Computing with polynomials in SymPy

Mateusz Paprocki <mattpap@gmail.com>

Wrocław University of Technology

July 10, 2010

Presentation plan

- Short introduction to SymPy
- Polynomials manipulation module
- Tour of module's functionality
- Use cases of the module
  - Mathematics: Graph coloring
  - Embedding: Code generation
- Notes on the internal implementation
  - Multi–level structure
  - Ground types
  - Pure mode Cython
- Future plans

What is SymPy?

- A pure Python library for symbolic mathematics

```
>>> from sympy import *
>>> x = Symbol('x')

>>> limit(sin(pi*x)/x, x, 0)
pi

>>> integrate(x + sinh(x), x)
(1/2)*x**2 + cosh(x)

>>> diff(_, x)
x + sinh(x)
```

What is SymPy?

- A pure Python library for symbolic mathematics

```
>>> from sympy import *
>>> x = Symbol('x')

>>> limit(sin(pi*x)/x, x, 0)
pi

>>> integrate(x + sinh(x), x)
(1/2)*x**2 + cosh(x)

>>> diff(_, x)
x + sinh(x)
```

How to get involved?

- Visit our main web site:
  - `www.sympy.org`
- and additional web sites:
  - `docs.sympy.org`
  - `wiki.sympy.org`
  - `live.sympy.org`
- Contact us on our mailing list:
  - `sympy@googlegroups.com`
- or/and IRC channel:
  - `#sympy` on FreeNode
- Clone source repository:

```
git clone git://git.sympy.org/sympy.git
```

My role in the project

A short historical background

- cooperation started in March 2007
  - a few simple bugfixes and improvements
- next came Google Summer of Code 2007
  - algorithms for solving recurrence relations
  - algorithms for definite and indefinite summations
- and this is how it works:
  - algorithms for symbolic integration
  - algebraic structures, polynomials
  - expression simplification, ...
- else:
  - Google Summer of Code 2009, 2010 mentor (PSU, PSF)
  - EuroSciPy 2009 and 2010, py4science (UC Berkeley)
  - master's thesis

My role in the project

A short historical background

- cooperation started in March 2007
  - a few simple bugfixes and improvements
- next came Google Summer of Code 2007
  - algorithms for solving recurrence relations
  - algorithms for definite and indefinite summations
- and this is how it works:
  - algorithms for symbolic integration
  - algebraic structures, polynomials
  - expression simplification, ...
- else:
  - Google Summer of Code 2009, 2010 mentor (PSU, PSF)
  - EuroSciPy 2009 and 2010, py4science (UC Berkeley)
  - master's thesis

Why we need polynomials in SymPy?

- polynomials are useful on their own
  - many problems can be stated in terms of polynomials
- polynomials are ubiquitous in symbolic algorithms
  - symbolic integration, summation
    - resultant, GCDs, factorization, ...
  - simplification of expressions
    - GCDs, factorization, ...
  - truncated power series
    - fast arithmetics, compositions, ...
  - solvers
    - roots, Gröbner bases, ...
  - ...

Why we need polynomials in SymPy?

- polynomials are useful on their own
  - many problems can be stated in terms of polynomials
- polynomials are ubiquitous in symbolic algorithms
  - symbolic integration, summation
    - resultant, GCDs, factorization, ...
  - simplification of expressions
    - GCDs, factorization, ...
  - truncated power series
    - fast arithmetics, compositions, ...
  - solvers
    - roots, Gröbner bases, ...
  - ...

Why polynomials not expressions?

- take advantage of
  - additional knowledge
    - monomial orderings, ...
  - dedicated data structures
  - polynomials manipulation algorithms
- efficiency concerns

```
In [1]: var('x')

In [2]: f = Add(*[ k*x**k for k in xrange(1, 1000) ])

In [3]: %timeit f + 3*f;
1 loops, best of 3: 773 ms per loop

In [4]: F = Poly(f, expand=False)

In [5]: %timeit F + 3*F;
100 loops, best of 3: 12.8 ms per loop
```

Why polynomials not expressions?

- take advantage of
  - additional knowledge
    - monomial orderings, ...
  - dedicated data structures
  - polynomials manipulation algorithms
- efficiency concerns

```
In [1]: var('x')

In [2]: f = Add(*[ k*x**k for k in xrange(1, 1000) ])

In [3]: %timeit f + 3*f;
1 loops, best of 3: 773 ms per loop

In [4]: F = Poly(f, expand=False)

In [5]: %timeit F + 3*F;
100 loops, best of 3: 12.8 ms per loop
```

Module's functionality

- classical arithmetics
- GCD, LCM, cofactors
- functional decomposition
- square–free decomposition
- factorization into irreducibles
- root counting, isolation, finding
- Gröbner bases, monomial orderings
- polynomial remainder sequences
- subresultants, resultant, discriminant
- special polynomials
- polynomial forms
- ...

Module's functionality

Factorization of polynomials

- finite fields

```
In [1]: factor(x**2 + 1, modulus=2)
Out[1]: (1 + x)**2
```

- rational numbers

```
In [2]: factor(x**6 + 1)
Out[2]: (1 + x**2)*(1 - x**2 + x**4)

In [3]: factor(expand((x + 101*y + 703*z)**20))
Out[3]: (x + 101*y + 703*z)**20
```

- algebraic numbers

```
In [4]: factor(x**4 + 1, extension=I)
Out[4]: (I + x**2)*(-I + x**2)

In [5]: factor(x**2 - 2*y**2, extension=sqrt(2))
Out[5]: (x + y*2**(1/2))*(x - y*2**(1/2))
```

Module's functionality

Solving inequalities

- inequalities

```
In [1]: a = Symbol('a', real=True)

In [2]: solve(a**2 - 2 > 0, a)
Out[2]: [(-oo, -2**(1/2)), (2**(1/2), oo)]
```

- systems of inequalities

```
In [3]: solve([a**2 - 2 < 0, a**2 - 1 > 0], a)
Out[3]: [(-2**(1/2), -1), (1, 2**(1/2))]
```

- relational syntax

```
In [4]: solve([a**2 - 2 < 0, a**2 - 1 > 0], a, relational=True)
Out[4]: Or(And(-2**(1/2) < a, a < -1), And(1 < a, a < 2**(1/2)))
```

- symbolic roots

```
In [5]: solve(a**5 + 2*a**3 + 3*a + 4 > 0, a)
Out[5]: [(RootOf(a**5 + 2*a**3 + 3*a + 4, 0), oo)]
```

Module's functionality

Gröbner bases and monomial orderings

- monomial orderings

```
In [1]: LT(x*y**7 + 2*x**2*y**3, order='lex')
Out[1]: 2*x**2*y**3

In [2]: LT(x*y**7 + 2*x**2*y**3, order='grlex')
Out[2]: x*y**7
```

- Gröbner bases

```
In [3]: var('x1,x2,i1,i2,i3')
Out[3]: (x1, x2, i1, i2, i3)

In [4]: f1, f2, f3 = x1**2 + x2**2, x1**2*x2**2, x1**3*x2 - x1*x2**3

In [5]: G = groebner([f1 - i1, f2 - i2, f3 - i3], wrt='x1,x2')

In [6]: reduced(x1**7*x2 - x1*x2**7, G, wrt=[x1, x2])[1]
Out[6]: -i2*i3 + i3*i1**2

In [7]: _.subs({i1: f1, i2: f2, i3: f3}).expand()
Out[7]: x2*x1**7 - x1*x2**7
```

Module's functionality

Generating special polynomials

- symmetric polynomials

```
In [1]: symmetric_poly(2, x, y, z)
Out[1]: x*y + x*z + y*z
```

- cyclotomic polynomials

```
In [2]: cyclotomic_poly(10, x)
Out[2]: 1 - x + x**2 - x**3 + x**4
```

- orthogonal polynomials

```
In [3]: hermite_poly(6, x)
Out[3]: -120 + 720*x**2 - 480*x**4 + 64*x**6
```

- Swinnerton–Dyer polynomials

```
In [4]: swinnerton_dyer_poly(3, x)
Out[4]: 576 - 960*x**2 + 352*x**4 - 40*x**6 + x**8
```

Module's functionality

Polynomial forms: Horner

- univariate polynomials

```
In [1]: horner(x**6 + 2*x**3 + 3*x**2 + 4*x + 5)
Out[1]: 5 + x*(4 + x*(3 + x*(2 + x**3)))

In [2]: horner(a*x**6 + b*x**3 + c*x**2 + d*x + e, x)
Out[2]: e + x*(d + x*(c + x*(b + a*x**3)))
```

- multivariate polynomials

```
In [3]: sum(i*j*x**i*y**j, (i, 1, 2), (j, 1, 2))
Out[3]: x*y + 2*x*y**2 + 2*y*x**2 + 4*x**2*y**2

In [4]: horner(_3)
Out[4]: x*(y*(1 + 2*y) + x*y*(2 + 4*y))

In [5]: horner(_3, wrt=y)
Out[5]: y*(x*(1 + 2*x) + x*y*(2 + 4*x))
```

Module's functionality

Computing symmetric reduction

- symmetric polynomials

```
In [1]: symmetrize(x**2 + y**2)
Out[1]: (-2*x*y + (x + y)**2, 0)

In [2]: symmetrize(x**2 + y**2, formal=True)
Out[2]: (-2*s2 + s1**2, 0, {s1: x + y, s2: x*y})
```

- non–symmetric polynomials

```
In [3]: symmetrize(x**2 - y**2)
Out[3]: (-2*x*y + (x + y)**2, -2*y**2)

In [4]: symmetrize(x**2 - y**2, formal=True)
Out[4]: (-2*s2 + s1**2, -2*y**2, {s1: x + y, s2: x*y})
```

Implemented algorithms

Only most remarkable cases here

- square–free decomposition
  - Yun
- factorization into irreducibles
  - finite fields
    - Berlekamp, Zassenhaus, Shoup
  - rational numbers
    - Cantor–Zassenhaus, Wang
- functional decomposition
  - Landau–Zippel
- Gröbner bases
  - Buchberger
- root isolation
  - continued fractions, Collins–Krandick

Use cases of the module

And SymPy in general

- solving mathematical problems
  - e.g. in teaching mathematics
  - Why?
    - uses single, easy to learn programming language
    - core algorithms exposed to the user
  - Example: graph k–coloring
- embedding in other software
  - e.g. code generation, preprocessing
  - Why?
    - small library with no dependencies
  - Example: C code generation

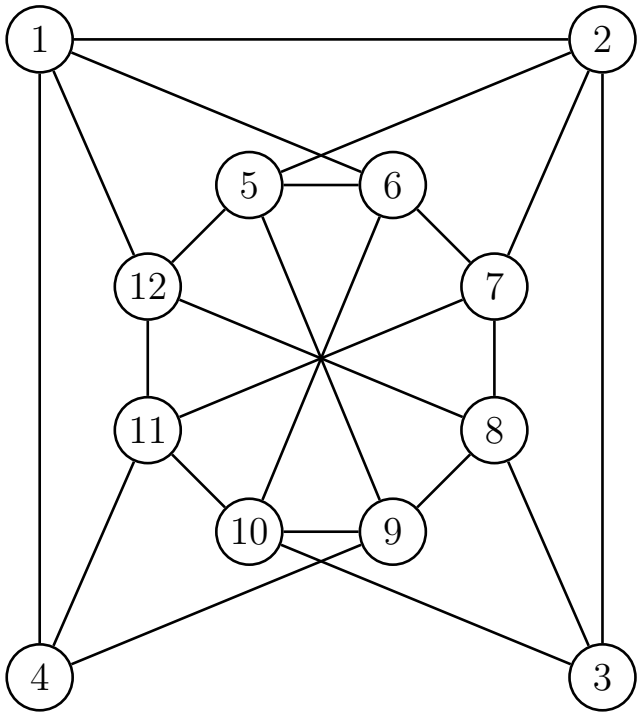
Use cases of the module

And SymPy in general

- solving mathematical problems
  - e.g. in teaching mathematics
  - Why?
    - uses single, easy to learn programming language
    - core algorithms exposed to the user
  - Example: graph k–coloring
- embedding in other software
  - e.g. code generation, preprocessing
  - Why?
    - small library with no dependencies
  - Example: C code generation

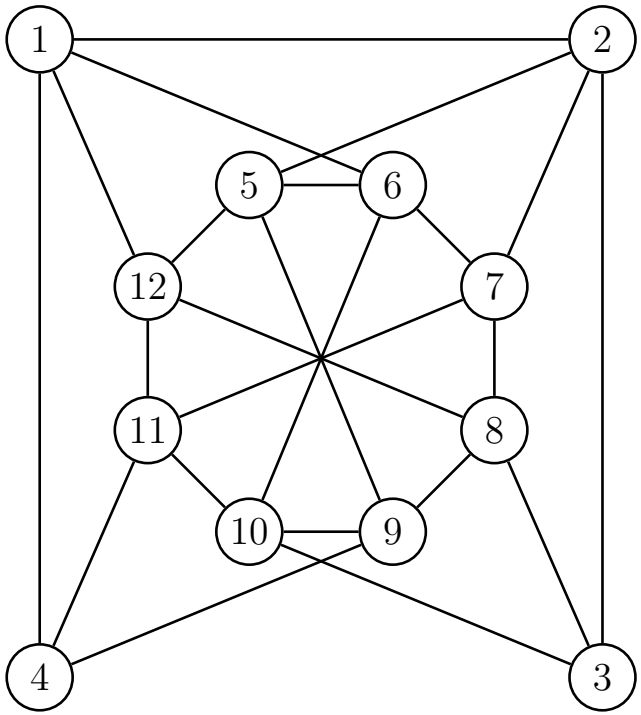
Use cases of the module

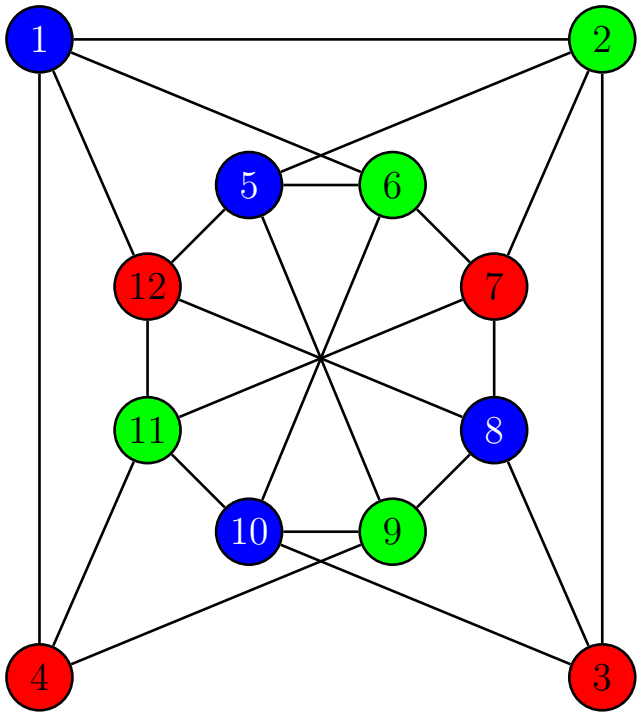
Graph k–coloring with Gröbner bases (1)



Use cases of the module

Graph k–coloring with Gröbner bases (1)





Use cases of the module

Graph k–coloring with Gröbner bases (2)

Given a graph $\mathcal{G}(V, E)$ we write two sets of equations:

- I_k — allow one of k colors per vertex

$$I_k = \{x_i^k - 1 : i \in V\}$$

- $I_{\mathcal{G}}$ — adjacent vertices have different colors assigned

$$I_{\mathcal{G}} = \{x_i^{k-1} + x_i^{k-2}x_j + \ldots + x_i x_j^{k-2} + x_j^{k-1} : (i,j) \in E\}$$

Next we solve $I_k \cup I_{\mathcal{G}}$ using the Gröbner bases method.

Use cases of the module

Graph k–coloring with Gröbner bases (2)

Given a graph $\mathcal{G}(V, E)$ we write two sets of equations:

- I_k — allow one of k colors per vertex

$$I_k = \{x_i^k - 1 : i \in V\}$$

- $I_\mathcal{G}$ — adjacent vertices have different colors assigned

$$I_\mathcal{G} = \{x_i^{k-1} + x_i^{k-2}x_j + \ldots + x_i x_j^{k-2} + x_j^{k-1} : (i,j) \in E\}$$

Next we solve $I_k \cup I_\mathcal{G}$ using the Gröbner bases method.

Use cases of the module

Graph k–coloring with Gröbner bases (2)

Given a graph $\mathcal{G}(V, E)$ we write two sets of equations:

- I_k — allow one of k colors per vertex

$$I_k = \{x_i^k - 1 : i \in V\}$$

- $I_{\mathcal{G}}$ — adjacent vertices have different colors assigned

$$I_{\mathcal{G}} = \{x_i^{k-1} + x_i^{k-2}x_j + \ldots + x_i x_j^{k-2} + x_j^{k-1} : (i,j) \in E\}$$

Next we solve $I_k \cup I_{\mathcal{G}}$ using the Gröbner bases method.

Use cases of the module

Graph k–coloring with Gröbner bases (2)

Given a graph $\mathcal{G}(V, E)$ we write two sets of equations:

- I_k — allow one of k colors per vertex

$$I_k = \{x_i^k - 1 : i \in V\}$$

- $I_{\mathcal{G}}$ — adjacent vertices have different colors assigned

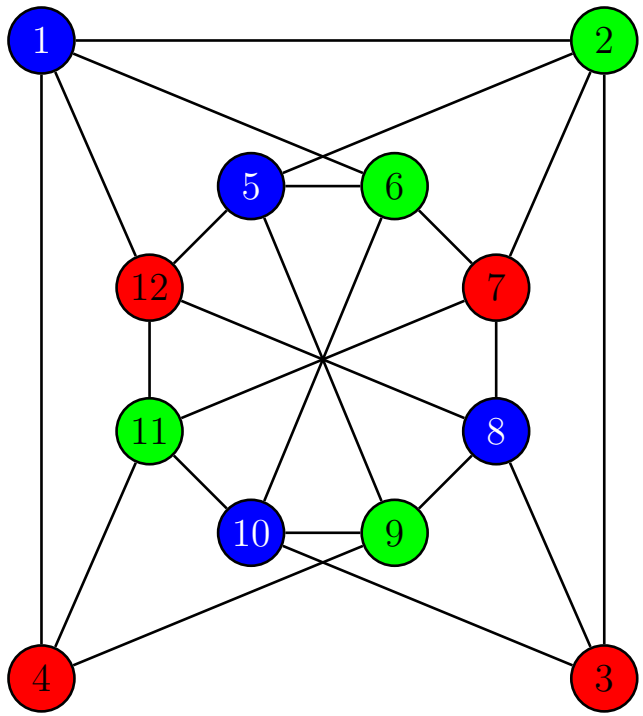
$$I_{\mathcal{G}} = \{x_i^{k-1} + x_i^{k-2}x_j + \ldots + x_i x_j^{k-2} + x_j^{k-1} : (i,j) \in E\}$$

Next we solve $I_k \cup I_{\mathcal{G}}$ using the Gröbner bases method.

Use cases of the module

Graph k–coloring with Gröbner bases (3)

$$\{x_1 + x_{11} + x_{12},$$
$$x_2 - x_{11},$$
$$x_3 - x_{12},$$
$$x_4 - x_{12},$$
$$x_5 + x_{11} + x_{12},$$
$$x_6 - x_{11},$$
$$x_7 - x_{12},$$
$$x_8 + x_{11} + x_{12},$$
$$x_9 - x_{11},$$
$$x_{10} + x_{11} + x_{12},$$
$${x_{11}}^2 + x_{11}x_{12} + x_{12}^2,$$
$${x_{12}}^3 - 1\}$$



Use cases of the module

Graph k–coloring with Gröbner bases (4)

Here is how to solve 3–coloring problem in SymPy:

```
In [1]: V = range(1, 12+1)
In [2]: E = [(1,2),(2,3),(1,4),(1,6),(1,12),(2,5),(2,7),
(3,8),(3,10),(4,11),(4,9),(5,6),(6,7),(7,8),(8,9),(9,10),
(10,11),(11,12),(5,12),(5,9),(6,10),(7,11),(8,12)]

In [3]: X = [ Symbol('x' + str(i)) for i in V ]
In [4]: E = [ (X[i-1], X[j-1]) for i, j in E ]

In [5]: I3 = [ x**3 - 1 for x in X ]
In [6]: Ig = [ x**2 + x*y + y**2 for x, y in E ]

In [7]: G = groebner(I3 + Ig, X, order='lex')

In [8]: G != [1]
Out[8]: True
```

Use cases of the module

Graph k–coloring with Gröbner bases (4)

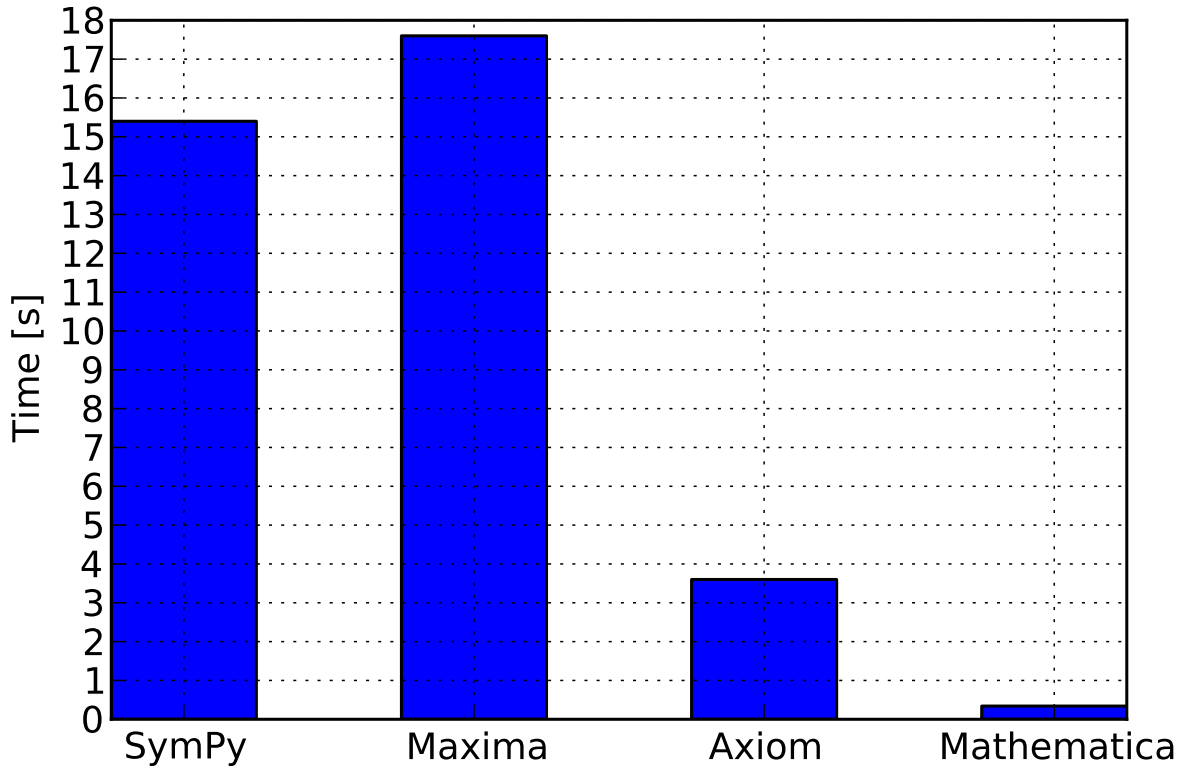


Figure: Average timing of Gröbner basis computation

Use cases of the module

C code generation (1)

The problem:

- transform expressions into Horner form
- generate equivalent C code
  - e.g. for fast evaluation
- assume we can't use `pow()` function

How we will approach the problem?

- define a new printer to generate code
- use `horner()` to transform expressions

Use cases of the module

C code generation (1)

The problem:

- transform expressions into Horner form
- generate equivalent C code
  - e.g. for fast evaluation
- assume we can't use `pow()` function

How we will approach the problem?

- define a new printer to generate code
- use `horner()` to transform expressions

Use cases of the module

C code generation (2)

Lets define a dedicated printer for the task:

```
from sympy.printing import StrPrinter

class CPrinter(StrPrinter):
    """Print Lambda as C function and unroll Pow. """

    counter = 0

    def _print_Lambda(self, expr):
        self.counter += 1

        return """long _L%i(long %s) {\n  return (%s);\n}""" % \
            (self.counter, expr.args[0], self.doprint(expr.args[1]))

    def _print_Pow(self, expr):
        if expr.exp.is_Integer:
            return '*'.join([str(expr.base)]*int(expr.exp))
        else:
            return StrPrinter._print_Pow(self, expr)
```

Use cases of the module

C code generation (3)

Lets generate some code using the new `CPrinter`:

```
In [1]: from sympy import horner, Lambda

In [2]: from sympy.abc import x

In [3]: f = horner(x**6 + 2*x**3 + 3*x**2 + 4*x + 5)

In [4]: print CPrinter().doprint(Lambda(x, f))
Out[4]:
double _L1(double _x) {
  return (5 + (4 + (3 + (2 + _x*_x*_x)*_x)*_x)*_x);
}
```

Multi–level structure

- user level: L3

```
>>> f3 = x**10 - 1
>>> %timeit factor_list(f3)
100 loops, best of 3: 5.57 ms per loop
```

- user level: L2

```
>>> f2 = Poly(x**10 - 1, x, domain='ZZ')
>>> %timeit f2.factor_list()
100 loops, best of 3: 2.15 ms per loop
```

- internal level: L1

```
>>> f1 = DMP([1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1], ZZ)
>>> %timeit f1.factor_list()
100 loops, best of 3: 1.90 ms per loop
```

- internal level: L0

```
>>> f0 = [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1]
>>> %timeit dup_factor_list(f0, ZZ)
100 loops, best of 3: 1.88 ms per loop
```

Ground types

```
In [1]: from sympy.polys.algebratools import ZZ_python, ZZ_gmpy

In [2]: f = Poly(x**3 - 5, domain=ZZ_python())

In [3]: map(type, f.rep.all_coeffs())
Out[3]: [<type 'int'>, <type 'int'>, <type 'int'>, <type 'int'>]

In [4]: g = Poly(x**3 - 5, domain=ZZ_gmpy())

In [5]: map(type, g.rep.all_coeffs())
Out[5]: [<type 'mpz'>, <type 'mpz'>, <type 'mpz'>, <type 'mpz'>]

In [6]: f + g
Out[6]: Poly(2*x**3 - 10, x, domain='ZZ')

In [7]: map(type, _.rep.all_coeffs())
Out[7]: [<type 'mpz'>, <type 'mpz'>, <type 'mpz'>, <type 'mpz'>]
```

Ground types

Benchmark: Small coefficients

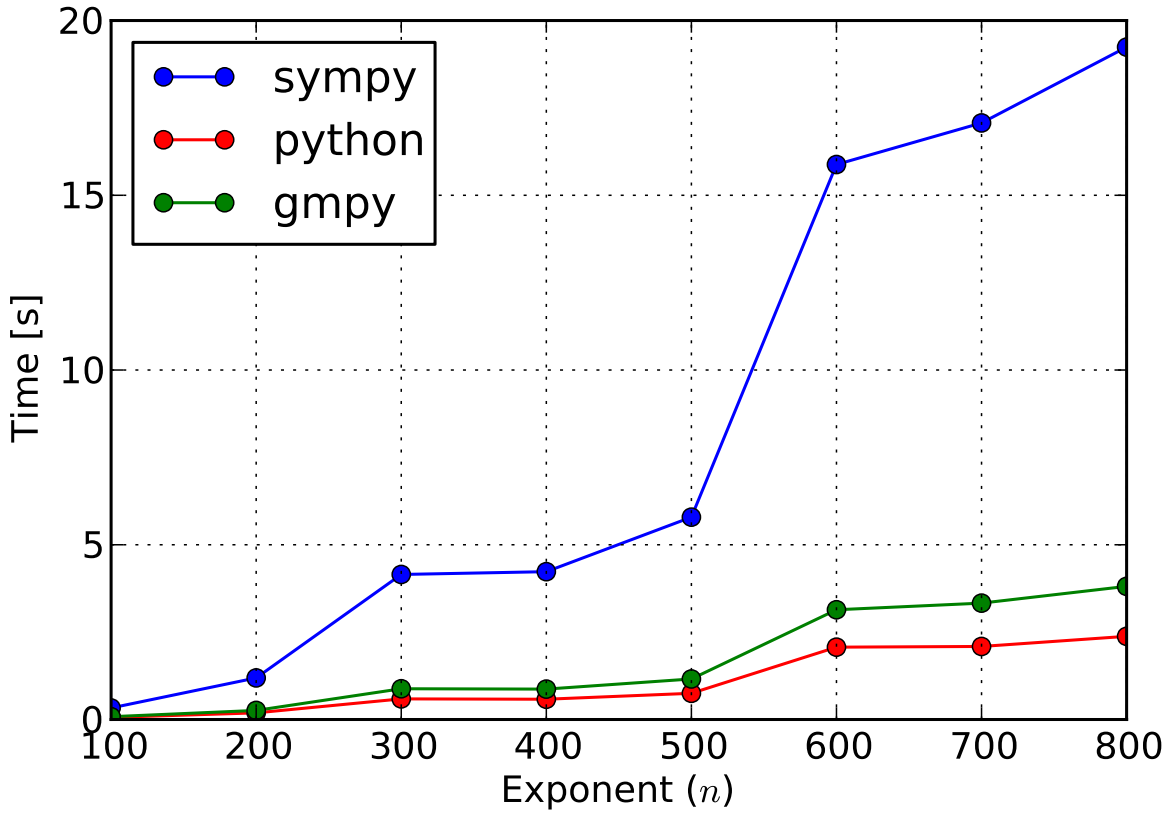


Figure: Factoring time of $x^n - 1$ polynomial

Ground types

Benchmark: Large coefficients

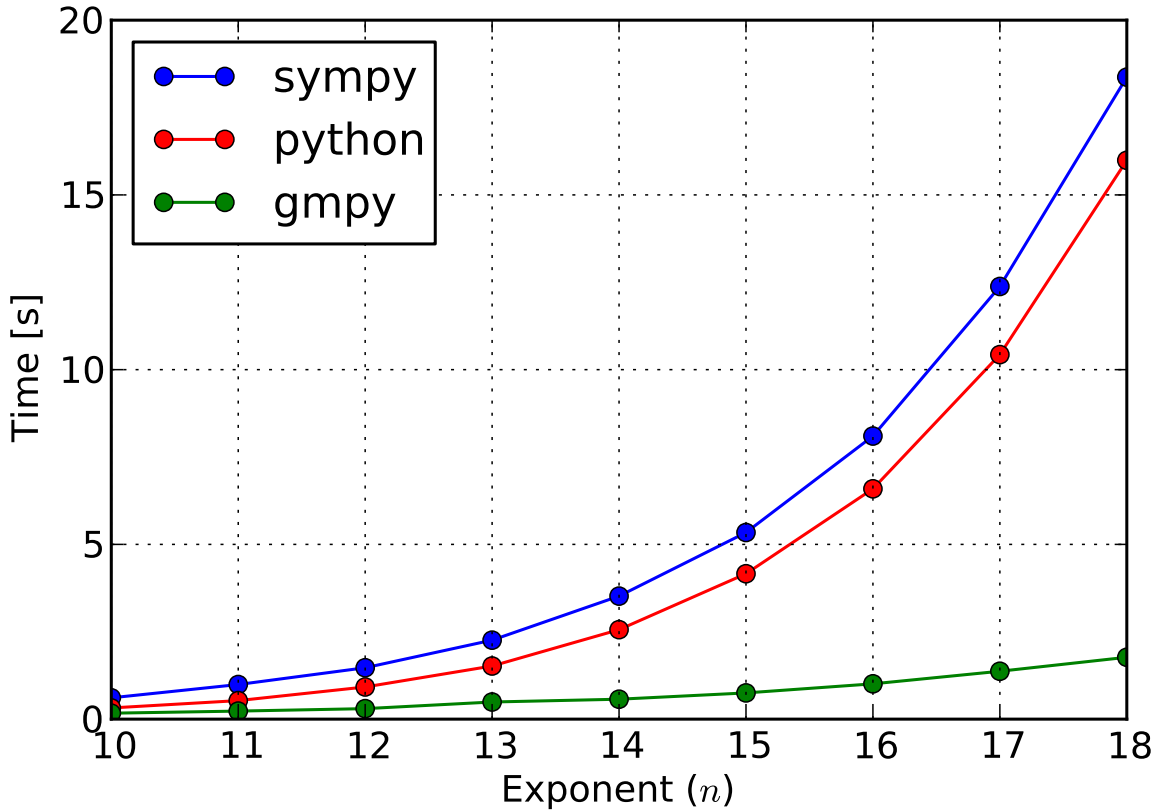


Figure: Factoring time of $(1234x + 123y + 12z + 1)^n$ polynomial

Pure mode Cython

Why want to use Cython?

- we want speed improvements
- we don't want to write C/C++ by hand

Why we want to use pure mode Cython?

- actually we don't want to write Cython either
- allows us to have a single source base
- it is very cheap to use

Pure mode Cython

Why want to use Cython?

- we want speed improvements
- we don't want to write C/C++ by hand

Why we want to use pure mode Cython?

- actually we don't want to write Cython either
- allows us to have a single source base
- it is very cheap to use

Using pure mode Cython

How to use pure mode Cython in SymPy?

- install Cython (`www.cython.org`) on your system
- annotate selected functions with `cythonized` decorator
- choose which variables should be considered as native
- type `make` in your shell and rerun SymPy

Example:

```
@cythonized('i,n')
def dup_scale(f, a, K):
    """Compute ``f(a*x)`` in ``K[x]``. """
    f, n, b = list(f), dup_degree(f), a

    for i in xrange(n-1, -1, -1):
        f[i], b = b*f[i], b*a

    return f
```

Using pure mode Cython

How to use pure mode Cython in SymPy?

- install Cython (`www.cython.org`) on your system
- annotate selected functions with `cythonized` decorator
- choose which variables should be considered as native
- type `make` in your shell and rerun SymPy

Example:

```
@cythonized('i,n')
def dup_scale(f, a, K):
    """Compute ``f(a*x)`` in ``K[x]``. """
    f, n, b = list(f), dup_degree(f), a

    for i in xrange(n-1, -1, -1):
        f[i], b = b*f[i], b*a

    return f
```

Using pure mode Cython

How to use pure mode Cython in SymPy?

- install Cython (`www.cython.org`) on your system
- annotate selected functions with `cythonized` decorator
- choose which variables should be considered as native
- type `make` in your shell and rerun SymPy

Example:

```
@cythonized('i,n')
def dup_scale(f, a, K):
    """Compute ``f(a*x)`` in ``K[x]``. """
    f, n, b = list(f), dup_degree(f), a

    for i in xrange(n-1, -1, -1):
        f[i], b = b*f[i], b*a

    return f
```

Using pure mode Cython

How to use pure mode Cython in SymPy?

- install Cython (`www.cython.org`) on your system
- annotate selected functions with `cythonized` decorator
- choose which variables should be considered as native
- type `make` in your shell and rerun SymPy

Example:

```
@cythonized('i,n')
def dup_scale(f, a, K):
    """Compute ``f(a*x)`` in ``K[x]``. """
    f, n, b = list(f), dup_degree(f), a

    for i in xrange(n-1, -1, -1):
        f[i], b = b*f[i], b*a

    return f
```

Using pure mode Cython

How to use pure mode Cython in SymPy?

- install Cython (`www.cython.org`) on your system
- annotate selected functions with `cythonized` decorator
- choose which variables should be considered as native
- type `make` in your shell and rerun SymPy

Example:

```
@cythonized('i,n')
def dup_scale(f, a, K):
    """Compute ``f(a*x)`` in ``K[x]``. """
    f, n, b = list(f), dup_degree(f), a

    for i in xrange(n-1, -1, -1):
        f[i], b = b*f[i], b*a

    return f
```

Using pure mode Cython

How cythonized decorator works?

- with Cython

```
def cythonized(specs):
    arg_types = {}

    for spec in specs.split(','):
        arg_types[spec] = cython.int

    return cython.locals(**arg_types)
```

- without Cython

```
def cythonized(specs):
    return lambda f: f
```

Using pure mode Cython

How cythonized decorator works?

- with Cython

```
def cythonized(specs):
    arg_types = {}

    for spec in specs.split(','):
        arg_types[spec] = cython.int

    return cython.locals(**arg_types)
```

- without Cython

```
def cythonized(specs):
    return lambda f: f
```

Using pure mode Cython

Benchmark: Polynomial exponentiation

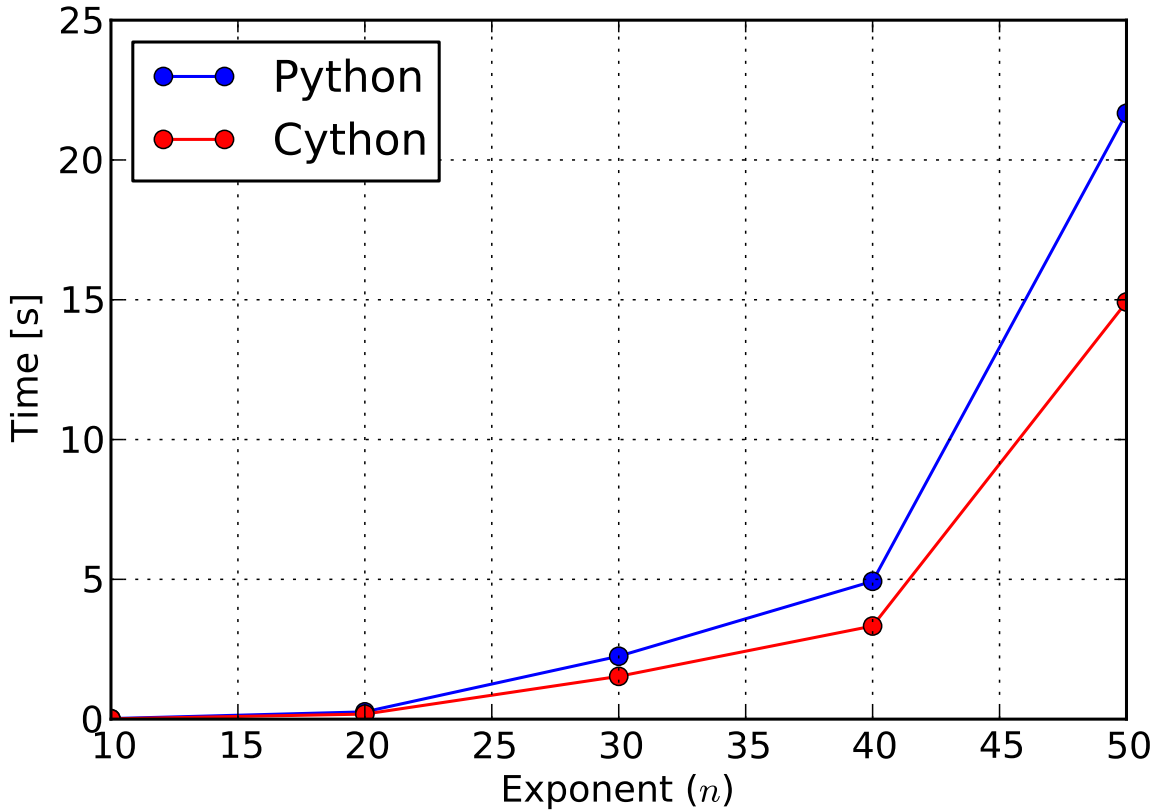


Figure: Computation time of $(27x + y^2 - 15z)^n$

Using pure mode Cython

Benchmark: Polynomial factorization

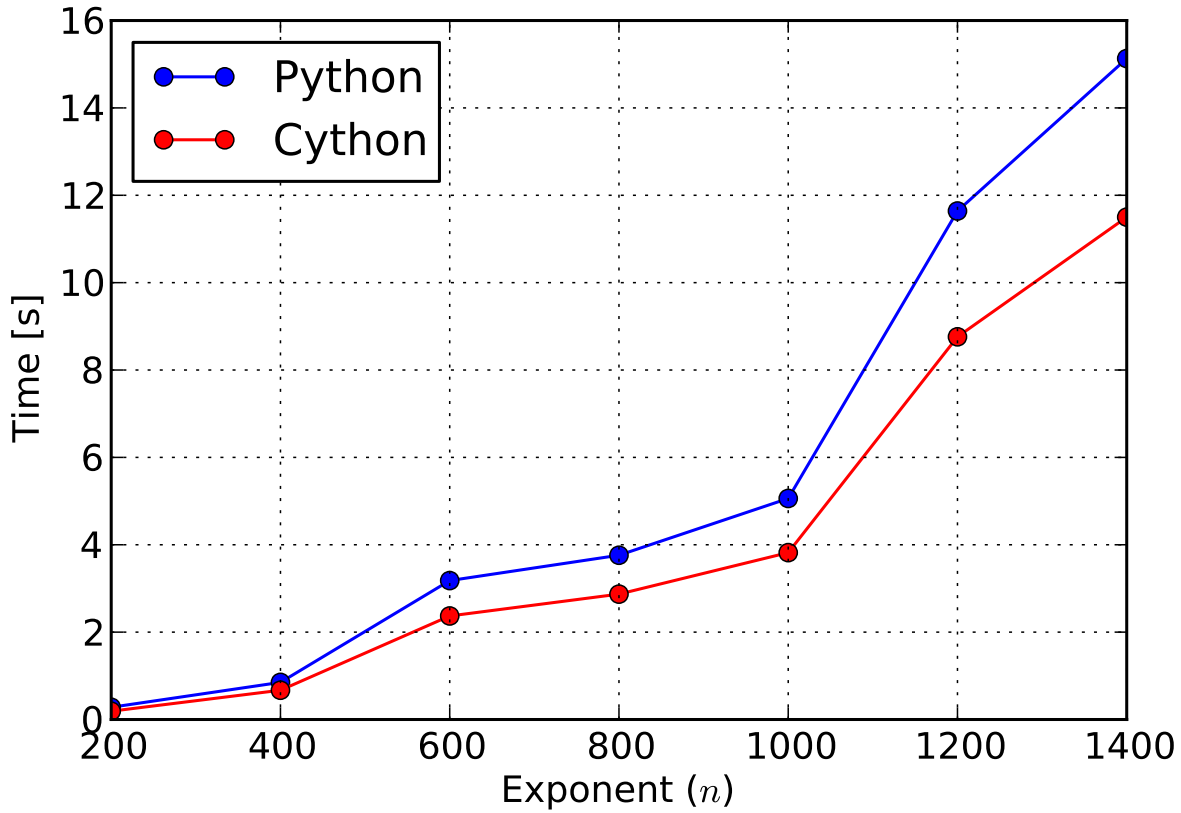


Figure: Factoring time of $x^n - 1$ polynomial

Future plans

- implement better algorithms
  - factorization, Gröbner bases, ...
- write more extensive documentation
- use the module for practical things
  - GSoC project: Algorithms for Symbolic Integration

Thank you for your attention!

Questions, remarks, discussion . . .