New features of Symbolica

Ben Ruijl

Jun 12, 2025

Symbolica

Symbolica

Symbolica (https://symbolica.io) is a symbolic-numerical framework

- Provides world-class performance
- Intuitive and easy to use
- Easy to integrate into existing software (Rust and Python API)
- Easily extendable by users
- Free for hobbyists, 1 core per device free for non-commercial use

A different way to structure programs

- In HEP it is common to daisy chain libraries
- Use files or pipes for input/output
- Parsing is error prone and slow
- An artificial split between core logic and symbolic data
- Leads to arcane data representations
  - Tree: `node(0)*node(1,0)*node(2,0)*node(3,1)`
  - Graph: `v(1,2,3)*v(1,2,5)*v(3,5)`

A different way to structure programs

- Symbolica is a library, so expressions are just another data type
- Allows for more flexible manipulation of mathematical expressions
- For example, they can be a key in a hashmap

```python
from symbolica import *
x, y = S('x', 'y')
m = {x: 1, y**2 + 1: 2}
```

Use the exact data type that you need

- `Atom` for generic expressions
- `MultivariatePolynomial` for polynomials over a ring:
  - `MultivariatePolynomial<Q, u16>`: over a rationals
  - `MultivariatePolynomial<Z, u16>`: over integers
  - `MultivariatePolynomial<FiniteField<u32>, u16>`: over a finite field
  - `MultivariatePolynomial<AlgebraicExtension, u16>`: over an extension (e.g. $Q\left(\sqrt{2}\right)$)
- `MultivariatePolynomial<Q, i16>` for ‘polynomials’ with poles
- `RationalPolynomial` for rational polynomials
- These types satisfy certain **traits**

Domains

- Rust traits can be used to abstract over nested mathematical domains
- Each subdomain has more properties than its superdomain

Domains

- Rust traits can be used to abstract over nested mathematical domains
- Each subdomain has more properties than its superdomain

rings $\supset$ commutative rings $\supset$ integral domains $\supset$ integrally closed domains $\supset$ GCD domains $\supset$ unique factorization domains $\supset$ principal ideal domains $\supset$ Euclidean domains $\supset$ fields $\supset$ algebraically closed fields

Rings

```rust
pub trait Ring {
    type Element;

    fn add(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
    fn sub(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
    fn mul(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
    fn neg(&self, a: &Self::Element) -> Self::Element;
    fn zero(&self) -> Self::Element;
    fn one(&self) -> Self::Element;
    fn nth(&self, n: u64) -> Self::Element;
    fn pow(&self, b: &Self::Element, e: u64) -> Self::Element;
    fn is_zero(a: &Self::Element) -> bool;
    fn is_one(&self, a: &Self::Element) -> bool;
}
```

Example

```
struct FiniteField {  modulus: u32 }

impl Ring for FiniteField {
    type Element = u32;

    fn add(&self, a: &u32, b: &u32) -> u32 {
        let r = *a as u64 + *b as u64;
        if r >= self.modulus as u64 {
            (r - self.modulus as u64) as u32
        } else {
            r as u32
        }
    }

    fn is_zero(a: &u32) -> bool {
        *a == 0
    }
}
```

Euclidean domains and fields

```
pub trait EuclideanDomain: Ring {
    fn rem(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
    fn gcd(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
}
```

Euclidean domains and fields

```
pub trait EuclideanDomain: Ring {
    fn rem(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
    fn gcd(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
}

pub trait Field: EuclideanDomain {
    fn inv(&self, a: &Self::Element) -> Self::Element;
    fn div(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
}
```

Abstraction example

- In an IBP program one can represent the expression $I(1,2,1,1,2)\frac{\varepsilon^2+m^2}{\varepsilon}$ as:

```
struct Term {
      indices: Vec<isize>,
      coefficient: Atom, // general expression
}
```

Abstraction example

- In an IBP program one can represent the expression $I(1,2,1,1,2)\frac{\varepsilon^2+m^2}{\varepsilon}$ as:

```
struct Term {
    indices: Vec<isize>,
    coefficient: RationalPolynomial<Z, u16>,
}
```

Abstraction example

- In an IBP program one can represent the expression $I(1,2,1,1,2)\frac{\varepsilon^2+m^2}{\varepsilon}$ as:

```
struct Term<R: EuclideanDomain> {
    indices: Vec<isize>,
    coefficient: RationalPolynomial<R, u16>,
}
```

Abstraction example

- In an IBP program one can represent the expression $I(1,2,1,1,2)\frac{\varepsilon^2+m^2}{\varepsilon}$ as:

```
struct Term<F: Field> {
    indices: Vec<isize>,
    coefficient: F::Element,
}
```

Abstraction example

- In an IBP program one can represent the expression $I(1,2,1,1,2)\frac{\varepsilon^2+m^2}{\varepsilon}$ as:

```
struct Term<F: Field> {
    indices: Vec<isize>,
    coefficient: LRUCache<F::Element>,
}
```

Row reduction

```rust
impl<T: Field> Matrix<T> {
    fn row_reduce(&mut self) -> Matrix<T> {
        for j in 0..self.n_cols {
            for i in 0..self.nrows {
                if !self.ring.is_zero(self[i, j]) {
                    for k in 0..self.n_rows {
                        if k != i {
                            let factor = self.ring.div(&self[k, j], &self[i, j]);
                            for l in 0..self.n_cols {
                                self[k, l] = self.ring.sub(&self[k, l],
                                                          self.ring.mul(&self[i, l], &factor));
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Selective normalization

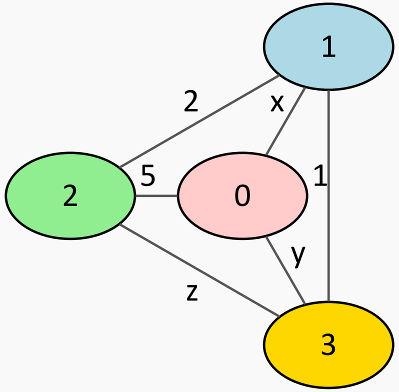
- For atoms, it is not clear what the 'normal' form is
- Some simplifications may be too slow or make the expression worse
- Customize the behaviour of atoms

```
let field = AtomField {
    statistical_zero_test: false,
    cancel_check_on_division: true,
    custom_normalization: None,
};

field.div(&parse!("x^2+2x+1"), &parse!("x+1"))
```

yields $x + 1$

Data representation

Graph



Matrix

$$\begin{pmatrix} 0 & x & 5 & y \\ x & 0 & 2 & 1 \\ 5 & 2 & 0 & z \\ y & 1 & z & 0 \end{pmatrix}$$

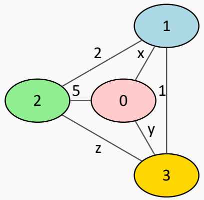
Math

$$e(0,1,x)e(0,2,5)e(0,3,y)$$
$$e(1,2,2)e(1,3,1)e(2,3,z)$$

- Different representations of the **same** data
- Different algorithms between representations

Representation in Symbolica

Graph




```
x, y, z = S('x', 'y',
'z')
g = Graph(4)
g.add_edge(0, 1, x)
g.add_edge(0, 2, 5)
g.add_edge(0, 3, y)
g.add_edge(1, 2, 2)
g.add_edge(1, 3, 1)
g.add_edge(2, 3, z)
```

Matrix

$$\begin{pmatrix} 0 & x & 5 & y \\ x & 0 & 2 & 1 \\ 5 & 2 & 0 & z \\ y & 1 & z & 0 \end{pmatrix}$$

```
x, y, z = S('x', 'y',
'z')
m = Matrix.from_nested([
  [0, x, 5, y],
  [x, 0, 2, 1],
  [5, 2, 0, z],
  [y, 1, z, 0]
])
```

Math

$$e(0,1,x)e(0,2,5)e(0,3,y)$$
$$e(1,2,2)e(1,3,1)e(2,3,z)$$

```
x, y, z, e = S('x', 'y',
'z', 'e')
r = e(0,1,x)*e(0,2,5)*
    e(0,3,y)*e(1,2,2)*
    e(1,3,1)*e(2,3,z)
```

New features

Simplified Rust API

```rust
let x = symbol!("x");
let x_sym = symbol!("xs"; Symmetric);
let b = parse!("f(x,1)");

let c = b.replace(x).with(function!(x_sym, 3, 2, 1) * x + 1);
```

Simplified Rust API

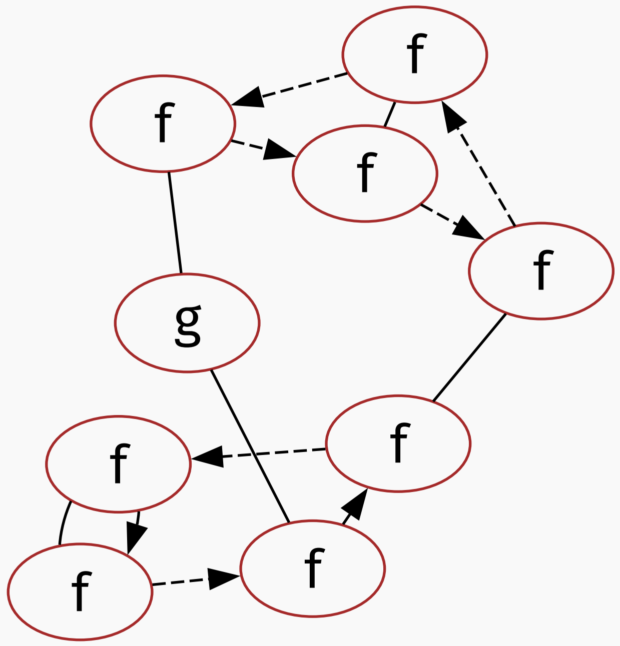
```rust
let x = symbol!("x");
let x_sym = symbol!("xs"; Symmetric);
let b = parse!("f(x,1)");

let c = b.replace(x).with(function!(x_sym, 3, 2, 1) * x + 1);
```

- Symbolica can easily be extended in Rust
- Any Rust code performs as fast as ‘native’ Symbolica code
- Manual pattern matching and replace of *x* to *z*:

```rust
let (x, y, z) = symbol!("x", "y", "z");
let expr = Atom::var(x) + y;
let result = expr.replace_map(|atom, _ctx, out| {
    if atom.get_symbol() == Some(x) { *out = z.into(); true } else { false }
});
```

Tensor symmetrization

- $f_{\text{cyc}}(\mu, \nu, \rho, \nu) f_{\text{cyc}}(\rho, \sigma, \sigma, \delta) g(\mu, \delta) =$ $f_{\text{cyc}}(\alpha, \gamma, \alpha, \mu) f_{\text{cyc}}(\varepsilon, \varepsilon, \delta, \gamma) g(\mu, \delta)$?
- Canonize (cycle)symmetric tensors using graphs
- Also works for nested tensors



ASM code generation

- Generate high quality ASM code for expression evaluation
- Bypasses long C++ compilation

```
void sigma(const std::complex<double> *params, std::complex<double> *out) {
  __asm__(
    "movupd 208(%2), %%xmm1\n\t"
    "movupd 224(%2), %%xmm2\n\t"
    "movapd %%xmm1, %%xmm0\n\t"
    "unpckhpd %%xmm0, %%xmm0\n\t"
    "unpcklpd %%xmm1, %%xmm1\n\t"
    "mulpd %%xmm2, %%xmm0\n\t"
    "mulpd %%xmm2, %%xmm1\n\t"
```

Namespaces

- Each defined symbol has a namespace
- This prevents name clashes and makes it easy to use multiple libraries
- By default it is the project where it is defined/parsed
- In Rust, the macros `symbol!` and `parse!` get the project name and the call line
- `symbol!("x")` defines `yourproject::x`
- `symbol!("dirac::x")` defines `dirac::x`
- To use a function `gamma` from the `dirac` library, write `dirac::gamma`

Namespaces in FORM?

dirac.frm:

```
#namespace "dirac"; * set namespace for this file
S x;
CF gamma;

#procedure gammasimplify()
  id gamma(x?, x?) = d_(x,x);
#endprocedure
```

test.frm:

```
S gamma; * default namespace is filename, so test::gamma
#include "dirac.frm"

L F = gamma(1,1) + dirac::gamma(2,2);
#call dirac::gammasimplify()
```

Other new features

- Evaluation output as tree of operations
- Floating point arithmetic with error tracking
- Graph generation
- Complex number coefficients: $2x + ix = (2 + i)x$
- Custom normalization and printing

Demo time