SymEngine: A Fast Symbolic Manipulation Library

Ondřej Čertík, Isuru Fernando, Thilina Rathnayake, Abhinav Agarwal, Sumith Kulal, Abinash Meher, Rajith Vidanaarachchi, Shikhar Jaiswal, Ranjith Kumar

September 4, 2017

Outline

SymEngine

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

- Symbolic manipulation library written in C++
- Thin wrappers to Python, Ruby, Julia, C and Haskell
- MIT licensed
- Started in 2012
- 46 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 (open-source or commercial)
- Serve as the core for SymPy and Sage, optionaly supporting PyDy
- Serve as the default symbolic manipulation library in other languages thanks to thin wrappers (Python, Ruby, Julia, C and Haskell)

Choice of Language

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)
- Number Theory
- Differentiation, Substitution
- Matrices and Sets
- Polynomials (Piranha, Flint)
- Series Expansion
- Solvers (Polynomial and Trigonometric)
- Printing, Parsing and Code Generation
- Numeric Evaluation (Double and 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.

Internals of SymEngine

How Add Class Works

- Add stores the various algebraic terms in a dictionary as variable-coefficient pairs, while separately storing the constant term of the expression
- Add uses `std::unordered_map` (hashtable) for the dictionary
  - $2xy^2 + 3x^2y + 5 \rightarrow \{xy^2 : 2, x^2y : 3\}; coeff = 5$
- Each object is reference counted (RCP), hence very fast implementation in Release mode

- Mul stores the various algebraic terms in a dictionary as base-exponent pairs, while separately storing the constant coefficient of the expression
- Mul uses `std::map` (red-black tree)
  - $2xy^2 \rightarrow \{x : 1, y : 2\}; \mathit{coeff} = 2$
- Each object is, like in the case of Add class, reference counted (RCP)

Internals of SymEngine

How Pow Class Works

- Pow just stores the base and exponent as individual RCP objects, no dictionaries are used for storage
  - $x^5 \rightarrow base = x; coeff = 5$

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 for practical purposes

SymEngine and SymEngine.py

Designing The Interface

- SymEngine.py uses Cython's native support for C++ constructs.
- Uses Cython's *libcpp* module for importing *bool*, *string*, *map*, *vector* and *pair* data types.
- Declares *set*, *multiset* and *unordered_map* directly from C++'s $< set >$ module, as Cython's *libcpp.set* does not support multi-template arguments to any of them.

```
cdef extern from "<set>" namespace "std":
    cdef cppclass set[T, U]:
```

SymEngine and SymEngine.py

Designing The Interface

- Additionally, maintains .pxd files with cdef extern from blocks and (if existing) the C++ namespace name:

```
cdef extern from "<symengine/symbol.h>" namespace
"SymEngine":
```

- In these blocks, we declare SymEngine's classes as cdef cppclass blocks:

```
cdef cppclass Symbol(Basic):
```

- And then declare SymEngine's public names (variables, methods and constructors):

```
Symbol(string name) nogil
string get_name() nogil
```

SymEngine and SymEngine.py

Working With SymEngine's Data Types

- Cython classes implemented for data types available in SymEngine, and Python classes for types currently unavailable.
- As soon as a class object is called, a SymEngine equivalent object is created and passed to a dedicated function (*c2py*).
- The function takes the object and returns the corresponding Cython or Python counterpart for usage.
- Conversely, another dedicated function (*sympy2symengine*) takes a Python object and returns the SymEngine equivalent.

- Since specific classes are created for each data type, the functionalities can be directly called, just as in the case of SymPy.
- Most of the test cases derive directly from SymPy's test suite for filtering out inconsistencies and finding the fundamental differences.

- 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?

- 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.Mod(x, 2)
>>> assert str(e) == "x + Mod(x, 2)"
>>> assert isinstance(e, Add)

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

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

SymPy, SymEngine and SymEngine.py

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)
- SymEngine.py accounts for this incompatibility by having a Python class implemented for `ImaginaryUnit` returning `I`
- Singleton class also implemented in SymEngine.py to account for SymPy's Singleton pattern.

SymPy, SymEngine and SymEngine.py

Speeding Up - Past Strategy

- Create an old_core_api.py module, which will define the API to the core, the implementation will just import things from the current core.
- All client code (that is, the rest of SymPy that uses the core) will access things from the core through old_core_api.py only.
- Each method accepts SymPy objects, converts to SymEngine, calls SymEngine's counterpart, and converts the result back to SymPy. Then it validates the result by calling SymPy's class directly and compares the final expressions.
- Remove the validation and remove the SymPy's core, that is not used at this point. Tests must still pass, since we didn't change any results from the previous step.

SymPy, SymEngine and SymEngine.py

Speeding Up - Current Approach

- Define a file backend.py in SymPy's core, for providing optional support of SymEngine's routines through USE flags.

```
USE_SYMENGINE = os.getenv('USE_SYMENGINE', '0')
USE_SYMENGINE = USE_SYMENGINE.lower()
                in ('1', 't', 'true')
if USE_SYMENGINE:
    from symengine import ...
else:
    from sympy import ...
```

- Shift all the viable imports used in a particular module of interest to import from backend.py

```
sympy/liealgebras/weyl_group.py
from sympy.core.backend import Matrix, eye ..
```

SymPy, SymEngine and SymEngine.py

Speeding Up - Current Approach

- Hence, SymEngine's routines are directly used for backend computations whenever the USE flag is set.
- As such, no SymPy->SymEngine->SymPy conversion cycle is required, leading to maximum performance improvement and minimal changes.
- When the USE flag is unset, routines are imported from SymPy core itself.

Sage, SymEngine and SymEngine.py

Using SymEngine in Sage

- Every particular class in SymEngine.py, having a corresponding data type in Sage, has a callable *_sage_*() sub-routine.
- Hence the conversion of SymEngine objects to Sage compatible type is handled through the above sub-routine itself.

```
assert Integer(12)._sage_() == sage.Integer(12)
```

- Additionally, every particular class also has a callable *_sympy_*() sub-routine, for converting objects to SymPy specific types, which is accessed through *sympify* function. This allows us to do the following:

```
assert Integer(12) == sympify(sage.Integer(12))
```

PyDy, SymEngine and SymEngine.py

Using SymEngine in PyDy

- PyDy, short for Python Dynamics, is a tool kit written in the Python programming language to enable the study of multibody dynamics.
- Directly uses the APIs of SymPy's mechanics module which currently has the optional SymEngine usage option, keeping the code-related changes minimal.
- Hence, the idea here is to use SymEngine in the same way as used by many SymPy modules, through optional flags and shifting the following imports:

```
from sympy import symbols ...
to
from sympy.core.backend import symbols ...
```

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

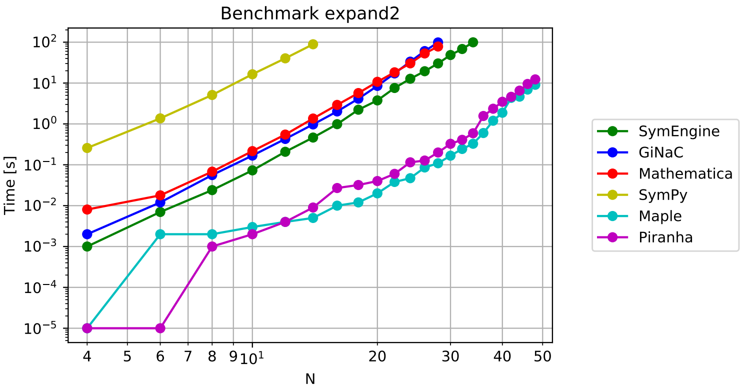
- $e = (x + y + z + w)^n$
- $f = e * (e + w)$
- Measure time taken for expanding f

-
```
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

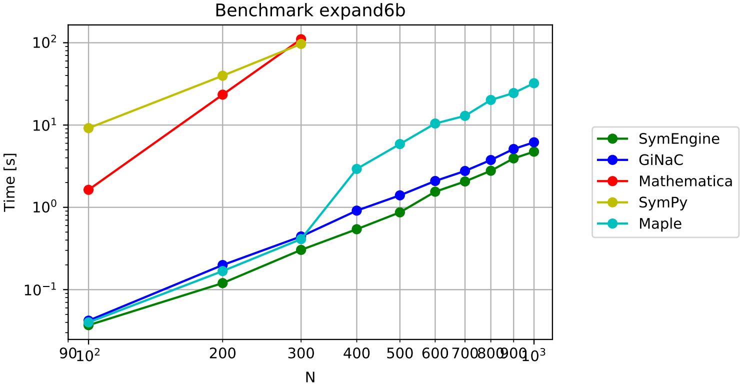
Modified GiNaC Benchmark

```
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

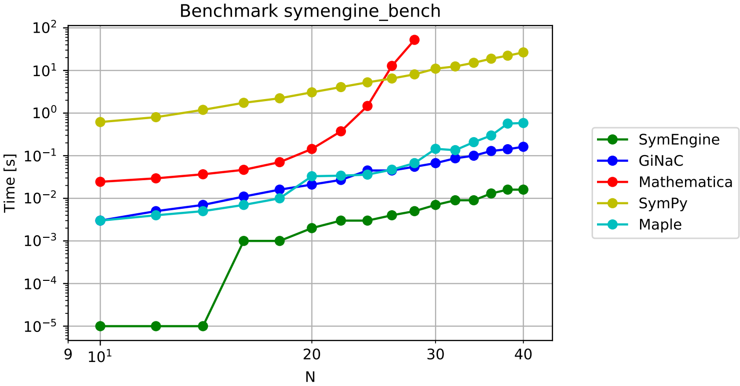


- Series expansion of $sin(cos(x+1))$ around $x = 0$
-

```
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();
```

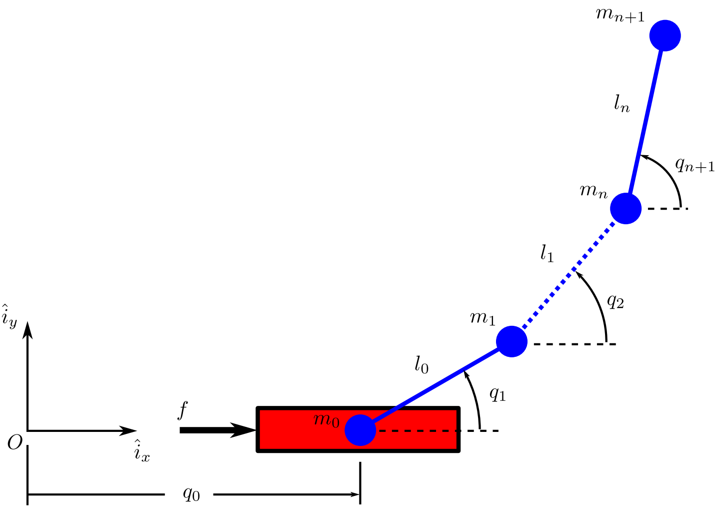
Benchmarks

SymEngine Benchmark



Benchmarks

PyDy Benchmark



Benchmarks

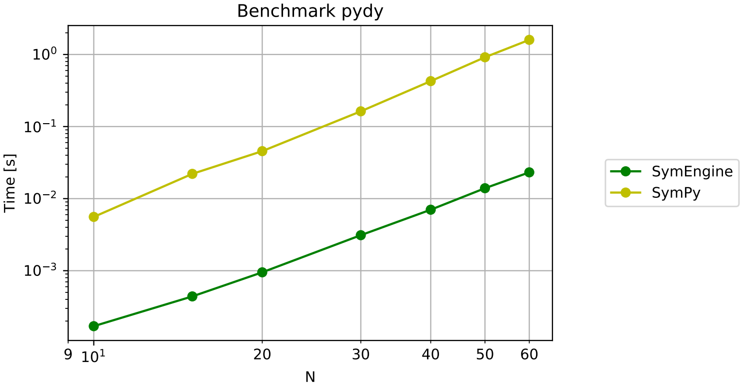
PyDy Benchmark

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 and Haskell)
- Easily usable as an optional backend in SymPy, Sage and PyDy

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
- https://github.com/symengine/symengine.hs

Mailinglist:

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

Gitter:

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