PTNT: Process Tensor Network Tomography

Fazeleh Kazemian, Angus Southwell, John Kam, Giacomo Pantaleoni, Remmy Zen, Gavin Brennen, Tom Stace, Kavan Modi, Gregory White

Self-consistent process-tensor tomography with tensor networks, physics-aware runtime layouts, and heterogeneous execution

Hosted at Monash University   Supported by ASCA

What does “noise” mean in a quantum computer?

Ideal picture

- We ask for a gate.
- The same gate should do the same physical action every time.
- Errors should be small, local, and memoryless.

Real hardware

The same commanded gate can behave differently because the device remembers the past, controls drift, and operations interfere.

Three important noise mechanisms

- **Environment memory**: a hidden bath keeps history, so an error at time t can depend on earlier events.
- **Control drift**: the pulse electronics change slowly, so the “same” gate is not identical.
- **Crosstalk / spillage**: driving one qubit can disturb another qubit or the environment.

Goal: Learn these effects from experimental data instead of hiding them inside a simple Markovian error bar.

What is the actual problem?

What we observe

- many circuit rows
- control sequences
- measurement records
- shot counts / empirical probabilities

What is hidden

- multitime process noise
- imperfect local controls
- correlations across time and space

What we want

- a learned process model
- learned control imperfections
- predictions that remain self-consistent

This is an inverse problem

We do not simulate forward from a known model. We infer a hidden model from many imperfect experimental rows.

Why this matters

For quantum computing

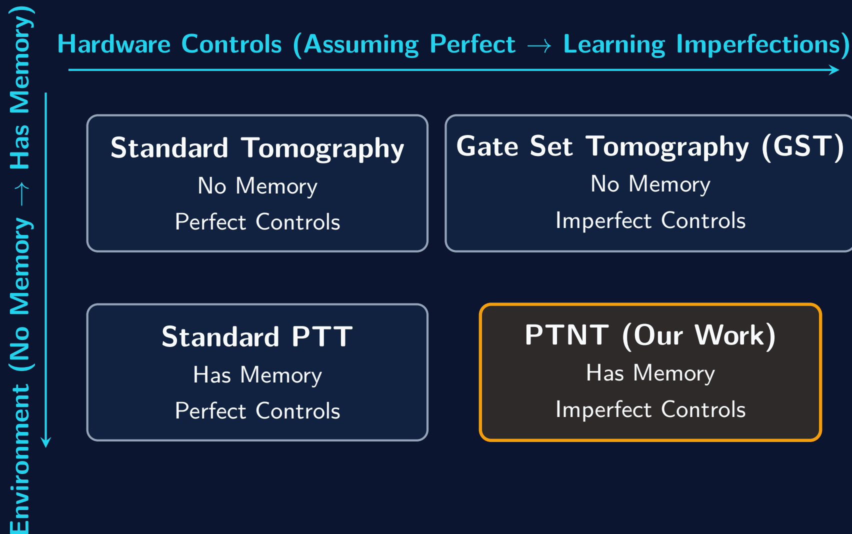
- **Quantum error correction**: decoders and thresholds depend on the real noise, not the idealized one.
- **Control and calibration**: different noise sources require different fixes.
- **Noise-aware compilation**: better models can improve how we choose or decompose gates.

For hardware and systems teams

- richer diagnostics for device behavior
- better separation between background process noise and control-side problems
- a reusable software workflow for large experimental datasets

Where PTNT sits among existing methods

Does the environment have memory—meaning, is it non-Markovian?are the controls perfect?



Where PTNT sits among existing methods

question	GST	standard PTT	PTNT
learn controls jointly?	Yes	No	Yes
models process memory?	No	Yes	Yes
models control correlations?	No	No	Yes
models spillage / process-control interplay?	No	No	Yes
computational burden	lower	medium	highest

Key message

PTNT closes a real gap: it learns multitime process noise and imperfect controls together.

Markovian means "the current error does not depend on past hidden state." PTNT is built for the cases where that assumption fails.

PTNT as an AI-style differentiable inverse problem (engine)

1. The Guess: Per-row prediction (forward pass)

For one empirical row r, PTNT predicts:

$$p_\theta(r) = \text{contract}(\Upsilon_\theta, \tau_r)$$

Here, Υ_θ is the learned process-tensor model and τ_r is the tester built from the observed row.

2. The Reality Check: Training objective on a mini-batch $\mathcal{B}$

$$\mathcal{L}_\mathcal{B}(\theta) = -\sum_{r\in\mathcal{B}} w_r \log p_\theta(r) + \lambda_{\text{caus}} C(\theta) + \lambda_{\text{TP}} T(\theta)$$

Calculates a penalty score for how wrong the prediction was compared to the experimental data.

3. The Fix: Backward pass and update

$$\nabla_\theta \mathcal{L}_\mathcal{B}(\theta) \quad \Longrightarrow \quad \theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}_\mathcal{B}(\theta)$$

Automatic differentiation runs backward through the tensor-network contraction, then the optimizer updates the parameters.

Why the learned object is a process tensor

A normal quantum channel is not enough

A usual channel describes one input state and one output state: $\rho_{\mathrm{out}} = \Lambda(\rho_{\mathrm{in}})$.
But our experiment has many interventions through time: $A_0,\ A_1,\ \ldots,\ A_{T-1},\ \Pi_x$.
So the hidden object must describe how the device responds to a whole **sequence** of controls.

Process tensor view

The process tensor is the multitime object:

$$\Upsilon_{T:0} \in \mathcal{B}(\mathcal{H}_{o_T} \otimes \mathcal{H}_{i_T} \otimes \cdots \otimes \mathcal{H}_{o_0})\,.$$

A circuit row becomes a tester τ_r, and the prediction is: $p_\theta(r) = \mathrm{contract}(\Upsilon_\theta, \tau_r)$.

PTNT is not learning one Markovian error channel. It is learning a **multitime memory object**: the process tensor that maps whole experimental histories to probabilities.

Why this becomes an HPC problem

Dominant time cost per training step

$$\mathcal{T}_{\mathrm{step}} \approx O(|\mathcal{B}|\, C_{\mathrm{contract}} + C_{\mathrm{reg}})\,.$$

The exact contraction cost depends on qubit count, time depth, bond dimension, and contraction policy.

Dominant memory cost per step

$$\mathcal{M}_{\mathrm{step}} \approx O(P_\theta + W_{\mathrm{contract}} + S_{\mathrm{batch}})\,.$$

Model parameters, contraction workspace, and the chosen batch representation all matter.

Dense baseline warning in the code

The dense `single_pt` baseline scales as

$$N_{\mathrm{params}} = r\, 2^{n_q(2n_t+1)}.$$

case	value	implication
$n_q = 4$, $n_t = 4$, $r = 2$	1.374×10^{11}	complex parameters
complex64 storage	≈ 1.0 TiB	before optimizer overhead
complex128 storage	≈ 2.0 TiB	before contractions

Conclusion: without structure, even moderate process-tensor models become memory-prohibitive. PTNT therefore uses a PEPO / LPDO tensor-network ansatz.

What the PTNT software does end to end

Input	Canonical objects	Compiled runtime	Training & report
Simulate, shot data, or saved dataset file	`PTNTDataset` and `ControlLibrary`	Dense local arrays or compiled event data	PEPO / LPDO model, TNOptimizer, metrics, artifacts

For open-source users, this is exposed through CLI + YAML, Python API, saved datasets, and generated reports.

Key systems idea: one public schema, two runtime layouts

handle memory: how software hand the data

Stable public schema

- `PTNTDataset`
- `ControlLibrary`
- one API for simulate / shot_data / file

Users describe experiments once. The runtime then chooses the right machine layout.

Two lowered runtime layouts

- **Local fast path**: dense operator arrays for regular single-site workloads.
- **Generic nonlocal path**: compiled control kernels plus bucketed compiled event datasets for irregular spacetime supports.

Note

The contribution is not "a universal new data structure." It is a **domain-specific runtime design** that keeps one public schema while lowering into different machine layouts that match the workload.

Physics-aware runtime data layout

Public schema: PTNTDataset + ControlLibrary

Local fast path

Dense operator arrays for common local workloads

- Full-U: $(B, Q, T+1, 2, 2)$
- RZ mode: $(B, Q, 3(T+1)+1, 2, 2)$

Generic nonlocal path

uses two numeric objects. Bucketed struct-of-arrays runtime for event-based data

CompiledControl Kernels

1q / 2q unitary tables
1q / 2q Kraus tables
integer index maps

CompiledEvent Dataset

unique circuits: U
measurement rows: R
bucket metadata

- Store unique circuit programs once
- Keep weighted measurement rows separate
- Fewer Python objects in the hot path

Main message: The runtime chooses the cheapest faithful lowering: dense local tensors for regular structure, bucketed compiled events for irregular nonlocal structure. We ensure both perfect correctness and maximum hardware efficiency

What the two layouts buy us

Question	Local fast path	Generic compiled-event path
Best case	local unitary or local instrument rows with regular spacetime structure	nonlocal, sparse, or irregular event-based rows
Main time win	regular dense batches, cheap indexing, cheap batch assembly, simpler contraction setup	avoid rebuilding repeated circuit programs, reduce host-side object overhead during preprocessing and batching
Main memory win	contiguous numeric arrays and better cache/device reuse	store unique circuits once (U) and keep measurement rows separate (R); much better when many rows share the same circuit skeleton
What it protects	throughput and device regularity	correctness and generality for nonlocal workloads
JAX / GPU fit today	strongest fit: dense numeric arrays map naturally to batched device execution	improving fit: already compiled and bucketed, but still less regular on the current reference backend

Exact data structures and how they map to JAX / GPU

Local fast path

$$\texttt{fullU}[B, Q, T{+}1, 2, 2]$$
$$\texttt{RZ}[B, Q, 3(T{+}1){+}1, 2, 2]$$

Interpretation

- one batch axis B
- one qubit axis Q
- one time axis
- each local operator is a 2×2 block

Device mapping

dense arrays $\rightarrow$ `jax.asarray`
$\rightarrow$ probability eval $\rightarrow$ autodiff

Best for: regular dense batches.

Generic compiled-event path

Control kernels

$$\texttt{single_unitaries}(N_1, 2, 2),\ \texttt{pair_unitaries}(N_2, 4, 4)$$
$$\texttt{single_kraus}(N_1, R_1, 2, 2),\ \texttt{pair_kraus}(N_2, R_2, 4, 4)$$

Bucketed event data

circuits: $(U, n_{\text{slots}}, k_{1\,\text{max}})$ and $(U, n_{\text{slots}}, k_{2\,\text{max}})$
rows: (R, M_{max}) site arrays + row weights

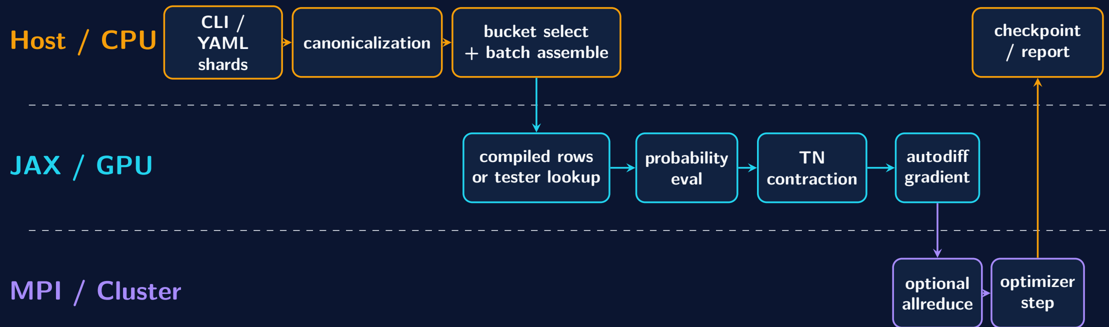
Current mapping

compiled tables $\rightarrow$ batch assemble $\rightarrow$ device work

Today: partly compatibility-oriented on irregular paths.

local dense arrays are best when every site is regular; compiled-event buckets are best when many rows share irregular circuit structure.

Parallel workflow: how a PTNT job is forked



Parallel axes:
rows/circuits, mini-batches, and optional distributed ranks.

the host prepares canonical data, the device computes gradients, and MPI reduces before the optimizer steps.

Engineering choices that protect correctness and performance

SEngineering choices that protect correctness and performancemall choices, when building a software, we need to handle two things:

Correctness side

- explicit PTM / Choi / SuperOp conventions prevent silent interpretation bugs
- control semantics are separated from where controls are allowed to act
- deterministic parameter ordering keeps template binding reproducible

HPC side

- controls are compiled before training so the hot path sees dense numeric kernels
- the CLI caps BLAS / OpenMP threads before importing heavy dependencies
- JAX GPU preallocation is disabled by default for shared systems

These are the software choices that reduce ambiguity, improve reproducibility, and make the workflow safer on clusters and shared GPU nodes.

Execution policy: where PTNT runs today

rows / circuits → mini-batches → JAX device kernels → optimizer step

Stage	CPU / host	GPU / JAX
CLI / YAML / shards	High	–
dataset canonicalization	High	Low
compiled row / batch assembly	High	Low
dense probability evaluation	Low	High
TN contraction	Low	High
autodiff gradient	Low	High
optimizer step	Med	Med
checkpoint / report	High	–

This is an architectural execution map, not a sampled hardware profile.

What PTNT already supports

- mini-batch training
- causality and TP regularization
- optional pure-JAX backend
- validation tracking & best-model retention

Ongoing....

Dense numeric batches are the most device-friendly path. Generic nonlocal batches are already compiled and bucketed, but some current workflows may still use compatibility-oriented tester handling.

What is already strong, and what is the next milestone?

Strong today

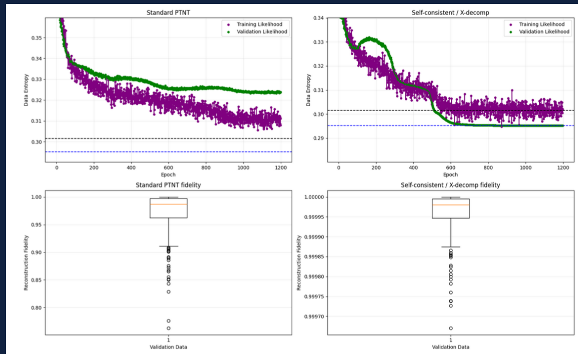
- canonical dataset and control abstractions
- dual runtime layouts matched to local vs nonlocal structure
- compiled numeric kernels for event-based execution
- mini-batching, JAX autodiff, optional MPI hooks
- end-to-end nonlocal event-based training path

Next HPC milestone

- push more of the generic compiled-event path into a native numeric contraction backend
- reduce compatibility-oriented tester materialization
- make the irregular nonlocal path as device-friendly as the dense local path
- turn architectural GPU readiness into consistent time-to-solution wins

Main result: better local models and strong local GPU speedup

Model quality on local PTNT workloads



Standard PTNT vs Self-consistent / X-decomp: validation likelihood and reconstruction fidelity improve substantially when control imperfections are modeled jointly.

Single-node local GPU speedup

Local workload	**GPU speedup**
Standard PTNT	**13.45**×
Self-consistent / X-decomp	**12.03**×

This plot shows whether the learned process-tensor model is becoming consistent with the observed data also the local fast path is already a regular dense numeric layout, so it is the most device-friendly PTNT workload today.

Takeaways

1. Real quantum hardware is noisy in ways that simple Markovian models miss.
2. PTNT tackles the right scientific problem: it learns multitime process noise and imperfect controls together.
3. The hard part is computational: repeated tensor-network contractions, large datasets, and irregular runtime structure.
4. The software contribution is a clear AI+HPC workflow: one public schema, two runtime layouts, batched optimization, and a path toward stronger GPU-native execution.

PTNT is a case study in how quantum characterization, scientific machine learning, and HPC software design must work together.

Questions?

Why model control drift?

Ideal gate assumption

Standard models assume the commanded gate is identical every time.

In practice, pulse electronics drift, causing the physical rotation to vary slightly.

Control drift variable

In the RX-error template, each circuit i receives a latent angle:

$$\epsilon_i = \texttt{err_X} \text{ (physical deviation)}$$

The circuit then applies $R_X(\epsilon_i)$ around the physical SX pulses.

Why this matters

If ϵ_i is correlated across circuits, the control system has memory.

This manifests directly as **control-side non-Markovianity**.

Main message

Keep PTNT as the physics model, but integrate a more robust model for the hidden control-drift trajectory.

Drifting Randomness (Non-Markovian)= The values are still technically random, but they are correlated across time. The system "remembers" what the error was a millisecond ago

Hardcoded baseline: pink-ish cumulative drift

Current implementation: Simulating the drift

1. **White noise** ($\xi_i \sim \mathcal{N}(0, 1)$): *Get a random, unrelated number (like a dice roll).*
2. **Cumulative sum** ($x_i = \sum_{j \leq i} \xi_j$): *Keep a running total to create a wandering path ("memory").*
3. **Normalize** ($x_i \leftarrow x_i / \max_k |x_k|$): *Squish the path down so the largest peak is exactly 1.*
4. **Scale** ($\epsilon_i = \epsilon_{\text{max}} x_i$): *Stretch the path to match our maximum allowed physical error.*
5. **Bind** ($\epsilon_i \rightarrow \texttt{err_X}$): *Apply this slowly drifting error as the latent angle in the circuit.*

Comparing distributions

Different random seeds create completely different traces. Therefore, we must compare statistical distributions of these errors, not individual traces.

distinction

This creates an analytic long-memory baseline (a cumulative random walk), which mimics, but is not exactly, ideal pink noise.

Diffusion extension: learned latent control drift

Baseline (Hardcoded)

$$\epsilon_i = f(\text{seed})$$

Pro: Simple and perfectly reproducible.

Con: The drift "shape" is manually chosen by us, not the hardware.

Diffusion prior (Learned)

$$\epsilon_{1:N} \sim p_\phi(\epsilon)$$

A generative diffusion model learns the true probability distribution over bounded drift trajectories.

PTNT stays physical

The process tensor remains PEPO / LPDO. The loss still enforces physical constraints.

not replacing QM with AI. Diffusion *only* proposes the latent `err_X` trajectory.

Software design

```
control_noise = none | pink | diffusion_drift
```

Generate a sequence of errors (from time 1 to N) by sampling from a probability distribution (p) that our neural network (ϕ) learned.

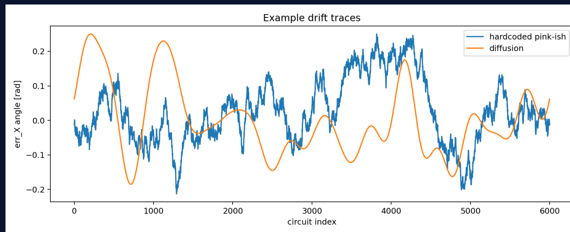
Diffusion extension: learned latent control drift

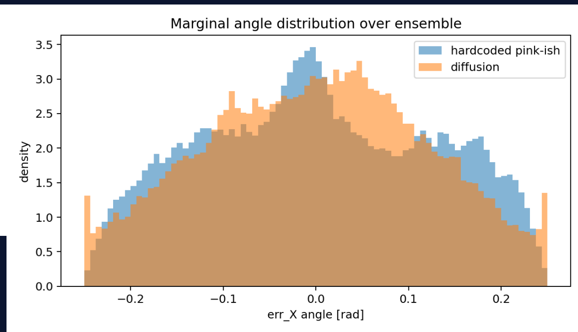
Diffusion prior (Learned)

1. Start with real or synthetic drift traces.
2. Add random Gaussian noise to them step by step.
3. Train a denoising model to predict and remove that added noise.
4. After training, sampling works in reverse: start from pure random noise and repeatedly denoise it until it becomes a realistic drift trace.

So diffusion is not directly “solving quantum mechanics.” It is only learning a probability distribution over possible latent control-drift trajectories.
Note: we do not train diffusion directly on the raw 6000-dimensional drift trace(long smooth signals):
drift trace $\rightarrow$ Fourier coefficients (compact way to describe smooth low-frequency signals) $\rightarrow$ diffusion then we are training in coefficient space.

Validation: compare ensembles, not single traces



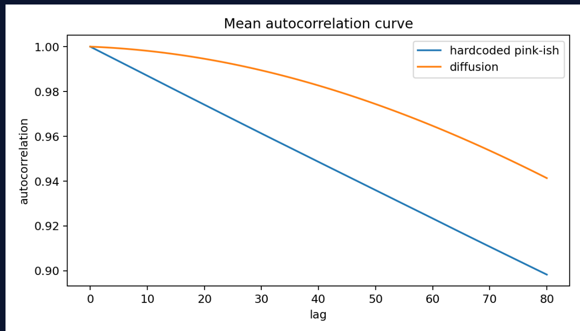


Diffusion stays bounded and slow, overlaps the marginal distribution, and shows longer memory in the autocorrelation curve.

The histogram shows that both methods stay inside the same physical angle range and have overlapping angle distributions.

Both hardcoded pink-ish drift and diffusion drift are random generators, so we compare their statistics, not one trace against one trace. The diffusion version gives a learned latent control-drift prior that stays bounded, remains slow/correlated, and can be passed through the same PTNT pipeline

Validation: compare ensembles, not single traces



Interpretation

lag : how far apart in circuit index are the two drift values we are comparing?
Autocorrelation: "If I know the control error now, how much can I predict the control error later?"
The autocorrelation plot shows both have long memory. In this run, diffusion decays more slowly, so it has stronger long-range correlation.

conclusion

Claim 1

The hardcoded baseline is an analytic cumulative-drift generator. It is useful, but it fixes the drift family by hand.

Claim 2

The diffusion model is a learned latent-control-noise prior. It generates bounded, slow, correlated `err_X` drift.

Claim 3

The physics model remains PTNT. Diffusion does not replace the PEPO/LPDO process tensor or the physical loss.

Final sentence

Diffusion moves PTNT from a hand-coded control-drift assumption toward a learned latent-control-noise prior.