SymEngine, a fast symbolic manipulation library

Ondřej Čertík, Isuru Fernando, Thilina Rathnayake, Abhinav Agarwal, Sumith Kulal, Abinash Meher, Rajith Vidanaarachchi

July 15, 2016

Outline

SymEngine

- Introduction
- Features
- Demo (Python, Ruby, Julia)
- Why C++, how to write safe code
- Internals of SymEngine
- Roadmap for using SymEngine in SymPy
- Benchmarks

- Symbolic manipulation library written in C++
- Thin wrappers to Python, Ruby, Julia, C, ...
- `https://github.com/symengine/symengine`
- `https://github.com/symengine/symengine.py`
- `https://github.com/symengine/symengine.rb`
- `https://github.com/symengine/symengine.jl`
- MIT licensed
- Started in 2012
- 35 contributors
- Runs on Linux (GCC, Clang, Intel), OS X (GCC, Clang), Windows (MSVC, MinGW, MinGW-w64)
- Part of the SymPy organization, but the C++ library is Python independent

Introduction
Goals

- Be the fastest symbolic manipulation library (opensource or commercial)
- Serve as the core for SymPy and Sage
- Serve as the default symbolic manipulation library in other languages thanks to thin wrappers (Python, Ruby, Julia, C, ...)

What language to use

Problem

SymPy speed is sometimes insufficient

- Handling of very large expressions
- Large calculations using small/medium size expressions

Let's fix that

- We tried: pure Python/PyPy, Cython, C
- Investigated Julia, Rust, Scala, Javascript, ...
- Chose C++

Current Features

- Core (symbols, +, -, *, /, **)
- Elementary Functions (sin, cos, Gamma, Erf)
- Differentiation, substitution
- Matrices
- Polynomials (Piranha, Flint)
- Series expansion
- Printing, parsing
- Numeric evaluation (double/arbitrary precision)

Demo

Demo Time

Why Pure C++

- **Fast** in Release mode, but safe in Debug mode
- Compiler helps (not as good as Scala or Haskell, but much better than Python)
- Just one language to learn, thus easy to maintain (as opposed to several intertwined layers such as C + Cython + Python)
- Thin wrappers (that core developers do not need to maintain), all functionality in C++
- Easier to create bindings to other languages like Python, Julia, Ruby and Haskell.

Why Pure C++: Fast in Release Mode

- Allows direct memory handling (allocation, deallocation, access)
- Allows to tweak how and when things are done
- It is possible to go to bare metal
- Allows reasonably high level abstractions (simple, maintainable code)

Why Pure C++: Safe in Debug Mode

- Reference counted pointers `Teuchos::RCP` (from Trilinos)
- Checks for dangling and null pointers (exception is raised)
- No raw pointers/references (use `Ptr` and `RCP`)
- Use a safe subset of C++
- Few other rules, e.g. how to use `Ptr` and `RCP` properly
- Possible to visually verify in a PR (pull request) review
- Hopefully eventually there are plugins to Clang to check automatically (since the rules are simple and static)
- As fast as raw pointers in Release mode (but it could segfault)

Conclusion: the code cannot segfault or have undefined behavior in Debug mode — always get an exception at runtime, or a compile error.

- Add uses `std::unordered_map` (hashtable)
  - $2xy^2 + 3x^2y + 5 \rightarrow \{xy^2 : 2, x^2y : 3\}; \mathit{coeff} = 5$
- Mul uses `std::map` (red-black tree)
  - $2xy^2 \rightarrow \{x : 1, y : 2\}; \mathit{coeff} = 2$
- Pow just stores the base and exponent
- Each object is reference counted (RCP), very fast implementation in Release mode

Internals of SymEngine

Extensibility using visitor pattern

- All algorithms implemented using visitor pattern
- Algorithm is implemented in its own file, separate from the core
- Two virtual function calls (can be implemented in third party code or user code)
- Special version with just one virtual function call (faster, but must be compiled as part of the SymEngine source code)
- The speed difference between the two is minor

SymPy, SymEngine and the interface

Using SymPy in SymEngine

- SymEngine will convert any SymPy object to a corresponding SymEngine object before doing any operation

```
>>> from symengine import symbols, Add
>>> import sympy
>>> x = symbols("x")
>>> y = sympy.symbols("y")
>>> x + y
x + y
>>> type(x+y)
<type 'symengine.lib.symengine_wrapper.Add'>
```

- What if there is no corresponding SymEngine object?

SymPy, SymEngine and the interface

Using SymPy in SymEngine

- SymEngine will keep a reference to a SymPy object if there is no corresponding SymEngine object using Python/C API. SymEngine will use Python callbacks to evaluate the SymPy object

```
>>> e = x + sympy.loggamma(x)
>>> assert str(e) == "x + loggamma(x)"
>>> assert isinstance(e, Add)

>>> f = e.subs({x : 10})
>>> assert f == 10 + log(362880)

>>> f = e.subs({x : 2})
>>> assert f == 2
```

SymPy, SymEngine and the interface

Using SymEngine in SymPy

- `>>> from sympy.core.backend import symbols, sin, diff`
- Most things can be used unmodified
- Few things are fundamentally different (e.g. SymPy stores `I` as `ImaginaryUnit`, SymEngine has a `Complex` class)
- We will have a compatibility layer, probably similar to Python 2 and 3 support using the same source base.

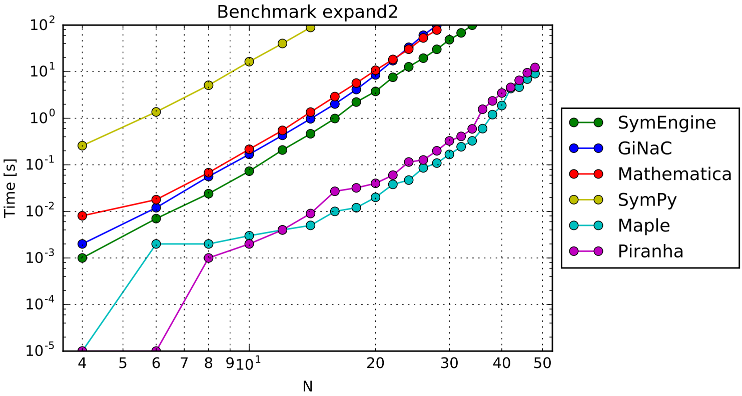
Benchmarks were run in a Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz running Ubuntu 16.04 with gcc 5.4.0

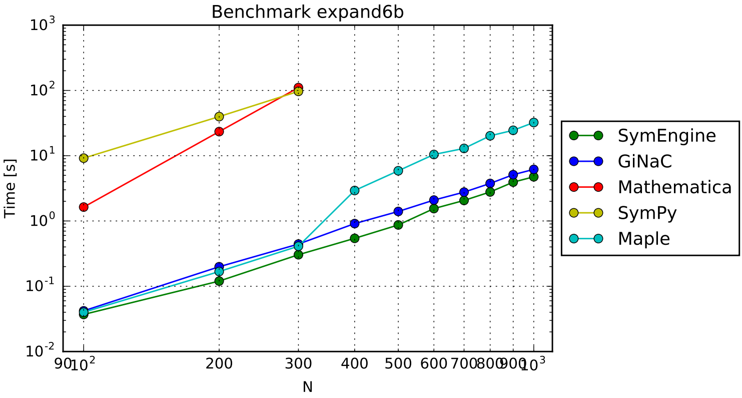
- SymEngine master (with GMP and FLINT)
- GiNaC 1.6.6
- SymPy 1.0
- Mathematica 10.2.0.0
- Maple 2015.2

Benchmarks

Expand benchmark

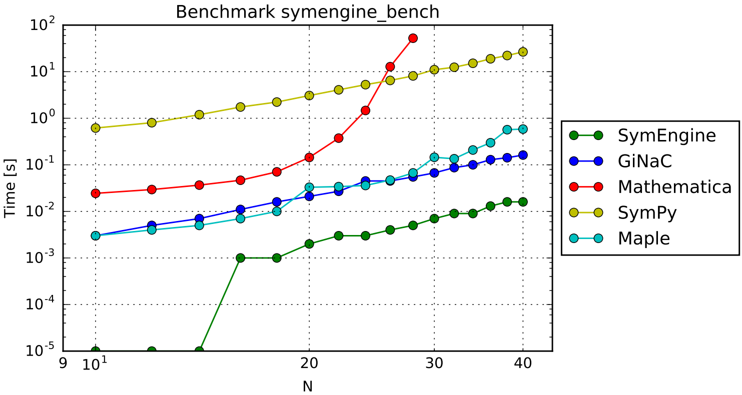
- $e = (x + y + z + w)^n$
- $f = e * (e + w)$
- Measure time taken for expanding f

- ```
  using SymEngine
  using TimeIt

  @vars x y z w
  n = 30
  e = (x + y + z + w)^n
  f = e * (e + w)
  @timeit expand(f)
  ```
  ```

Benchmarks

Expand benchmark



Benchmarks

Modified GiNaC benchmark

- Let e be the expanded sum of 2 symbols $\{a_0, a_1\}$ and $n-2$ trigonometric functions $\{sin(a_2), sin(a_3)...sin(a_{n-1})\}$ squared: e $\leftarrow (a_0 + a_1 + \sum_{i=2}^{n-1} sin(a_i))^2$
- Substitute $a_0 \leftarrow -\sum_{i=2}^{n-1} sin(a_i)$
- Expand e again so it collapses to a_1^2

Benchmarks

Modified GiNaC benchmark

```
from symengine import symbols, sin
from time import clock
n = 100
a0, a1 = symbols("a0, a1")
t = sum([sin(symbols("a%s" % i)) for i in range(2, n)])
e = a0 + a1 + t
f = -t

t1 = clock()
e = (e**2).expand()
e = e.xreplace({a0: f})
e = e.expand()
t2 = clock()
```

Benchmarks

Modified GiNaC benchmark



Benchmarks

SymEngine Benchmark

- Series expansion of $sin(cos(x + 1))$ around $x = 0$

- ```
  RCP<const Symbol> x = symbol("x");
  int n = 15;
  RCP<const Basic> ex = sin(cos(add(integer(1), x)));
  auto t1 = std::chrono::high_resolution_clock::now();
  RCP<const Basic> res = series(ex, x, n);
  auto t2 = std::chrono::high_resolution_clock::now();
  ```
  ```

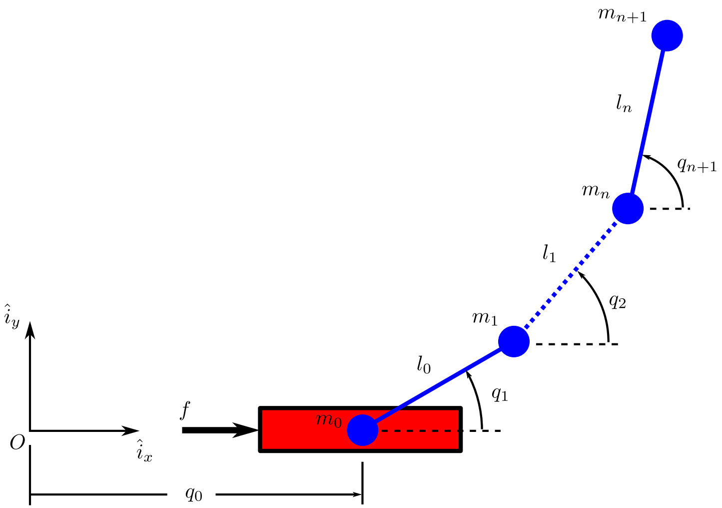
Benchmarks

SymEngine Benchmark



Benchmarks

PyDy Benchmark



Benchmarks

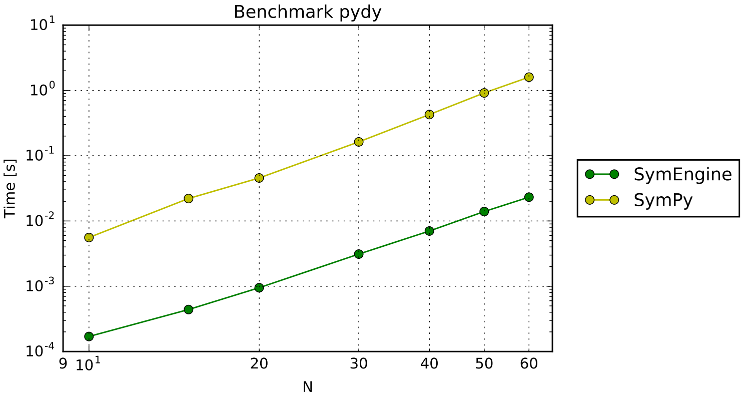
PyDy Benchmark

n	SymEngine + SymPy	SymPy only	Speedup
10	0.17 s	5.58 s	32.8x
15	0.44 s	22.07 s	50.1x
20	0.95 s	45.59 s	47.9x
30	3.11 s	162.80 s	52.3x
40	7.02 s	427.16 s	60.8x
50	13.95 s	915.83 s	65.6x
60	23.16 s	1596.37 s	68.9x

Table: Results

Benchmarks

PyDy Benchmark



Summary

- SymEngine aims to be the fastest C++ symbolic manipulation library
- Thin wrappers to other languages (Python, Ruby, Julia, C, ...)
- Easily usable as an optional backend in SymPy (and Sage)

Thank You

GitHub:

- `https://github.com/symengine/symengine`
- `https://github.com/symengine/symengine.py`
- `https://github.com/symengine/symengine.rb`
- `https://github.com/symengine/symengine.jl`

Mailinglist:

- `http://groups.google.com/group/symengine`

Gitter:

- `https://gitter.im/symengine/symengine`