Idenso

Symbolic tensor identities for Spenso and Symbolica

1 Overview

Idenso is the symbolic identity and simplification layer for tensors encoded in the form that Spenso can parse from Symbolica expressions. It provides representation symbols, index tooling, reversible "cooking" of subexpressions, selective expansion, and identities for Dirac, metric, epsilon, and color algebra.

> **Symbolic, not component expansion**
> Idenso rewrites abstract tensor expressions. Functions such as `expand_mink` distribute factorized terms that carry selected index families; they do not turn every tensor into a dense array of components. Concrete tensor storage and network execution belong to Spenso.

1.1 Choose a task

- To install Symbolica and simplify one metric contraction with its bundled community modules, follow the Python guide.
- To apply the same identity through the native API, use the Rust guide.
- To verify the rewrite as a controlled, observable pass, follow the controlled identity tutorial and the Python function reference.
- To isolate dummy-index namespaces or cook a large expression, use the algebra guide with the exact `IndexTooling` and `Cookable` Rustdoc.
- To audit a sign or normalization in a Dirac/color pass, use the source-backed shipped color and Dirac convention reference.

1.2 A controlled rewrite pipeline

There is no universally correct "simplify everything" order. A robust workflow makes each phase explicit:

- initialize the standard representations and tensor symbols;
- inspect dangling indices and normalize or wrap dummy-index namespaces when combining expressions;
- expand only the sectors needed by the next identity pass;
- simplify gamma, metric, or color structures as appropriate;
- canonicalize and compare results using the conventions of the consuming calculation.

Expansion can grow an expression quickly, so perform it as late and selectively as possible. Wrapping indices is especially important before multiplying independently constructed expressions: equal printed index names can otherwise acquire an unintended contraction.

1.3 Representations and cooking

The representation layer defines spin-fundamental, color-fundamental, color-sextet, bispinor, and color-adjoint types together with their duality. Importing the published Idenso community module registers the related Symbolica symbols and Spenso tags before expressions are built.

The `Cookable` API can replace selected function-like subexpressions or index payloads with compact symbols and later reverse the operation. Use reversible settings when the original expression must be recovered. A cooked symbol is an intermediate encoding, not a portable serialized physics result unless the associated mapping and settings are retained.

Idenso is consumed by GammaLoop and its tensor conventions are shared with Spenso. Continue to the interface guide, native Rustdoc, and generated Python reference for supported functions, signatures, and feature gates.

Contributors changing representation syntax, symbolic-network parsing, index ownership, or rewrite order should also read the source-audited Idenso implementation architecture.

2 Simplify your first tensor expression

Idenso supplies representation-aware identities through both Symbolica's Python community module and a native Rust crate. Choose the interface that already owns your symbolic expressions; both routes simplify a Minkowski metric contraction while preserving its one free index.

Python
Use the bundled community module for notebooks, exploratory algebra, and Python pipelines. This is the shortest supported installation path.

Use Idenso from Python →

Rust
Use the crate when a Rust application owns Symbolica expressions and should compose Idenso's metric, Dirac, color, or cooking passes directly.

Use Idenso from Rust →

The interface guide maps both routes to the same identity families and explains their feature and registration boundaries.

3 Using Idenso from Python

Symbolica `2.2.0` bundles Idenso and Spenso as native community modules. This example contracts one four-dimensional Minkowski metric with a vector and verifies the exact remaining expression.

3.1 Install and verify the modules

```
python -m venv .venv
. .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install "symbolica==2.2.0"
python -c "import symbolica.community.idenso; import symbolica.community.spenso"
```

There is no standalone `idenso` Python wheel. Import the community module before constructing expressions, and follow Symbolica's installation and license terms.

3.2 Simplify one metric contraction

Save this as `idenso_quickstart.py`:

```
from symbolica.community.idenso import initialize, list_dangling, simplify_metrics
from symbolica.community.spenso import Representation, TensorName

initialize()
rep = Representation.mink(4)
mu, nu = rep("mu"), rep("nu")
g, q = TensorName.g(), TensorName("q")
```

```
expression = g(mu, nu) * q(mu)
reduced = simplify_metrics(expression)

assert reduced == q(nu).to_expression()
assert len(list_dangling(reduced)) == 1
print(reduced)
```

Run `python idenso_quickstart.py`. Success means the explicit metric disappears and `q(nu)` remains. The structural equality check is stronger than comparing printed text, whose formatting may change between Symbolica releases.

> **Make rewrite stages observable**
> Check dangling indices before and after each pass. When multiplying expressions built with independent dummy-index namespaces, wrap those namespaces before simplification so equal printed names do not create an accidental contraction.

Continue with the controlled identity tutorial for a verification workflow, then use the algebra guide to compose metric, Dirac, color, and cooking stages deliberately.

4 Using Idenso from Rust

This route constructs the same indexed metric contraction as the Python example, applies Idenso's native `MetricSimplifier`, and checks the exact reduced Symbolica expression.

> **Use the current Git API until the next crate release**
> The published `idenso 0.3.0` predates the current module layout and uses an older Symbolica dependency. Mixing its API with the current manual can create incompatible `Atom` types. The pinned Git dependency below matches this documentation; switch to the next published Idenso version once it carries the same API.

4.1 Create a Rust project

Use Rust 1.85 or newer:

```
cargo new idenso-quickstart
cd idenso-quickstart
cargo add idenso \
  --git https://github.com/alphal00p/gammaloop.git \
  --rev 41e0eddb39c7c668074af483ac1d566e46c247a8
cargo add symbolica@2.2.0 --no-default-features
```

Replace `src/main.rs` with:

```
use idenso::shorthands::metric::MetricSimplifier;
use symbolica::parse_lit;

fn main() {
    idenso::representations::initialize();

    let expression = parse_lit!(
        spenso::g(spenso::mink(4, 0), spenso::mink(4, 1))
            * p(spenso::mink(4, 1))
    );
    let reduced = expression.simplify_metrics();

    assert_eq!(reduced, parse_lit!(p(spenso::mink(4, 0))));
```

```
    println!("{reduced}");
}
```

Run `cargo run`. Success means the assertion passes and the vector retains index `0` after the metric contracts index `1`. Symbolica's license terms apply to the native crate as well as the Python module.

Continue with the algebra guide before composing multiple identity families or multiplying expressions with independently allocated dummy indices.

5 Tutorial

This tutorial uses Idenso's Python community module to contract one four-dimensional Minkowski metric tensor with a vector. It is intentionally small: the point is to establish the registered tensor syntax and observe one algebra pass before composing a larger Dirac or color pipeline.

For the shortest installation and first-run path, begin with Using Idenso from Python. Then return here to inspect the identity pass and extend it into an algebra pipeline.

5.1 Prerequisites

Idenso is mounted at `symbolica.community.idenso`; it is not installed by `pip install idenso`. Install the published Symbolica distribution, which bundles the Idenso and Spenso community modules, and verify the actual environment first:

```
python -m pip install --upgrade symbolica
python -c "import symbolica.community.idenso; import symbolica.community.spenso"
```

If that fails, check the installed Symbolica version before continuing. Generating a `.pyi` file does not mount the native module. Source embedders can build a custom community assembly.

5.2 Simplify one metric contraction

Save the following as `metric_first.py`:

```
from symbolica.community.idenso import list_dangling, simplify_metrics
from symbolica.community.spenso import Representation, TensorName

rep = Representation.mink(4)
mu = rep("mu")
nu = rep("nu")

g = TensorName.g()
q = TensorName("q")
expression = g(mu, nu) * q(mu)

free_before = list_dangling(expression)
reduced = simplify_metrics(expression)
free_after = list_dangling(reduced)

assert len(free_before) == 1
assert len(free_after) == 1
print("reduced:", reduced)
```

Run `python metric_first.py`. Success means both rank assertions pass, the metric is removed from the reduced expression, and the result is a rank-one expression carrying `nu`. The rewrite is abstract: the registered Minkowski metric supplies index compatibility, but this example does not substitute a numerical signature. Printed output can vary with the installed Symbolica version; inspect the expression structure instead of comparing exact text.

Verification scope and cost
The docs harness compiles this Python source without importing native modules; it syntax-checks the environment probe without running it. Execute both steps in a provisioned community-module environment to verify the rank-one invariant. The script should take seconds after startup; building or installing Symbolica community modules is the expensive prerequisite and may take substantially longer.

Import before constructing expressions
Importing `symbolica.community.idenso` registers Idenso's representation and tensor symbols. Construct the Spenso-compatible expression after that import so Idenso's matchers see the intended slots and tensor names.

5.3 Grow this into an algebra pipeline

Keep transformations observable while developing:

- call `list_dangling` before and after a pass to catch accidental contractions;
- use `wrap_dummies` before multiplying expressions built in independent index namespaces;
- expand only the Minkowski, bispinor, or color sector needed by the next pass;
- apply `simplify_metrics`, `simplify_gamma`, and `simplify_color` as distinct phases;
- canonicalize only after the physics-specific identities required by the calculation are explicit.

5.4 Troubleshooting and next steps

- `ModuleNotFoundError` for `symbolica.community.idenso` means the installed Symbolica build does not include the native module; a `.pyi` type stub or source checkout alone is not enough.
- If the metric remains unchanged, construct its slots with the same `Representation` and abstract index objects as the vector. Plain Symbolica functions do not automatically carry Spenso tensor structure.
- If a free index disappears unexpectedly, inspect repeated names and use `wrap_dummies` before combining separately constructed expressions.
- Continue with the syntax-and-algebra manual, then use the Python and Rust API references for signatures and feature gates.

6 Syntax, indices, and algebra passes

Idenso consumes the Symbolica function form emitted for Spenso structures. Function heads carry representation meaning and their arguments carry abstract indices or tensor data. Import the Idenso community module before parsing so Symbolica attributes, Spenso tags, and dual representations are registered in one deterministic order.

6.1 Indices and cooking

Free indices describe the result; repeated compatible indices describe contractions. Before combining independently built expressions, wrap or rename dummy-index namespaces so equal printed names do not create accidental contractions. Cooking temporarily replaces selected index payloads or function subexpressions with compact symbols. Retain the generated map and settings whenever uncooking is required.

Cooking is especially useful before expensive canonicalization, but it changes what downstream matchers can see. Uncook before applying an identity whose pattern depends on the hidden tensor head or index structure.

6.2 Keep independent dummy namespaces independent

Two factors may legitimately print the same local dummy name before they are multiplied. Wrap each factor with a distinct header first, while leaving its external indices untouched:

```python
import symbolica as sp
from symbolica.community.idenso import list_dangling, wrap_dummies
from symbolica.community.spenso import Representation, TensorName

rep = Representation.euc(3)
mu = rep("mu")
nu = rep("nu")
rho = rep("rho")
g = TensorName.g()
p = TensorName("p")
q = TensorName("q")

left = g(mu, nu) * p(mu)
right = g(mu, rho) * q(mu)
safe_product = wrap_dummies(left, sp.S("lhs")) * wrap_dummies(right, sp.S("rhs"))

assert len(list_dangling(safe_product)) == 2
```

The stable invariant is two free indices, `nu` and `rho`; the two occurrences of local `mu` belong to separate contractions after wrapping. The generated `wrap_dummies` reference records the Python signature, while the exact `IndexTooling` Rustdoc covers the underlying Rust boundary.

> **Interpret index failures before simplifying**
> More or fewer than two dangling indices means a name collided or a slot's representation or duality differs from the intended one. An unchanged plain Symbolica function means it was not constructed through Spenso tensor names/representations, or the Idenso module was imported only after parsing. Correct those structural issues before metric, Dirac, or color simplification.

6.3 Metric and epsilon operations

Metric contraction raises, lowers, or identifies compatible Lorentz indices according to the registered representation. Epsilon identities depend on dimension, ordering, and sign conventions; expand them only when the next pass benefits from the larger expression. Keep Minkowski expansion selective to avoid distributing unrelated scalar factors.

6.4 Dirac and color algebra

Dirac passes simplify gamma chains, traces, slashes, and spinor-compatible contractions. Color passes handle fundamental/adjoint deltas, generators, structure constants, and registered group parameters. Apply one algebra family at a time and inspect the intermediate expression; an all-at-once fixed-point loop can obscure which convention produced a sign or normalization.

> **Canonical does not mean physically equivalent by itself**
> Canonical ordering makes structurally equivalent expressions comparable under the registered rules. On-shell relations, gauge choices, dimension-specific identities, and model parameter substitutions must still be requested explicitly.

6.5 Schoonschip network parsing

The Schoonschip path parses large gamma-chain expressions into the same Spenso-compatible network model before contraction. It resolves aliases and normalizes the expression before constructing the network. Spenso then plans and executes contractions, while Idenso supplies the representation and algebra rules. Avoid substituting scalar parameters too early: doing so can substantially increase intermediate expression size even when the resulting tensor network is unchanged.

Concrete syntax and rewrite cases are documented in the Spenso/Symbolica syntax note and the rendered shipped color and Dirac convention reference. The Schoonschip parsing guide shows how normalization, network construction, and contraction fit together.

Tensor storage and execution remain owned by Spenso.

7 Shipped color and Dirac rules

This page states the conventions implemented by Idenso's public simplifiers. It is deliberately shorter than the implementation source map: a physicist should be able to decide whether a rule applies without reading FORM internals, proposed patterns, or historical validation commands.

7.1 Input and scope

Idenso rewrites representation-aware Spenso expressions after `initialize()`. A printed function that merely resembles `gamma`, `t`, or `f` is not enough: its Lorentz, bispinor, fundamental, and adjoint slots must carry the registered representations and compatible dimensions. Unrecognized or unsupported structures remain in the expression rather than acquiring tensor meaning from their names.

The public Python entry points use the default `GammaSimplifySettings` and `ColorSimplifySettings`. Rust callers can choose other settings, so record those settings when a normal form is part of a reproducible result.

7.2 Dirac convention

For compatible Lorentz and bispinor representations, Idenso uses the Clifford relation

$$\{\gamma^\mu, \gamma^\nu\} = 2\eta^{\mu\nu}1.$$

The metric object and its signature belong to the registered Spenso representation; the simplifier does not silently replace it by a numerical diagonal metric. Ordinary gamma chains support adjacent contractions and odd/even trace recursion in a dimension-compatible form. In the explicit four-dimensional representation, the checked trace identities include

$$\operatorname{tr}(\gamma^\mu\gamma^\nu) = 4\eta^{\mu\nu},$$

$$\operatorname{tr}(\gamma^\mu\gamma^\nu\gamma^\rho) = 0,$$

and

$$\operatorname{tr}(\gamma^\mu\gamma^\nu\gamma^\rho\gamma^\sigma) = 4(\eta^{\mu\nu}\eta^{\rho\sigma} - \eta^{\mu\rho}\eta^{\nu\sigma} + \eta^{\mu\sigma}\eta^{\nu\rho}).$$

Chisholm reductions, gamma-zero conjugation, chiral projectors, and gamma-five identities are strictly four-dimensional in the implementation. In particular, the shipped gamma-five trace convention is

$$\operatorname{tr}(\gamma_5\gamma^\mu\gamma^\nu\gamma^\rho\gamma^\sigma) = 4\varepsilon^{\mu\nu\rho\sigma}.$$

This equation fixes Idenso's internal epsilon normalization: it contains no additional factor of i. It is not a prescription for gamma five in dimensional regularization. Expressions using a specific scheme such as 't Hooft–Veltman or Larin require an explicit scheme-aware conversion outside this default simplifier. The optional three-gamma epsilon expansion exists in the Rust settings but is disabled by the default Python entry point.

The maintained runtime cases are the FORM-reference Dirac tests and the dimension-gating tests.

7.3 Color convention

Fundamental generators are written $(T^a)_i^{\ j}$. Idenso keeps the trace normalization T_R and Casimirs symbolic unless an explicit Rust conversion setting specializes them. The implemented normalization is

$$\operatorname{tr}\big(T^a T^b\big) = T_R\delta^{ab},$$

$$\sum_a (T^a)^j_i (T^a)^l_k = T_R\left(\delta^l_i \delta^j_k - N_c^{-1}\delta^j_i \delta^l_k\right),$$

$$\sum_a (T^a)^j_i (T^a)^k_j = C_F \delta^k_i,$$

and

$$\sum_{c,d} f^{acd} f^{bcd} = C_A \delta^{ab}.$$

For the conventional fundamental representation of $\mathrm{SU}(N_c)$, one may subsequently choose $T_R = \frac{1}{2}$, $C_A = N_c$, and $C_F = \frac{N_c^2-1}{2N_c}$. Those substitutions are a specialization, not assumptions imposed on every expression by `simplify_color`.

The default color pass evaluates supported closed traces and expands supported cross-chain fundamental Fierz contractions. Open lines, higher invariant tensors, or unsupported structures may remain explicit in the result; that is a partial normal form, not an exception. The normalization and free/dummy-index placement above are locked by the FORM-reference color tests.

7.4 What is shipped

Rule family	Status	Boundary
Ordinary Clifford chains and traces	Shipped	Compatible dimensions; default Python path.
Chisholm, gamma zero, projectors, and gamma five	Shipped	Explicit four-dimensional representations only.
Three-gamma epsilon expansion	Opt-in	Rust `GammaSimplifySettings`; disabled by the Python default.
Color traces, Casimirs, structure contractions, and fundamental Fierz	Shipped	Registered color representations; invariants remain symbolic by default.
Every FORM `color.h` identity and high-order invariant	Not implied	Only rules represented in the current Idenso implementation and tests are public behavior.

> **Do not mix algebraic and physical assumptions**
> Idenso's canonical form records what follows from its registered algebra rules. On-shell relations, gauge choices, external-state sums, dimensional-regularization schemes, and model substitutions are separate assumptions and must be applied explicitly.

The implementation-oriented source mapping, FORM excerpts, target patterns, historical commands, and unresolved provenance questions are retained in the Idenso FORM color/Dirac source map. They are contributor evidence, not additional claims about the public runtime surface.

8 Rust and Python APIs

8.1 Rust package

The `idenso` crate exposes representation types and syntax macros together with several rewrite families:

- `IndexTooling` covers canonicalization, conjugation, index wrapping, and dangling-index inspection for Symbolica atoms;
- `Cookable`, `CookSettings`, and the cook filters control reversible or flattening encodings;
- `SelectiveExpand` expands terms by recognized representation families;
- `dirac`, `color`, `epsilon`, and shorthand modules implement algebra-specific rewrites;
- `representations::initialize` installs the standard representation and tensor symbols.

Representation helper macros such as `bis!`, `cof!`, and `coad!` construct the symbolic forms expected by Spenso and Idenso. The Rust orientation leads to the revision-specific Rustdoc for their accepted forms, return types, feature gates, and source locations. APIs behind `bincode`, `reference-cases`, `python`, and `python_stubgen` are available only when the matching Cargo feature is enabled.

8.2 Python community module

Part of Symbolica community
Import Idenso from `symbolica.community.idenso`. The `idenso` Cargo package supplies this community module when built with its Python feature; it is not a standalone PyPI distribution and is not included in the GammaLoop wheel merely because the crates share a repository.

Install the published assembly with `python -m pip install --upgrade symbolica`, then verify it with `python -c "import symbolica.community.idenso"`. There is no `pip install idenso` fallback. For a source build, add this crate to the external symbolica-community assembly with the `python` feature and call `IdensoModule`'s `SymbolicaCommunityModule` registration when that extension is assembled. Building the Rust crate alone does not add the community module to an already installed Symbolica package.

The generated Python API records exact signatures and defaults. Its operations group naturally into four phases:

- setup: importing the community module registers its symbols;
- expansion: `expand_bis`, `expand_mink`, `expand_mink_bis`, `expand_metrics`, and `expand_color`;
- index preparation: `wrap_indices`, `wrap_dummies`, `list_dangling`, `cook_indices`, and `cook_function`;
- algebra: `dirac_adjoint`, `simplify_gamma`, `to_dots`, `simplify_metrics`, and `simplify_color`.

```python
from symbolica.community.idenso import list_dangling, simplify_metrics
from symbolica.community.spenso import Representation, TensorName

minkowski = Representation.mink(4)
mu = minkowski("mu")
nu = minkowski("nu")
metric = TensorName.g()
momentum = TensorName("p")
expression = metric(mu, nu) * momentum(mu)

external_indices = list_dangling(expression)
reduced = simplify_metrics(expression)
assert len(external_indices) == 1
assert len(list_dangling(reduced)) == 1
```

Idenso does not define a second parser syntax: the example constructs a Spenso-compatible Symbolica expression and then applies one Idenso transformation. Keep transformations separate while developing a pipeline so expression growth and convention changes remain observable.

For implementation details, start with the Rust API and the Python binding.

9 Version history

> **History coverage**
> This version of Idenso is 0.3.0, but the published changelog currently ends at 0.2.5. Release notes for 0.3.0 are not yet available, so the history below is incomplete for that version.

9.1 Available history

The rendered Idenso history records the published history through 0.2.5 and links to its canonical source.

When upgrading Idenso, pay particular attention to representation initialization, dummy-index handling, canonicalization, and Dirac, metric, and color conventions. A rewrite change can alter the shape or ordering of a symbolic result without changing the mathematical intent, so callers that persist or compare printed expressions should verify their chosen canonical form.

9.2 Upgrade checklist

Re-run representative identities and verify initialization, dummy-index allocation, canonicalization, and printed-expression expectations.

9.3 Reproducibility record

For reproducible upgrades, record the Idenso version and enabled features with the calculation. Do not assume that changes between 0.2.5 and 0.3.0 are fully represented in the available notes.

10 Idenso versions

This page renders the canonical Idenso changelog source as part of the Idenso website, navigation, and search.

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

11.1 Unreleased

11.2 0.2.5 - 2026-01-08

11.2.1 Fixed

- fix metric initialization and update to symbolica 1.2
- fix initialize to also load ETS

11.2.2 Other

- update linnet

11.3 0.2.4 - 2025-12-15

11.3.1 Fixed

- fix canonization in presence of duals
- fix gamma normalisation
- fix warnings
- fix `add_assign` with sparse tensors
- fix projp

11.3.2 Other

- Import initialize function
- Update symbolica to 1.1.0
- update linnet
- align to atom in argument of cof for Nc
- update linnet
- add readmes
- formatting and add ref for parametric
- update to 0.17 `pyo3_stub_gen`
- update linnet and make classattr to classmethods
- proper imports and feature flags for python
- move initialize to api
- Release spenso-macros 0.3
- update to symbolica 1.0
- update sympolica to 0.20
- Properly precontract scalars
- Add back pyo3-stub-gen-derive
- update to current dev symbolica and add pattern for pslash + m conjugate
- update to linnet 0.13.1
- working canonize (up to functions) and dirac adjoint
- spenso canonize buggy
- `robust_conj` but with lots of `gamma_0s`
- first try conj
- working pow execution with wrapping
- add function generic key

11.4 0.2.2 - 2025-08-18

11.4.1 Fixed

- fix cook indices
- fix clippy errors

11.4.2 Other

- update linnet to crates
- all tests pass
- Fix Multiple Heads Bug
- update linnet
- update linnet
- update linnet
- update parse problem and add spenso:: to all symbols
- make all symbols take spenso namespace
- operate directly on coefficient list
- remove expand

11.5 0.2.1 - 2025-07-15

11.5.1 Fixed

- fix gamma algebra again

11.5.2 Other

- update versions

11.6 0.2.0 - 2025-07-15

11.6.1 Fixed

- fix `wrap_indices`
- fix gamma algebra≈

11.6.2 Other

- allow arbitrary arguments for to dots

Python API reference

The supported Python packages available in this version are listed below. The HTML manual provides a separate page for every public class and function; this printable edition keeps a compact inventory. Rust APIs use canonical Rustdoc in the HTML manual.

`idenso-community`

Exports registered by idenso.

`cook_function`

Convert a single function call into a flattened variable symbol.

```
Convert a single function call into a flattened variable symbol.

Transforms a function expression with arguments into a single symbolic variable
whose name encodes both the function name and its arguments. This is the
atomic version of `cook_indices()`, operating on individual function calls
rather than complete expressions.

**Function Cooking Transform:**
- Simple function: `f(a, b)` → `f_a_b`
- Nested arguments: `tensor(rep(mu))` → `tensor_rep_mu`
- Multiple arguments: `gamma(mu, alpha, beta)` → `gamma_mu_alpha_beta`
- Complex names: `my_function(x, y)` → `my_function_x_y`


**Constraints:**
- Input must be a single function call (not sum, product, etc.)
- Arguments must be cookable (symbols, numbers, simple functions)
- Cannot cook expressions containing polynomials or complex structures

# Arguments
- `self_`: expression representing a single function call to cook

# Returns
Expression containing the flattened variable symbol.

# Raises
`TypeError` if input is not a cookable function or contains invalid argument types.

# Examples:
```python
import symbolica as sp
from symbolica.community.idenso import cook_function

# Simple function cooking
f = sp.S('f')
a, b = sp.S('a','b')

cooked = cook_function(f(a, b))
print(cooked)  # Outputs: f_a_b
```

def cook_function(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

cook_indices

Convert complex nested index structures into flattened symbolic names.

Convert complex nested index structures into flattened symbolic names.

Transforms hierarchical index expressions within tensor function arguments
into simplified, flat symbolic representations. This "cooking" process is
essential for pattern matching, simplification, and computational efficiency
when dealing with complex tensor expressions.

Index Cooking Transformation:
- Nested structure: `mink(4, f(g(h(μ))))` → `mink(4, f_g_h_mu)`
- Function chains: `lorentz(up(mu))` → `lorentz(up_mu)`
- Complex arguments: `tensor(rep(dim,type(idx)))` → `tensor(rep(dim,type_idx))`

Scope:
- Only affects indices appearing as function arguments
- Preserves top-level function structure
Arguments
- `self_`: expression containing complex nested index structures

Returns
Expression with flattened, simplified index names.

Examples:
```python
from symbolica.community.spenso import TensorName, Slot, Representation
import symbolica as sp
from symbolica.community.idenso import wrap_indices, cook_indices

T = TensorName("T")
rep = Representation.euc(3)
# With slots (creates TensorIndices)
mu = rep("mu")
nu = rep("nu")
x = sp.S("x")
tensor_with_args = T(mu, nu, x)  # T(mu, nu; x)
print(tensor_with_args)
print(
    cook_indices(wrap_indices(tensor_with_args.to_expression(), sp.S("wrap")))
)
```

```
def cook_indices(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

dirac_adjoint

Return the Dirac adjoint of a Symbolica tensor expression.

Return the Dirac adjoint of a Symbolica tensor expression.

Idenso takes the symbolic complex conjugate, reverses compatible open bispinor
chains, and
inserts the registered `gamma0` factors required at dangling bispinor slots.
The input must use the representation-aware Spenso forms registered by
`initialize()`.

```
# Examples
```python
>>> from symbolica.community.idenso import dirac_adjoint, initialize, list_dangling
>>> from symbolica.community.spenso import Representation, TensorName
>>> initialize()
>>> initialize() is None  # Registration is idempotent.
True
>>> bispinor = Representation.bis(4)
>>> spinor = TensorName("u")(bispinor("alpha")).to_expression()
>>> adjoint = dirac_adjoint(spinor)
>>> len(list_dangling(adjoint)) == 1
True
>>> "gamma0" in str(adjoint)
True
```

Arguments
- `self_`: a Spenso-compatible tensor expression.

Returns
The representation-aware Dirac adjoint.
```

```
def dirac_adjoint(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

`expand_bis`

```
Expand products around factors carrying registered bispinor indices.
```

```
Expand products around factors carrying registered bispinor indices.

# Arguments
- `self_`: a factorized Spenso-compatible expression.

# Returns
The expression distributed around its bispinor-bearing factors. No explicit spinor
components are substituted.

# Examples
```python
>>> from symbolica.community.idenso import expand_bis, initialize
>>> from symbolica.community.spenso import Representation, TensorName
>>> initialize()
>>> bispinor = Representation.bis(4)
>>> alpha, beta = bispinor("alpha"), bispinor("beta")
>>> u, v, w = TensorName("u"), TensorName("v"), TensorName("w")
>>> u_alpha = u(alpha).to_expression()
>>> v_beta, w_beta = v(beta).to_expression(), w(beta).to_expression()
>>> factorized = u_alpha * (v_beta + w_beta)
>>> expand_bis(factorized) == u_alpha * v_beta + u_alpha * w_beta
True
```
```

```
def expand_bis(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

expand_color

Expand products around registered color factors.

Expand products around registered color factors.

Fundamental, antifundamental, adjoint, color-chain, color-trace, and supported invariant
factors form the selected sector. This only distributes the symbolic expression; use `simplify_color()` separately to apply SU(N) identities.

Arguments
- `self_`: a factorized Spenso-compatible expression.

Returns
The expression distributed around its color-bearing factors.

Examples
```python
>>> from symbolica.community.idenso import expand_color, initialize
>>> from symbolica.community.spenso import Representation, TensorName
>>> initialize()
>>> adjoint, fundamental = Representation.coad(8), Representation.cof(3)
>>> antifundamental = fundamental.dual()
>>> generator = TensorName.t()
>>> t_a = generator(
...     adjoint("a"), fundamental("i"), antifundamental("j")
... ).to_expression()
>>> t_b = generator(
...     adjoint("b"), fundamental("k"), antifundamental("l")
... ).to_expression()
>>> t_c = generator(
...     adjoint("c"), fundamental("m"), antifundamental("n")
... ).to_expression()
>>> factorized = t_a * (t_b + t_c)
>>> expand_color(factorized) == t_a * t_b + t_a * t_c
True
```

```
def expand_color(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

expand_metrics

Expand products around registered metric tensors.

Expand products around registered metric tensors.

This is a structural expansion only. It neither contracts the metrics nor substitutes a
dimension or signature; call `simplify_metrics()` separately for supported contractions.

Arguments
- `self_`: a factorized Spenso-compatible expression.

```
# Returns
The expression distributed around its metric factors.

# Examples
```python
>>> from symbolica.community.idenso import expand_metrics, initialize
>>> from symbolica.community.spenso import Representation, TensorName
>>> initialize()
>>> minkowski = Representation.mink(4)
>>> metric = TensorName.g()
>>> g_mn = metric(minkowski("mu"), minkowski("nu")).to_expression()
>>> g_rs = metric(minkowski("rho"), minkowski("sigma")).to_expression()
>>> g_ab = metric(minkowski("alpha"), minkowski("beta")).to_expression()
>>> factorized = g_mn * (g_rs + g_ab)
>>> expand_metrics(factorized) == g_mn * g_rs + g_mn * g_ab
True
```
```

```
def expand_metrics(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

`expand_mink`

`Expand products around factors carrying registered Minkowski indices.`

```
Expand products around factors carrying registered Minkowski indices.

This is a selective symbolic expansion: Minkowski-bearing factors become polynomial
variables while unrelated sectors remain coefficients. It does not substitute
explicit
four-vector components or choose a metric signature.

# Arguments
- `self_`: a factorized Spenso-compatible expression.

# Returns
The expression distributed around its Minkowski-bearing factors.

# Examples
```python
>>> from symbolica.community.idenso import expand_mink, initialize
>>> from symbolica.community.spenso import Representation, TensorName
>>> initialize()
>>> minkowski = Representation.mink(4)
>>> mu, nu = minkowski("mu"), minkowski("nu")
>>> p, q, r = TensorName("p"), TensorName("q"), TensorName("r")
>>> p_mu = p(mu).to_expression()
>>> q_nu, r_nu = q(nu).to_expression(), r(nu).to_expression()
>>> factorized = p_mu * (q_nu + r_nu)
>>> expand_mink(factorized) == p_mu * q_nu + p_mu * r_nu
True
```
```

```
def expand_mink(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

expand_mink_bis

Expand products around factors carrying Minkowski or bispinor indices.

Expand products around factors carrying Minkowski or bispinor indices.

This combines the selection patterns of `expand_mink()` and `expand_bis()` in one coefficient pass. Other representation families remain in the coefficient sector.

Arguments
- `self_`: a factorized Spenso-compatible expression.

Returns
The expression distributed around both selected representation families.

Examples
```python
>>> from symbolica.community.idenso import expand_mink_bis, initialize
>>> from symbolica.community.spenso import Representation, TensorName
>>> initialize()
>>> minkowski, bispinor = Representation.mink(4), Representation.bis(4)
>>> p_mu = TensorName("p")(minkowski("mu")).to_expression()
>>> q_mu = TensorName("q")(minkowski("mu")).to_expression()
>>> u_a = TensorName("u")(bispinor("a")).to_expression()
>>> v_a = TensorName("v")(bispinor("a")).to_expression()
>>> factorized = (p_mu + q_mu) * (u_a + v_a)
>>> expected = p_mu * u_a + p_mu * v_a + q_mu * u_a + q_mu * v_a
>>> expand_mink_bis(factorized) == expected
True
```

```
def expand_mink_bis(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

initialize

Register Idenso's built-in representations and algebra symbols with Symbolica.

Register Idenso's built-in representations and algebra symbols with Symbolica.

Symbolica calls this during community-module initialization. Calling it again is safe and ensures the standard Lorentz, spinor, and color objects exist.

```
def initialize() -> None:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

list_dangling

Lists the dangling (external, uncontracted) indices present in the expression.

Lists the dangling (external, uncontracted) indices present in the expression.

Identifies and returns all indices that are not summed over (i.e., not dummy indices). These are the "free" indices that appear in the final result and determine the tensor rank of the expression. For dualizable representations, downstairs indices are represented wrapped in `dind(...)`.

This is essential for:

```
- Verifying index conservation in tensor equations
- Determining the rank and structure of tensor expressions
- Debugging index contractions

# Arguments
- `self_`: tensor expression to analyze

# Returns
A list of expressions, each representing a free (dangling) index.

# Examples:
```python
from symbolica.community.spenso import TensorName, Slot, Representation
import symbolica as sp
from symbolica.community.idenso import (
    list_dangling,
)

T = TensorName("T")
rep = Representation.euc(3)
# With slots (creates TensorIndices)
mu = rep("mu")
nu = rep("nu")
x = sp.S("x")
tensor_with_args = T(mu, nu, nu, x)  # T(mu, nu; x)
# print(tensor_with_args)
print(list_dangling(tensor_with_args.to_expression()))
```
def list_dangling(self_: Expression) -> list[Expression]:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

simplify_color

Simplify registered Spenso color chains, traces, generators, and structure constants.

Simplify registered Spenso color chains, traces, generators, and structure constants.

With the default Python settings, the simplifier evaluates supported closed traces and
expands contractions between generators on separate fundamental chains. Its normalization
conventions are

- `Tr(T^a T^b) = TR δ^{ab}`;
- `Σ_a (T^a)_i^j (T^a)_k^l = TR (δ_i^l δ_k^j - δ_i^j δ_k^l/Nc)`;
- `Σ_a (T^a)_i^j (T^a)_j^k = CF δ_i^k`;
- `Σ_{c,d} f^{acd} f^{bcd} = CA δ^{ab}`.

`CA = Nc`, `CF = (Nc² - 1)/(2Nc)`, and `TR = 1/2` are the conventional fundamental SU(Nc) specialization, not identities imposed on every input. The default simplifier keeps
representation invariants symbolic where possible; explicit dimension substitution is a
separate Rust setting.

```
# Examples
```python
>>> from symbolica import E
>>> from symbolica.community.idenso import initialize, simplify_color
>>> initialize()
>>> generators = E('''
...     t(coad(Nc^2-1,a),cof(Nc,i),dind(cof(Nc,j)))
...     * t(coad(Nc^2-1,a),cof(Nc,k),dind(cof(Nc,l)))
... ''', default_namespace="spenso")
>>> simplified = simplify_color(generators)
>>> "t(" not in str(simplified) and "g(" in str(simplified)
True
```

Arguments
- `self_`: expression containing SU(N) color structures

Returns
The simplified expression. Unsupported or open indexed structures may remain
explicitly in
the result; their presence is not an error.

Notes
Only representation-aware Spenso color forms are recognized. Plain Symbolica
functions with
similar names are left unchanged.
```

```
def simplify_color(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

simplify_gamma

```
Simplify registered Spenso gamma chains and traces with Idenso's default rules.
```


```
Simplify registered Spenso gamma chains and traces with Idenso's default rules.
```


```
The dimension-generic part applies compatible Clifford anticommutation, adjacent
contractions, and ordinary odd/even trace recursion. Chisholm identities, gamma-five
anticommutation and traces, gamma-zero conjugation, and chiral-projector rules are
applied
only when the expression carries explicit four-dimensional Minkowski and bispinor
representations.

This function does not select or implement a dimensional-regularization gamma-five
scheme.
Its gamma-five rules are strictly four-dimensional, and the default Python entry
point does
not enable the optional three-gamma epsilon expansion available through Rust
settings.
Gamma factors must use the Spenso representation-aware forms registered by
`initialize()`;
unrecognized plain Symbolica functions are left unchanged.

# Examples
```python
>>> from symbolica import E
```
```

```
>>> from symbolica.community.idenso import initialize, simplify_gamma
>>> initialize()
>>> trace = E('''
...     gamma(bis(4,a),bis(4,b),mink(4,mu))
...     * gamma(bis(4,b),bis(4,a),mink(4,nu))
... ''', default_namespace="spenso")
>>> simplified = simplify_gamma(trace)
>>> "gamma(" not in str(simplified) and "g(" in str(simplified)
True
```

Arguments
- `self_`: expression containing gamma matrix products and traces

Returns
The simplified expression with gamma algebra applied.

```
def simplify_gamma(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

simplify_metrics

```
Simplifies contractions involving metric tensors and identity tensors.
```

Simplifies contractions involving metric tensors and identity tensors.

Applies fundamental tensor algebra rules for metric and identity tensors:

Metric tensor rules:
- `gᵘᵛ pᵥ → pᵘ` (index raising/lowering)
- `gᵘᵛ gᵥρ → gᵘρ` or `δᵘρ` (metric composition)
- `gᵘᵤ → D` (dimension of spacetime)
- `ηᵘᵛ pᵥ → pᵘ` (flat metric contractions)

Identity tensor rules:
- `δᵘᵛ pᵥ → pᵘ` (Kronecker delta contraction)
- `δᵘᵤ → D` (trace of identity)

The function recognizes metrics as `spenso::g(...)`

Arguments
- `self_`: expression containing metric/identity tensor contractions

Returns
The simplified expression with metric rules applied.

Examples:
```python
from symbolica.community.idenso import simplify_metrics, to_dots
from symbolica.community.spenso import Representation, TensorName
q = TensorName("q")
g = TensorName.g()
rep = Representation.euc(3)
# With slots (creates TensorIndices)
mu = rep("mu")
nu = rep("nu")
```

```
print(simplify_metrics(g(mu, nu) * q(mu)))
```

```
def simplify_metrics(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

`to_dots`

Converts contracted Lorentz/Minkowski indices into dot product notation.

Converts contracted Lorentz/Minkowski indices into dot product notation.

Automatically identifies and converts patterns like `p(mink(D, mu)) * q(mink(D, mu))` into the compact, representation-carrying `dot(p(mink(D)), q(mink(D)))` notation. This
simplification is essential for physics calculations involving four-vectors.

The function recognizes:
- Contracted vector indices: `pᵘqᵤ → p·q`
- Multiple contractions: `pᵘqᵤrᵛsᵥ → (p·q)(r·s)`
- Self-contractions: `pᵘpᵤ → p²`

Arguments
- `self_`: expression containing contracted Minkowski vector indices

Returns
The expression with vector contractions converted to dot products.

Examples:
```python
from symbolica.community.idenso import to_dots
from symbolica.community.spenso import Representation, TensorName
p = TensorName("p")
q = TensorName("q")
rep = Representation.euc(3)
# With slots (creates TensorIndices)
mu = rep("mu")
nu = rep("nu")

print(to_dots( p(mu)*q(mu)))
```

```
def to_dots(self_: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

`wrap_dummies`

Wraps only the dummy (contracted) indices within the expression using a header symbol.

Wraps only the dummy (contracted) indices within the expression using a header symbol.

Similar to `wrap_indices`, but selectively identifies and wraps only contracted indices (those appearing once upstairs and once downstairs, or twice in a self-dual representation), leaving external (dangling) indices untouched.

```
This is crucial for proper index management in tensor calculations.

Contracted indices are those that:
- Appear in both upper and lower positions (for dualizable reps)
- Appear twice in the same position (for self-dual reps)
- Are summed over (Einstein summation convention)

# Arguments
- `self_`: input expression containing both dummy and free indices
- `header`: symbol to use as wrapper function name for dummy indices only

# Returns
A new expression with only contracted indices wrapped.

# Examples:
```python
from symbolica.community.spenso import TensorName, Slot, Representation
import symbolica as sp
from symbolica.community.idenso import simplify_metrics, wrap_dummies

T = TensorName("T")
rep = Representation.euc(3)
# With slots (creates TensorIndices)
mu = rep("mu")
nu = rep("nu")
x = sp.S("x")
tensor_with_args = T(mu, nu, nu, x)  # T(mu, nu; x)
# print(tensor_with_args)
print(wrap_dummies(tensor_with_args.to_expression(), sp.S("wrap")))

```

def wrap_dummies(self_: Expression, header: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

`wrap_indices`

Wrap all abstract indices with a header symbol # Arguments - `self_`: input expression containing tensor indices - `header`: symbol to use as the wrapper function for all indices # Returns Expression with all indices wrapped by the header symbol.

```
Wrap all abstract indices with a header symbol

# Arguments
- `self_`: input expression containing tensor indices
- `header`: symbol to use as the wrapper function for all indices

# Returns
Expression with all indices wrapped by the header symbol.

# Examples:
```python
from symbolica.community.spenso import TensorName, Slot, Representation
import symbolica as sp
from symbolica.community.idenso import wrap_indices
```
```

```
T = TensorName("T")
rep = Representation.euc(3)
# With slots (creates TensorIndices)
mu = rep("mu")
nu = rep("nu")
x = sp.S("x")
tensor_with_args = T(mu, nu, x)  # T(mu, nu; x)
print(tensor_with_args)
print(wrap_indices(tensor_with_args.to_expression(), sp.S("wrap")))

```

```
def wrap_indices(self_: Expression, header: Expression) -> Expression:
```

Requires features: `python, python_stubgen`

Exhaustive reference · Source

Version information

Save these identifiers with published results and include them in issue reports so that collaborators can reproduce the same software environment.

- **Documentation channel:** `latest`
- **Documentation route:** `/products/idenso/latest/`
- **Source revision:** `e51747446aa724c3ffac5c42c4103d090c09c6b0`

Package versions and enabled features

- `gammaloop/gammalooprs` — `0.3.4` (`no optional features`)
- `gammaloop/gammaloop-api` — `0.1.0` (`features: cli`)
- `gammaloop/gammaloop` — `0.3.4` (`features: python_api, python_abi, python_stubgen`)
- `linnet/linnet` — `0.17.0` (`features: nodestore-vec, serde, bincode, drawing, symbolica`)
- `linnet/linnet-py` — `0.1.0` (`features: extension-module, abi3-py310`)
- `spenso/spenso` — `0.6.0` (`features: shadowing`)
- `spenso/spenso-macros` — `0.3.2` (`features: shadowing`)
- `spenso/spenso-hep-lib` — `0.1.7` (`no optional features`)
- `spenso/spynso3` — `0.1.2` (`features: python_stubgen`)
- `idenso/idenso` — `0.3.0` (`features: bincode, reference-cases`)
- `idenso/idenso` — `0.3.0` (`features: python, python_stubgen`)
- `vakint/vakint` — `0.1.2` (`no optional features`)
- `vakint/vakint` — `0.1.2` (`features: symbolica_community_module, python_stubgen`)