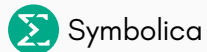


Symbolica: a modern computer algebra system

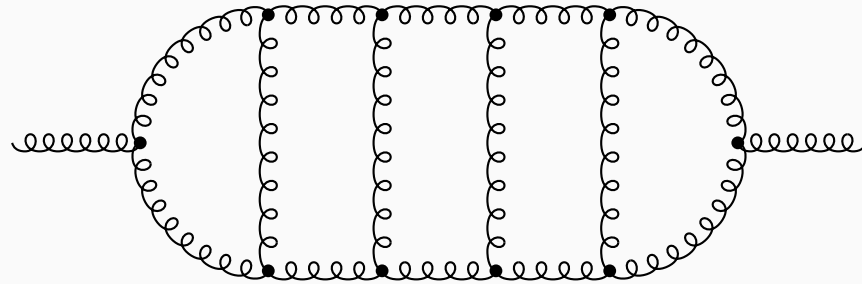
Turning particle-physics code into sustainable research software

Ben Ruijl · Ruijl Research · Swiss RSE Day 2026

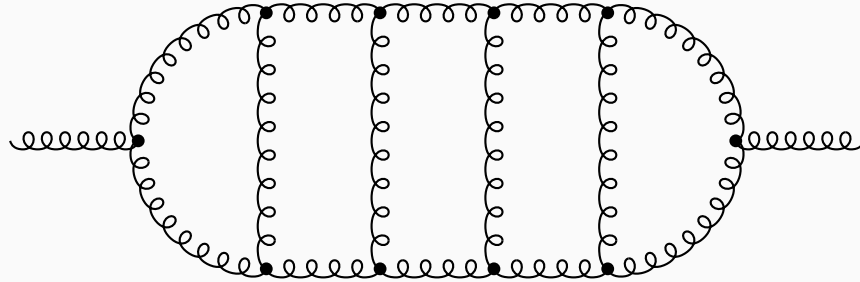
Aug 31, 2026



- Former particle physicist, making predictions for collider experiments
- Computed Feynman diagrams using custom computer algebra software



- Former particle physicist, making predictions for collider experiments
- Computed Feynman diagrams using custom computer algebra software



- Diagram above represents 12 million integrals!
- Predictions take months of computing

Developing software inside academia

- Most research software is written by non-coders, for their own research
- Publishing software not greatly rewarded, maintaining it even less so
- The 'best' software is often kept secret
- Writing a long-lived, user-friendly library requires planning and another funding model



Quanta**magazine**



PARTICLE PHYSICS

Crucial Computer Program for Particle Physics at Risk of Obsolescence

By MATT VON HIPPEL

December 1, 2022

Maintenance of the software that's used for the hardest physics calculations rests almost entirely with a retiree. The situation reveals the problematic incentive structure of academia.

Goals for Symbolica

- From-scratch computer algebra system in modern language
- Heavy focus on performance and low memory footprint
- No new scripting language: library in Rust and Python
- Make use of notebooks and existing ecosystems (Jupyter, Marimo, WASM)
- Straightforward API and sane defaults
- Convenient I/O and pretty printing

Source-available model

- Source available and open development; selected components are MIT-licensed
- Free for hobbyists, one core per instance free for non-commercial use
- Academic/commercial licenses fund continued maintenance and support
 - Grant models are too unpredictable for long-term sustainability
- Community extensions provide an open contribution surface
 - Free use of standalone packages that depend on Symbolica

ETH zürich



u^b UNIVERSITÄT
BERN



Symbolica used in published calculations

- Users report >10x performance over Mathematica

2026

Kermanschah & Vicini

Two-loop diphoton and triphoton production via quark loops

JHEP 03 (2026) 004 • arXiv:2510.18801

Kermanschah & Vicini

Axial triangles in $gg \rightarrow Z\gamma$ at two loops in QCD

arXiv:2603.03171

Naterop & Stoffer

LEFT at two loops: dimension-six operators

JHEP 02 (2026) 016 • arXiv:2507.08926

Capatti, Fraaije, Hirschi, Huber, Ruijl & Shao

NNLO $\gamma\gamma \rightarrow Q\bar{Q}$ from Local Unitarity

JHEP 03 (2026) 068 • arXiv:2509.13439

2025–2024

Naterop & Stoffer

LEFT at two loops: dimension-six baryon-number-violating operators

JHEP 07 (2025) 237 • arXiv:2505.03871

Naterop & Stoffer

LEFT at two loops: dimension-five effects

JHEP 06 (2025) 007 • arXiv:2412.13251

Davies, Schönwald & Steinhauser

Three-loop large- N_c virtual corrections to $gg \rightarrow HH$

JHEP 08 (2025) 192 • arXiv:2503.17449

Smirnov & Zeng

FIRE 6.5: Feynman integral reduction with a new simplification library

Computer Physics Communications 302 (2024) 109261

Straightforward API: series expansion

Python

```
from symbolica import *  
x = S('x')  
a = (1-x.cos())/x.sin()  
a.series(x, 0, 4)
```

Rust

```
use symbolica::prelude::*;  
let x = symbol!("x");  
let a = parse!("(1-cos(x))/sin(x)");  
a.series(x, 0, 4).unwrap()
```

$$\frac{1}{2}x + \frac{1}{24}x^3 + \mathcal{O}(x^5)$$

Fast expression evaluation

```
x, y, z, f, g = S('x', 'y', 'z', 'f', 'g')
e = x + 22/7 + x.cos() + f(g(x+1), x*2)
ev = e.evaluator([x], {(f, (y, z)): y**2 + z**2*y**2, (g, (y,)): y * 5})
ev.evaluate([[5.]])
```

- Fast bridge between symbolic manipulation and numerical evaluation
- Rewrite using Horner schemes
- Remove common subexpressions
- Generate inline ASM code + JIT compile
- Near-instant compilation: gcc and clang can take weeks!
- 1000x faster than a naive evaluation, same quality as -O3

```
Z[0] = x;
Z[1] = 1;
Z[2] = 2;
Z[3] = 5;
Z[4] = 22/7;
Z[5] = Z[0] + Z[1]
Z[5] = Z[3] * Z[5]
Z[6] = Z[0] * Z[2]
Z[5] = Z[5] * Z[5]
Z[6] = Z[5] * Z[6] * Z[6]
Z[5] = Z[5] + Z[6]
Z[6] = cos(Z[0])
Z[6] = Z[0] + Z[4] + Z[5] + Z[6]
```

Rust + Python

Why Rust

- C++-level performance, with memory errors caught at compile time

```
std::string* p;
{
    std::string value = "hello";
    p = &value;
} // value is destroyed

std::cout << *p; // undefined behavior
```

```
let p;
{
    let value = String::from("hello");
    p = &value;
}
println!("{p}");
error[E0597]: `value` does not live long enough
5 |         p = &value;
  |             ^^^^^^ borrowed value does not live long
  |             enough
6 |     }
  |     - `value` dropped here while still borrowed
7 |     println!("{p}");
  |         - borrow later used here
```

- Traits allow for generic algorithms without runtime overhead
- Easily create python bindings with `pyo3`

The importance of typed Python

- Auto-completion and inline documentation increase productivity
- Modern LSPs like `ty` are very fast, but work best with typed APIs
- Use stub file `.pyi` to provide type hints for shared object files
- Golden rule: use one output type
- Use `@overload` in `.pyi`

Computational graphs in Python

- Make Python API as fast as Rust by creating a computational graph:

```
x_, f = S('x_', 'f')
```

```
t = T().repeat(  
    T().expand(),  
    T().replace(f(x_), f(x_ - 1) + f(x_ - 2), x_ > 1)  
)
```

```
t(f(100)) # execute the transformation chain
```

- A transformer creates an AST that is executed in Rust

Why is computer algebra hard?

Why is computer algebra hard?

- Let us look at a simple example: row reduction
- Used for solving linear systems

$$\begin{pmatrix} 6. & 5. \\ 2. & 3. \end{pmatrix} \rightarrow \begin{pmatrix} 6. & 5. \\ 0. & 1.3333 \end{pmatrix}$$

- Many linear algebra packages around for this
-

Why is computer algebra hard?

- Let us look at a simple example: row reduction
- Used for solving linear systems

$$\begin{pmatrix} 6. & 5. \\ 2. & 3. \end{pmatrix} \rightarrow \begin{pmatrix} 6. & 5. \\ 0. & 1.3333 \end{pmatrix}$$

- Many linear algebra packages around for this
- Let us try to do it exactly:

$$\begin{pmatrix} 6 & 5 \\ 2 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 6 & 5 \\ 0 & ? \end{pmatrix}$$

$$3 - \frac{2}{6}5 = \frac{3}{1} - \frac{10}{6} = \frac{3 \cdot 6 - 10 \cdot 1}{1 \cdot 6} = \frac{8}{6} \stackrel{\text{gcd}}{=} \frac{4}{3}$$

Why is computer algebra hard?

- What if we add a parameter?

$$\begin{pmatrix} t^2 + 3t + 2 & 1 + t \\ 2 + t & t + 4 \end{pmatrix} \rightarrow \begin{pmatrix} t^2 + 3t + 2 & 1 + t \\ 0 & ? \end{pmatrix}$$

$$t + 4 - \frac{2 + t}{t^2 + 3t + 2}(1 + t) = \frac{t^3 + 6t^2 + 11t + 6}{t^2 + 3t + 2} \stackrel{\text{gcd}}{=} t + 3$$

- Much more bookkeeping
- How is this greatest common divisor computed?

Euclidean GCD algorithm

$$a = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$

$$b = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

Euclidean GCD algorithm

$$a = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$

$$b = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

$$a \bmod b = r_1 = -\frac{5}{9}x^2 + \frac{1}{9}x^2 - \frac{1}{3}$$

$$b \bmod r_1 = r_2 = -\frac{117}{25}x^2 - 9x + \frac{411}{25}$$

$$r_1 \bmod r_2 = r_3 = -\frac{102500}{6591} + \frac{233150}{19773}x$$

$$r_2 \bmod r_3 = r_4 = -\frac{1288744821}{543589225}$$

- Coefficients are rational numbers now...

Euclidean algorithm with pseudo-remainders

Pseudo-remainders avoid rational numbers but lead to huge coefficients:

$$r_1 = -15x^4 + 3x^2 - 9$$

$$r_2 = 15795x^2 + 30375x - 5953$$

$$r_3 = 1254542875143750x - 1654608338437500$$

$$r_4 = 12593338795500743100931141992187500$$

- Main challenge: avoid intermediate expression swell
- Requires carefully crafted code and algorithms
- Still, code should be generic

Mathematical traits: Rust abstractions

```
pub trait Ring {  
    type Element: Clone + PartialEq + Eq;  
  
    fn add(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
    fn mul(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
}
```

Mathematical traits: Rust abstractions

```
pub trait Ring {  
    type Element: Clone + PartialEq + Eq;  
  
    fn add(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
    fn mul(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
}  
  
pub trait EuclideanDomain: Ring {  
    fn quot(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
    fn rem(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
}
```

Mathematical traits: Rust abstractions

```
pub trait Ring {  
    type Element: Clone + PartialEq + Eq;  
  
    fn add(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
    fn mul(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
}  
  
pub trait EuclideanDomain: Ring {  
    fn quot(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
    fn rem(&self, a: &Self::Element, b: &Self::Element) → Self::Element;  
}  
  
pub trait Field: EuclideanDomain {  
    fn inv(&self, a: &Self::Element) → Self::Element;  
}
```

Integers

```
pub struct Z {} // empty structure representing the ring of integers
```

```
impl Ring for Z {  
    type Element = Integer; // an arbitrary-precision integer  
    fn add(&self, a: &Integer, b: &Integer) → Integer {  
        a + b  
    }  
    fn mul(&self, a: &Integer, b: &Integer) → Integer {  
        a * b  
    }  
}
```

```
impl EuclideanDomain for Z { ... }
```

- Cannot implement Field, as $1/2$ is not an integer

Polynomials

A polynomial over a ring is an element of a polynomial ring!

```
struct PolynomialRing<R: Ring> {  
    coeff_ring: R,  
}  
  
impl<R: Ring> Ring for PolynomialRing<R> {  
    type Element = Polynomial<R>;  
}
```

Polynomials

A polynomial over a ring is an element of a polynomial ring!

```
struct PolynomialRing<R: Ring> {  
    coeff_ring: R,  
}  
  
impl<R: Ring> Ring for PolynomialRing<R> {  
    type Element = Polynomial<R>;  
}
```

Polynomials with field coefficients are a Euclidean domain:

```
impl<R: Field> EuclideanDomain for PolynomialRing<R> { ... }
```

Generic algorithms

- The Euclidean algorithm can be implemented generically for any Euclidean domain

```
fn gcd<R: EuclideanDomain>(ring: &R,  
                           mut a: R::Element,  
                           mut b: R::Element) → R::Element  
{  
    while !ring.is_zero(&b) {  
        let r = ring.rem(&a, &b);  
        (a, b) = (b, r);  
    }  
    a  
}
```

Representing an expression

Tree representation?

```
enum Atom {  
    Number(i64, i64),  
    Variable(u64),  
    Function(u64, Vec<Atom>),  
    Pow(Box<(Atom, Atom)>),  
    Mul(Vec<Atom>),  
    Add(Vec<Atom>),  
}
```

- Advantage: easy for pattern matching in Rust
- Disadvantage: takes 32 bytes per node and fragments in memory
- We want to store millions of terms in memory
- $xyf(1, 2, x^2)$ takes up $9 \cdot 32 = 288$ bytes

Compact linear representation

- Use tag-length-value format to store a tree in a byte array
- Compress size of two numbers into a single byte, followed by the numbers
- $f(x, \frac{2}{5})$ is coded in only 15 bytes as

$$\left\{ \overbrace{3}^F, \overbrace{10, 0, 0, 0}^L, 17, \underbrace{1}_f, \underbrace{2}_V, \underbrace{2}_V, \underbrace{1}_V, \underbrace{0}_x, \overbrace{1}^N, 17, 2, 5 \right\}$$

- Byte array is never exposed in logic, it is wrapped in Rust structs

Artificial intelligence usage

- AI shines when correctness is cheap to verify
- Porting and improving established code
- Exploratory API work and refactoring made easier
- Optimizing against tests and benchmarks

Watch for

- Bug workarounds and niche solutions
- Proper attribution and unreliable references

AI application: porting Rubi to Symbolica

- Translated Rubi's 7000+ symbolic integration rules from Mathematica to Symbolica
- Simple rule:

```
Int[a_/x_,x_Symbol] := a*Log[x] /; FreeQ[a,x]
```

AI application: porting Rubi to Symbolica

- Translated Rubi's 7000+ symbolic integration rules from Mathematica to Symbolica
- Simple rule:

```
Int[a_/x_,x_Symbol] := a*Log[x] /; FreeQ[a,x]
```

- Complex rule:

```
Int[x_^m_.*(a_.+b_.*ArcCot[c_.*x_^n_])^p_,x_Symbol] :=  
With[{k=Denominator[m]},  
k \[Star] Subst[Int[x^(k*(m+1)-1)*(a+b*ArcCot[c*x^(k*n)])^p,x],x,x^(1/k)]] /;  
FreeQ[{a,b,c},x] && IGtQ[p,1] && IGtQ[n,0] && FractionQ[m]
```

AI application: porting Rubi to Symbolica

- Translated Rubi's 7000+ symbolic integration rules from Mathematica to Symbolica
- Simple rule:

```
Int[a_/x_,x_Symbol] := a*Log[x] /; FreeQ[a,x]
```

- Complex rule:

```
Int[x_^m_.*(a_.+b_.*ArcCot[c_.*x_^n_])^p_,x_Symbol] :=  
With[{k=Denominator[m]},  
k \[Star] Subst[Int[x^(k*(m+1)-1)*(a+b*ArcCot[c*x^(k*n)])^p,x],x,x^(1/k)]] /;  
FreeQ[{a,b,c},x] && IGtQ[p,1] && IGtQ[n,0] && FractionQ[m]
```

- Correctness verified by tracing the rules in Mathematica and forcing the same path
- Would have taken months by hand, done by AI in two weeks

Results

- `symbolica-integrate`: 111.24 s
- Rubi 4.17.3.0 in Mathematica 13.2: 155.78 s

Results

- symbolica-integrate: 111.24 s
- Rubi 4.17.3.0 in Mathematica 13.2: 155.78 s

```
x = S('x')
result, overview, steps = (x/(x+1)).integrate_with_steps(x)
print(overview)
```

yields:

$$\begin{aligned}\int x/(1+x) \, dx &= \int 1-1/(1+x) \, dx \\ \int 1-1/(1+x) \, dx &= \int 1 \, dx + \int 1/(-1-x) \, dx \\ \int 1 \, dx &= x \\ \int -1/(1+x) \, dx &= -\log(1+x) \\ &= x - \log(1+x)\end{aligned}$$

Fast, easy to use, and built to last

Rust core

Performance, memory safety, and reusable mathematical abstractions

Python API

Familiar notebooks with autocomplete and inline documentation

Sustainable

Open development, published use, verifiable AI

Try Symbolica (symbolica.io)

PYTHON

```
$ pip install symbolica
```

RUST

```
$ cargo add symbolica
```