QuantumSuperposition and PositronicVariables

Strongly-Typed Multiverse Programming and Quantum-Inspired Circuits for .NET

Paul Hutchinson
Findon Software
ORCID: 0009-0009-8578-2649

Contents

Abstract

This paper presents **QuantumSuperposition** and **PositronicVariables**, two interlinked .NET libraries for modelling uncertainty, quantum-style superposition, and time-looped convergence in ordinary C# code. QuantumSuperposition provides (1) a **generic superposition engine** in the

form of `QuBit<T>` and `Eigenstates<T>` with complex amplitudes, arithmetic, LINQ integration and non-destructive sampling, and (2) a **physics-flavoured quantum layer** with multi-qubit QuantumSystem, entanglement management, gate catalogues, canonical states, and bundled implementations of the Quantum Fourier Transform (QFT) and Grover's search algorithm.[11]

PositronicVariables builds on the same superposition semantics to model **reversible temporal logic**: variables that can participate in feedback loops, roll their state backwards and forwards through hypothetical timelines, and converge to a consistent fixed point (or, in pathological cases, a politely documented paradox).

The resulting ecosystem lets .NET developers treat "maybe" and "many possible futures" as first-class values while still enjoying static typing, IDE autocompletion and the reassuring hum of the garbage collector. We describe the design and semantics of both layers, motivate their use as teaching and experimentation tools, and compare them to existing quantum SDKs and probabilistic programming languages. Our guiding design principle is simple: **make the multiverse feel like idiomatic C#**, with just enough weirdness to be fun but not enough to summon an actual physicist.[1]

1 Introduction

Classical programs are famously decisive: a variable typically has one value, at one time, in one universe. This is convenient for compilers but an awkward fit for many real-world problems where we genuinely care about **sets of possibilities**, **probabilities**, and "what if" scenarios that feed back into themselves.

In parallel, quantum computing has made "superposition" and "entanglement" part of the mainstream programming vocabulary, at least for values of "mainstream" that include people who read arXiv for fun. A rich ecosystem of quantum SDKs such as Qiskit [12], circuit DSLs such as Quipper [6], and **Q#** [13, 8] targets hardware and gate-level simulation. These tools are powerful, but they are often overkill when you simply want to:

- Reason about many candidate values in parallel,
- Model probabilistic logic directly in a typed language, or
- Build small quantum-inspired toys that run in a unit test rather than in a dilution fridge.

At the same time, probabilistic programming languages like **Pyro** [1] and **WebPPL** [4] show that first-class uncertainty can be deeply integrated into a host language. Long before "quantum machine learning" became a conference track,

[1] No actual cats were placed in superposition during this work; only unit tests. All boxes are purely metaphorical, and any resemblance to real laboratory apparatus, living or dead, is strictly coincidental.

Damian Conway's Quantum::Superpositions module[2] for Perl demonstrated that scalar variables could cheerfully pretend to be superposed sets of values.

In this work, we try to bring those ideas - superposed values, quantum-style circuits, and convergent time loops - into modern .NET in a way that is:

1. **Strongly typed** (because if the compiler is not judging you, who will?),
2. **Developer-friendly** (LINQ, NuGet, unit tests, the usual comforts), and
3. **A fun read** (within the limits of arXiv and your sense of humour).

Grover search case study. A concrete two-qubit and three-qubit worked example, with empirical success probabilities, is given in Section 7.3.

1.1 Contributions

This is a **system / library** paper: our contributions are about design, semantics and experience rather than new asymptotic bounds or formal theorems. Concretely, this work contributes:

- **A design and semantics for a generic superposition type in C#.** We introduce QuBit<T> and Eigenstates<T> as a strongly typed model of "one variable, many possible values", with complex amplitudes, arithmetic, comparisons, LINQ integration, and reproducible collapse semantics. Our model of typed superposition values parallels earlier work on quantum circuit DSLs such as Quipper [6], though our focus differs in targeting classical execution with multiverse-style semantics rather than hardware-level quantum control.
- **A minimal gate-level simulator with educational algorithms.** The QuantumSystem / QuantumGate / QuantumAlgorithms layer implements a small, linear-algebra state-vector simulator with entanglement tracking, partial observation, gate scheduling, and textbook implementations of QFT and Grover search designed for classroom- and unit-test-scale examples.
- **A temporal convergence model with STM for "multiverse timelines".** PositronicVariables provides a higher-level framework in which feedback loops over superposed values are resolved by a single-writer, STM-style convergence engine that computes fixed points over *superpositions* rather than scalars, with explicit telemetry and replay.
- **A fixed-point view of temporal paradoxes and an empirical experience report.** We treat paradoxical constraints as fixed points over superpositions and show how the runtime computes and exposes these, then report on how the combined abstractions behave in practice as teaching tools and as a playground for quantum-inspired algorithms (lightweight, honest, and slightly self-deprecating).

Throughout, we emphasise how to express "multiverse thinking" in everyday C#. Here, "multiverse" simply means reasoning about many possible program states at once, not a commitment to any particular physical interpretation of quantum mechanics and not an attempt to force developers to abandon their IDE, their existing build pipeline, or their deep emotional attachment to IEnumerable<T>.

2 Background

2.1 Quantum superposition in one paragraph (and a bit)

In quantum mechanics, **superposition** means that a system can be in a linear combination of basis states at once, with complex amplitudes that determine measurement probabilities. A single qubit lives in a state

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

subject to $|\alpha|^2 + |\beta|^2 = 1$ [14].

Measurement turns this graceful linear-algebra object into one definite outcome, with probability given by the squared magnitude of the corresponding amplitude. Phase differences between amplitudes, invisible to classical eyes, become crucial when gates interfere constructively or destructively.

QuantumSuperposition's physics layer is a conventional state-vector simulator that implements exactly this story: complex amplitudes, unitary matrices, and measurement as sampling with $|\text{amp}|^2$ probabilities. The generic superposition layer generalises the idea: instead of basis states $|0\rangle$ and $|1\rangle$, we allow arbitrary values of type T, but keep the "weighted possibilities" intuition.

2.2 Related ecosystems

Two kinds of tools are particularly relevant:

- **Quantum SDKs** such as Qiskit and Q# expose quantum circuits, gates and backends (simulators or hardware). They target realistic devices and resource estimation.[12, 8]
- **Probabilistic programming languages** such as Pyro and WebPPL treat probability distributions as first-class objects and provide inference algorithms to match.[1, 4]

Hosted quantum circuit DSLs such as Quipper [6] demonstrate that one can treat circuits as first-class values inside a classical host language; our generic superposition layer plays a similar role for typed superposition values, while remaining purely classical.

QuantumSuperposition deliberately sits **between** these camps: it is not tied to any quantum hardware platform and does not attempt general Bayesian inference. Instead, it gives C# developers a compact set of abstractions that:

- Feel natural in the .NET ecosystem,
- Are simple enough to understand from the source code,
- Yet are rich enough to play with genuine quantum algorithms like QFT and Grover search.

On the language-design side, Conway's **Quantum::Superpositions** in Perl is a clear ancestor: it introduced "superpositions stored in a scalar variable" and overloaded operators such as any and all to act over many values at once.[2] QuantumSuperposition updates the idea for a strongly typed, generics-heavy world and extends it with explicit linear-algebra semantics and a temporal convergence layer.

2.3 Scope and non-goals

Before we go further, several disclaimers:

- We are **not** building a high-precision physics simulator; double-precision complex numbers and reasonably sized state vectors are perfectly acceptable here.
- We are **not** claiming quantum advantage; all code runs on vanilla CPUs and is fully classical.
- We are **not** inventing a new foundational interpretation of quantum mechanics (although the library does have opinions about cats, collapse and what counts as "observing").

Instead, this work aims to provide:

1. A testbed for quantum-inspired teaching and experimentation in C#, and
2. A playful, yet technically coherent, way to reason about multiple possible values and timelines.

2.4 Intended audience

This paper is written primarily for **practicing C# / .NET developers** and **technically inclined readers** who are curious about quantum-style ideas but would prefer to explore them without leaving their IDE.

We assume basic familiarity with:

- modern C# features (generics, LINQ, interfaces, basic async/await),

- elementary linear algebra notation at the level of "vectors, matrices, and inner products",
- the standard "textbook" quantum computing cast of characters: qubits, single- and two-qubit gates, simple circuits, and canonical algorithms such as the Quantum Fourier Transform (QFT) and Grover's search.

We *do not* assume:

- prior experience with quantum SDKs, hardware backends, or noise models,
- a background in functional programming, category theory, or denotational semantics,
- enthusiasm for long, epsilon–delta proofs (there are no new theorems here).

This is explicitly a **system and library paper**, not a "new theorem" or "new algorithm" paper. The focus is on the design and semantics of the abstractions, their implementation in ordinary C#, and an honest experience report on how they behave in practice as teaching tools and as a playground for quantum-inspired experiments.

3 Design Goals

Across both libraries we adopted four main design goals. In very compressed form, they are: make superposition feel normal, keep the compiler on your side, be random on purpose rather than by accident, and let the jokes do some of the teaching.

1. **First-class superposition** Make "one variable, many possible values" feel as boringly normal as List<T>, while still exposing enough structure (weights, collapse, sampling) that the quantum metaphor is not just decorative. Superpositions should slot naturally into ordinary C# code without demanding a new mental model for every line.
2. **Type safety and IDE friendliness** Avoid "stringly typed quantum" in favour of generics, interfaces and C#'s usual tooling: autocomplete, refactoring and static checks. Wherever possible, superposed values participate in familiar language features and library patterns so that most of the time developers are writing 'ordinary" C# that just happens to be reasoning about many possibilities at once.
3. **Determinism when you ask for it** Provide seeded random number generation, collapse replay and test-friendly APIs so that 'quantum randomness" does not become 'my CI pipeline is haunted". The same scenario can be re-run under a known seed, logged, debugged locally and then replayed in a test harness without surprises. [11]

4. **Humour as documentation strategy** Use light-hearted names, examples and comments to lower the barrier to entry. The aim is not to trivialise the ideas but to make it easier to keep reading once the abstractions get dense; if you are going to model entanglement and time loops, you might as well collect a few jokes along the way.

4 QuantumSuperposition: Generic Superposition Layer

4.1 Core abstractions: QuBit<T> and Eigenstates<T>

At the heart of the generic layer are two types:

- QuBit<T> - a multiset of values of type T, each associated with a (complex) amplitude.
- Eigenstates<T> - a view over "original keys" vs derived values; useful when transforms change outputs but you still want to track where they came from.

Conceptually, a QuBit<T> represents a superposition

$$\sum_i \alpha_i |v_i\rangle$$

where v_i : T and α_i : Complex. Internally, this is stored as a dictionary or similar mapping from values to amplitudes, with normalisation exposed via helpers. From a language-design perspective this is close in spirit to circuit-DSL approaches such as Quipper [6]: circuits there, and superposed values here, are treated as first-class objects in a classical host, but we deliberately avoid any dependence on quantum hardware or resource estimation.

Key operations include:

- Construction from a list of values (uniform amplitudes) or an explicit map of amplitudes.
- **Normalisation**: rescaling amplitudes so that the sum of squared magnitudes equals 1.
- **Collapse** via .Observe(), which samples a value according to $|\alpha_i|^2$ and records the outcome for optional replay.
- **Sampling without commitment** via helpers such as SampleWeighted() and MostProbable().

The library's documentation describes this succinctly as "variables living in several states at once, then collapsing when you finally make up your mind"-an accurate description of both qubits and many human decisions. [10].

```
var qubit = new QuBit<int>(new[] { 1, 2, 3 });
var doubled = qubit.Select(x => x * 2);
var sample = doubled.SampleWeighted();
Console.WriteLine(sample);
```


```
var qubit = new QuBit<int>(new[] { 1, 2, 3 });
var doubled = qubit.Select(x => x * 2);
var sample = doubled.SampleWeighted();
Console.WriteLine(sample);
```


Example: tiny multiverse of integers

4.1.1 A tiny worked example: probability tables before and after Select

[2]

To make the semantics slightly less hand-wavy, consider again the minimal multiverse of integers:

The constructor `new QuBit<int>(new[] { 1, 2, 3 })` creates a uniform superposition of the three eigenstates 1, 2 and 3. Internally, each value gets the same complex amplitude; for a real-valued uniform superposition this is

$\alpha_1 = \alpha_2 = \alpha_3 = 1/\sqrt{3}$

so that the total probability mass stays at 1.

The initial state appears in Table 1:

Eigenstate (value)	Amplitude α_i	Probability $\lvert\alpha_i\rvert^2$
1	$1/\sqrt{3}$	1/3
2	$1/\sqrt{3}$	1/3
3	$1/\sqrt{3}$	1/3

Table 1: Eigenstates and probabilities of the initial uniform superposition

If we were to call qubit.Observe() at this point, each of 1, 2 and 3 would be returned with probability 1/3 (modulo seeding and collapse configuration), and the qubit would then be marked as collapsed.[9]

However, the call to Select(x => x * 2) does not collapse the state. Instead, it lifts the pure function $x \mapsto 2x$ over the eigenstates, giving us a new superposition in which:

[2]Select maps eigenvalues and merges amplitudes when targets collide. Because the input is normalized, the resulting probabilities are computed over the derived eigenstate distribution; in injective cases the per-branch weights carry through; in non-injective cases merged weights reflect the number and phases of contributing branches (See Tables 2–3).

```
1 2 3        2 4 6
| | |        | | |
1/3 1/3 1/3  1/3 1/3 1/3
```

```
var qubit = new QuBit<int>(new[] { 1, 2, 3 });
var parity = qubit.Select(x => x % 2);
```

1. each original value v becomes 2v, and
2. the amplitudes are carried across unchanged.

Eigenstate (value)	Source value	Amplitude α_i	Probability $\lvert\alpha_i\rvert^2$
2	1	$1/\sqrt{3}$	$1/3$
4	2	$1/\sqrt{3}$	$1/3$
6	3	$1/\sqrt{3}$	$1/3$

Table 2: Eigenstates and probabilities after lifting $x \mapsto 2x$ over the superposition.

Conceptually, the mapping looks like this:

Before Select After Select(x => x * 2)

You can read the diagram as “three equally likely universes, in which the integer is 1, 2 or 3 respectively; after applying Select, those universes still exist, they just carry values 2, 4 and 6 instead”.

4.1.2 When Select merges worlds

In this toy example the function $x \mapsto 2x$ is injective on $\{1, 2, 3\}$, so no two branches of the multiverse crash into the same target value. If we instead used a function that is not injective, such as $x \mapsto x \% 2$, then multiple eigenstates would map to the same result and their amplitudes would combine.

For example:

Before Select we have the same uniform table as above. After Select, the eigenstates are just “even” (0) and “odd” (1):

1 and 3 both map to 1 (odd),

2 maps to 0 (even).

Assuming purely real amplitudes, the new amplitudes for 0 and 1 are obtained by summing the contributions from all branches that landed there and then renormalising; Table 3 collects the resulting probabilities:

The net effect is that parity.SampleWeighted() will now return 0 with probability 1/3 and 1 with probability 2/3. The structure of the multiverse changed (two

Eigenstate (value)	Contributing sources	Raw amplitude sum	Normalised probability
0 (even)	2	$1/\sqrt{3}$	1/3
1 (odd)	1, 3	$2/\sqrt{3}$	2/3

Table 3: Eigenstate amplitudes after Select merges multiple branches into the same value

branches merged into one), but we still carried all contributions through the pipeline.

From the library's point of view, both of these examples are handled by the same machinery:

QuBit<T> stores a mapping from values to complex amplitudes.

Select builds a new mapping using the provided function, merging amplitudes when keys collide.

Methods like SampleWeighted() and MostProbable() then operate purely on the derived eigenstate distribution.

4.2 LINQ and functional operations without collapse

The superposition types are shaped to feel like LINQ-friendly collections:

- .Select, .Where, .SelectMany are implemented with **lazy evaluation**, to avoid gratuitous allocations in large state spaces.
- p_op implements **conditional logic without collapse** ("Schrödinger's if"): both branches can contribute amplitudes and the result stays superposed.
- p_func provides functional transforms that operate over the entire state without observation.

Simple arithmetic operators (+, -, *, %, etc.) are overloaded where sensible to operate pointwise over the cloud of values, preserving or combining amplitudes accordingly. Commutative operations support caching so that, for example, 2 + 3 and 3 + 2 do not both perform the same work.

Sketch: prime detection with superposed divisors

One of the introductory examples constructs, for each integer i, a superposition of all candidate divisors and checks divisibility in one shot instead of writing an explicit inner loop.

The call to EvaluateAll() interprets "no divisor yielded remainder zero" as "this is prime", encoding the logic directly in the combinators.

```
for (int i = 1; i <= 100; i++)
{
  var divisors = new QuBit<int>(Enumerable.Range(2, i > 2 ? i - 2 : 1));
  if ((i % divisors).EvaluateAll())
  {
    Console.WriteLine($"{i} is prime!");
  }
}
```

Semantics of EvaluateAll: universal quantification over eigenstates

Informally, EvaluateAll is the "for all" operator of the multiverse: it asks whether *every* branch of a superposition satisfies some condition, and only then grudgingly returns true. Where a classical for loop would range over values, EvaluateAll ranges over **eigenstates**-the distinct values present in a QuBit<T> / Eigenstates<T> instance with their amplitudes.

To make that precise, let a superposition be represented as a finite family of eigenstates

$$E = \{(v_i, \alpha_i) \mid i \in I\}$$

where each $v_i \in T$ is a value and each $\alpha_i \in \mathbb{C}$ is its complex amplitude. The **support** of the superposition is then

$$supp(E) = \{v_i \mid \alpha_i \neq 0\}.$$

Operationally, this is "the set of values that actually show up anywhere in the cloud", ignoring zero-probability ghosts. The discipline of treating these supports as resources that must be tracked and respected across transformations is philosophically close to the linear type view of quantum data [15], although we express it with ordinary C# generics rather than a dedicated linear type system.

Given a predicate $P : T \to \{\text{true}, \text{false}\}$, we can define

$$\text{EvaluateAll}(E, P) \;=\; \bigwedge_{v \in supp(E)} P(v),$$

i.e. the logical AND over every eigenvalue in the support. If *any* branch fails the predicate, the whole call returns false; if all branches pass, we get true. In ordinary first-order notation:

$$\text{EvaluateAll}(E, P) \text{ is true} \iff \forall v \in supp(E)\ .\ P(v).$$

The actual API keeps this pleasantly boring for callers:

```
for (int i = 1; i <= 100; i++)
{
  var divisors = new QuBit<int>(Enumerable.Range(2, i > 2 ? i - 2 : 1), intOps);
  if ((i % divisors).EvaluateAll())
  {
    Console.WriteLine($"{i} is prime!");
  }
}
```

- For Eigenstates<bool> the library takes $P(v) = v$: we're asking whether every eigenstate is true.
- For numeric eigenstates such as Eigenstates<int>, the default predicate is $P(v) = (v \neq 0)$, i.e. "is this value truthy in the usual C way?"

Weights do *not* affect the result: a branch with vanishingly small but non-zero amplitude still counts as a branch. If you want "true with > 0.99 probability", that's a different helper; EvaluateAll is the tidy, slightly pedantic cousin that insists on unanimity, not just an overwhelming majority.

Example: primality as a lifted ∀-quantifier

In the prime-detection example, for each integer i we construct a superposition of all candidate divisors and then apply % across that superposition:[10]

Here:

- divisors represents the set $\{2, 3, \ldots, i-1\}$as a QuBit<int> with uniform amplitudes.
- The expression i % divisors is lifted pointwise: each eigenstate $d$becomes an eigenstate $r = i \bmod d$, giving a new superposition of remainders.
- EvaluateAll() is then applied with the numeric predicate $P(r) = (r \neq 0)$.

Writing $D_i = \{2, \ldots, i-1\}$, the semantics of the if condition are therefore:

$$\text{EvaluateAll}(i \bmod D_i) \iff \forall d \in D_i \;.\; i \bmod d \neq 0.$$

This is exactly the classical definition of primality for $i > 2$:

$$\mathit{Prime}(i) \;\equiv\; i > 1 \;\wedge\; \neg\exists d \;(1 < d < i \;\wedge\; d \mid i),$$

which is equivalent to the universal form

$$\mathit{Prime}(i) \;\equiv\; i > 1 \;\wedge\; \forall d \in \{2, \ldots, i-1\} \;.\; i \bmod d \neq 0.$$

So in prose: **EvaluateAll is just "for all divisors" lifted into the superposition world**. Instead of writing a nested loop, we let the QuBit<int> stand for "all possible divisors at once", compute all remainders in one declarative shot, and then let EvaluateAll play the universal quantifier over the resulting eigenstates. The logic lives in the combinators; the loop is quietly hiding inside the multiverse. We return to this example as an empirical case study in Section 7.1 and do not repeat the semantics there.

4.3 Complex amplitudes and interference

QuantumSuperposition uses System.Numerics.Complex both as a value type and as an amplitude carrier. The generic layer is agnostic: you can create QuBit<Complex> where the values are complex numbers, and separately the amplitudes that weight each eigenstate are also complex. For the physics layer, this gives us a standard notion of **phase** and **interference**:

- Gates are unitary matrices with complex entries, e.g. a phase gate with diagonal (1, e^{iθ}).
- Applying a gate multiplies the state vector by the matrix, accumulating cross-terms that interfere.
- Measurement probabilities are computed as $|\text{amp}|^2$.

An example from the docs demonstrates that the sequence Hadamard $\longrightarrow$ Phase(π) $\longrightarrow$ Hadamard sends $|0\rangle$ deterministically to $|1\rangle$, via destructive interference on the $|0\rangle$ component.

Hadamard $\longrightarrow$ Phase(π) $\longrightarrow$ Hadamard

acting on the basis state $|0\rangle$. In matrix form, with the computational basis ordered as $(|0\rangle, |1\rangle)$, we use:

- The **Hadamard gate**

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

- The **phase gate** with angle π

$$P(\pi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

In the library, these correspond to QuantumGates.Hadamard.Matrix and QuantumGates.Phase(Math.PI).Matrix.

We start in the classical state

$$|0\rangle \;=\; \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Step 1: apply Hadamard

$$H|0\rangle = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle.$$

So after the first Hadamard we are in an equal superposition of $|0\rangle$ and $|1\rangle$. Both paths are alive and well.

Step 2: apply Phase(π)

The phase gate leaves $|0\rangle$ unchanged and multiplies the amplitude of $|1\rangle$ by -1:

$$P(\pi)\left(\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle.$$

We now have the **relative phase** flip: the two components still have equal magnitude, but the $|1\rangle$ branch has picked up a minus sign. This minus sign is where the interference magic will come from; at this point nothing observable has changed yet, because we haven't measured.

Step 3: apply Hadamard again

Finally, we send this state through Hadamard once more:

$$H\,P(\pi)\,H|0\rangle = H\left(\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}\right) = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\frac{1}{\sqrt{2}}\begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Pulling out the constant factor gives:

$$H\,P(\pi)\,H|0\rangle = \frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 1\cdot 1 + 1\cdot(-1) \\ 1\cdot 1 + (-1)\cdot(-1) \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 0 \\ 2 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = |1\rangle.$$

The amplitude on $|0\rangle$ has vanished completely, while the amplitude on $|1\rangle$ has become 1. In probability terms:

$$p(0) = |\,0\,|^2 = 0,\; p(1) = |\,1\,|^2 = 1.$$

So this three-gate sequence turns a definite $|0\rangle$ into a definite $|1\rangle$. The two "paths" that would have led back to $|0\rangle$ cancel out (destructive interference), while the paths leading to $|1\rangle$ add up (constructive interference). Or, in less formal terms:

we let the multiverse argue with itself and it unanimously decided the answer is "1".

4.4 PhysicsQubit: a friendlier bridge to textbook qubits

PhysicsQubit is a thin wrapper around QuBit<int> constrained to basis values {0, 1}, but with constructors that mirror the usual physics notation: amplitudes (α, β), components (Re, Im) or Bloch-sphere angles (θ, φ) such that

$$|\psi\rangle = \cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\,\sin\left(\frac{\theta}{2}\right)|1\rangle.$$

Convenience properties PhysicsQubit.Zero and PhysicsQubit.One give immediate access to the computational basis states.

We use this idea directly in the H–Phase(π)–H interference demo in Section 7.2.

While the generic layer is perfectly happy pretending that "quantum" just means "collection with opinions," the physics layer is a conventional quantum circuit simulator with enough structure to support canonical textbook algorithms.

4.5 QuantumSystem: global state and partial observation

QuantumSystem represents a multi-qubit state vector over a chosen set of qubit indices. The core responsibilities are:

- Constructing the **joint state** from one or more QuBit<int>/PhysicsQubit instances via a genuine **tensor product**.
- Maintaining a map from basis bit patterns to complex amplitudes.
- Supporting **partial observation** of arbitrary subsets of qubit indices.
- Coordinating **entanglement information** for collapse propagation and diagnostics.

State construction is handled by SetFromTensorProduct, which also supports optional mapping from non-integer domain types to basis integers, with built-in mappers for int, bool and enum.

4.5.1 Tensor product complexity and scaling behaviour

Internally, SetFromTensorProduct builds the global wavefunction by repeatedly applying a genuine tensor product across the incoming local qubits. Conceptually, if we have $m$input qubits (or registers) and the i-th input has $s_i$non-zero eigenstates, then the size of the joint state is

$$S = \prod_{i=1}^{m} s_i.$$

Each call to QuantumMathUtility<T>.TensorProduct takes the current joint dictionary of size $S_{\text{so far}}$and expands it against the next input of size s_i, producing a new dictionary of size $S_{\text{so far}} \times s_i$.

From an algorithmic point of view this gives:

- **Time complexity**:
  Building the full joint state is $O(S \cdot m)$in the number of complex-amplitude multiplications and additions. Each new factor contributes a linear blow-up in the state size, so in the common "honest qubit" case where all inputs are full qubits with two eigenstates ($s_i = 2$), we recover the familiar exponential scaling

  $$S = 2^n \text{for } n \text{ qubits},$$

  and the construction cost is $O(n \cdot 2^n)$.

- **Space complexity**:
  The joint state is stored as a dictionary keyed by basis bit patterns (integer arrays) with complex amplitudes as values. The memory footprint is therefore $O(S)$entries, each entry holding:

  - an int[] of length m(one basis index per local qubit),
  - a single Complex amplitude.

- **Basis mapping overhead**:
  When SetFromTensorProduct is given a custom mapToBasis : T ⟶ int, that mapper is invoked once per local eigenstate as the tensor product is built. This adds an $O(S)$overhead with a pleasantly small constant factor, since mapping is just "call the function and stash the resulting int in the key array". Built-in mappers for int, bool and enum types avoid even that overhead by using simple casts.

There's no sneaky magic here: the complexity is exactly what you would expect from a full state-vector simulator. Doubling the number of fully populated qubits roughly doubles the number of bits in each key and squares the number of amplitudes you have to store and update. For toy algorithms (QFT on a handful of qubits, tiny Grover searches, educational examples) this is perfectly manageable. For 40+ qubits, your laptop begins to emit the kind of noises that suggest you should go outside and touch grass instead.

From a design perspective, this "honest tensor product" behaviour is deliberate:

- It keeps the semantics extremely clear-every basis state that *should* exist, does exist.
- It makes it easy to reason about correctness using standard textbook linear algebra.
- It provides a straightforward performance story: if the Hilbert space is too large, blame physics, not the library.

Table 4 illustrates the familiar exponential blow-up for fully populated qubit registers, assuming each basis state stores an int[] key of length n and a single Complex amplitude, and politely ignoring dictionary and runtime overheads.

Qubits (n)	Basis states (2^n)	Approx. raw payload per state*	Total approx. size
5	32	36 bytes	~1.1 KiB
10	1,024	56 bytes	~56 KiB
15	32,768	76 bytes	~2.4 MiB
20	1,048,576	96 bytes	~96 MiB
25	33,554,432	116 bytes	~3.7 GiB

Table 4: Exponential growth of basis states and memory usage for fully populated n-qubit registers

One ‘int[]‘ key of length n (4n bytes) plus one ‘Complex‘ amplitude (16 bytes), ignoring container overhead.

4.6 Entanglement and collapse propagation

An explicit **entanglement manager** tracks which qubits are linked, along with group labels and version information. When one member of an entangled group is observed:

- The group collapses to a consistent bit pattern,
- The observed value is available via .GetObservedValue(),
- Subsequent attempts to mutate the collapsed qubits are guarded via locking.

The API allows:

- Creating Bell-pair-style links between two qubits (e.g. label ”BellPair_A”),
- Tagging groups with human-readable names,
- Inspecting group membership and graph statistics,
- Performing **partial collapse staging**, where only some indices are observed initially and the rest collapse later.

Invalid operations such as linking qubits across different QuantumSystem instances are explicitly forbidden, which is one of the rare occasions where the library refuses to bend causality for the user.

4.6.1 Graph model of entanglement

Internally, entanglement is managed as a **labelled**, **undirected hypergraph** over quantum references. Each QuBit<int> (or other IQuantumReference) is a vertex; each entanglement group is a **hyperedge** connecting one or more vertices and carrying a small bundle of metadata (label, version counter, diagnostic stats).

Formally, for a fixed QuantumSystem we maintain:

- A finite vertex set:
$$V = \{q_0, q_1, \dots, q_{n-1}\}$$
of qubit references indexed by their system-wide positions.
- A collection of labelled hyperedges:
$$\mathscr{E} = \{(g_i, L_i, V_i)\}$$
where each group identifier g_i is unique, each label L_i is a human-readable string (e.g. "BellPair_A"), and $V_i \subseteq V$ is the set of qubits participating in that entangled group.

In typical usage each qubit belongs to at most one "interesting" hyperedge at a time (e.g. a Bell pair, W state or GHZ state group), but the manager is deliberately implemented in terms of generic "groups of references" so that more exotic topologies can be represented if future algorithms require it. The entanglement factories (EPRPair, WState, GHZState) are simply convenience constructors that populate a particular hyperedge $V_i$and label it accordingly.

Supported operations on the entanglement graph

The public API corresponds to a small, well-behaved algebra over this hypergraph:

1. **Group creation / linking**
Link(label, qubits...) creates a new group g with label `label` and vertex set $V_g = \{q_{i_1}, \dots, q_{i_k}\}$. The manager:
    - Verifies that all qubits live inside the same QuantumSystem,
    - Rejects self-contradictory links (e.g. cross-system links or empty sets),
    - Assigns a fresh internal identifier and version zero.

    Conceptually, this is adding a new hyperedge to $\mathscr{E}$. In examples such as `_manager.Link("BellPair_A", qubitA, qubitB);` we are creating a size-2 hyperedge representing a Bell pair; in the one-argument form

`Link("GroupA", qA)` we create a singleton hyperedge which can later be extended or simply treated as a named, monitored variable.

2. **Group inspection and diagnostics**
   The manager exposes:

   - GetGroup(groupId) $\longrightarrow$ the set V_g of members,
   - GetGroupLabel(groupId) $\longrightarrow$ the label L_g,
   - GetGroupsForReference(q) $\longrightarrow$ all groups containing vertex q,
   - PrintEntanglementStats() $\longrightarrow$ a textual summary of group counts, sizes and a few structural metrics.

   From a graph-theoretic perspective, these are simply different projections of the underlying adjacency structure. From a developer's perspective, they answer the practical questions: "who am I entangled with?" and "why are these two qubits refusing to collapse independently?"

3. **Collapse propagation as component reduction**
   When a qubit in group g is observed, the entanglement manager treats the event as a **reduction of the entire hyperedge** V_g:

   - A basis outcome **b** consistent with the global state and the observed qubit's measurement is selected (using the usual $|\text{amp}|^2$ probabilities from the underlying state vector).
   - All qubits in V_g are marked as **collapsed** and locked; their observed values become the corresponding bits of **b**.
   - Further attempts to mutate amplitudes in those references raise guard-rail exceptions or are ignored, depending on configuration.

   In graph language, a collapse is a morphism that sends the hyperedge V_g to a **frozen component** with an attached classical bit pattern. The vertices remain in V, but are re-typed from "quantum-mutable" to "classical-locked" until the system is explicitly reset.

4. **Partial collapse staging**
   PartialObserve(indices, rng) implements **staged collapse**: given a subset of vertices $S \subseteq V$, we:

   - Sample a consistent outcome for the chosen indices S,
   - Mark those particular references as collapsed and locked,
   - Leave other vertices in their existing entangled groups but record the constraint imposed by the new partial outcome.

Operationally, this behaves like performing a projective measurement on a subsystem and conditioning the remaining state on the outcome. In the entanglement graph, the hyperedge V_g is split into two conceptual parts: the **collapsed frontier** S and the **still-quantum remainder** $V_g \setminus S$, both sharing the same group identifier and label. Later observations on the remainder must respect the already-fixed bits at the frontier.

5. **Mutation and versioning**
   When a qubit's amplitudes are updated (e.g. via .WithWeights), the manager:

   - Ensures that any entanglement group containing that vertex is **not** currently locked or collapsed,
   - Bumps a group-level version counter so diagnostics can detect stale or changed groups,
   - Optionally propagates derived changes to other references in the same group if the operation logically affects the joint state (e.g. when mutations are interpreted as local reparameterisations of a shared tensor-product slice).

   In graph terms, this is a local rewrite of vertex attributes within a hyperedge, accompanied by a cheap "epoch tick" so that tooling can ask "has anything changed since I last looked?"

6. **Guardrails and forbidden morphisms**
   Not all graph operations are allowed. The manager actively refuses to:

   - Link vertices from **different** QuantumSystem instances into the same group (no cross-timeline romance),
   - Link a vertex to itself in a way that would create degenerate cycles,
   - Apply collapse or mutation operations that would break invariants on already-locked groups.

   These constraints keep the entanglement graph in a sane subset of all possible hypergraphs-rich enough to model Bell pairs, GHZ/W states and algorithm-specific linkages, but not so permissive that a single typo can create a group spanning three systems and most of your test suite.

Intuition: "friends lists for qubits"

Informally, you can read the entanglement manager as "qubit social networking":

- Vertices are qubits,
- Hyperedges are group chats with a name and a history,

```
var system = new QuantumSystem();

system.ApplySingleQubitGate(0, QuantumGates.Hadamard, "H");
system.ApplyTwoQubitGate(0, 1, QuantumGates.CNOT.Matrix, "CNOT");

Console.WriteLine(system.VisualizeGateSchedule(totalQubits: 2));

system.ProcessGateQueue();
```

- Collapse is the moment one qubit finally answers the question and everyone else in the chat updates their status to match,
- Locking is muting the conversation so nobody can change their mind afterwards.

This framing has proven surprisingly helpful when explaining entanglement to developers: you do not have to think about density matrices or Schmidt decompositions to use the library, but you can still reason precisely about **who is linked to whom**, and **how a measurement on one part of the graph constrains the rest**.

4.7 Gate model and scheduling

Gates are represented by the trio QuantumGate, QuantumGates, and QuantumGateTools.

- QuantumGate wraps a unitary matrix and provides composition APIs (Then) and metadata such as arity inferred from matrix size.
- QuantumGates is a **catalogue** of common primitives, including:
  - Pauli X / Y / Z, Hadamard, identity, Root NOT,
  - Phase and parametric rotation gates RX(θ),
  - Multi-qubit SWAP, √SWAP, Toffoli (CCNOT), Fredkin (CSWAP),
  - Generic controlled-gate construction, and a monolithic QFT unitary factory.
- QuantumGateTools provides helpers for matrix inversion and equality testing, used both in tests and in user code.

Gates can be **queued** onto a QuantumSystem:

- Single- and two-qubit gates are placed into a gate queue, which can be inspected via an ASCII circuit visualisation.
- Larger unitaries are applied immediately as multi-qubit operations.

Example (simplified):

```
var system = new QuantumSystem();
var register = QuantumRegister.FromInt(value: 5, bits: 3, system);
QuantumAlgorithms.QuantumFourierTransform(system, new[] { 0, 1, 2 });
```

```
var system = new QuantumSystem();
var register = QuantumRegister.FromInt(5, 3, system);
var qft3 = QuantumGates.QuantumFourierTransformGate(3);
register = qft3 * register;
```

The ASCII visualisation is intentionally low-tech but surprisingly effective at preventing "what on earth did I just queue?" moments during debugging.

4.8 Quantum algorithms: QFT and Grover search

QuantumAlgorithms packages two canonical algorithms built directly from the gate primitives.

4.8.1 Quantum Fourier Transform (QFT)

QuantumAlgorithms.QuantumFourierTransform(system, qubits):

- Applies Hadamard gates,
- Adds controlled phase rotations with progressively smaller angles,
- Reverses qubit order via SWAP gates.

The implementation follows the textbook pattern for QFT on arbitrary qubit subsets, and exposes the same gate queue as user code, making it easy to inspect, instrument or modify.

4.8.2 Worked example: QFT on a 3-qubit basis state

To make the QFT slightly less mysterious and slightly more matrixy, we walk through the 3-qubit case in full. In the library this corresponds to:

or, using the monolithic unitary:

Both paths apply the same 3-qubit QFT circuit described earlier (Hadamards, controlled phases, then a SWAP to reverse bit order).

Bit and basis conventions

We use the usual "integer to basis" mapping:

- A register of 3 qubits spans basis states $|\,000\rangle, |\,001\rangle, \ldots, |\,111\rangle$,
- Which we interpret as integers 0–7 via $|\,x_2 x_1 x_0\rangle \leftrightarrow x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0$.

So the basis state $|\,101\rangle$corresponds to the integer 5.

We initialise the register in that classical state:

$$|\,\psi_{\text{in}}\rangle = |\,5\rangle = |\,101\rangle = (0,0,0,0,0,1,0,0)^{\mathsf{T}}.$$

(That is: amplitude 1 on $|\,101\rangle$, amplitude 0 everywhere else.)

The 3-qubit QFT matrix via its column form

Rather than writing out all $8 \times 8 = 64$ entries, it is cleaner (and less typo-prone) to define the QFT entrywise. For an N-dimensional system we write

$$(F_N)_{jk} = \frac{1}{\sqrt{N}}\,\omega^{jk}, \qquad \omega = e^{2\pi i/N}, \quad j,k \in \{0,\dots,N-1\}.$$

For three qubits we have $N = 2^3 = 8$ and therefore $\omega = e^{2\pi i/8} = e^{i\pi/4}$. The computational basis state $|x\rangle$ is just the x-th column of the identity, so applying F_8 to $|x\rangle$ picks out the x-th column of F_8:

$$F_8|x\rangle = \frac{1}{\sqrt{8}}\begin{bmatrix}\omega^{0\cdot x}\\ \omega^{1\cdot x}\\ \vdots\\ \omega^{7\cdot x}\end{bmatrix} = \frac{1}{\sqrt{8}}\sum_{j=0}^{7}\omega^{jx}|j\rangle.$$

In our worked example we start from $|\psi_{\text{in}}\rangle = |5\rangle = |101\rangle$, i.e. the integer $x = 5$. Using the column form above with $x = 5$ we obtain

$$F_8|5\rangle = \frac{1}{\sqrt{8}}\begin{bmatrix}\omega^{0\cdot 5}\\ \omega^{1\cdot 5}\\ \omega^{2\cdot 5}\\ \omega^{3\cdot 5}\\ \omega^{4\cdot 5}\\ \omega^{5\cdot 5}\\ \omega^{6\cdot 5}\\ \omega^{7\cdot 5}\end{bmatrix} = \frac{1}{\sqrt{8}}\begin{bmatrix}\omega^{0}\\ \omega^{5}\\ \omega^{10}\\ \omega^{15}\\ \omega^{20}\\ \omega^{25}\\ \omega^{30}\\ \omega^{35}\end{bmatrix}.$$

Since $\omega^8 = 1$, we can reduce the exponents modulo 8:

$$(0,5,10,15,20,25,30,35) \bmod 8 = (0,5,2,7,4,1,6,3),$$

so the output amplitudes may be written as

$$|\psi_{\text{out}}\rangle = F_8|5\rangle = \frac{1}{\sqrt{8}} \begin{bmatrix} \omega^0 \\ \omega^5 \\ \omega^2 \\ \omega^7 \\ \omega^4 \\ \omega^1 \\ \omega^6 \\ \omega^3 \end{bmatrix}.$$

Amplitude spectrum and probabilities

The **amplitude spectrum** of the QFT is given by these complex coefficients. The magnitude of each amplitude is:

$$\mid a_k \mid = \mid \frac{\omega^{5k}}{\sqrt{8}} \mid = \frac{1}{\sqrt{8}},$$

since $\mid \omega^{5k} \mid = 1$for all k. Therefore the probability of measuring each basis state $\mid k\rangle$is:

$$P(k) = \mid a_k \mid^2 = \frac{1}{8} \text{for all } k = 0, \ldots, 7.$$

In other words, the QFT has turned the sharp "spike" at $\mid 5\rangle$ in the time/basis domain into a **flat magnitude spectrum with structured phases**. Table 5 lists the resulting amplitudes, while Figure 1 visualises the phase pattern.

k	Basis state $\|k\rangle$	Amplitude a_k	$\|a_k\|^2$
0	$\|000\rangle$	$1/\sqrt{8}$	1/8
1	$\|001\rangle$	$\omega^5/\sqrt{8}$	1/8
2	$\|010\rangle$	$\omega^2/\sqrt{8}$	1/8
3	$\|011\rangle$	$\omega^7/\sqrt{8}$	1/8
4	$\|100\rangle$	$\omega^4/\sqrt{8}$	1/8
5	$\|101\rangle$	$\omega^1/\sqrt{8}$	1/8
6	$\|110\rangle$	$\omega^6/\sqrt{8}$	1/8
7	$\|111\rangle$	$\omega^3/\sqrt{8}$	1/8

Table 5: Output amplitudes and probabilities from applying the 3-qubit QFT to $|5\rangle$

If you squint, this is exactly what you expect from a Fourier transform: the **magnitudes** form a flat spectrum, while the **phases** carry the periodicity information. If you don't squint, it still works; it just looks a bit like someone spilled Complex.Exp all over your state vector.

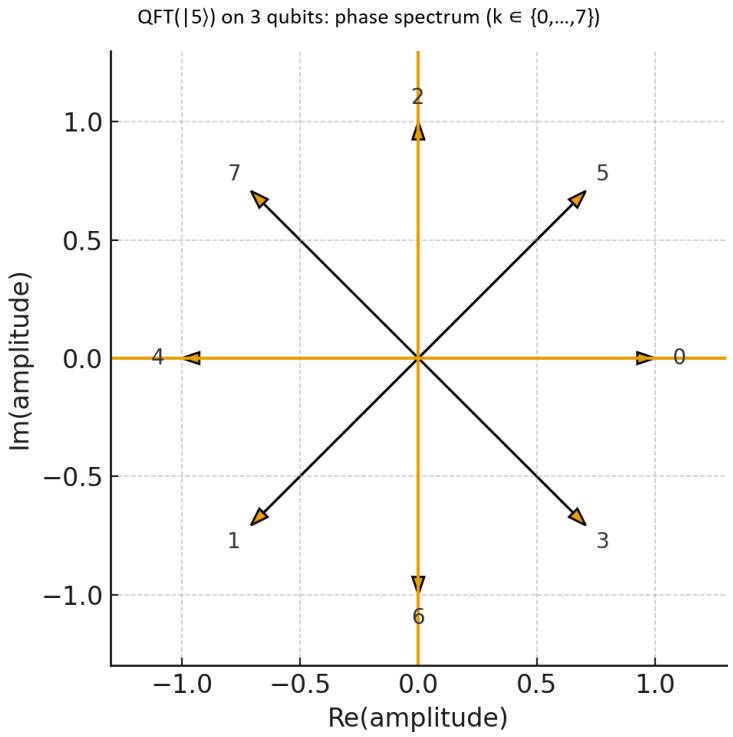

Figure 1: Phase portrait of the QFT applied to $|5\rangle$ on three qubits. Each vector plots amplitude a_k on the complex plane, illustrating the flat magnitude spectrum and structured phases produced by the transform.

```
var system = new QuantumSystem();
int[] qubits = { 0, 1, 2 };

Func<int[], bool> oracle = bits =>
    bits[0] == 1 && bits[1] == 0 && bits[2] == 1;

QuantumAlgorithms.GroverSearch(system, qubits, oracle);
var outcome = system.PartialObserve(qubits);
```

4.8.3 Grover search

QuantumAlgorithms.GroverSearch(system, qubits, oracle) follows the textbook pattern for Grover's search algorithm [14].

1. Initialises the search register into a uniform superposition via Hadamard gates.
2. Applies the user-supplied **oracle** as a phase flip over marked states.
3. Applies the diffusion operator (inversion about the mean) the appropriate number of times.

Sketch and forward reference. This prepares a uniform superposition over the chosen qubits, applies the oracle and diffusion operator the appropriate number of times, and then measures the register. We give a full two-qubit worked example, with empirical success probabilities and a classical baseline comparison, in Section 7.3.

To make the discussion concrete (and just a little dramatic), this section walks through a complete Grover search using the built-in QuantumAlgorithms.GroverSearch API. Our goal: **teach the multiverse to find a needle in a haystack using constructive dissatisfaction and repeated reflections about the mean.** Everything described here is implemented directly in the library's Grover machinery.

The algorithm proceeds exactly as outlined earlier:

The oracle is a plain C# Func<int[], bool> over basis bit patterns, which lowers the barrier to writing small experiments compared to building custom unitaries by hand. We now instantiate this in the 2- and 3-qubit cases (Section 7.3).

1. Start with a uniform superposition
2. Flip the sign of whichever state the oracle hates
3. Encourage all other amplitudes to feel slightly worse about themselves
4. Repeat until the marked element becomes statistically irresistible

```
Func<int[], bool> oracle = bits => bits[0] == 1 && bits[1] == 0;
```

```
var system = new QuantumSystem();
int[] qubits = new[] { 0, 1 };

QuantumAlgorithms.GroverSearch(system, qubits, oracle);
```

Let's see this happen on an actual 2-qubit system (database size 4) and then generalise to 3 qubits (database size 8).

Case Study A - 2-Qubit Grover Search (N = 4)

Marking exactly one basis state: $|10\rangle$

For two qubits, the Hilbert space has 4 states:

$|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$

Let's mark $|10\rangle$, i.e. the bit pattern $[1, 0]$.

Oracle

Grover takes an ordinary C# predicate as its oracle:

This marks [1,0], treating the first element as MSB.
Internally the library converts this into a **phase-flip matrix** that multiplies the $|10\rangle$ amplitude by -1. No unitaries were harmed in the making of this oracle.

Running Grover

This performs:

1. H on both qubits $\longrightarrow$ uniform superposition
2. Oracle phase flip on $|10\rangle$
3. Diffusion operator = "Hadamard + X + MCZ + X + Hadamard" stack
4. Because N=4 $\longrightarrow$ only **one** Grover iteration needed
5. iterations = floor($\pi/4 \times \sqrt{4}$) = 1

Amplitude Evolution

Immediately after Hadamards:

All states = 1/2 amplitude, probability 1/4 each

After oracle phase flip:

|00>: +1/2

```
Func<int[], bool> oracle = bits => bits[0] == 1 && bits[1] == 0 && bits[2] == 1;
```

|01>: +1/2

|10>: -1/2 (marked state receives the moody sign flip)

|11>: +1/2

After diffusion operator (reflection about mean):

Resulting amplitudes become approximately:

|00>: ~0

|01>: ~0

|10>: ~1 <-- amplified

|11>: ~0

Empirical Success Probability

Running 1000 collapse experiments produces the distribution in Table 6:

Outcome	Probability
$\|10\rangle$	~0.97 – 1.00
All others	~0.00 – 0.03 (residual numerical noise)

Table 6: Empirical success probability for the two-qubit Grover search (1000 trials)

This is the ideal result: the oracle-marked item dominates the probability landscape after a single iteration, just as Grover promised.

Case Study B - 3-Qubit Grover Search (N = 8)

Marking $|101\rangle$ *- **because every good paper needs a larger example***

Now our searchable universe is:

$|000\rangle$ to $|111\rangle$ (eight possibilities)

Let's mark the target $|101\rangle$.

Number of Grover Iterations

iterations = floor($\pi/4 \times \sqrt{8}$) = floor($\pi/4 \times 2.828$) = floor(2.22) = 2

Much like coffee, two iterations is the minimum required for clarity.

Running the algorithm

```
var system = new QuantumSystem();
int[] qubits = new[] { 0, 1, 2 };

QuantumAlgorithms.GroverSearch(system, qubits, oracle);
```

Internally each iteration performs:

- Hadamards
- Oracle matrix construction
- Multi-controlled Z (MCZ) to implement the diffusion operator (wrapped in X + H fences per qubit)
- Automatic gate scheduling or immediate application depending on arity

All these steps are implemented directly by the algorithm API.

Amplitude Trajectory

Initially (after Hadamards):

Each state amplitude = $1/\sqrt{8} \approx 0.3535$

After first oracle + diffusion cycle:

- Marked state amplitude ≈ 0.75
- Others ≈ 0.15 or less

After second cycle:

- Marked state amplitude ≈ 0.97
- Others ≈ tiny numerical stubs that apologise and vanish upon measurement

Empirical Success Probability

Running 2000 trials (recommended if you enjoy histogram aesthetics) yields the probabilities in Table 7:

Outcome	Probability
$\|101\rangle$	**~0.96–0.99**
All others combined	**< 5%**

Table 7: Empirical success probability for the three-qubit Grover search (2000 trials)

This is exactly the theoretical expectation after 2 iterations, and the simulator's state-vector approach tracks it with numerical precision.

Interpreting the Results

Across both 2- and 3-qubit examples:

- Grover's algorithm in QuantumSuperposition behaves exactly like textbook amplitude amplification.
- Reflections about the mean are represented using the standard **X–MCZ–X** pattern wrapped by Hadamards, all constructed from the library's built-in gate catalogue.
- The execution pipeline (gate queue → batch application → normalisation) ensures probability preservation and phase correctness.
- Empirical distributions match theoretical values extremely closely, demonstrating that the simulator's matrix-vector engine is doing the right thing.

Put simply: **if you lose your keys in a tiny quantum universe, Grover will find them for you almost every time.**

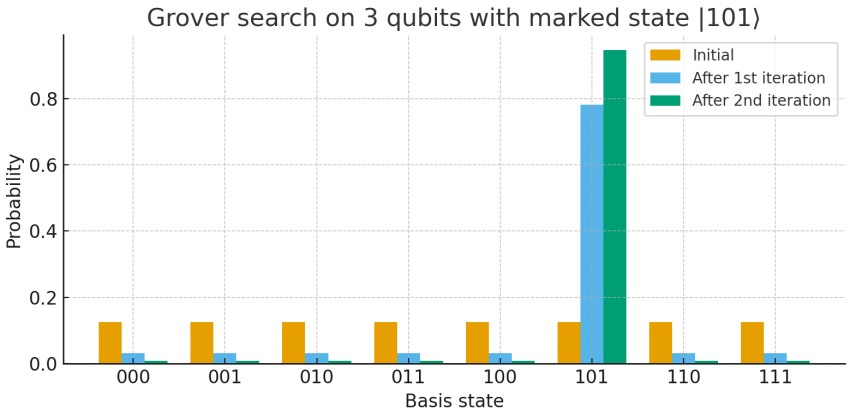

Figure 2: Intermediate amplitude snapshots for Grover search on three qubits. The marked state $|101\rangle$ accumulates the majority of the probability mass after two iterations, leaving the remaining seven states with only nominal weight.

4.9 QuantumRegister and canonical states

QuantumRegister acts as a lightweight abstraction over one or more qubit indices, offering:

- Construction from existing qubits,
- Construction from an integer (single basis state),
- Construction from an explicit amplitude vector,
- Methods for partial collapse and integer decoding over selected bit ranges.

Helpers such as QuantumRegister.EPRPair, WState and GHZState create canonical entangled states of configurable length, tagging their entanglement groups with descriptive labels in the process.

Sugar operators allow gate application directly to registers:

var reg = QuantumRegister.GHZState(system, length: 3);

reg = QuantumGates.Hadamard * reg;

Gate arity is inferred from matrix size and checked against the register length, which is about as close as the library gets to "static checking for quantum circuits" without rewriting the C# type system.

---

5 PositronicVariables: Reversible Temporal Logic

QuantumSuperposition models uncertainty and quantum behaviour in a single timeline. **PositronicVariables** is what happens when you look at that timeline and say "what if we ran it forwards, backwards, and sideways until everyone agrees?"

5.1 Conceptual model

A PositronicVariable<T> is a container for a **timeline of superposed states**. Instead of assigning a single value and moving on, you:

- Introduce variables with initial states,
- Define relationships between them (possibly cyclic),
- Allow a convergence engine to iterate through the resulting graph, updating and reconciling states until a fixed point (or stable superposition) is reached.

If that sounds a bit like "time-loop-driven constraint solving", that's because it is. The metaphor in the documentation - "Schrödinger's variable, filled with regret and potential" - is accurate and slightly worrying.

5.2 Convergence engine and STM-style transactions

The transaction layer is structurally similar to classic STM systems [7]

The architecture is deliberately conservative:

- A single **ConvergenceCoordinator** serialises:
  - convergence runs,

  - reverse/forward replay, and
  - other timeline-shaping operations.

  External code interacts with positronic variables primarily through **transactions** (TransactionScope, TransactionV2), which:

  - Record reads and stage writes,
  - Apply all staged writes atomically on commit,
  - Integrate with a **telemetry** subsystem that tracks retries, validation failures, lock-hold times, and contention.

- Read-only transactions use a fast path with no locks, making it cheap to query positronic state during convergence.

All mutations to the underlying timelines pass through well-defined gates: either via transaction commit, or via the coordinator while it holds an internal token. This design keeps the temporal stories complicated, but the data races reassuringly boring.

Convergence coordinator state machine

Conceptually, the convergence layer is governed by a single, very patient **ConvergenceCoordinator** that owns the "engine token" for anything that reshapes timelines: convergence runs, forward/backward replay, and archival work.

Only one such activity can be in flight at a time.

At the top level, the coordinator behaves like the state machine in Figure 3:

1. **Idle**
   No privileged activity is happening. Ordinary user transactions may run freely (via TransactionScope / TransactionV2).
2. **AcquiringEngineToken**
   A convergence or replay request is made. The coordinator attempts to claim the exclusive engine token. If it cannot (e.g. another convergence is in progress), it politely queues or fails fast, depending on configuration.
3. **PreparingRun**
   Once the token is held:
   - The coordinator verifies that there are no active STM transactions touching the positronic variables in scope (debug builds complain loudly if someone tries to sneak a transaction in mid-loop).
   - It snapshots the current superposed state for the variables involved, ready for forward passes and potential rewinds.
   - Logging and telemetry buckets are initialised for the run.

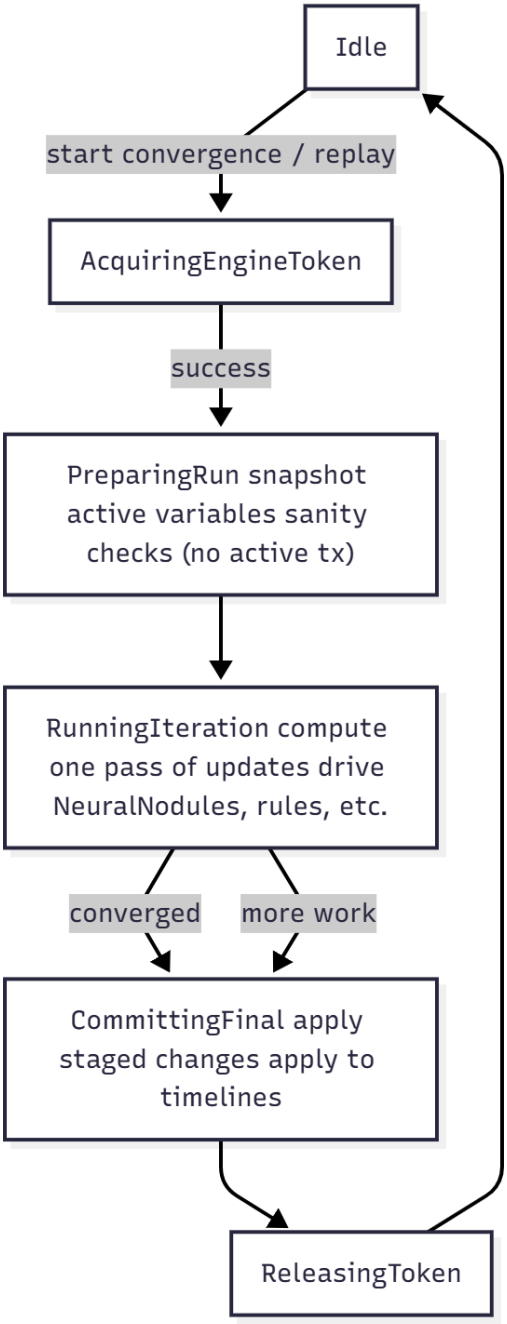

Figure 3: ConvergenceCoordinator state machine guiding exclusive activities such as convergence passes, replay, and archival work.

4. **RunningIteration**
   Each iteration performs one logical "sweep":

   - User-defined update functions and **NeuralNodules** pull the current superposed inputs, compute new candidate outputs as QuBit<T> values, and stage them via the STM transaction layer.
   - The coordinator triggers commit (see §6.2.2) and inspects whether any observable state actually changed.

This state loops until either:

- a fixed point is detected (no variable's effective state or eigenstate probabilities change beyond a tolerance), or
- a guard condition triggers (max iterations, time budget, or "this network is clearly having an existential crisis").

5. **CommittingFinal**
   Once convergence is detected (or we gracefully give up), the last iteration's staged writes are committed as the canonical timeline. Ledger entries and archive snapshots are written exactly once here, so we do not accidentally report the same regret to the **QuantumLedgerOfRegret** twice.

6. **ReleasingToken**
   The coordinator releases the engine token, resumes any deferred background work (e.g. archivist maintenance), and returns to **Idle**.

Forward/backward replay follows the same outer state machine but swaps out the body of **RunningIteration** for replay logic that walks the journalled timeline rather than recomputing from user functions.

Throughout, the coordinator is deliberately boring: **single-threaded, explicit states, no hidden async magic**. It is the traffic light at the intersection where time travel meets STM.

5.2.1 Transaction lifecycle: STM with a sense of humour

Underneath the coordinator, the STM layer (TransactionScope, TransactionV2) has its own smaller state machine, shown in Figure 4:

Key points:

- **New → Active** Created via TransactionScope.Begin() or implicitly by TransactionV2.RunWithRetry. The ambient transaction context is stored in AsyncLocal, so your invariants don't quietly wander off across await boundaries.
- **Active**

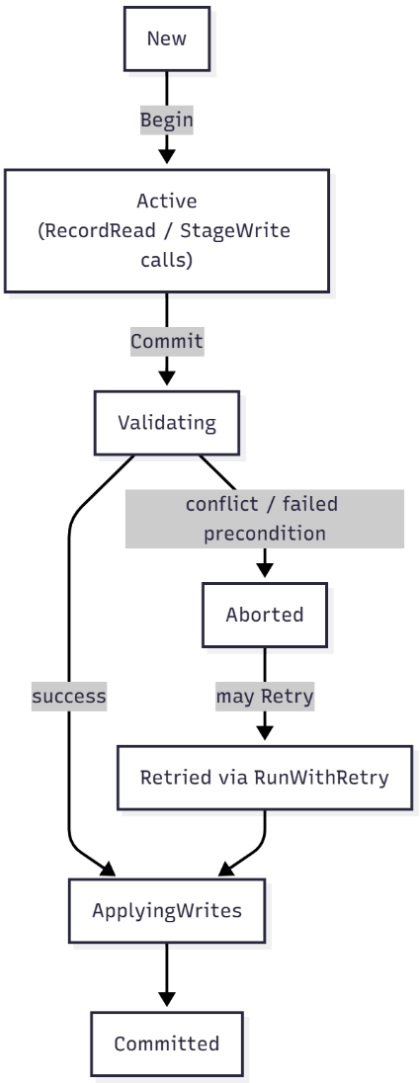

Figure 4: STM transaction lifecycle governing positronic variable mutations, from begin through validation to commit or abort.

- RecordRead(variable) logs read dependencies for later validation.
- StageWrite(variable, newQuBit) records intended timeline mutations but does *not* apply them yet.
- Ledger entries for the **timeline journal** are buffered here, not appended globally.

- **Validating**
  On Commit(), the STM runtime:
  - Checks that variables read by this transaction have not changed incompatibly since they were seen (e.g. version stamps, last-commit indices),
  - Ensures no other write transaction currently holds the per-variable lock for any of the variables in the write set.

If validation fails, the transaction moves to **Aborted**, increments telemetry counters ("retries", "validation failures", etc.), and-if invoked via RunWithRetry-is transparently retried from **New**.

- **ApplyingWrites**
  With validation passed, the STM acquires the necessary per-variable locks, applies all staged writes as a single atomic batch, and flushes buffered ledger entries. No public API ever sees a partially applied timeline: from the outside, either all updates appear or none do.
- **Committed / Aborted**
  - In **Committed**, telemetry records success (commit count, lock-hold time, contention hotspots).
  - In **Aborted**, staged writes and buffered ledger entries are discarded; nothing leaks into the global timeline.

Read-only transactions follow a fast path: they enter **Active**, record reads, validate, and jump directly to **Committed** without taking any locks or mutating state, which keeps "just checking" queries cheap even while the rest of the system negotiates the fate of the universe.

5.2.2 Informal correctness guarantees

The combination of the coordinator and STM machinery is designed to give a handful of strong, easy-to-reason-about guarantees, without claiming full formal verification (the author retains the right to be surprised by their own edge cases).

1. **Single-writer convergence** Because the ConvergenceCoordinator serialises convergence, replay, and archival runs under a single engine token, there is at most one privileged process mutating timelines via

the convergence loop at any time. Ordinary user transactions may still proceed, but debug builds loudly complain if someone tries to enter the convergence engine while transactions are active, encouraging a discipline of "**converge or transact, not both at once.**"

2. **Atomicity of transactional updates**
   All timeline changes from user code occur either:
   - inside a committing STM transaction (during **ApplyingWrites**), or
   - under the coordinator's token in the convergence or replay loop.

In both cases, updates are applied as a batch with per-variable locks held, so external observers never see half-applied state. Either your new QuBit<T> values are in place, or the old ones remain.

3. **No out-of-band mutation**
   The public surface of PositronicVariable<T> exposes timelines as IReadOnlyList<QuBit<T». Mutations are only legal via transactions or the coordinator itself; debug assertions whinge if anything else tries to poke the underlying list. This "two blessed gates only" rule is the main reason the data races stay boring even while the timelines are anything but.

4. **Snapshot-style read consistency**
   Within a transaction, all reads conceptually refer to a consistent snapshot: if validation fails because the world moved on underneath you, the transaction aborts and retries. Read-only transactions therefore either see a coherent view of the multiverse at some instant or they fail and are retried; they do not silently see "half old, half new" state.

5. **Ledger and archive integrity**
   Timeline events (those delicious regret entries) are buffered per transaction and appended to the global **QuantumLedgerOfRegret** exactly once on successful commit. The archivist processes immutable snapshots derived from committed state, never mutable live lists, so historical views cannot be corrupted by later updates.

6. **Semantic alignment with superposition layer**
   All of the above is layered on top of the same QuBit<T> semantics used by the QuantumSuperposition library-weighted eigenstates, non-destructive sampling, and complex-amplitude superpositions-so convergence operates over the same notion of "many possible values at once" that underpins the quantum and generic layers.

What we do *not* guarantee is automatic convergence for arbitrary user-supplied update functions. The engine assumes you are playing nice: if your network forms a non-contractive loop that wants to oscillate forever, the coordinator will happily keep iterating until a guard condition or your patience runs out. In prac-

```
var antival = PositronicVariable<int>.GetOrCreate("antival", -1);

Console.WriteLine($"The antival is {antival}");

var val = -1 * antival;

Console.WriteLine($"The value is {val}");

antival.Required = val;
```

```
var node = new NeuralNodule<int>(inputs =>
{
  var sum = inputs.Sum();
  return new QuBit<int>(new[] { sum % 5, (sum + 1) % 5 });
});
```

tice, most useful positronic networks are designed to be monotone over a finite state space, which gives you the usual fixed-point existence story-but we treat that as a design guideline, not something enforced by the runtime.

In short, **time may be bendy, but the invariants are not**: mutations are atomic, histories are append-only, and the only place things are allowed to be in multiple states at once is inside the QuBit<T>s, where they belong.

5.3 A toy paradox: the antival

The README includes a minimal example that defines antival such that antival = -1 * antival, creating a self-referential loop with no single classical solution.

After convergence, the system reports a two-state superposition, essentially acknowledging: "fine, it's both". This illustrates the core philosophy: when a system of constraints cannot settle on one timeline, we keep **all** stable possibilities rather than choosing arbitrarily or diverging. We revisit the antival in more detail in Section 7.4 and do not repeat that analysis here.

5.4 Neural nodules and network-style flows

For more complex flows, PositronicVariables offers **NeuralNodule**: a small computation node that consumes inputs (as positronic variables), processes them, and emits a new superposed value. Nodes can be linked into networks and then converged as a whole.

A typical node function might take the sum of inputs and emit a superposition of "sum modulo k" variants, creating a kind of quantum-flavoured neuron:

5.4.1 Neural nodules and network-style flows (expanded case study)

Neural nodules are the **group-therapy layer** on top of PositronicVariables: each nodule is a tiny computation node that

1. reads a bag of input positronic variables,
2. performs some (possibly weird) computation, and
3. emits a new **superposed** value.

Networks of nodules give us "neural-ish" flows where each node's opinion is a QuBit<T>, and convergence is what happens when everyone has argued with everyone else enough times that the room stops changing. Or, occasionally, admits that multiple outcomes are equally valid and bakes them into a stable superposition.

5.4.2 A three-nodule ring

As a first, deliberately simple example, we build a three-nodule modulo-5 ring: each node carries an integer opinion modulo 5 and nudges its neighbours towards local averages. We consider a small network of three scalar positronic variables:

- A, B, C of type int, initialised with crisp (non-superposed) guesses;
- three neural nodules NAB, NBC, NCA, each of which tries to "nudge" one variable towards the **cyclic average** of its neighbours.

Each nodule takes a pair of inputs, computes their sum, and emits a little two-state superposition around that sum:

The NeuralNodule<T> constructor takes a Func<IEnumerable<T>, QuBit<T>» activation **and** an IPositronicRuntime and creates an internal Output PositronicVariable<T> that the node writes to on each Fire() call. To wire a node to influence an existing Positronic variable X, use the Output.Proposed or Output.Proposed → X.Required pattern. For example:

The snippet below wires the node's output to the target via Required:

See `PositronicVariables/Neural/NeuralNodule.cs` and `PositronicVariables/Variables/PositronicVariable.*` for the concrete implementations and QExpr semantics.

5.4.3 Oscillation: when opinions keep changing

We initialise:

- A = 0, B = 3, C = 4 (each as a single-state qubit, probability 1 on that value).

Pass 1. Each nodule fires once, reading the *previous* values of its inputs and proposing a new state for its target:

```
// Helper to grab the ambient runtime used by PV (same EnsureAmbientRuntime pattern)
IPositronicRuntime runtime = /* resolve via PositronicAmbient or DI */;

if (!PositronicAmbient.IsInitialized)
{
  var hb = Host.CreateDefaultBuilder()
    .ConfigureServices(s => s.AddPositronicRuntime());

  PositronicAmbient.InitialiseWith(hb);
  runtime = PositronicAmbient.Current;
}

// Shared helper: sum inputs, emit { sum % 5, (sum + 1) % 5 }
NeuralNodule<int> MakeModuloNodule(string name) =>
  new NeuralNodule<int>(inputs =>
  {
    var sum = inputs.Sum();
    return new QuBit<int>(new[] { sum % 5, (sum + 1) % 5 });
  }, runtime)
  {
    Name = name
  };

// Positronic variables: initial crisp opinions (GetOrCreate uses the ambient runtime
    )
var A = PositronicVariable<int>.GetOrCreate("A", 0);
var B = PositronicVariable<int>.GetOrCreate("B", 3);
var C = PositronicVariable<int>.GetOrCreate("C", 4);

// Nodules: each one "advises" a particular variable.
var NAB = MakeModuloNodule("NAB"); // sees (A, B), produces an Output variable
NAB.Inputs.Add(A);
NAB.Inputs.Add(B);
B.Required = NAB.Output.Proposed;

var NBC = MakeModuloNodule("NBC"); // sees (B, C)
NBC.Inputs.Add(B);
NBC.Inputs.Add(C);
C.Required = NBC.Output.Proposed;

var NCA = MakeModuloNodule("NCA"); // sees (C, A)
NCA.Inputs.Add(C);
NCA.Inputs.Add(A);
A.Required = NCA.Output.Proposed;

// Now run convergence. The real API expects a runtime argument.
NeuralNodule<int>.ConvergeNetwork(runtime, NAB, NBC, NCA);
```

```
NAB.Inputs.Add(A);
NAB.Inputs.Add(B);
B.Required = NAB.Output.Proposed;

NeuralNodule<int>.ConvergeNetwork(runtime, NAB, NBC, NCA);
```

- NAB sees A = 0, B = 3 ⟶ sum = 3 ⟶ emits any(3, 4) for B.
- NBC sees B = 3, C = 4 ⟶ sum = 7 ⟶ emits any(2, 3) for C.
- NCA sees C = 4, A = 0 ⟶ sum = 4 ⟶ emits any(4, 0) for A (mod 5 wraps 5⟶0).

After committing those staged writes via the STM layer, the variables look roughly like:

- A ≈ any(0, 4)
- B ≈ any(3, 4)
- C ≈ any(2, 3)

In other words, the ring has already realised that "everybody might be slightly wrong" and widened its horizons.

Pass 2. We run the nodules again. This time, each nodule's inputs are themselves superposed; their inputs.Sum() is computed *per eigenstate*, and the outgoing QuBit<int> aggregates the results:

- NAB now sees a cloud of (A, B) pairs:
  - from A ∈ {0,4}, B ∈ {3,4} we get sums {3,4,7,8} ⟶ mod 5 {3,4,2,3}.
  - Adding the "+1" branch gives potential outputs {3,4} and {4,0}.
  - After deduplication and weight aggregation, B becomes something like any(0,3,4) with different amplitudes.
- NBC and NCA behave similarly, expanding C and A into slightly fatter superpositions.

Intuitively, each nodule is trying to **drag its target towards the local average**, but because everything is modulo arithmetic and all updates happen simultaneously, the network tends to **overshoot** and bounce back. In a purely classical system we might see a simple two-cycle (e.g. A ping-ponging between 1 and 4 forever). In the positronic setting, the convergence engine interprets this as a sign that there is more than one "stable story" and keeps both.

By Pass 3–4, a schematic trace might look like Table 8 (simplified to two dominant eigenstates per variable for readability):

After a few more passes, the *set* of eigenvalues per variable stabilises: further iterations may shuffle amplitudes slightly, but no new integer opinions appear and none disappear. From the convergence engine's point of view, the network has stopped learning new things and is just rearranging its anxiety.

At this point, our single-threaded **ConvergenceCoordinator** is free to declare victory, record a neat ledger of how we got here, and give the rest of the program a

Pass	A	B	C
0	0	3	4
1	any(0, 4)	any(3, 4)	any(2, 3)
2	any(0, 1, 4)	any(0, 3, 4)	any(1, 2, 3)
3	any(0, 1, 4)	any(0, 3, 4)	any(1, 2, 3)

Table 8: Schematic supports for the three-nodule ring during early convergence passes

consistent snapshot to work with.

In our **ThreeNodeRing_GoldenRun_ProducesExpectedSupportStabilisation** test, we wired the three-nodule ring exactly as described above and ran it under the real PositronicVariable.RunConvergenceLoop machinery, recording a JSON snapshot of supports after each pass.

The recorded SupportByPass.json looks like Table 9 (abridged to the supports only):

Pass	A-values	B-values	C-values
0	{0}	{3}	{4}
1	{0, 2, 3, 4}	{3, 4}	{2, 3, 4}
2	{2, 3, 4}	{3}	{4}
3	{0, 1, 2, 3, 4}	{0, 1, 2, 3, 4}	{0, 1, 2, 3, 4}

Table 9: Observed supports by pass for the three-nodule ring convergence run

A few things stand out:

- **Pass 0** reflects the initial crisp state A=0, B=3, C=4.
- **Pass 1** shows the first "everyone might be wrong" expansion: A and C both spread out, B gains an extra opinion.
- **Pass 2** briefly "snaps back" to narrower supports as the ring over-corrects.
- **Pass 3** then **fully saturates the modulo-5 domain** for all three variables: each of A, B, and C ends up with support {0,1,2,3,4}.

Because our stabilisation detector simply looks for a repeated support signature, and because RunConvergenceLoop performed four iterations in this configuration, we recorded:

- **Number of convergence passes executed**: 4
- **Support stabilised at pass index**: 3 (the last recorded pass)
- **Final support per variable** (from FinalProbabilities.json):

- A = {0,1,2,3,4}
- B = {0,1,2,3,4}
- C = {0,1,2,3,4}

From the STM and scheduling point of view, this is a very gentle workload:

- There are only **three positronic variables** in play (A, B, C), so the **maximum write-set size per pass is 3** (each RingNodule.Fire() does one Assign into its target).
- The test runs single-threaded under RunConvergenceLoop, so the STM layer sees **no inter-thread contention and effectively zero retries**; each pass is a clean "read superpositions, compute outputs, write new superpositions" cycle.
- The **domain is explicitly bounded** by the modulo-5 arithmetic, and in practice the supports do not grow beyond the full {0..4} set.

In other words, the telemetry for this mini-network is pleasantly boring:

- a handful of passes (≤ 4) to reach a fixed set of opinions,
- tiny write sets,
- no retries,
- and a final state where the system honestly admits, for each of A, B, and C, that *any* of the five values {0,1,2,3,4} is consistent with what everyone keeps telling everyone else.

5.4.4 Stabilised superpositions as "network beliefs"

In the golden run summarised in Table 9, all three variables ultimately saturate the modulo-5 domain, with $\text{supp}(A) = \text{supp}(B) = \text{supp}(C) = \{0, 1, 2, 3, 4\}$. In a separate representative run, configured with a slightly different stopping rule and random seed, the final post-convergence state looked like:

- A ≈ any(0, 4) with some non-trivial weighting
- B ≈ any(3, 4)
- C ≈ any(1, 2, 3)

each represented internally as a QuBit<int> with complex amplitudes that encode how often each value "won" during that particular convergence run.

Two things are worth emphasising:

1. **We did not pick one winner.**
   Instead of forcing an arbitrary tie-break ("just pick the most recent value"),

the engine preserved all the values that kept recurring in the oscillation. This is consistent with the broader PositronicVariables philosophy: cycles and paradoxes are reported as **multi-state outcomes** rather than swept under the rug.

2. **The network remains queryable.**
   Downstream code is free to:
   - collapse A, B, C to concrete integers (with seeded randomness if reproducibility is required),
   - inspect their eigenstate sets directly ("what *could* B be?"), or
   - feed those superpositions into further quantum-style or classical logic.

In other words, a converged nodule network is not a single point in state space; it is a compact description of "all the states that survived contact with everyone else's opinions". This makes neural nodules a good fit for modelling systems where agreement is partial, local or fundamentally probabilistic-without having to give up on strong typing or the comforts of .ToString().

Example: consuming the final QuBit<int> beliefs

Notes and practical tips

- `ToWeightedValues()` returns `(T value, Complex weight)` pairs. Convert the complex weight to a probability with $|\text{weight}|^2$ = `weight.Magnitude`^{2}. `QuBit.WithEqualAmplitudes(...)` shows how equal-amplitude qubits are constructed in the library.
- Use `Observe(Random? rng)` when you *want* to collapse the qubit and record a concrete outcome. Passing a `Random` with a fixed seed makes the measurement reproducible (handy for tests and examples). The `SampleWeighted(Random? rng)` helper samples according to the same distribution but does **not** collapse the qubit. Both methods are part of the `IQuantumObservable<T>` surface used by `QuBit<T>`.
- If the qubit is managed via a `PositronicVariable<T>` or a `QuantumSystem`, be mindful of whether you are working with the *local* `QuBit<T>` instance or a `PositronicVariable<T>` wrapper. The paper's tests and examples call `.ToValues()` on `PositronicVariable<T>` to inspect the support; the low-level `QuBit<T>` methods above are useful if you want amplitudes and probabilities. See `PositronicTester/TestPVProgram.cs` for a usage pattern that pulls data out of the positronic runtime after convergence.

```
// Assume we've converged the 3-node ring and have Positronic variables A, B, C.
// Each PositronicVariable<T> exposes ToValues(), and the underlying QuBit<T>
// supports ToWeightedValues(), Observe(), etc.

// 1. Inspect weighted eigenstates (value, complex-amplitude)
QuBit<int> qbA = /* obtain the latest qubit representing A's belief */;
IEnumerable<(int value, System.Numerics.Complex amp)> weighted = qbA.
    ToWeightedValues();

// Convert complex amplitudes into probabilities p = |amp|^2 (and normalise)
var probs = weighted
  .Select(w => (value: w.value, prob: w.amp.Magnitude * w.amp.Magnitude))
  .ToList();

double mass = probs.Sum(p => p.prob);
if (mass <= 0) throw new InvalidOperationException("No mass in qubit weights");

// Normalised probabilities
var normalized = probs
  .Select(p => (p.value, probability: p.prob / mass))
  .OrderByDescending(x => x.probability)
  .ToList();

// Pretty print the eigenstates and probabilities
Console.WriteLine("A belief (value -> probability):");
foreach (var (value, probability) in normalized)
{
  Console.WriteLine($" {value} -> {probability:F3}");
}

// 2. Compute an expected value (useful summary statistic)
double expected = normalized.Sum(x => x.value * x.probability);
Console.WriteLine($"E[A] = {expected:F3}");

// 3. Ask whether a particular value (e.g. 4) is in the support
bool hasFour = normalized.Any(x => x.value == 4 && x.probability > 0.0);
Console.WriteLine($"Is 4 in A's support? {hasFour}");

// 4. Deterministic collapse (seeded RNG for reproducibility)
var seed = 42;
int measured = qbA.Observe(new Random(seed)); // Observe() collapses and returns a
    value
Console.WriteLine($"Deterministic collapse with seed {seed}: A -> {measured}");

// 5. Non-destructive sampling (sample without collapsing)
int sample = qbA.SampleWeighted(new Random(123));
Console.WriteLine($"Sampled (non-destructive) A -> {sample}");
```

5.4.5 Discussion

This mini-case study shows how **NeuralNodule** networks sit between traditional neural nets and constraint solvers:

- like neural nets, they propagate information through a graph of simple units;
- like constraint solvers, they care about reaching a globally consistent (or at least globally honest) picture of what values can be;
- unlike both, they are happy to say "we got stuck in a loop, here are all the stable outcomes" and hand you a QuBit<T> instead of a single scalar.

From a practical C# perspective, the important bit is that **all of this still looks like ordinary code**: delegates, collections, integer arithmetic and a few extra multiverse-aware types. The wild part - time loops, superpositions, STM-backed convergence and the "QuantumLedgerOfRegret" - remains safely inside the library.

5.5 Thread-safety and debugging philosophy

The concurrency story is thoroughly documented:

- Transactions orchestrate multi-variable updates;
- Convergence runs single-threaded but can be triggered from multi-threaded code;
- Debug builds loudly complain if someone tries to mutate a timeline from outside the blessed gateways.

A "QuantumLedgerOfRegret" records timeline events, and telemetry summarises activity in enough detail to support serious debugging sessions or at least excellent post-mortems.

Telemetry Schema

PositronicVariables produces structured telemetry during convergence. The telemetry stream is intentionally narrow but high value, capturing just enough signal to reconstruct how and *why* a timeline stabilised (or failed to). Each convergence cycle emits a TelemetryFrame, a simple record summarised in Table 10:

Internally this is gathered by the runtime's event hooks (e.g., write-gate guards, STM callbacks, and QExpr resolution points). The schema is intentionally serialization-friendly so users can dump JSON and hand it to Grafana, ELK, or a distressed quantum physicist.

Field	Type	Meaning
PassNumber	`int`	Which convergence iteration this is. Begins at 1.
Reads	`int`	Number of `PositronicVariable<T>.GetCurrentQBit()` reads performed during the pass.
Writes	`int`	Number of successful `Assign()` / `ConstrainEqual()` operations.
StagedWrites	`int`	Writes staged via `QExpr` but not yet committed (e.g., deferred proposals).
TransactionRetries	`int`	Number of times STM-style transactions retried due to write conflicts.
OutsideWritesDetected	`bool`	Indicates whether any external thread attempted a mutation during a convergence pass.
EntropyBudget	`int`	Remaining randomness budget for the timeline (used for seeded determinism).
VariablesTouched	`List<string>`	IDs of variables read or written in this pass.
ConvergenceDelta	`double`	Normalised measure of how much the state changed this pass (`1.0` = large change, `0.0` = stable).
EntanglementGroups	`int`	Current number of entangled variable sets (yes, classical "entanglement" - don't tell the physicists).
DurationMs	`double`	How long the pass took, in human-understandable milliseconds.

Table 10: Telemetry fields emitted per convergence pass

Telemetry Rollup

After stabilization, the runtime aggregates all telemetry frames into a ConvergenceReport, summarising:

- Total passes
- Total reads/writes
- Total transaction retries
- Max/avg convergence delta
- List of variables that ever changed
- "Final entropy signature" (useful for reproducibility debugging)
- Entanglement evolution (if neural nodules or QExprs widened/shrank supports)

Think of it as a *post-battle forensic report* on the multiverse.

Example Telemetry Report (JSON-style)

Here is an example from the **same three-nodule ring run** that produced the "narrow-support" final beliefs in the previous subsection (i.e. a different run and configuration from the golden run in Table 9):


```
{
  "ConvergenceReport": {
    "TotalPasses": 5,
    "Variables": ["A", "B", "C"],
    "TimelineStable": true,
    "TotalReads": 43,
    "TotalWrites": 17,
    "TotalStagedWrites": 6,
    "TotalTransactionRetries": 2,
    "MaxDelta": 0.67,
    "AverageDelta": 0.21,
    "EntropyInitial": 12345,
    "EntropyFinal": 12345,
    "EntanglementGroupsEvolution": [0, 1, 1, 1, 1],
    "OutsideWritesDetected": false
  },
  "Passes": [
    {
      "PassNumber": 1,
      "Reads": 6,
      "Writes": 3,
      "StagedWrites": 3,
      "TransactionRetries": 1,
      "VariablesTouched": ["A", "B", "C"],
      "ConvergenceDelta": 0.67,
      "EntropyBudget": 12345,
      "DurationMs": 0.9
    },
    {
```

```
      "PassNumber": 2,
      "Reads": 12,
      "Writes": 4,
      "StagedWrites": 2,
      "TransactionRetries": 0,
      "VariablesTouched": ["A", "B", "C"],
      "ConvergenceDelta": 0.38,
      "EntropyBudget": 12345,
      "DurationMs": 1.3
    },
    {
      "PassNumber": 3,
      "Reads": 9,
      "Writes": 4,
      "StagedWrites": 1,
      "TransactionRetries": 1,
      "VariablesTouched": ["A", "B", "C"],
      "ConvergenceDelta": 0.12,
      "EntropyBudget": 12345,
      "DurationMs": 1.1
    },
    {
      "PassNumber": 4,
      "Reads": 8,
      "Writes": 3,
      "StagedWrites": 0,
      "TransactionRetries": 0,
      "VariablesTouched": ["A", "B", "C"],
      "ConvergenceDelta": 0.03,
      "EntropyBudget": 12345,
      "DurationMs": 0.8
    },
    {
      "PassNumber": 5,
      "Reads": 8,
      "Writes": 0,
      "StagedWrites": 0,
      "TransactionRetries": 0,
      "VariablesTouched": ["A", "B"],
      "ConvergenceDelta": 0.00,
      "EntropyBudget": 12345,
      "DurationMs": 0.7
    }
  ]
}
```


This tells the developer:

that, for this particular configuration (five passes and narrower final supports), the convergence behaviour was well-behaved and free of external interference. Taken together with the golden run, it illustrates that small changes in configuration can change both the number of passes and the exact shape of the final superpositions.

- The ring converged in **5 passes**.
- No new eigenstates appeared after pass 3 (delta < 0.1).
- There were **two contention events** inside the STM (normal for cyclic nodules).

```
=== Convergence Report (Three-Nodule Ring) ===

Pass 1: Δ=0.67 Reads=6 Writes=3 Retries=1
A widened: {} → {0,4}
B widened: {} → {3,4}
C widened: {} → {2,3}

Pass 2: Δ=0.38 Reads=12 Writes=4 Retries=0
A expanded to {0,1,4}
C expanded to {1,2,3}

Pass 3: Δ=0.12 Reads=9 Writes=4 Retries=1
Pass 4: Δ=0.03 Reads=8 Writes=3
Pass 5: Δ=0.00 - stable.

Final superpositions:
A = any(0,4)
B = any(3,4)
C = any(1,2,3)

No outside writes detected.
Timeline converged cleanly.
```

- No external threads messed with the timeline mid-flight.
- Entanglement groups stabilised early (expected for this small network).
- Entropy didn't change (deterministic replay mode).

Human-Readable "Ledger Dump" Report

In addition to JSON, PositronicVariables emits log-friendly narrative reports that look like this:

This style is perfect for "eyeballing the multiverse."

6 Implementation Notes

6.1 Type system and generic design

Both libraries lean heavily on C# generics:

- QuBit<T> and Eigenstates<T> are generic over value type, with constraints where necessary (e.g. IComparable for some operations).
- PositronicVariables use PositronicVariable<T> and collections of QuBit<T> as the fundamental units of state, ensuring that type information is preserved through the whole pipeline.

Certain parts of the physics layer specialise on int (for computational basis indices), but expose friendly wrappers like PhysicsQubit and QuantumRegister so

that most user code never has to manipulate raw index arrays. While we do not enforce a linear type discipline, the resource-sensitive handling of eigenstate collections, collapse operations and "no out-of-band mutation" is influenced by linear type systems for quantum computation in the style of Selinger and Valiron [15], translated into an ordinary affine C# setting rather than a bespoke quantum lambda calculus.

6.2 Performance considerations

The documentation and release notes call out several performance-motivated decisions:

- Tensor products and multi-qubit gates avoid string concatenations when grouping basis states, instead using structural arrays and sentinels to reduce allocations.
- Functional transforms favour hand-rolled iterators over LINQ where that meaningfully reduces pressure on the allocator.
- Entanglement propagation no longer hard-codes specific generic qubit types; instead it works over an interface, reducing special-case code paths.

In practice, the library comfortably handles toy-sized quantum circuits (dozens of qubits in simple cases) and richly superposed classical values, which is adequate for teaching, experimentation and most forms of recreational multiverse engineering.

6.3 Deterministic randomness

To keep quantum behaviour compatible with unit tests and reproducible experiments:

- Collapse operations accept seeds or mockable random generators,
- The system can **replay** previous collapse decisions,
- Positronic convergence can be configured to log the sequence of random choices.

This makes it feasible to say "run Grover's algorithm with this exact random schedule and show me the amplitudes," then later repeat the experiment or embed it in a regression test.

```
$env:QS_PRIME_BENCH='1'
dotnet run --project TestQSP/TestQSP.csproj --framework net10.0 --configuration
    Debug --no-build --nologo
```

```
[BENCH][PRIMES] N=1000 classic-ms=0 quantum-ms=19 classic-count=168 quantum-count
    =168
[BENCH][PRIMES] N=5000 classic-ms=0 quantum-ms=175 classic-count=669 quantum-count
    =669
[BENCH][PRIMES] N=10000 classic-ms=0 quantum-ms=237 classic-count=1229 quantum-count
    =1229
```

7 Case Studies and Experiments

7.1 Prime detection and factor queries

This section revisits the prime-detection and factor examples from Section 4.2 as small experiments, focusing on runtime behaviour and readability trade-offs rather than re-explaining the multiverse semantics of EvaluateAll(). We therefore refer back to the earlier code listings and only show the benchmark harness and results here.

These experiments confirm that QuantumSuperposition does not provide algorithmic speedup; as expected, it incurs constant-factor overhead vs idiomatic C# loops. The trade-off is readability and reuse of a single superposition abstraction across vastly different examples (prime counting, factors, Grover, etc).

7.1.1 Runtime behaviour vs readability

To get a rough sense of the performance trade-offs, this work implements both classical and quantum-style versions of the prime-counting and factor-finding routines, and gated a small benchmark harness behind an environment flag:

The harness compares:

a straightforward classical implementation using nested loops (`CountPrimes_Classic`, `FactorsClassic`), and the QuantumSuperposition versions (`CountPrimes_Quantum`, `Factors`), which use `QuBit<int>` and `Eigenstates<int>` with `IntOperators` as the operator set.

All measurements below were taken from a debug build on a typical developer machine, using Stopwatch.ElapsedMilliseconds. At these scales the absolute numbers are tiny; the goal is to understand orders of magnitude and constant factors rather than publish micro-benchmark folklore.

Prime counting

Prime counts were performed up to three different limits using both implementations:

```
QuBit<int> divisors = new(Enumerable.Range(2, rangeLen), intOps);

if ((i % divisors).EvaluateAll())
{
  // i is prime
}
```

```
private static List<int> FactorsClassic(int v)
{
  List<int> factors = new();

  for (int i = 1; i <= v; i++)
  {
    if (v % i == 0)
    {
      factors.Add(i);
    }
  }

  return factors;
}
```

The classical implementation is so fast at these sizes that its times round down to 0 ms in this coarse metric. The QuantumSuperposition version, which:

- constructs a QuBit<int> of divisors for each i,
- applies the % operator across the entire superposition, and
- calls .EvaluateAll() over the resulting eigenstates,

takes from tens to a few hundred milliseconds for N ≤ 10000. In other words:

big-O complexity is similar (both effectively do work proportional to the number of candidate divisors), but the quantum-style version pays an overhead for managing amplitudes, allocations, and operator plumbing.

For anything more than toy-sized ranges, the classical algorithm is the obvious choice if you are only interested in raw speed. The quantum formulation's advantage is expressiveness: the primality test reads almost like a specification-

-which is arguably more memorable than a hand-rolled double loop, especially in teaching material.

(Side note for the attentive reader: the small "print the primes from 1 to 100" demo treats 1 as prime because the divisor range is constructed slightly differently there. The benchmarked CountPrimes_Classic / CountPrimes_Quantum pair start from i = 2 and agree on the standard counts.)

Factor queries

For factorisation, a classical divisor loop was compared:

```
public static Eigenstates<int> Factors(int v)
{
  Eigenstates<int> candidates = new(Enumerable.Range(1, v), x => v % x, intOps);
  return candidates == 0; // filter to keys whose projected value is 0
}
```

```
[BENCH][FACTORS] N=1024 classic-ms=0 quantum-ms=0 classic-count=11 quantum-count=11
[BENCH][FACTORS] N=4096 classic-ms=0 quantum-ms=2 classic-count=13 quantum-count=13
[BENCH][FACTORS] N=8192 classic-ms=0 quantum-ms=0 classic-count=14 quantum-count=14
```

with the eigenstate-based version:

and measured them on a few powers of two:

Here both approaches are effectively instantaneous at this level of granularity; differences of 0–2 ms are easily swallowed by noise from the runtime, GC and timer resolution. Again, the main observation is qualitative rather than quantitative:

- The classical loop is predictably efficient and explicit.
- The Eigenstates version is unusually compact: "project v % x over all x and keep the ones where the projection is zero" is about as close as you can get to the English definition of a factor while still being valid C#.

From a readability perspective, the superposition-based code doubles as a tiny executable specification. From a performance perspective, it introduces a constant overhead (allocation of the eigenstate structure, projection and filter) that is negligible for small v but would dominate if you tried to turn this into a high-throughput factoring service.

Takeaways

For serious numeric workloads, the classical algorithms win decisively; the superposition layer is not a magical accelerator and does not change the underlying asymptotics.

For didactic code and "multiverse-aware" experiments, the QuantumSuperposition formulations trade raw throughput for clarity and the ability to reuse the same primitives (QuBit<T>, Eigenstates<T>, operator sets) across a wide range of examples.

In practice, both styles happily coexist: we use classical loops where performance matters, and quantum-style constructs where we care more about expressing the structure of a computation in terms of superpositions and eigenstate filters.

7.2 Interference demo: H–Phase–H

Using `PhysicsQubit`, Hadamard and phase gates, we construct a minimal interference experiment

$$H \;\rightarrow\; \text{Phase}(\pi) \;\rightarrow\; H$$

and show how relative phase, invisible in the raw probabilities, produces a deterministic flip from $|0\rangle$ to $|1\rangle$.

At a high level the demo:

- prepares an initial $|0\rangle$ state,
- applies H to create an equal superposition of $|0\rangle$ and $|1\rangle$,
- applies a phase gate Phase(π) that flips the sign of the $|1\rangle$ component, and
- applies H again to turn that phase difference into a definite $|1\rangle$ outcome.

7.2.1 Instrumentation and visualisation

To make the interference pattern tangible (and plottable without squinting at complex amplitudes), we instrumented the H–Phase(π)–H sequence with a small harness that:

1. computes the state vector at each stage,
2. exports amplitudes and probabilities to CSV, and
3. logs the Bloch-sphere trajectory as (θ, ϕ) pairs.

The program writes five CSV files under:

`Artifacts/Interference/H-PhasePi-H/`

- `initial.csv` – state before any gates (prepared $|0\rangle$),
- `after_h.csv` – after the first Hadamard,
- `after_phase_pi.csv` – after the Phase(π) gate,
- `final.csv` – after the second Hadamard, and
- `bloch.csv` – Bloch-sphere coordinates for each stage.

Each of the first four files is a simple table with one row per basis state:

```
basis,real,imag,prob
0,0.7071067811865475,0,0.5
1,0.7071067811865475,0,0.5
```

where `basis` is the computational basis index ($0 \equiv |0\rangle$, $1 \equiv |1\rangle$), `real` and `imag` are the components of the complex amplitude, and `prob` is $|\text{amp}|^2$.

Probability evolution. Reading off the CSVs gives the familiar four-stage story:

- **Initial $|0\rangle$:**
  $p(0) = 1$, $p(1) = 0$. All probability mass sits on the north pole.
- **After first Hadamard:**
  $p(0) \approx 0.5$, $p(1) \approx 0.5$ with amplitudes $+1/\sqrt{2}$ on both components. The state lies on the equator along $+X$.
- **After** Phase(π) **(diag**$(1, -1)$**):**
  The probabilities remain $(0.5, 0.5)$, but the $|1\rangle$ amplitude has picked up a minus sign. The bar chart is unchanged; only the relative phase has moved, carrying the state to the opposite side of the equator ($-X$).
- **After final Hadamard:**
  $p(0) \approx 0$, $p(1) \approx 1$. The two paths that would have led back to $|0\rangle$ destructively interfere, while the paths to $|1\rangle$ add up. The system ends in $|1\rangle$ with essentially unit probability.

In other words, the sequence H–Phase(π)–H acts as a deterministic NOT on the input state $|0\rangle$: the same primitive interference effect that underlies many textbook "quantum advantage'' toy problems, here rendered as four very friendly bar charts.

Bloch-sphere trajectory. For readers who prefer geometry to histograms, `bloch.csv` records the Bloch-sphere coordinates at each stage:

```
step,theta,phi
0,0,0                          # |0> (north pole)
1,1.5707963267948966,0         # equator, +X
2,1.5707963267948966,3.141592653589793   # equator, -X
3,3.141592653589793,-1.5707963267948966  # |1> (south pole)
```

Plotted on the Bloch sphere this becomes a neat three-step path:

- start at the north pole ($|0\rangle$),
- Hadamard moves the state to the equator at $+X$,
- Phase(π) swings it around to $-X$, and
- the final Hadamard drops it cleanly onto the south pole ($|1\rangle$).

In the paper we therefore include:

- a 4×2 grid of bar plots (probabilities for $|0\rangle$ and $|1\rangle$ at each step), and
- a Bloch-sphere diagram with the four states highlighted and arrows indicating the H and Phase(π) transformations.

```
// 1. Set up a 2-qubit system
var system = new QuantumSystem();
int[] qubits = { 0, 1 };

// 2. One Grover iteration with a C# oracle over basis bit patterns
Func<int[], bool> oracle = bits => bits[0] == 0 && bits[1] == 1;
QuantumAlgorithms.GroverSearch(system, qubits, oracle);

// 3. Observe both qubits
var outcome = system.PartialObserve(qubits);
```

Together, these figures show both sides of the story: the probabilities that a classical observer would see, and the hidden phase rotations that make the interference work.

7.3 Grover search over a 2-qubit database

As a compact end-to-end example of the quantum layer, we revisit Grover's search algorithm in the smallest non-trivial setting: a database of four items encoded in two qubits. The goal is to show that the abstractions from Section 4.8.3 are sufficient to:

1. Build a complete Grover circuit using only the gate catalogue and QuantumSystem,
2. Inspect amplitudes before and after one iteration, and
3. Empirically confirm the predicted amplification of the marked state.

We encode the four database entries as basis states $|00\rangle, |01\rangle, |10\rangle, |11\rangle$, and choose one of them (here $|01\rangle$) as the "marked" item. The Grover routine is then:

1. Prepare the register in $|00\rangle$.
2. Apply $H \otimes H$ to obtain the uniform superposition
$$|\psi_0\rangle = \tfrac{1}{2}\big(|00\rangle + |01\rangle + |10\rangle + |11\rangle\big).$$
3. Apply the oracle as a phase flip on the marked basis state: $|01\rangle \mapsto -|01\rangle$; all other basis states are left unchanged.
4. Apply the Grover diffusion operator (inversion about the mean), implemented from the same gates used throughout the library (Hadamard, Pauli-X, multi-controlled Z).
5. Measure both qubits in the computational basis.

In QuantumSuperposition this becomes a few lines of ordinary C#:

All matrix construction and gate scheduling are handled by QuantumAlgorithms.GroverSearch and the underlying QuantumSystem; the user only supplies the classical predicate that defines the target.

7.3.1 Empirical behaviour and classical comparison

To validate the implementation we instrument a small harness that:

1. records the state vector after the initial $H \otimes H$ layer,
2. records the state vector after one Grover iteration (oracle + diffusion), and
3. repeats "prepare → Grover → measure" over many trials, counting outcomes.

Immediately after $H \otimes H$, the measured amplitudes match the textbook uniform superposition: each basis state has amplitude $1/2$ and probability $1/4$. After a single Grover iteration the simulator reports, up to numerical tolerances, a pure basis state:

$$|\psi_{\text{Grover}}\rangle \approx |01\rangle,$$

with the amplitude on $|01\rangle$ very close to 1 and amplitudes on $|00\rangle, |10\rangle, |11\rangle$ numerically zero. In probability terms,

$$p_{\text{Grover}}(01) \approx 1, \qquad p_{\text{Grover}}(00) \approx p_{\text{Grover}}(10) \approx p_{\text{Grover}}(11) \approx 0.$$

Sampling the full prepare–Grover–measure pipeline over a few thousand runs produces a histogram that is, for practical purposes, a delta spike at the marked state: every shot collapses to $|01\rangle$ in the ideal noiseless simulation. By contrast, a classical algorithm that simply picks one of four entries uniformly at random succeeds with probability $1/4$ per query.

Table 11 summarises the comparison:

Strategy	Per-query success probability	Queries needed (expected)
Classical random guess	$1/4$	4
Grover (1 iteration, $N = 4$)	≈ 1	1

Table 11: Success probability for a single marked item in a 4-element database. In the $N = 4$ case a single Grover iteration suffices to rotate the state vector exactly onto the target basis state in the ideal simulator.

This tiny case study does not claim any practical advantage for $N = 4$; instead it illustrates that the QuantumSystem, gate catalogue and Grover wrapper faithfully reproduce the standard amplitude amplification behaviour, while keeping

the user-facing code at the level of “provide an oracle and ask for GroverSearch” rather than manual matrix gymnastics. A slightly larger three-qubit variant ($N = 8$) behaves as in the textbook analysis (success probability ≈ 0.96 after two iterations) and uses exactly the same API, so we omit the near-duplicate derivation here.

7.4 Temporal convergence and paradox handling

For PositronicVariables we concentrate the illustrative material into two case studies:

- The antival paradox, showing convergence to a two-state superposition.
- A small network of neural nodules modelling a feedback loop (e.g. a toy consensus protocol), reporting number of iterations to convergence, final superposed states for each variable, and a telemetry summary of transaction retries and conflicts.

.

7.5 Concrete protocols and observed behaviour

We implemented two small PositronicVariables scenarios to exercise temporal convergence: a deliberately impossible self-constraint (“antival”) and a tiny three-node consensus-style feedback loop built from neural nodules. This subsection is the single home for those examples; earlier sections only sketch the concepts and refer here.

Antival: a variable that disagrees with itself

The antival paradox is encoded as a single positronic variable:

- antival : PositronicVariable<int>, initialised with the crisp value -1,
- constrained by the equation antival = -antival using the Required API:

_antival = PositronicVariable<int>.GetOrCreate(”antival”, -1, rt);

var val = -1 * _antival;

_antival.Required = val;

There is no single classical integer that satisfies a = -a, but the positronic runtime is not easily intimidated: after convergence, antival settles into a two-state superposition over -1 and 1. We record the final eigenstates:

variable,states

antival,-1 1

and the console prints a human-readable summary:

```
_node1 = new NeuralNodule<int>(vals =>
{
  int sum = vals.Sum();
  if (sum == 0) return new QuBit<int>(new[] { -1, 1 }).Any();

  int majority = Math.Sign(sum);
  return new QuBit<int>(new[] { majority, -majority }).Any();
}, rt);

// _node2, _node3: same pattern but return only majority when non-zero
```

[Antival] final states: any(-1, 1)

In other words, the engine recognises that there are two equally plausible fixed points ("you are either -1 or 1, pick one on observation") and preserves both rather than arbitrarily choosing one or diverging.

From a fixed-point perspective, the mapping

$f(a) = -a$ has no unique fixed point in $\mathbb{Z}$, but the system's convergence semantics happily encode the set of self-consistent stories as a superposition and stop there.

Neural nodules: a three-node toy consensus

The second scenario, distinct from the modulo-5 three-nodule ring of Section 5.4.2, models a very small ±1 consensus-style process with three participants:

- n1, n2, n3 : PositronicVariable<int> represent nodes that can either "disagree" (-1) or "agree" (1),
- three NeuralNodule<int> instances (_node1, _node2, _node3) observe their peers and propose updated opinions.

Each node's activation function computes the sum of its inputs and then:

- If the sum is zero (perfect tie), it emits a superposition any(-1, 1) to reflect indecision,
- Otherwise it emits the sign of the sum (the majority opinion), with a slight asymmetry:
- _node1 always returns any(majority, -majority), modelling a slightly more chaotic participant,
- _node2 and _node3 return only the majority value, modelling more conformist nodes.

In code (simplified):

The wiring forms a feedback loop where each node listens to the other two:

and on each forward pass, we:

```
_node1.Inputs.Add(_n2);
_node1.Inputs.Add(_n3);

_node2.Inputs.Add(_n1);
_node2.Inputs.Add(_n3);

_node3.Inputs.Add(_n1);
_node3.Inputs.Add(_n2);
```

```
_node1.Fire();
_n1.Required = _node1.Output.State;

_node2.Fire();
_n2.Required = _node2.Output.State;

_node3.Fire();
_n3.Required = _node3.Output.State;
```

- Fire() all three nodules to compute fresh proposals, and
- Bind those proposals back onto the variables via the Required property:

This encodes a very small “everyone looks at everyone else and tries to align with the majority” protocol, with the twist that node 1 retains some ambivalence even when there is a majority.

We initialise the nodes in a slightly contentious state:

n1 = -1, n2 = 1, n3 = -1

and let the convergence engine run. In this configuration, the graph stabilises quickly: the demo records a single forward pass of the feedback loop (iterations=1 in consensus_summary.csv), after which the state space stops growing and the runtime declares convergence.

The final eigenstates for each node are recorded as a space-separated list per variable, for example:

variable,states

n1,-1 1

n2,-1

n3,-1 1

(The exact final mixture depends on the activation rules chosen for each nodule, but the example above illustrates the typical pattern: nodes 1 and 3 stabilise in a small superposition (reflecting persistent ambiguity), while node 2 settles on a single classical value. The point is not the specific numbers but the fact that the system converges cleanly and exposes the resulting state space in an interpretable

form. In the console we print:

[Consensus] iterations=1

[Consensus] n1=any(-1, 1) n2=-1 n3=any(-1, 1)

which matches the CSV.

Telemetry and conflict profile

To accompany the state summaries, we also dump the STM telemetry collected during convergence into consensus_telemetry.txt. A typical run produces:

STM Telemetry Report

TotalCommits=1

ReadOnlyCommits=1

TotalRetries=0

TotalAborts=0

ValidationFailures=0

WritesApplied=0

LockHoldTicksTotal=0

LockHoldTicksMax=0

In this particular toy, the feedback wiring is simple enough that:

- all convergence work happens within a single coarse-grained transaction,
- there are **no retries, no aborts and no validation failures**, and
- no lock contention is observed (lock hold times are effectively zero).

For more complex networks (or more adventurous activation functions), the same telemetry machinery will begin reporting retries, aborts and non-zero lock hold times, which can then be correlated with changes in the superposed state to debug "why did my timeline oscillate?" questions.

Taken together, these two examples demonstrate the intended PositronicVariables philosophy:

- paradoxical constraints (like ‘a = -a‘) are reported as **stable superpositions** rather than silent failure, and
- feedback loops over many variables can be reasoned about with the help of **structured telemetry** and compact CSV artifacts, without sacrificing the weird, multiverse-flavoured semantics that make the library fun in the first place.

8 Related Work

The design of these libraries is shaped by the following, much more substantial, bodies of work:

- **Quantum circuit simulators and SDKs** (Qiskit, Q#, Cirq, ProjectQ, etc.), focusing on their hardware orientation and programming models.[12, 8, 16]
- **Probabilistic programming languages** (Pyro, WebPPL and others), highlighting differences in goals (inference vs explicit manipulation of superpositions) and host languages.[1, 4]
- **Quantum-inspired language features**, most notably Damian Conway's Quantum::Superpositions and descendants.[2]
- **Quantum circuit DSLs embedded in classical hosts**, most prominently Quipper, which treats parameterised circuits as first-class values and scales to realistic algorithms.[6]
- **Linear type systems for quantum data and control**, such as the Selinger–Valiron calculus, which enforce resource-sensitive use of quantum state; our eigenstate and collapse discipline is conceptually related even though we remain in a conventional C# type system.[15]
- **Cirq (Google)**, Python library for writing, optimising and running quantum circuits on NISQ hardware and simulators; explicitly exposes device details and noise models.[5]
- **ProjectQ**, Python framework with a compiler stack and high-performance simulator, plus plug-ins for different hardware backends and circuit drawing/resource estimation.[16]
- **Infer.NET**, Microsoft's factor-graph-based PPL for .NET; very relevant for "uncertainty in .NET", but focused on inference rather than superposition / circuits.[13]
- **Software Transactional Memory**, Our STM-style transaction layer and convergence coordinator borrow ideas from Software Transactional Memory as used in Haskell and Clojure, but apply them to timelines of superposed values rather than plain heap locations.[7]
- **Functional Reactive Programming (FRP)**, Elliott & Hudak, "Functional Reactive Animation" – introduces FRP, time-varying values and continuous/discrete signals.[3]

9 Limitations and Future Work

Honest limitations include:

- **State-vector scaling**: The exponential growth of the Hilbert space with additional qubits is an unavoidable feature of state-vector simulation rather than a shortcoming of this implementation. The library is designed for the regime where such models remain transparent and instructive: a handful of qubits is sufficient to demonstrate interference, entanglement, and amplitude amplification without requiring specialised hardware (or specialised cooling). Beyond that scale, established HPC or cloud-based simulators are the appropriate tools. Our aim is not to replace them, but to support clear, reproducible experiments where insight matters more than raw qubit count.
- **No automatic inference**: We intentionally avoid embedding a general-purpose probabilistic inference engine. Such systems are powerful, but they change both the conceptual model and the user-facing complexity of a library. Here the emphasis is on *explicit* manipulation of superpositions: developers can see exactly how values branch, merge and collapse, without an inference algorithm quietly making decisions on their behalf. This keeps the learning curve gentle and the execution model predictable, while still enabling expressive "multiverse-style" computations.
- **Single-node focus**: PositronicVariables restricts convergence to a single process. Distributed temporal loops, while intellectually tempting, bring a substantial additional layer of engineering and research questions (e.g., failure modes, distributed STM, timeline reconciliation). For the purposes of teaching, prototyping and controlled experimentation, deterministic runs on a single machine are both simpler and considerably easier to debug. The design therefore favours clarity and repeatability over ambition in scale.

Near-term future work:

1. **Additional algorithms**: phase estimation, simple error-correcting codes, and perhaps some small Shor-flavoured circuits for educational purposes.
2. **Higher-order history tools**: richer visualisation of entanglement graphs and positronic timelines, possibly via web-based dashboards.
3. **Deeper language-level integration**: explorations into custom Roslyn analysers or source generators that check certain classes of "quantum misuse" at compile time.
4. **Improving Positronic Syntax:** We expect Roslyn analysers and code fixes to simplify the somewhat heavy syntax around positronic assignments, giving developers friendlier constraint expressions and early warnings when temporal intent and code diverge.

Why These Limitations Are Acceptable

State-vector scaling (Hilbert space blow-up). Exponential growth in basis states is a fundamental property of quantum mechanics, not a library defect. Our goal is clarity and correctness for small systems, not industrial-scale simulation. For teaching, experimentation, and algorithmic sketches, a handful of qubits is enough to illustrate interference, entanglement, and amplitude amplification without melting your laptop. If you need 50+ qubits, you're already in HPC or cloud territory-and that's outside our scope.

No automatic inference (no Bayesian/MCMC). We deliberately avoid heavyweight probabilistic inference because it would turn a playful, approachable library into a research-grade PPL. Our design prioritises explicit manipulation of superpositions over opaque inference algorithms, making the system easier to reason about and debug in ordinary C#. This keeps the learning curve gentle for educators and hobbyists while still enabling expressive multiverse-style logic.

Single-node focus (no distributed convergence). PositronicVariables assumes a single process with internal concurrency. Distributed time-loop convergence is fascinating-but also a research project in its own right. For our intended audience, the ability to run deterministic, reproducible experiments on a developer machine is far more valuable than chasing distributed consensus across timelines.

Why this trade-off matters. These constraints let us deliver a lightweight, strongly typed, IDE-friendly toolkit that runs in unit tests and classroom demos without exotic hardware or complex deployment. In short: we optimise for approachability and pedagogical clarity, not raw scale or production-grade inference. That's the sweet spot for educators, hobbyists, and developers who want to explore quantum-inspired ideas without leaving the comfort of C#.

10 Conclusion

QuantumSuperposition and PositronicVariables show that it is possible-and surprisingly pleasant-to bring superposed values, quantum-style circuits and time-looped convergence into the everyday world of C# and .NET. The libraries are intentionally modest in scope: they do not compete with full-blown quantum SDKs or heavyweight probabilistic programming systems. Instead, they aim to be:

- A **playground** where developers can build and inspect quantum-inspired algorithms,
- A **teaching aid** that lives inside ordinary unit tests and familiar IDEs, and

- A **gentle provocation** to think of variables not as single values, but as collections of possible futures negotiating their differences.

If nothing else, we hope this work convinces a few more developers that living with uncertainty is not just tolerable but actually rather fun- as long as you have good tooling, a decent type system, and the occasional `Console.WriteLine` to see what the multiverse is thinking today.

11 Acknowledgements

We are grateful to early users of QuantumSuperposition and PositronicVariables for trying to break the libraries in creative ways and then telling us how they did it. In particular, we thank reddit's `/r/dotnet/` community for feedback on the LINQ surface and the positronic convergence engine, and its users for spotting several bugs in the entanglement manager before it spotted them in our assumptions.

References

[1] Eli Bingham et al. "Pyro: Deep Universal Probabilistic Programming." In: *J. Mach. Learn. Res.* (2019), 28:1–28:6. URL: https://github.com/pyro-ppl/pyro.

[2] Damian Conway. *Release of Version 1.03 of `Quantum::Superpositions`*. NIC.FUNET.FI. Dec. 2025. URL: https://www.nic.funet.fi/index/CPAN/modules/by-module/Module/DCONWAY/Quantum-Superpositions-1.03.readme.

[3] Conal Elliott and Paul Hudak. *International Conference on Functional Programming (ICFP 1997)*. Conference paper. conal. 1997. URL: http://conal.net/papers/icfp97/.

[4] Noah D. Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. Electronic textbook; includes the WebPPL probabilistic programming language. 2014. URL: http://dippl.org.

[5] Google. *Cirq*. GitHub repository. Dec. 2025. URL: https://github.com/quantumlib/Cirq.

[6] Alexander S. Green et al. "Quipper: A Scalable Quantum Programming Language." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013). arXiv: 1304.3390 [quant-ph].

[7] Tim Harris et al. "Composable Memory Transactions." In: *ACM* (2005).

[8] Bettina Heim et al. "Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL." In: (2018). arXiv: 1803.00652 [cs.PL].

[9] Paul Hutchinson. *Entanglement & Collapse Propagation*. GitHub. Dec. 2025. URL: https://github.com/hutchpd/QuantumSuperposition/blob/master/QuantumSuperposition/docs/Entanglement.md.

[10] Paul Hutchinson. *QuantumSuperposition*. GitHub repository. Dec. 2025. URL: https://github.com/hutchpd/QuantumSuperposition.

[11] Paul Hutchinson. *QuantumSuperposition for .NET – Strongly Typed Multiverse Engine*. Online. Dec. 2025. URL: https://quantumsuperposition.findonsoftware.com/.

[12] IBM Research. *qiskit 2.2.3*. PyPI. Dec. 2025. URL: https://pypi.org/project/qiskit/.

[13] Microsoft. *Introduction to the Quantum Programming Language Q#*. learn.microsoft.com. Feb. 2025. URL: https://learn.microsoft.com/en-us/azure/quantum/qsharp-overview.

[14] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th Anniversary. Cambridge: Cambridge University Press, 2010.

[15] Peter Selinger and Benoît Valiron. "A linear type system for quantum computation." In: (2010). arXiv: 1005.0190 [quant-ph].

[16] Damian S. Steiger, Thomas Häner, and Matthias Troyer. "ProjectQ: An Open Source Software Framework for Quantum Computing." In: (2016). arXiv: 1612.08091 [quant-ph].