tensor4all-rs

Tensor Network Computing with Vibe Coding

Hiroshi Shinaoka

January, 2026

https://github.com/tensor4all/tensor4all-rs

What is Vibe Coding?

A development style where you code through **intuitive, iterative dialogue with AI**

- Express intent in natural language
- AI generates and refines code
- Rapid trial-and-error cycles
- **Type system catches AI mistakes**

Rust is ideal: compile-time type checking instantly catches bugs in AI-generated code.

AI Agents I Used

Claude Opus 4.5 — One of the best AI coding models

- **Claude Code** (CLI tool by Anthropic)
  → Focuses on autonomous development
- **Cursor** (AI-powered IDE)
  → Better for collaborative editing with human

Both provide agentic coding capabilities:
code generation, error fixing, and refactoring

Typical Workflow

1. **Analyze**: Ask agent to analyze the original Julia library
2. **Plan**: Agent asks questions and presents a plan
   → Human approves the plan
3. **Generate**: Agent generates source code and tests
   → Autonomously fixes compilation and test errors
4. **Review**: Human reviews the code
   - Focus on global structure and logic. The compiler handles memory safety, lifetimes, etc.
5. **Refactor**: If better ideas emerge, ask agent for large-scale changes
   → Agent can refactor entire architecture autonomously

Typical Workflow (2): Issue-based

For larger projects with multiple tasks:

1. **Investigate**: Ask agent to analyze codebase and identify issues
   → Agent creates GitHub issues via `gh` command
2. **Accumulate**: Build up a backlog of well-defined issues
3. **Resolve**: Agent picks and solves issues one by one
   → Each issue becomes a focused PR

Advantage: Clear task boundaries, easy to track progress, parallelizable with multiple agents

Design Philosophy of tensor4all-rs

- **Modular architecture**
  Independent crates enable fast compilation and isolated testing
- **ITensors.jl-like dynamic structure**
  Flexible Index system preserves intuitive API
- **Static error detection**
  Rust's type system catches errors at compile time
- **Multi-language support**
  C-API exposes full functionality to Julia and Python

Project Structure

Crate	Description
tensor4all-core	Index, Tensor, SVD, QR, LU
tensor4all-simplett	Simple TT/MPS with canonical forms
tensor4all-tensorci	TCI algorithms
tensor4all-quanticstci	High-level Quantics TCI
tensor4all-capi	C FFI for language bindings
tensor4all-treetn	Tree tensor networks (WIP)
tensor4all-itensorlike	ITensors.jl-like API (WIP)
tensor4all-quanticstransform	Quantics transformation operators (WIP)
quanticsgrids	Quantics grid structures

Full functionality implemented in Rust and exposed to other languages via C-API.

Type Correspondence with ITensors.jl

ITensors.jl	tensor4all-rs
`Index{Int}`	`Index<Id, NoSymmSpace>`
`ITensor`	`TensorDynLen<Id, Symm>`
`Dense` / `Diag`	`Storage::DenseF64` / `DiagF64`
`A * B`	`a.contract_einsum(&b)`
`cutoff`	`rtol` (= √cutoff)

Truncation:

$$\frac{\left\| A - A_{\text{approx}} \right\|_F}{\|A\|_F} \leq \text{rtol}$$

Index System: ITensors.jl vs QSpace

Aspect	ITensors.jl	QSpace
Central entity	Index	Tensor
Index identity	UUID (auto)	itag name (string)
Connection	Share same Index	Same itag + opposite direction
Direction	Undirected	Directed (Ket/Bra)

tensor4all-rs supports **both** via `IndexLike` trait with `ConjState`

Design Challenge: Extensibility

Initial focus: Dense tensors without symmetry

Challenge: Maintain compatibility with different index systems (ITensors.jl, QSpace, etc.) and keep code generic for future extensions

Problem: A single Index type cannot support all use cases

Solution: Define minimal trait requirements for Index and Tensor
→ Tree TN algorithms are generic over any type implementing the traits

Trait: A set of requirements a type must satisfy (similar to C++ concepts)

IndexLike Trait

```rust
pub trait IndexLike: Clone + Eq + Hash {
    type Id: Clone + Eq + Hash;

    fn id(&self) -> &Self::Id;
    fn dim(&self) -> usize;
    fn conj_state(&self) -> ConjState;  // Undirected, Ket, Bra
    fn conj(&self) -> Self;
    fn is_contractable(&self, other: &Self) -> bool;
}
```

Contractability Rules

Two indices are contractable if:

1. Same `id()` and `dim()`
2. Compatible `ConjState`:
   - `(Ket, Bra)` or `(Bra, Ket)` → **contractable**
   - `(Undirected, Undirected)` → **contractable**
   - Mixed directed/undirected → **forbidden**

TensorLike Trait

```rust
pub trait TensorIndex: Sized + Clone {
    type Index: IndexLike;
    fn external_indices(&self) -> Vec<Self::Index>;
    fn replaceind(&self, old: &Self::Index, new: &Self::Index) -> Result<Self>;
}

pub trait TensorLike: TensorIndex {
    fn tensordot(&self, other: &Self, pairs: &[(Self::Index, Self::Index)]) -> Result<Self>;
    fn factorize(&self, left_inds: &[Self::Index], options: &FactorizeOptions)
        -> Result<FactorizeResult<Self>>;
    fn conj(&self) -> Self;
    fn norm_squared(&self) -> f64;
    // ... other operations
}
```

Tree TN algorithms are generic over any `TensorLike` implementor.

Contraction Methods

Explicit contraction — specify index pairs:

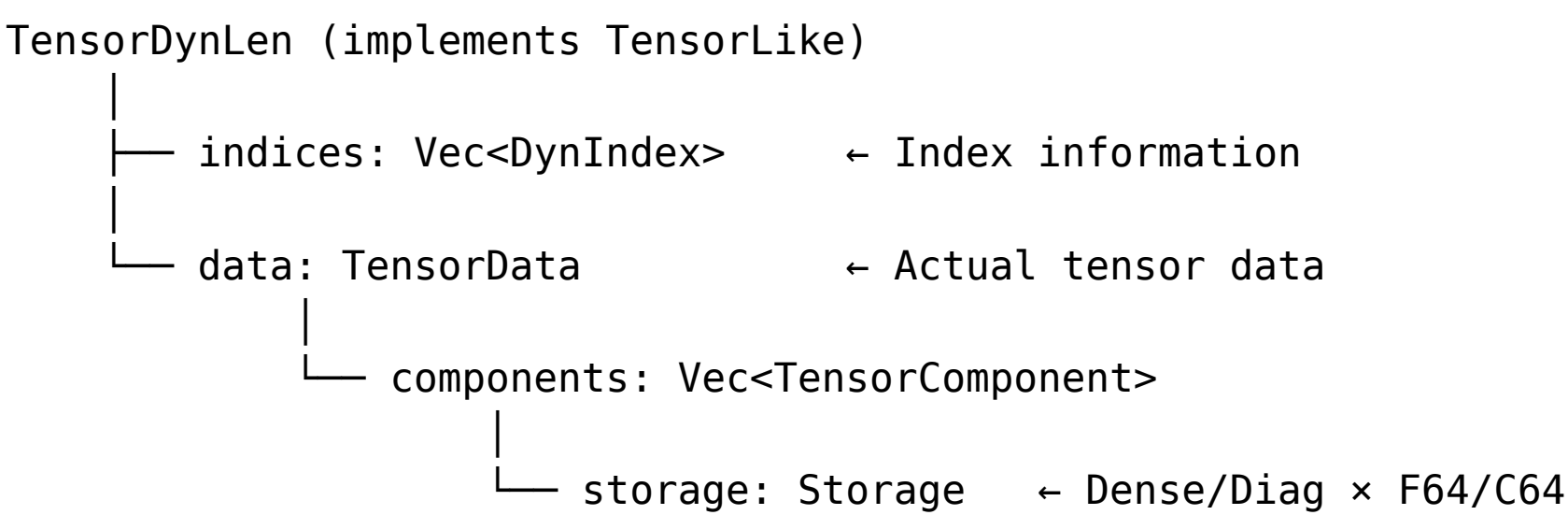
```
// Contract indices i from A with j from B
let c = a.tensordot(&b, &[(i, j)])?;
```

Einsum-style contraction — automatic matching by index ID:

```
// Indices with same ID are automatically contracted
let c = TensorLike::contract_einsum(&[a, b, c])?;
```

`tensordot`: explicit control / `contract_einsum`: convenient for networks

Type Hierarchy Overview



- **TensorDynLen**: User-facing tensor type (implements `TensorLike`)
- **TensorData**: Lazy outer product of tensor components
- **Storage**: Low-level data storage (`mdarray` backend)

DynIndex: Default Index Type

Type alias: `DynIndex = Index<DynId, TagSet>`

```
pub struct Index<Id, Tags> {
    pub id: Id,      // Unique identifier
    pub dim: usize,  // Dimension
    pub tags: Tags,  // String tags for labeling
}
```

- **DynId**: UUID-based unique identifier (like ITensors.jl)
- **TagSet**: ITensor-compatible string tags for labeling (e.g., `"Site,n=1"`)
- Implements `IndexLike` trait

TensorData: Lazy Outer Products

Stores tensors as **lazy outer products** of components:

```
pub struct TensorData {
    pub components: Vec<TensorComponent>,  // Storage + indices
    pub external_ids: Vec<DynId>,          // User-facing order
}
```

Advantages:

- Diagonal × Dense = lazy (no memory explosion)
- Actual expansion only when needed
- Permutations are tracked, not executed

Storage: Backend Layer

Defined in `tensor4all-tensorbackend` crate:

```
pub enum Storage {
    DenseF64(DenseStorageF64),  // Dense real
    DenseC64(DenseStorageC64),  // Dense complex
    DiagF64(DiagStorageF64),    // Diagonal real
    DiagC64(DiagStorageC64),    // Diagonal complex
}
```

Backend libraries:

- `mdarray`: Multi-dimensional array storage
- `mdarray-linalg`: Linear algebra operations (SVD, QR, LU)

Contraction Path Optimization

Problem: `TensorData` can contain mixed Dense + Diag components
→ Need optimal contraction order for the component list

Example (SVD-like): `U(i,j)` × `s(j)` × `V(j,k)` where `s` is diagonal
- Index `j` is a **hyperedge** (shared by 3 tensors)
- Naive order may expand diagonal unnecessarily

Solution: omeco (Rust port of OMEinsumContractionOrders.jl)
- GreedyMethod: $O(n^2 \log n)$ near-optimal ordering
- Also supports TreeSA (simulated annealing)

Workflow: `TensorData.components` → omeco → contraction tree → execute

Rust: Index & Tensor

```rust
use tensor4all_core::{Index, Tensor};

// Create indices with tags
let i = Index::new_dyn_with_tag(2, "Site,n=1")?;
let j = Index::new_dyn_with_tag(3, "Link")?;

// Create tensors and contract
let a = Tensor::random(&[i.clone(), j.clone()]);
let b = Tensor::random(&[j.clone(), k.clone()]);
let c = a.contract_einsum(&b)?;  // Contract on shared index j
```

Rust: TensorTrain

```rust
use tensor4all_simplett::{TensorTrain, AbstractTensorTrain};

// Create a constant tensor train with local dimensions [2, 3, 4]
let tt = TensorTrain::<f64>::constant(&[2, 3, 4], 1.0);

// Evaluate at a specific multi-index
let value = tt.evaluate(&[0, 1, 2])?;

// Compress with tolerance (rtol=1e-10, maxrank=20)
let compressed = tt.compressed(1e-10, Some(20))?;
```

Rust: TCI

```rust
use tensor4all_tensorci::{crossinterpolate2, TCI2Options};

let f = |idx: &Vec<usize>| -> f64 {
    ((1 + idx[0]) * (1 + idx[1]) * (1 + idx[2])) as f64
};

let options = TCI2Options { tolerance: 1e-10, ..Default::default() };
let (tci, ranks, errors) = crossinterpolate2(
    f, None, vec![4, 4, 4], vec![vec![0, 0, 0]], options)?;
```

Julia: Index & Tensor

```julia
using Tensor4all.ITensorLike

i = Index(2, tags="Site,n=1")
j = Index(3, tags="Link")
k = Index(2, tags="Site,n=2")

t1 = Tensor([i, j], randn(2, 3))
t2 = Tensor([j, k], randn(3, 2))
result = contract(t1, t2)  # Contract on shared index j
```

Julia: TensorTrain

```julia
using Tensor4all.SimpleTT

# Create and manipulate tensor trains
tt = TensorTrain(tensors)

# Orthogonalize and truncate
orthogonalize!(tt, 2)
truncate!(tt; maxdim=3, rtol=1e-10)

# Evaluate and sum
println("Sum: ", sum(tt))
```

Julia: TCI

```julia
using Tensor4all.TensorCI

f(i, j, k) = Float64((1 + i) * (1 + j) * (1 + k))
tt, err = crossinterpolate2(f, [4, 4, 4]; tolerance=1e-10)

println(tt(0, 0, 0))  # 1.0
println(tt(3, 3, 3))  # 64.0
```

Python: Index & Tensor

```python
from tensor4all import Index, Tensor
import numpy as np

i = Index(2, tags="Site")
j = Index(3, tags="Link")

data = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float64)
t = Tensor([i, j], data)

arr = t.to_numpy()  # Convert back to NumPy
```

Python: TensorTrain

```python
from tensor4all import TensorTrain

# Create tensor train
tt = TensorTrain.constant([2, 3, 4], 1.0)

# Evaluate and compress
value = tt.evaluate([0, 1, 2])
compressed = tt.compressed(rtol=1e-10, maxrank=20)

print("Sum:", tt.sum())
```

Python: TCI

```python
from tensor4all import crossinterpolate2

def f(i, j, k):
    return float((1 + i) * (1 + j) * (1 + k))

tt, err = crossinterpolate2(f, [4, 4, 4], tolerance=1e-10)

print(tt(0, 0, 0))  # 1.0
print(tt(3, 3, 3))  # 64.0
```

Future Extensions

- Tree TCI
- GPU acceleration, Automatic differentiation
- Quantum number symmetries: U(1), Z_n, SU(2), SU(N)