressions: atomic (symbols, numbers) and composite (operation + operands) expressions, inquire information.
- Pattern matching, substitutions of sub-expressions.
- Support for various mathematical concepts: arithmetics, calculus, polynomials, matrices, sets, logic, functions, operators, etc.
- Simplifications to some canonical form:
  `(x + x) -> 2*x`, `(x and x) -> x`, etc.
- Changes in the form of expressions: expanding, factorizations, rewriting.
- Simplifications with assumptions:
  `sqrt(x)**2 -> x` iff `x>0 is True`.
- Calculus: differentation, integration, series, limits, etc.
- Arithmetics: booleans, integers, rational numbers, complex numbers, arbitrary precision numbers
- Convert symbolic expressions to various forms: C/C++, Fortran, TeX, pretty-printing, etc.
- ...

A selection of available Python based CAS-es:

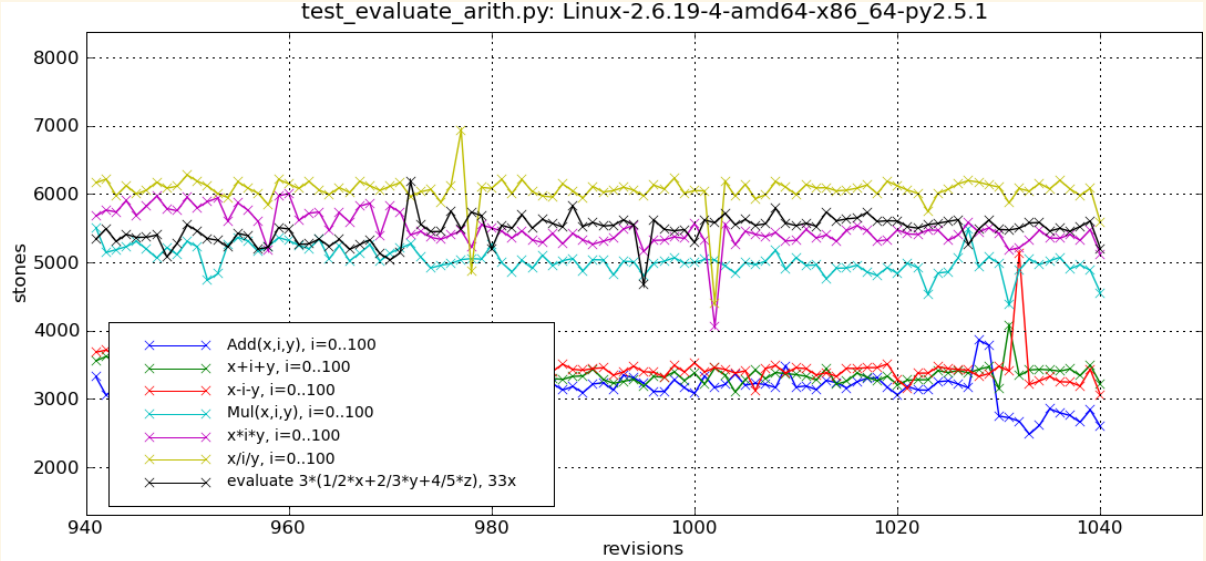
CAS	Advanatages	Disadvantages
Sage	most full-featured, lots of developers (20–50), fastest components	huge (≥ 800MB), ships more than one needs, slow components(read: interfaces)
Sympy	pure Python, many developers (8–15), many features	slow, core robustness needs work, calculus oriented
Sympycore	fastest pure Python core (10–300x faster than Sympy), facilitates various algebras	not many features implemented, few developers (2)

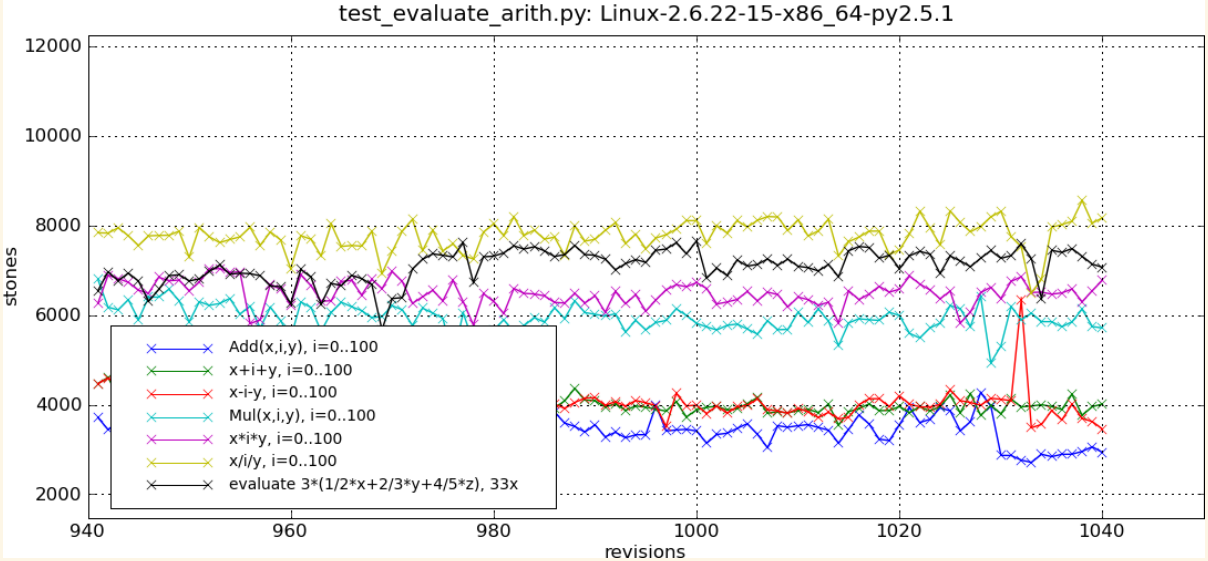
Sympycore — development

- About 10 000 LOC
- Test coverage is currently 65%
- Continuous control over performance



- Authors: Pearu Peterson, Fredrik Johansson


test_evaluate_arith.py: Linux-2.6.19-4-amd64-x86_64-py2.5.1
stones
revisions
8000
7000
6000
5000
4000
3000
2000
940
960
980
1000
1020
1040
Add(x,i,y), i=0..100
x+i+y, i=0..100
x-i-y, i=0..100
Mul(x,i,y), i=0..100
x*i*y, i=0..100
x/i/y, i=0..100
evaluate 3*(1/2*x+2/3*y+4/5*z), 33x


test_evaluate_arith.py: Linux-2.6.22-15-x86_64-py2.5.1
stones
revisions
12000
10000
8000
6000
4000
2000
940
960
980
1000
1020
1040
Add(x,i,y), i=0..100
x+i+y, i=0..100
x-i-y, i=0..100
Mul(x,i,y), i=0..100
x*i*y, i=0..100
x/i/y, i=0..100
evaluate 3*(1/2*x+2/3*y+4/5*z), 33x

Sympycore versus Sympy and Swiginac

Executed code	sympy / sympycore
`Add(x,random integer,y)`, 2000x	39.0
`Mul(x,random integer,y)`, 2000x	26.4
`sum(x**i/i,i=1..400)`	317
`expand((x+z+y)**20 * (z+x)**9)`	126
`expand((x+2*y+3*z)**20).subs(x,z).subs(y,z)`	32.2
`f=(x/(1+sin(x**(y+x**2)))**2)`, `f=f.diff(x)`, 5x	42.3
`f=4*x**3+y**2*x**2+x+1`, `f=integrate(f)`, 10x	20.8
`match((x*y**2)**sin(x+y**2), (v*w)**sin(v+w))`	9.87
`polynomial division P(1,2,3,4,5)/Q(1,2,3/4)`	333
`fem_test.py`	16.5
	swiginac / sympycore
`Add(x,random integer,y)`, 2000x	11.2
`Mul(x,random integer,y)`, 2000x	4.55
`sum(x**i/i,i=1..400)`	48.5
`expand((x+z+y)**20 * (z+x)**9)`	0.700
`expand((x+2*y+3*z)**20).subs(x,z).subs(y,z)`	0.282
`f=(x/(1+sin(x**(y+x**2)))**2)`, `f=f.diff(x)`, 5x	0.633

- Sympy has lots of features, uses caching
- Sympycore concentrates on core performance, one small extension module
- Swiginac has less features, computation is fast (GiNaC), interface is slow (SWIG)

Sympycore secret notes — implementation

Things to keep in mind:

- object creation in Python is slow
- function/method calls in Python are slow
- attribute access in Python is slow
- loops in Python are slow
- working with builtin types in Python can be fast (C speed)

Things to follow:

- avoid creating temporary objects, use singletons (`is`)
- avoid deep function calls, inline functions
- use local variables, remember attributes and global names
- avoid loops in Python
- take advantage of the speed of Python builtin types (`dict`, `set`, `list`, `tuple`)
- use profile results to decide which parts should be written in C
- with immutable objects: true inplace operations possible until hash calculation

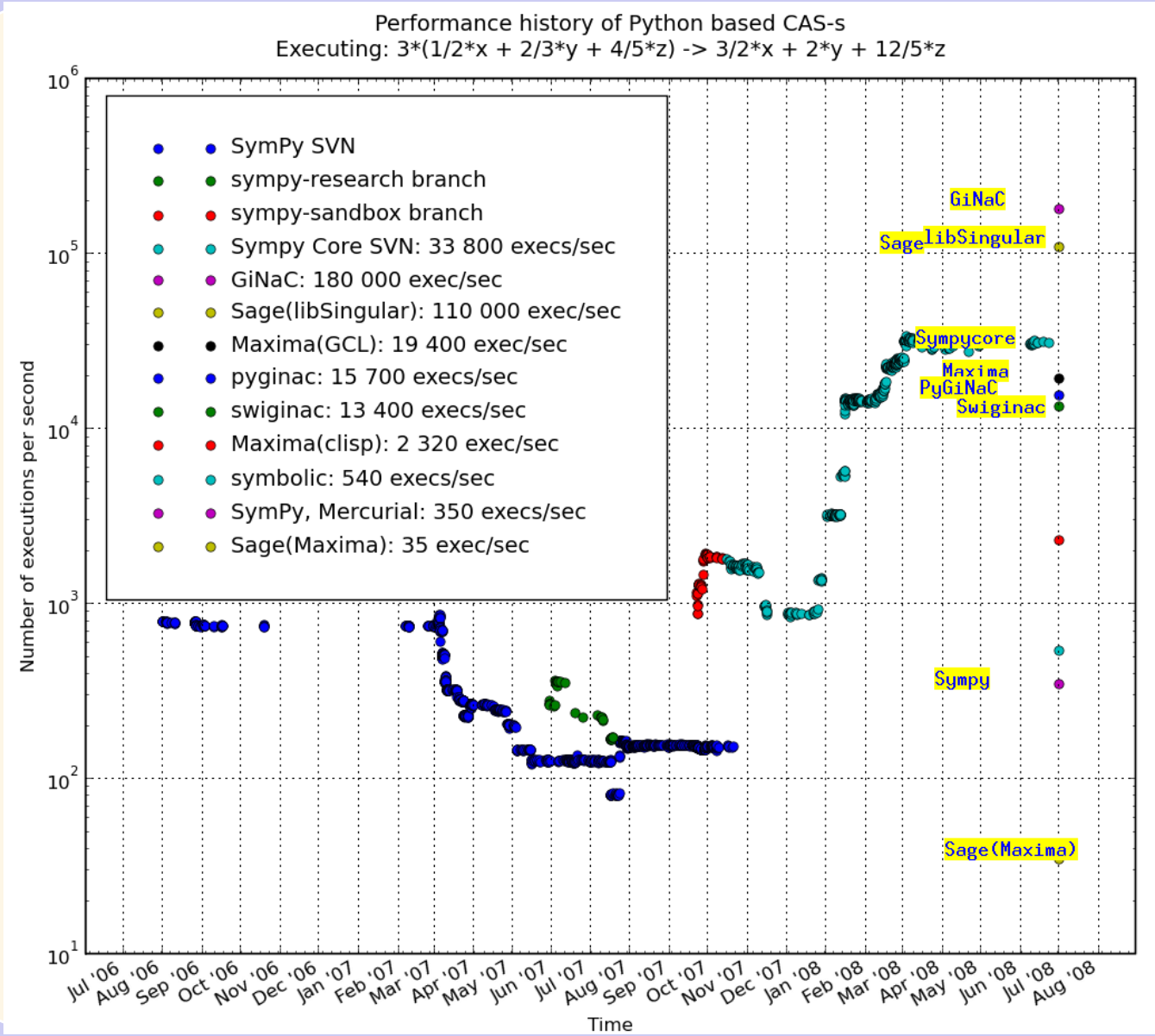
Sympycore secret notes — design

Things to keep in mind:

- theories of algebraic structures may seem complex, following their logic actually simplifies coding
- algorithms in text-books may be easy-to-understand but not always optimal
- there exist no ideal representation of mathematical concepts in a computer program, efficiency depends on a particular application and used algorithms

Things to follow:

- try different data representations for a particular mathematical concept, consider supporting multiple representation
- try different algorithms
- never underestimate the power of mathematics


Performance history of Python based CAS-s
Executing: 3*(1/2*x + 2/3*y + 4/5*z) -> 3/2*x + 2*y + 12/5*z
SymPy SVN
sympy-research branch
sympy-sandbox branch
Sympy Core SVN: 33 800 execs/sec
GiNaC: 180 000 exec/sec
Sage(libSingular): 110 000 exec/sec
Maxima(GCL): 19 400 exec/sec
pyginac: 15 700 execs/sec
swiginac: 13 400 execs/sec
Maxima(clisp): 2 320 exec/sec
symbolic: 540 execs/sec
SymPy, Mercurial: 350 execs/sec
Sage(Maxima): 35 exec/sec
GiNaC
Sage
libSingular
Sympycore
Maxima
PyGiNaC
Swiginac
Sympy
Sage(Maxima)
Number of executions per second
Time
Jul '06
Aug '06
Sep '06
Oct '06
Nov '06
Dec '06
Jan '07
Feb '07
Mar '07
Apr '07
May '07
Jun '07
Jul '07
Aug '07
Sep '07
Oct '07
Nov '07
Dec '07
Jan '08
Feb '08
Mar '08
Apr '08
May '08
Jun '08
Jul '08
Aug '08

Conclusions

- Sympycore — a research project, its aim is to seek out new high-performance solutions to represent and manipulate symbolic expressions in Python language
- Sympycore — fastest Python based CAS core implementation
- Our goal is to work in the direction of making Sympycore usable for Sympy