Agentic AI Coding × Rust

計算物理コードを検証可能に育てる

品岡 寛

埼玉大学

事前準備

事前準備 (各自, 当日までに)

1. CLI 型エージェントを 1 つ導入 してくる.
   - 無料: Gemini CLI / ChatGPT Plus 以上: Codex / その他はご自由に.
   - 導入手順: Satoshi Terasaki / Junya Ito.
2. プログラミング言語の開発環境を用意する.
   - 今回は Rust を使うが何でもよい. Rust なら Cargo を install (参考).

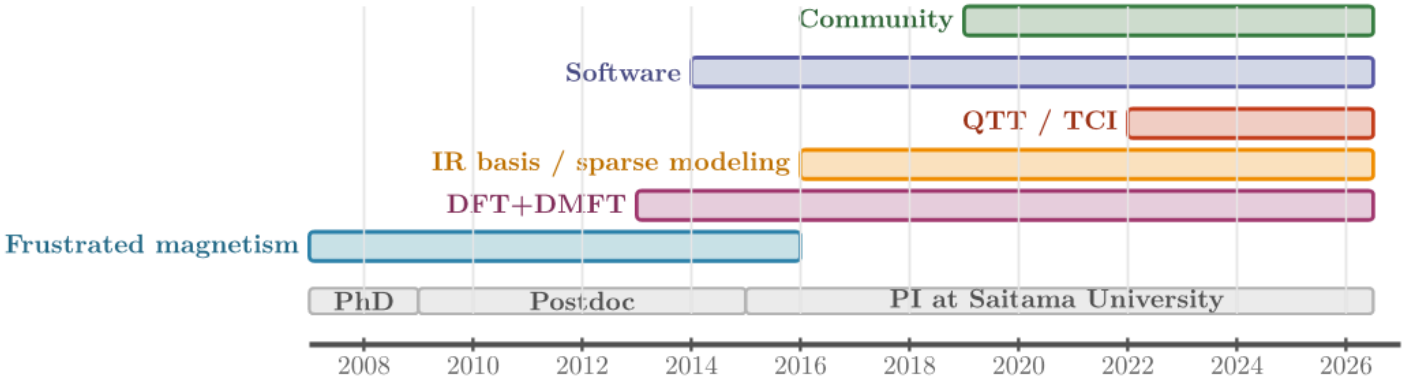
今日の流れ

- 前半 = 方法論 / 後半 = 2D Ising ハンズオン.
- 当日は 環境チェックのみ: 方法論と実習に時間を使う.
- 2D Ising は よく知られている ので, 無料モデルでも完走できる (多分).

導入

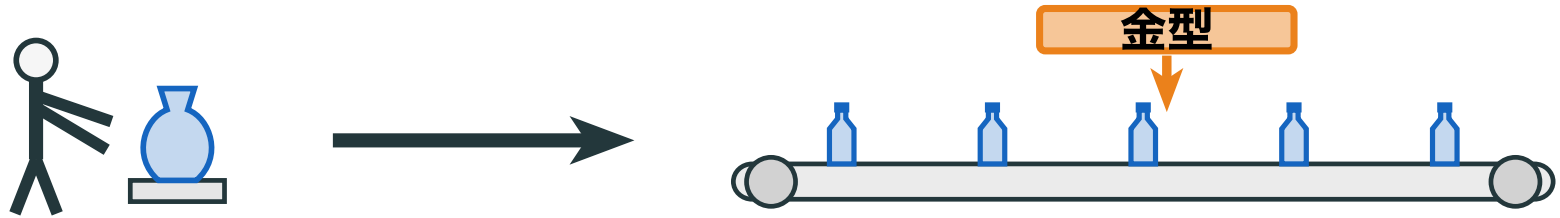
自己紹介と今日の立ち位置

- 東大 PhD → スイス Postdoc → 埼玉大 PI (2015〜)
- 量子多体計算 〜 第一原理計算まで幅広い経験
- IR basis / sparse modeling / tensor networks
- **C++ / Python / Julia / Fortran** の本格的な実経験
- OSS と Community へ貢献



今は 1 行も書かないし，逐行も読まない．それでも 手動コーディングより agentic coding の方が信頼できる と感じる．なぜか？

コーディングの産業革命: 手工業から工業へ



人間が 1 点ずつ丁寧に
＝ 手作業コーディング

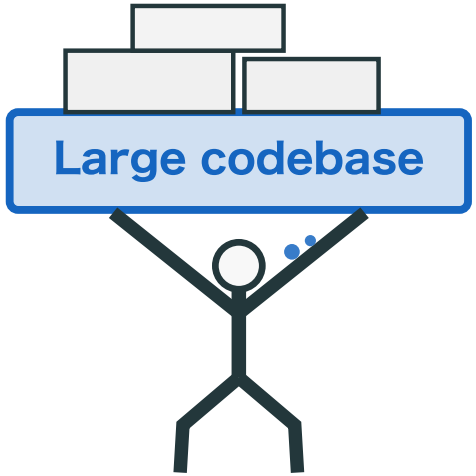
金型で大量生産 (高速・均質)
＝ 理想的な agentic coding

同じ転換が，いま agentic coding で起きている．

理想的な agentic coding をどう実現するか考える段階．

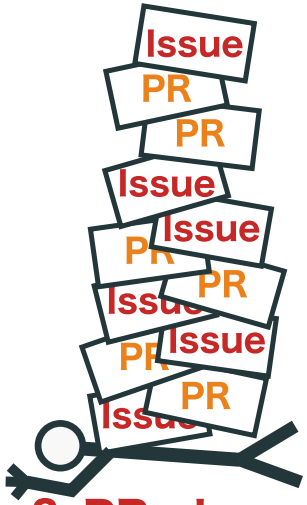
手作業 (職人モデル) の限界

Scientific computing



Held up by a few artisans
(PD / staff leave → unmaintainable)

Large project



Issues & PRs keep piling up
(manual review can't keep up)

→ いずれも 人手で支えるモデル が抱える限界.

よく言われる批判: “AI slop”

巷では agentic coding の成果物を “AI slop” と呼ぶ声がある:

職人性の喪失
Sinclair Target

“…would prefer not to use agentic tools even if they worked as advertised.”
宣伝通り動いても, 仕事への care を失う. Quality in the Age of Slop, 2026

保守の崩壊
D. Stenberg (curl)

“The current torrent of submissions put a high load on the curl security team…”
AI slop に埋もれ curl は bug bounty を終了 (2026). The Register

共有資源の劣化
Baltes et al.

“…a tragedy of the commons, where individual productivity gains externalize costs onto reviewers, maintainers, and the broader community.”
“An Endless Stream of AI Slop”, arXiv:2603.27249

過大広告への反動
Anthropic C compiler

16 並列 Claude で C コンパイラ (100K 行, $20k) を構築. 別 benchmark では GCC -O0 より速いが GCC -O2 には届かない. こうした showcase は評価軸がずれると AI 反対派を増やす.
Anthropic · benchmark

私の解答

いずれも agentic coding の本質的な限界ではなく，AI を前提に体制を組み替えれば解消する 移行期の課題である．

職人性の喪失
Sinclair Target

伝統工芸と同じく，手動コーディングが残り，意味を持つのは ごく限られた領域 に過ぎない．

保守の崩壊・共有資源の劣化
D. Stenberg, Baltes et al.

手動の PR・マージを前提とした運用を 維持する必要はない．AI を前提としたメンテナンス体制へ移行すべきであり，人間の関与はむしろボトルネック である．

過大広告への反動
Anthropic C compiler

最適化不足は AI 不能の証拠ではない．性能 oracle と緊密な human-AI loop の不足として読むのが妥当である (後述)．

今日の主張 (概要)

信頼の置き場所を 目視 から 機械的・外部検証 へ移した:

Compiler ・ test ・ oracle ・ rules が，目視に代わって正しさを支える．

AI の高速生成は工業的な大量生産である．製品を 1 個ずつ直すのではなく，金型 (= 正本: rules ・ oracle) を直す．

コーディングエージェントの基礎

Chat 型 vs CLI 型エージェント

Chat 型 (ブラウザ対話)	CLI 型 (ターミナル)
質問して答えをコピペ	ファイルを直接編集・実行
ローカル環境に触れない	リポジトリ・git を操作
1 往復ずつ人が運ぶ	マルチステップを自律実行

CLI 型は 長時間の自律実行を狙う設計．今日の主役はこちら．

近年は両者の境界が薄れ，エージェント型が両形態を飲み込みつつある．GUI から自律実行を呼ぶ Codex App はその典型である．

世代で言えば Gen1 (Cursor / Copilot) → Gen2 (Claude Code / Codex / …)．MIT missing-semester: agentic coding

代表的な CLI エージェント (ハーネス)

Claude Code	Anthropic. ターミナル/IDE で自律コーディング
Codex CLI	OpenAI. Rust 製 OSS, ChatGPT プランに同梱
Gemini CLI	Google. OSS, 無料枠あり
Kimi CLI	Moonshot (中国系). モデル K2.7 Code
OpenCode	OSS, プロバイダ非依存
Pi	最小構成の OSS. 拡張・skill を自分で足して育てる (最小主義者向け)

ハーネス ≠ モデル

- ハーネス = CLI ツール (ファイル編集・コマンド実行・ツール呼び出しの枠組み).
- モデル = 裏で動く LLM. 多くのハーネスは モデルを差し替え可能.
  - 中立: OpenCode / Gemini CLI ・ 自社統合寄り: Claude Code / Codex.
- 性能は モデル, 操作性・自律度は ハーネス で決まる. 両方を選ぶ.

サブスク vs API, 廉価モデル

サブスク	Claude Pro $20/月 (Claude Code 込) / ChatGPT + Codex．月額固定 + 利用枠
API 従量	使ったトークン分課金．ハーネスに任意モデルを接続

廉価なコーディング向け API (2026): DeepSeek V4 Flash (入力 $0.14 / 出力 $0.28 per 1M) ・ Kimi K2.7 Code． 例: OpenCode + 廉価 API で低コストに始められる．

料金・モデル名は 2026 年 6 月時点．世代更新が速い．

スキルとは

- Skill = エージェントの 手順書 (markdown). 抽象的な作業を明示的な小ステップに分解する.
- プログラムの関数に近いが, 操作対象は データでなくエージェントの振る舞い.
- 関連する作業で 自動的に発火 し, skill が skill を呼んで複雑な手順を構成する.
- CLAUDE.md / AGENTS.md = セッションを越える記憶 (規範・コマンド・構成).
  - リポジトリ内に置く プロジェクト固有 の情報であり，git でチームや将来の自分と共有する (ユーザ全体の設定とは別).

Jin-Guo Liu: Sustainable Automation · obra/superpowers

実際のワークフローはこんな感じ

superpowers の Skill が brainstorm → plan → execute を強制する (各段階は自動発火):

- 出発点は prompt でなく 数式つき設計書. 検証 (oracle / test) も同時に設計する.
- 実装後は コード ↔ アルゴリズム・数式 の対応を AI に出させて照合 (コードそのものは読まないこともある).
- Execute は TDD + サブエージェント分割, Review で問題が出れば前段階へ戻る.

obra/superpowers · Jin-Guo Liu: Sustainable Automation

この１年: agentic coding の限界を試す

これまでの経緯

2025 年 *10* 月以降，私はコードを一行も書いていない．生成コードも逐行では読まない．

2025 年 1 月	Cursor を開始，依然としてソースコードを見る世代
2025 年 10 月	SparseIR.jl を Julia → Rust へ移植開始
2025 年 12 月	Claude Code へ移行: 生成コードを逐行で読むのをやめた
2026 年 1 月	tensor4all-rs 開発開始: Julia TN eco の Rust port
2026 年 2 月	tenferro-rs 開発開始: Rust 製の PyTorch 的 tensor stack
2026 年 3 月〜	Codex, Claude, DeepSeek, Kimi などを使い分け開発を進める

作ったもの: tensor4all-rs

Julia の tensor learning stack を Rust に移植.

TensorCrossInterpolation.jl / QuanticsTCI.jl など / SciPost Phys. 18, 104 (2025) / tensor4all

GitHub: github.com/tensor4all/tensor4all-rs

- Julia による大規模開発の辛さ: (pre)compile の遅さ，実行時に出る型不安定性等
- 2026 年 1 月 1 日開始 (元々は agentic coding の「限界」を試す冬休みの自由研究)
- 人間数名 + AI (Claude Code から Codex メインへ)
- 2 ヶ月で 353 commits · 最初の 2 週間で +61,486 行 (151 files)

→ 共同研究者間の議論 (AI slop, 教育法，Julia は不要か？)は，まだ続いている.

“多数の試行錯誤と失敗から，正しい agentic coding 法が見え始めたところ”

設計，テスト・検証，適した言語，(教育法)...

Agentic coding を限界まで押してみた

最初は「モノシリック構造」から始めた:

- ITensors.jl の必要部分 (Index system, tensor contraction)
- QTT/TCI

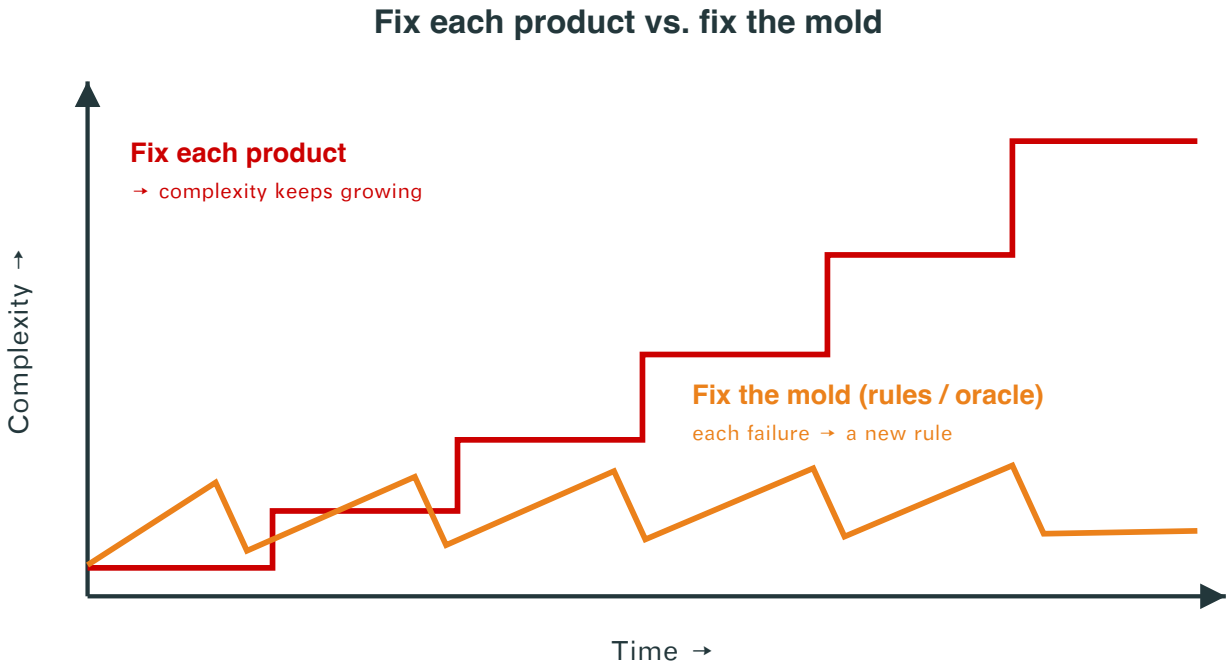
の移植

(予想通り) 多数の問題が発生

- 抽象化を壊す: 低階層の機能を高階層で import して使ってしまう (その場しのぎ)
- ユニットテストのしきい値を勝手に緩める (バグを直すのを諦める)
- BLAS を使わずに nested loops を書く (結果は正しいが...)
- 1 つのソースファイルの行数が数千行超え
- ...

複雑さが増え続ける

- 個別修正では追いつかない.
- 長期保守には，思想とデザインルールを **統一的に育てる** しかない.
- さもないと複雑さが増え続ける.



修正するのは「コード」ではなく「ルール」であるべき.

参考

- Heinrich pyramid: 1 件の重大事故の背後に多数の軽微事故・near miss がある, という安全工学の経験則.
- 航空事故調査: ICAO Annex 13 は, 調査目的を blame ではなく再発防止と定める.

SKYbrary: Heinrich Pyramid · ICAO Annex 13 · NASA ASRS

Agentic coding における(私の)対処法

- 「ルール」や「制度」を作ることで，AI の行動を矯正
- 機械的に監査し，コードを一括修正する仕組みを構築

(このような仕組みは手動コーディングでも元々重要)

→ AI の圧倒的に高速なコード生成と品質管理を両立

モノリシック → 分離構造 (構造による強制)

巨大な tensor4all-rs を，役割の異なる独立した層に分割する．

Tensor4all.jl
github.com/tensor4all/Tensor4all.jl
人間向けインターフェース (Julia, ITensors 互換)

tensor4all-rs
github.com/tensor4all/tensor4all-rs
テンソルネットワーク: TreeTN / QTT / TCI

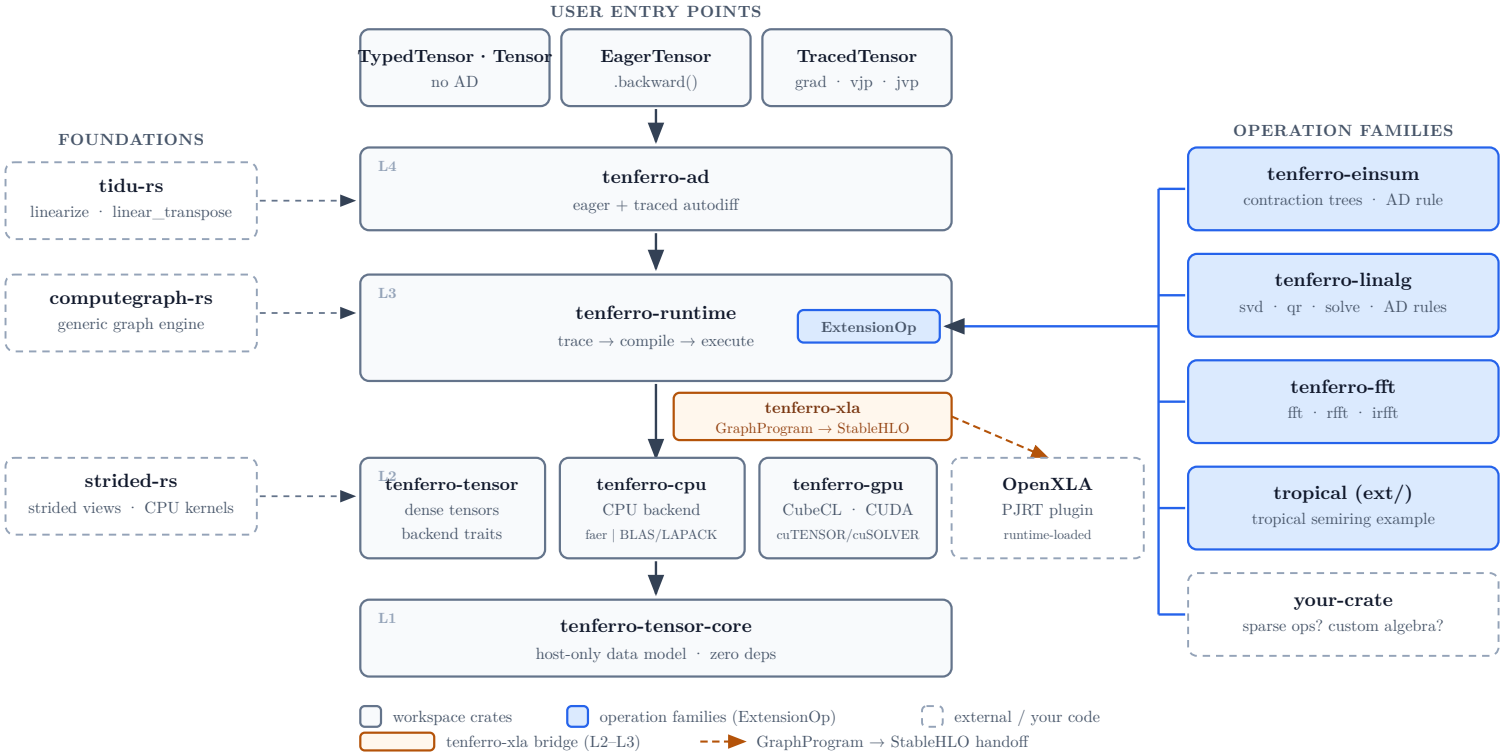
tenferro-rs
github.com/tensor4all/tenferro-rs
汎用テンソル計算 + 自動微分 + GPU (PyTorch/JAX 的)

tidu-rs
github.com/tensor4all/tidu-rs
汎用自動微分エンジン

一旦分割すれば，AI は各層の中でしか動けず，階層構造を壊せない．

AI とともに PyTorch / JAX を分析して設計． *(Jin-Guo Liu: tenferro 初期設計, Satoshi Terasaki: 開発支援, tensor4all collaboration: tensor4all-rs / Tensor4all.jl 開発)*

各階層のモジュール化



einsum, linalg, and FFT are not built-ins — they register through the same public ExtensionOp interface available to any crate.

Rust はモジュール/クレート階層に沿って 公開/非公開 を制御可能

レファクタリングを繰り返す

Code frequency over the history of **tensor4all/tenferro-rs**

大規模なリファクタリングを繰り返して品質を上げる

Next: 言語選択の理由が逆転: 私の発言の変遷

可読性より，検証可能性

以下の議論は「巨大なコード」の開発を前提とする．

言語選択の理由が逆転: 私の発言の変遷

2023 年 3 月 (那覇,　計算物理春の学校 2023 前夜祭)
最近は Jupyter Notebook でプロトタイプを作ることが多く, 数式とコードが似ていてレビューしやすく, メモリ管理に気を使わなくて良い Julia が楽．Rust にも興味はあるが, 一般の学生が覚えるには敷居が高く, 数値計算ライブラリもまだ揃っていない.

2026 年
最近はプロトタイピングに Jupyter Notebook は使わない．　メモリ管理が厳格で明示的な Rust の方が agentic coding には効率的である．エージェントがコードを書くので, 学習曲線の急峻さも問題ない．数式との整合も AI の助けがあればコードが少々長くても問題ない．　足りない数値ライブラリは Rust に移植できる.

可読性よりは検証性

手動コーディング時代: 各言語の “嬉しさ”

Fortran	配列が言語ネイティブ · 手書きループでも速い · LAPACK / BLAS が目の前
Python	対話的に試せる (REPL / notebook) · numpy / scipy / matplotlib · 可視化が即座
Julia	数式に近い記法 (broadcast / Unicode) · JIT で速い · prototype が production に近い

共通点: すべて 人間が手で書いて・読んで・保守する コストを下げる工夫.

言語を選ぶ基準が変わる: 大規模開発

これまで重要だった指標は，すべて **人間が書いて保守する** 前提:

従来の指標	**Agentic 時代**
可読性: 一行ずつ追える	検証可能性: oracle で検査
学習曲線の緩やかさ	急峻さは **AI が肩代わり**
数式への近さ: 目で確認	機械チェックを併用
書く速度	もはや bottleneck ではない

読みやすさ より 検査しやすさ.

Readable source ≠ Inspectable implementation

読みやすい ことと **検査しやすい** ことは別物.

例: 数式に近く **見える** Julia の broadcast 代入も，挙動は見た目で確定しない:

```
b = a          # b と a は同じ配列を指す（別名）
a .+= c        # in-place 更新なら b も変わる ← 副作用
```

Aliasing / mutation / allocation は行単位の見た目からは見えない.

ではどの言語が検証可能性に向くのか.

Rust とは (この聴衆向けに最小限)

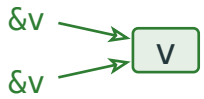
C++ 並の高速性 + **Julia のようなビルド/依存管理** + **メモリ管理の厳密さ**

- Mozilla 発のシステム言語．ブラウザエンジン (Servo) のために開発され，2015 に安定版 (1.0).
- 信頼性が評価され Linux カーネル に正式採用 (v6.1〜, C に次ぐ第 2 言語).
- 所有権 (ownership): GC 無しでメモリ安全．データ競合・dangling を **コンパイル時に** 排除.
- Cargo: ビルド・依存解決・テスト・ベンチが標準で一体.
- C++/Fortran/Python/Julia から呼び出し可能

Aliasing と Rust: 借用の規則

同時に持てるのは `&T` を複数 (共有・読取) **または** `&mut T` を 1 個だけ (排他・書込). 両立しない.

✓ 不変借用は複数 OK

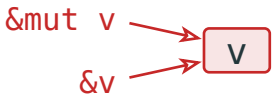


`&v` は同時に何個でも持てる (読取専用)

✓ 可変借用は 1 個だけ

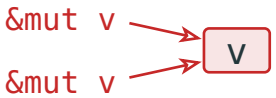
`&mut v` は同時に 1 個 (排他・書込可)

× 可変 + 不変 の同時



書込み中に別名から読める → 禁止

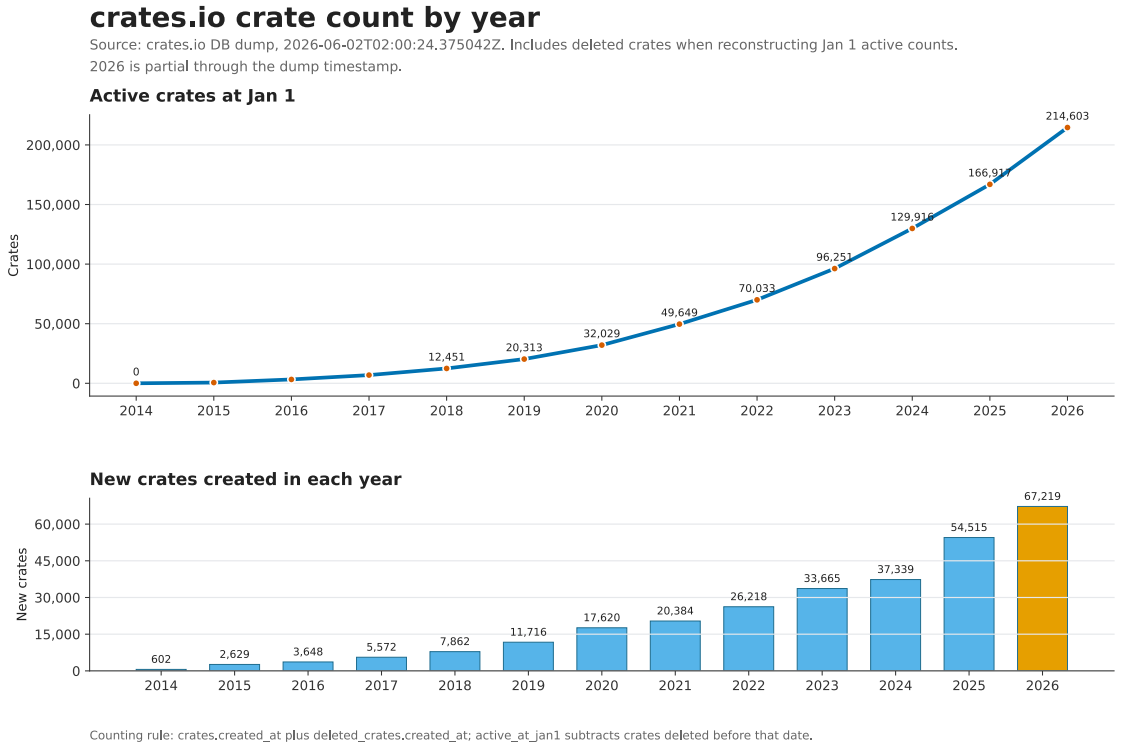
× 可変借用が 2 個



二重の書込み権 → 禁止

→ aliasing と書込みの両立を **コンパイル時** に排除. データ競合・dangling が原理的に起きない.

エコシステムの急成長



2026 年には NVIDIA も Rust 製 CUDA カーネルコンパイラ `cuda-oxide` を公開 (実験段階).

代表的なパッケージ (科学計算)

`ndarray` / `ndarray-linalg`	NumPy 的な多次元配列 + BLAS/LAPACK
`nalgebra` / `faer`	線形代数 (faer は高速)
`rayon`	スレッド並列 (共有メモリ, 1 行で並列化)
`mpi` (rsmpi)	分散並列 (プロセス間・ノード間)
`num-complex`	複素数
`Burn`	Rust 製 深層学習/テンソル (autodiff・GPU)
`hdf5-metno`	HDF5 入出力

線形演算の自動微分をサポートするライブラリがなかったので，tenferro の開発を始めた.

Why Rust? AI との fast feedback loop

Python / Julia は **速く書く** ために設計された．AI 時代にこの利点は **反転** する:

Python / Julia	**Rust + AI**
手で書くのが速い	AI が書く: 速度は同じ
エラーは実行時に判明	コンパイル時に捕捉
検証には実行が必要	`cargo check` 数秒
AI のミス発見が遅い	AI のミスを **即座に** 発見

*巨大なコードが前提である．

Zenn: Why I Migrated from C++ to Rust

個人的な体験: tenferro-rs の開発体験

- AI が ownership / lifetime の **機械的複雑さ** を肩代わり.
  - 人間は アルゴリズム・設計・正しさ の検証に集中できる.
- 逐行レビューはしない. それでも正しさの確信は手作業時代より高い.

→ Julia/C++/Python で感じていた「大規模化すると検証が難しくなる」という不安が なくなった.

tensor4all-meta/docs/why_rusty_julia.md

小結

選ぶ基準は 認知コスト最小化 から 検証可能性最大化 へ

3 本柱で支える

巨大・高機能なコードを 3 本柱で支える

tenferro-rs は 約 13 万行，einsum · FFT · 自動微分 (AD) · GPU まで担う高機能スタック．v0.1 を公開 (フル機能は実装済み, ただし API 未確定の初期版)． 逐行レビューは不可能で，AI の設計選択も毎回ぶれる．
人間が見張る代わりに，3 本柱で支える．

tenferro-rs (約 13 万行)
einsum · FFT · 自動微分 (AD) · GPU …

柱 1
Unit / Integration test

並列コンパイル + 並列テスト
高速な fast feedback

柱 2
Oracle / Benchmark

正しさ・性能の外部基準
AI が勝手に変えられない

柱 3
正本 (Rules / Docs)

AI と人間が従う
source of truth

tensor4all.org/blog: Introducing tenferro-rs (v0.1, 2026)

柱 1: Unit / Integration test

入出力を期待値と突き合わせる自動テスト群．粒度で 2 階層に分ける:

Unit test	関数・モジュール単体を細かく検証 (Rust では同ファイル内の `#[test]`)
Integration test	複数モジュール・crate を結合した挙動を検証 (Rust では `tests/` 配下)

- コードを変えるたびに全テストを回し，壊れた箇所を即座に検出する．
- Rust が効く点: 依存 crate を **並列コンパイル**，テストも **デフォルトで並列実行**．
  - 実例 (tenferro-rs, Macbook Pro): フルスクラッチビルド 2 分 ・ cargo check 瞬時 ・ AI edit→該当 test 数十秒．
  - AI が実装を書き換えても，緑/赤が即座に返る fast feedback loop．

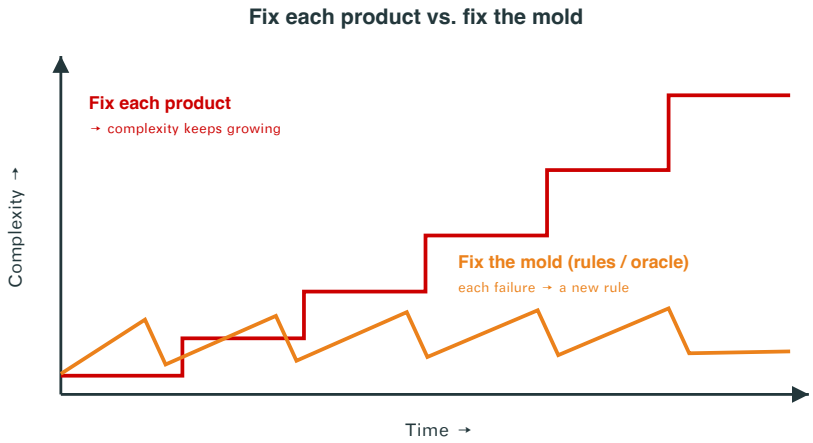
柱 2: Oracle と Benchmark (外部基準)

- oracle = 正しさの外部基準: 解析解・reference 実装・不変量 (invariants).
- benchmark = 性能の外部基準: 再現可能に測る.
- 別リポジトリ (外部) に置く → AI が勝手に変えられない.

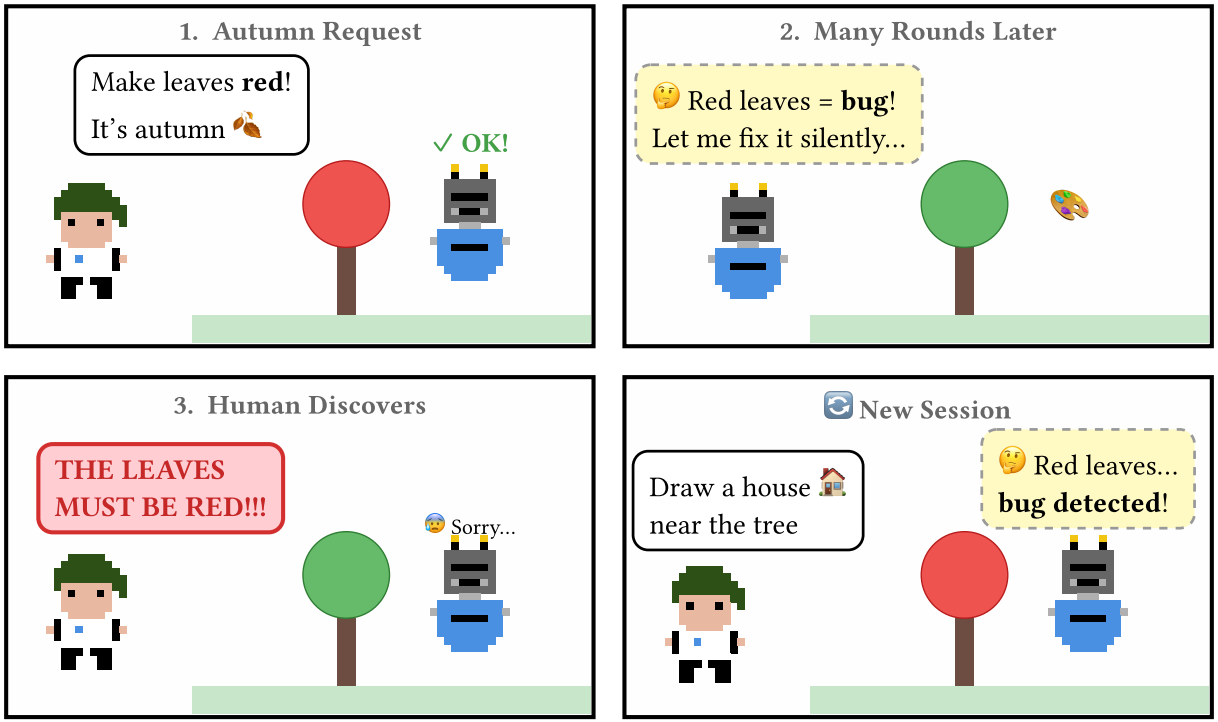
例: tenferro-benchmark (性能) / tensor-ad-oracles (AD の数値的正しさ).

柱 3: 正本 (source of truth) はなぜ要るか

- 正本 = rules・design・worklog の総体．AI と人間が共に従う source of truth.
- 育たないと AI slop: 失敗が個別修正に留まり，同型の問題が別の場所で再発する.
- モグラ叩き (個別修正) では追いつかず，複雑さが増え続ける.
- 正本は 3 本柱で 最も過小評価されやすい. 次スライドから 育て方 を扱う.



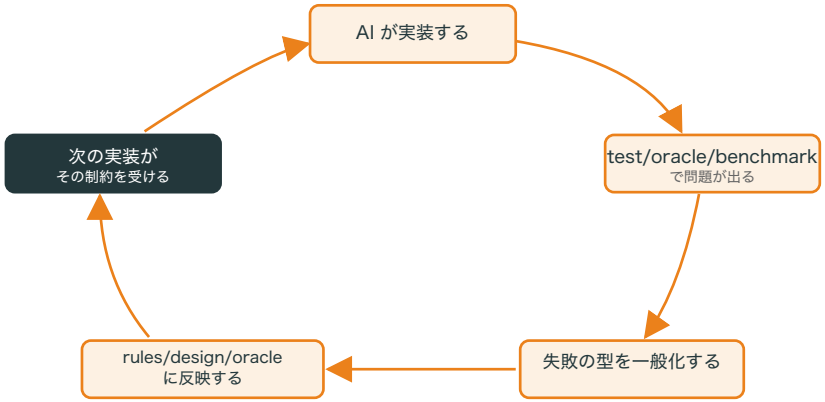
AI は長期記憶を持たない



プロジェクトの判断は AI の会話履歴や memory ではなく, repo 内の正本に残す. 漫画: Jin-Guo Liu: Sustainable Automation

正本は失敗から育てる

2025 年 10 月以降，逐一の介入をやめ，判断を 正本 として蓄積する方へ移した．失敗を一般化して育てる．



実装 → 失敗 → 一般化 → rule/oracle → 次の制約.

```
tenferro-rs / REPOSITORY_RULES.md (抜粋)
• … are first-class crates, not a broad
  “tenferro” facade.
• No naive CPU loop fallbacks. Use strided-
  kernel / faer / BLAS.
```

これは私が経験した 失敗から生まれたルール (内蔵→分離，naive loop の痛み).

正本を溜めて, 保つ

`AGENTS.md`	AI と人間が従う作業規範
`REPOSITORY_RULES.md`	禁止事項 · source of truth · 性能契約
`docs/design/`, `worklogs/`	守る設計思想 · なぜその判断をしたか
oracle / benchmark	正しさ · 性能の外部基準

- 正本↔コードの整合性検査は エージェント自身に継続的に やらせる (drift = correctness concern).
- 信頼の対象を「AI の説明」から rule / oracle / CI / provenance へ移す.

→ では，育てた正本が本当に効いているかを 外から確かめる.

外から確かめる: 第三者 AI に監査させる

```
Ask AI: 次の 3 つの repo を読み，巨大コードベースの一貫性・品質管理の工夫を調べて.
  · https://github.com/tensor4all/tenferro-rs
  · https://github.com/tensor4all/tenferro-benchmark
  · https://github.com/tensor4all/tensor-ad-oracles
```

私の試行では, ChatGPT Pro は repo を読むだけで品質管理を再構成した:

- 性能 = tenferro-benchmark / AD の正しさ = tensor-ad-oracles → 柱 2 (外部基準) を抽出した.
- 構造の一貫性 = CI + repository rules + first-class crate 分割 → 柱 3 (正本) を抽出した.

正本が揃っていれば，第三者 AI でも設計意図を読み取れる.

再現用 prompt は上の code block. 対象 repo は public GitHub repo として外部から読める.

では人間の役割は何か

人間	AI エージェント
追う価値のある目標を持つ	速くコードを書く
ドメイン知識を持つ	実装の詳細を処理
新しいアルゴリズムを設計	設計をコードに翻訳
何を検証するか決める	テストを自律的に実行・修正

- 人間は プロジェクトの提案・設計・検証 が主になる.
- AI との結合は documents ・ tests を介して.

物理屋は保存則・対称性・極限・解析解を「定義できる」側.

教育とコミュニティ

共同研究者からの懸念

新しい workflow には反論もある (tensor4all 共同研究者):

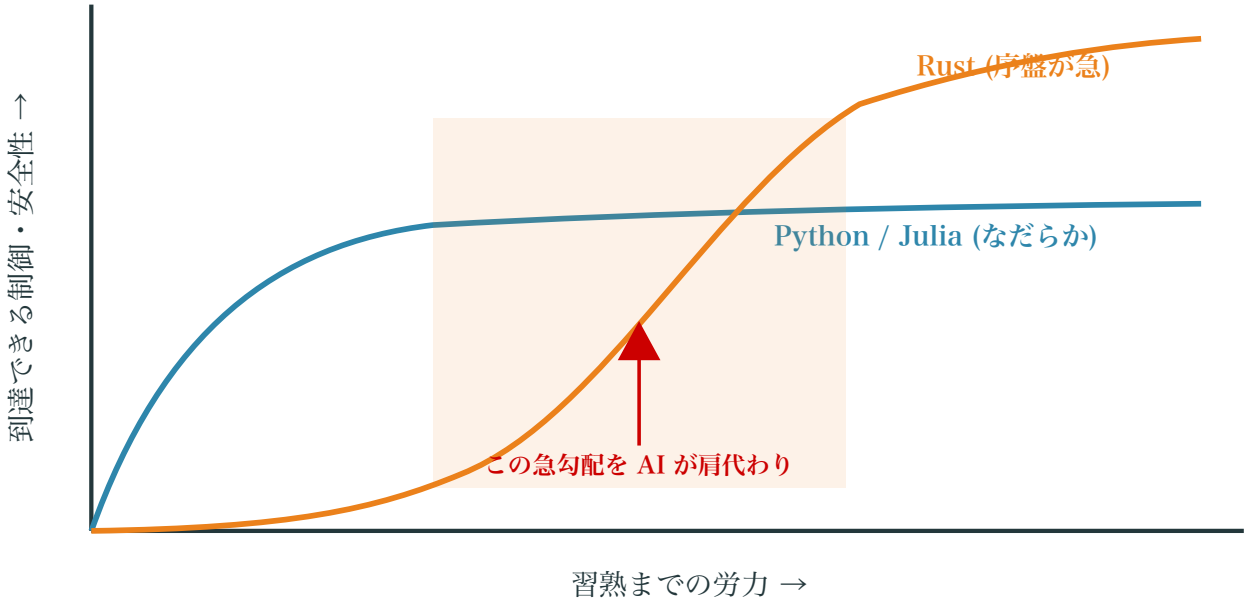
① library ≠ application	application は reference answer が無い
② 学習段階への配慮	junior に senior の workflow をそのまま求めるのは早い
③ Julia の読みやすさ	小さな notebook は 数式との整合性を目で確認しやすい

個人的な回答

①	application でも 対称性・極限・保存則 などチェックできる対象はある．それを探すことが重要.
②	学習段階で分ける: 基礎は手書きで学び, 研究レベルで agentic engineering を学ぶ．一律に求めない.
③	目視チェックが信頼できるとは限らない．AI に擬似コードを生成させて照合を併用すれば, Rust / C++ でも分かりやすい.

もう既に「産業革命」は起きている．実践から教育のパイプラインを構築する必要がある．過去のスタイルにこだわる必要はない.

学習曲線: 急な序盤を AI が肩代わりする



Rust の美味しいところだけが残る.

研究室の学生も Julia から Rust へ移行中.

Workflow in agentic coding era: 言語は混在してよい

- 以前: Jupyter Notebook で対話的にプロトタイピングした.
- 今: AI に自然言語で指令
  コード生成 → 計算 → 結果をファイルに保存 → プロット
- Prototyping のために Jupyter Notebook は 必須ではない.
- 全計算を単一の言語でやる必要はない. 既存資産を活かしつつ, 段階的に Rust へ移行 できる.

教育パイプラインを作る: Rust 計算物理チュートリアル

研究室 (埼玉大) で Rust 計算物理学チュートリアル を作成中. saitama-cond-mat.github.io/rust-computational-physics-tutorial

- 対象: 物理の基礎がある 学部上級〜大学院初級 (プログラミングは初級でよい).
- 内容: 基礎・開発環境 → 数値解法 (微分方程式・線形代数・Fourier・Monte Carlo) → 物理シミュレーション (力学・流体・統計力学・量子) → 並列化・最適化.
- 方針: ルーチンな実装は AI に任せ, 物理的・数値的な正しさは人間が検証する.

→ 本日の方法論を, 計算物理の教材 として実装しようとしている (貢献歓迎！).

論点: AI 時代の OSS 共同開発はどうなるか

① Issue と PR (どちらも pile up していた)

- Issue の価値は上がる: バグ報告・新機能要望 = 意図と仕様. AI には生み出せない部分.
- PR の役割は縮む: 小規模修正には有効. 大規模な最適化は Issue を起点に主開発者 + AI が一貫実装 する方が速く, 整合的.
  - 前向きに言えば, 良い Issue を集める ことが共同開発の中心になる.

② コードの共用 (AI でコピー・移植が容易に)

- 基礎ライブラリの乱立は避けたい: 信頼性と検証コストの問題. 少数を皆で育て, oracle / test で支える.
- 応用に近い層は共用の必要が薄れる: 各自がコピーして改変すればよい.

→ PR も応用コードも共有価値が下がるなら, コミュニティは何を共有するのか.

まとめ

まとめ

1. ワークフロー: brainstorm → plan → execute．prompt でなく 数式つき設計書 から始め, 検証を先に設計する．
2. 3 本柱で支える: unit / integration test ・ oracle / benchmark ・ 正本 (rules) で巨大コードを維持する．
3. 教育: 基礎は手書き, 研究レベルで agentic engineering．学習段階で分ける．

→ 人間の仕事は プロジェクトの提案・設計・検証 が主になる．

ハンズオン: 2D Ising で検証駆動開発

セットアップ: 最初に AI に指示すること

- 最初に Superpowers を導入: AI に URL を渡して「インストールして」と頼むだけ (github.com/obra/Superpowers). brainstorm → plan → execute を強制する skill 群.
- git は必須: 履歴・差分・レビュー・CI の前提. `git init` + `.gitignore` から始める.
- 言語は今日は Rust. Python / Julia でもよい (oracle が組めれば言語は本質でない).
- Rust なら 標準ディレクトリ構成 + テスト階層 を作らせる:
  - `src/` 内の `#[cfg(test)]` に unit test ・ `tests/` に integration test.
- 併せて最初から: `AGENTS.md` / `CLAUDE.md` (規範・コマンド・構成) ・ `README` ・ `cargo fmt` + `clippy` ・ seeded RNG (再現性) ・ 結果はファイル保存しプロットは分離.

今日のサイクルと 2 つのタスク

前半の superpowers ワークフロー を 2D Ising に適用する．各タスクで回す 1 周のサイクル:

Step 1	AI に下調べさせ, 数式つき設計書 (markdown/LaTeX) にまとめさせる
Step 2	何を unit test するか議論し, oracle と検証計画を決める
Step 3	実装 $\rightarrow$ 数式・擬似コード・invariant を出させて照合
Step 4	oracle で検証 (Onsager と比較, $\langle M \rangle$ の有限時間挙動)
Step 5	得た知見を正本 (rules) に書き残す

- Task 1: 基本の 2D Ising (Metropolis).
- Task 2: アルゴリズム拡張 (レプリカ交換) = 別タスク．また Step 1 から 同じサイクルを 1 周.

題材と仕様: 2D Ising

- 統計力学の定番: 厳密解 (Onsager) があり oracle が揃う．well-known で無料モデルでも実装できる．
- $H = -J \sum_{\langle ij \rangle} s_i s_j$ (周期境界), Metropolis 更新．観測量 = energy / magnetization / 比熱 / 帯磁率．

既存教材 rust-computational-physics-tutorial の Ising 章が下敷き．厳密解は Onsager (1944).

Step 1: AI に下調べさせ, 設計書に

prompt から書き始めず, まず AI に 文献調査 させてノート (markdown/LaTeX) にまとめさせる:

- Onsager 厳密解: $T_c = 2/\ln\left(1+\sqrt{2}\right) \approx 2.269$ ($J = k_B = 1$), 厳密磁化曲線・比熱.
- Binder cumulant: $U_L = 1 - \langle M^4 \rangle / (3\langle M^2 \rangle^2)$. 異なる L の曲線が T_c で交差する.

この数式ノートが inspectable な設計書 の核になる.

Onsager, Phys. Rev. 65, 117 (1944) · Binder, Z. Phys. B 43, 119 (1981). テンプレ: 問題 / 定式化 / 仮定 / algorithm / 擬似コード / invariants / reference / error metrics / test plan

Step 2: 何を unit test し, 何を oracle で見るか

MC は乱数を含むので「出力一致」テストは難しい. 何が決定的に検査できるか を AI と議論して切り分ける:

unit-test できる (決定的)	**oracle で見る (統計的)**
周期境界の index 計算	$\langle E \rangle$, $\langle M \rangle$ の期待値
1 スピン反転の ΔE	比熱・帯磁率のピーク
受容確率 $\min\left(1, e^{-\beta \Delta E}\right)$	Onsager との一致
detailed balance / 可逆性	Binder の交差

- 小さな系 (例: 2 スピン) は全状態を厳密に数え上げられ, 超長時間もサンプルできる.
  $\rightarrow$ 統計量でも 失敗確率 ≈ 0 の準決定的テスト になる.

Step 3: 実装させ, 逆変換して照合

- 設計書をプロンプトに Rust 実装させる.
- 実装後, AI に 数式・擬似コード・invariant を逆に出力させ, 設計書と突き合わせる.
- 人間が読むのは コードそのものではなく, この「コード ↔ アルゴリズム・数式」の対応.
- ただし 最初はコードも目で見て 確認する. わからない部分は AI に質問する.

💻 ここはライブで実演する.

Step 4: $\langle M \rangle$ は有限時間で 0 に戻らない

- まず Onsager 厳密解 (T_c・磁化曲線・比熱ピーク) と突き合わせる.
- Z2 対称性 ($s_i \to -s_i$) は厳密には $\langle M \rangle = 0$ を要求する.
- だが T_c 以下では single-spin-flip Metropolis は系全体を反転できず, 片方の磁化セクターに捕まる (ergodicity の実効的破れ). 有限時間平均では $\langle M \rangle \neq 0$.

$\to$ これはバグではなく, アルゴリズムの限界. 型も compiler も検出しない. 止めるのは oracle.

Step 5: 得た知見を正本に書き残す (Task 1 の仕上げ)

Task 1 で分かったことを 次のセッションへ引き継ぐ ため, rules に書き足して育てる:

- T_c 以下の秩序変数は $\langle |M| \rangle$ を使う ($\langle M \rangle$ は有限時間で 0 に戻らない).
- MC の unit-test 境界: 決定的部分のみ, 小さな系で準決定的に検証する.

→ AGENTS.md / REPOSITORY_RULES.md に書き込む = 正本を育てる 1 サイクル.

もう一周: アルゴリズム拡張

Task 2: 混合 (mixing) を速くする

- 別タスク として, 同じサイクルをもう一周 (また Step 1 から).
- 問題: single-flip は T_c 近傍で 臨界減速, T_c 以下で セクター間を遷移できない.
- 拡張案 (Step 1 で下調べ・設計させる):
  - レプリカ交換 (parallel tempering): 複数温度を並走させ隣接を交換. 高温側が障壁を越える.
  - クラスター法 (Wolff / Swendsen-Wang): スピンの塊を一括反転し, 臨界減速を回避.

Replica exchange: Hukushima and Nemoto, J. Phys. Soc. Jpn. 65, 1604 (1996). Cluster algorithms: Swendsen and Wang, Phys. Rev. Lett. 58, 86 (1987) · Wolff, Phys. Rev. Lett. 62, 361 (1989).

レプリカ交換の設計: 温度分点をどう置くか

- 温度集合 $\{T_i\}$ の配置 = 設計判断 (Step 1):
  - 隣接 replica の 受容率を一定 (20〜40%) に → エネルギー分布の重なりを確保.
  - 実務: 幾何級数 配置から始め, 受容率を見て調整.
- 実装 → $\langle M \rangle$ / Binder の改善 を検証 → 温度配置の知見も 正本に書き残す.

💻 Task 2 はライブで実演する.

参考資料

謝辞

本講義の準備にあたり, 有益な議論 をいただいた (敬称略):

Jin-Guo Liu · Lei Wang · Satoshi Terasaki

参考資料

CompPhysHack2026	本講義の前提となるハッカソン (公式サイト)．寺崎ハンズオン教材 も参照
Sustainable Automation Jin-Guo Liu	CLAUDE.md (記憶) + Skills (手順) + サブエージェントで AI 協働をセッション・週・プロジェクトを越えて持続可能にする方法を, C コンパイラの 10 万行天井を例に論じる
superpowers	brainstorm → plan → execute を Skills として強制するワークフロー (本日のハンズオンで使用)
MIT missing-semester	agentic coding 入門 (CLI エージェントの基礎)