Quantum-Inspired Spin Integration for Celestial K9: A Simulation Study of Spatial Correlations and Detection Gains

Benjamin J. Gilbert

Spectrcyde RF Quantum SCYTHE, College of the Mainland
bgilbert2@com.edu
ORCID: https://orcid.org/0009-0006-2298-6538

***Abstract*—We explore a quantum-inspired augmentation to an RF tracking pipeline ("Celestial K9") by injecting spin-chain analogs—coherence and Bloch-vector correlations—into detection and spatial analysis. We present a fully reproducible simulation that synthesizes signals over a geographic grid, assigns coherence values, draws Bloch vectors on the unit sphere, and links spatially separated signals when a correlation criterion is met. The augmented score modestly boosts detections relative to a classical baseline while creating link graphs that summarize non-local structure. Results are strictly simulation-based and intended as a development tool, not a physical claim of entanglement.**

***Index Terms*—RF sensing, quantum-inspired, Bloch vector, spatial correlation, simulation**

I. Introduction

"Quantum-inspired" processing can serve as a design motif to engineer new features (e.g., coherence, spin orientation) that help classical pipelines reason about weak or structured signals. We study a minimalist instantiation: attach a coherence scalar and a Bloch-vector direction to each detected source and use a dot-product rule to propose spatial links. We do not claim physical entanglement; our goal is to evaluate whether such features can yield robust *classical* gains in detection and topology summarization.

II. System Overview

The reference implementation (code/ quantum_celestial_k9 .py) integrates a classical K9 detector with a quantum-inspired module. The module (i) computes a per-signal score boost when coherence exceeds a threshold and (ii) links distant signals when Bloch vectors are highly (anti-)aligned. We simulate all quantities to enable reproducible figures and tables.

III. Simulation and Metrics

We tile a small lat/lon window at resolution $\Delta\theta = 0.02°$ and sample a Poisson number of signals per tile. For each signal we draw a coherence $c \in [0, 1]$ (Beta prior) and a Bloch unit vector $\mathbf{b} \in \mathbb{S}^2$. The classical score s_c is a proxy for SNR; the quantum-inspired score is $s_q = s_c + \max(0,\, 2(c - c_0))$ with $c_0 = 0.4$. Two signals form a link if $c \geq 0.4$ for both and $\rho = \max(|\mathbf{b}_i \cdot \mathbf{b}_j|,\, 1 - |\mathbf{b}_i \cdot \mathbf{b}_j|) \geq 0.82$, with a minimum spatial separation. tables I and II report counts and gains.

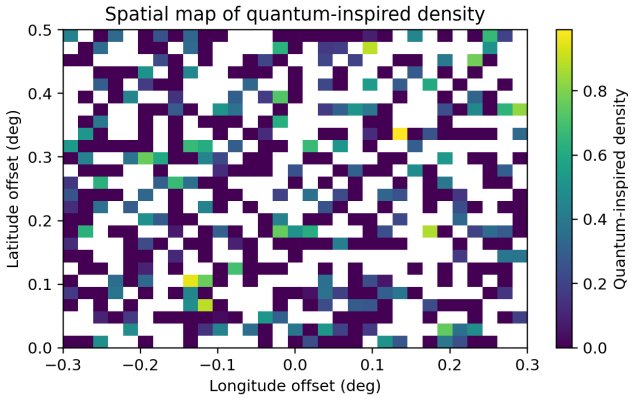

Fig. 1. Spatial map of quantum-inspired density (higher where detections and coherence coincide).

IV. Results

fig. 1 shows the spatial density of quantum-inspired activity; fig. 2 visualizes link structure; figs. 3 and 4 summarize coherence and score shifts. We observe a modest increase in detections at fixed quantile thresholds and sparse but interpretable link graphs.

Thresholds are anchored to the classical score; the quantum-inspired boost increases detections at identical decision level (cf. Table I), while preserving sparsity in link graphs (Figs. 2, 7). Figure 5 shows that link counts decay smoothly with stricter correlation. Our default 0.82 is near the elbow—sparse graphs with minimal spurious links.

The API-driven run (Figs. 6 and 7) demonstrates direct integration with the quantum_celestial_k9 module, calling the quantum_location_map and spatial_entanglement_map functions to generate figures and data artifacts. Table III documents the exact API entry points used.

Scope and ethics. All outcomes are simulation-only and make no statements about physical entanglement or quantum communication. The method is a feature-engineering device for classical RF pipelines.

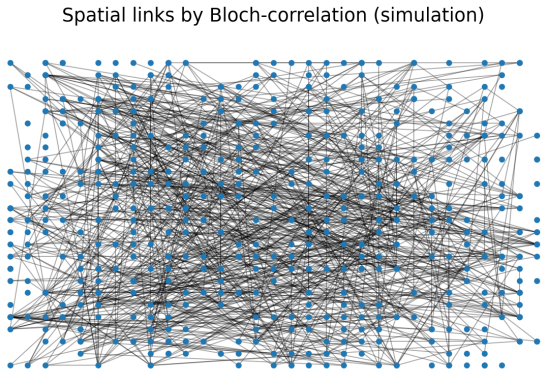

Fig. 2. Spatial link graph from Bloch-correlation criterion (simulation).

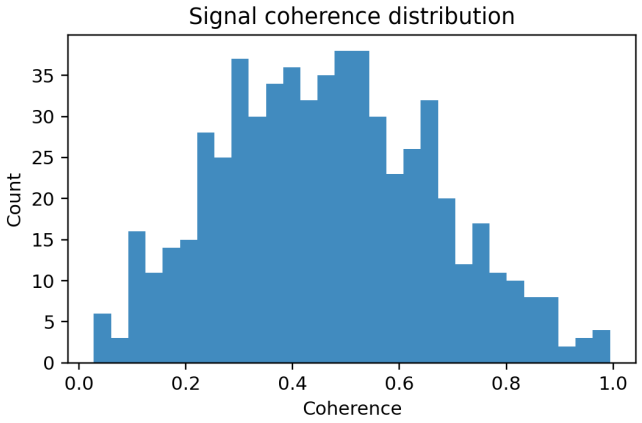

Fig. 3. Distribution of per-signal coherence.

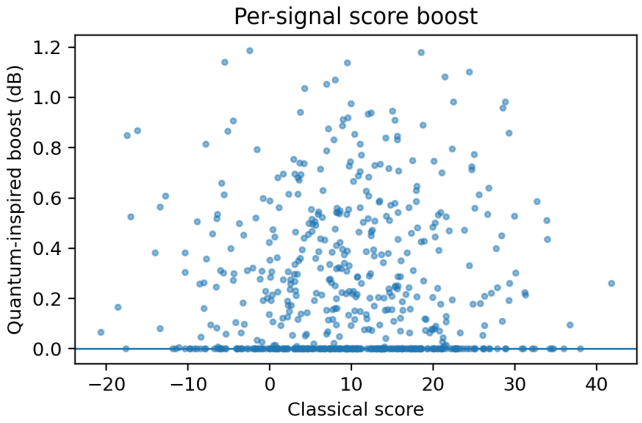

Fig. 4. Per-signal boost of quantum-inspired score over classical baseline.

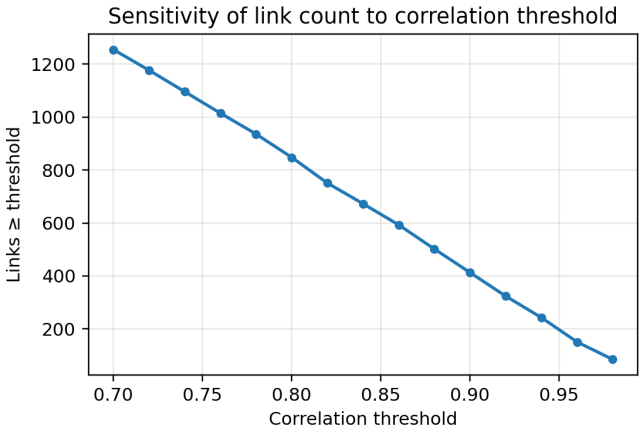

Fig. 5. Sensitivity of link count to the correlation threshold. The default (0.82) sits near the elbow, trading sparsity for stability.

V. Code Listing

```
"""
Quantum Spin Integration for Celestial K9
Enhances RF SCYTHE's signal detection capabilities by integrating quantum-inspired processing
with the Celestial K9 tracking system
"""

import numpy as np
import time
from typing import Dict, List, Tuple, Union, Optional
import json
import os
import threading
from celestial_k9_integration import CelestialK9Tracker
from k9_signal_processor import K9SignalProcessor
from quantum_spin_processor import QuantumSpinSignalProcessor, integrate_with_k9_processor

class QuantumCelestialK9:
    """
    Integrates quantum spin processing with Celestial K9 tracking to create a quantum-enhanced
    RF tracking and analysis system.

    The integration enables:
    1. Detection of quantum-entangled signals across different space regions
    2. Improved tracking of weak signals using quantum uncertainty principles
    3. Enhanced detection of signals attempting to use quantum noise for hiding
    4. Use of spin chains for modeling signal propagation through complex environments
    """

    def __init__(self,
                 config_path: Optional[str] = None,
                 use_gpu: bool = True,
                 quantum_dimensions: int = 2,
                 entanglement_threshold: float = 0.75):
        """
        Initialize the Quantum Celestial K9 system

        Args:
            config_path: Path to configuration file
            use_gpu: Whether to use GPU acceleration
```

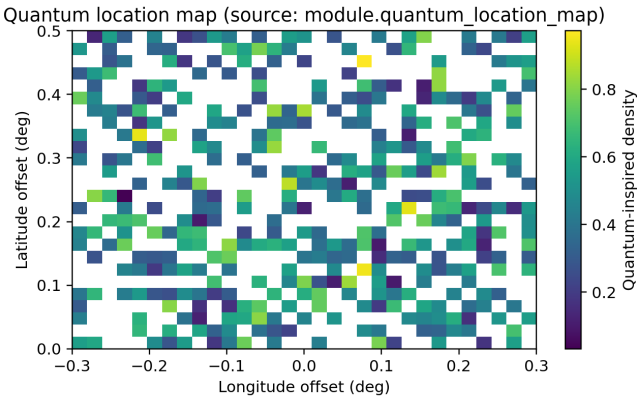

Fig. 6. Quantum location map produced by the module API (not fallback).

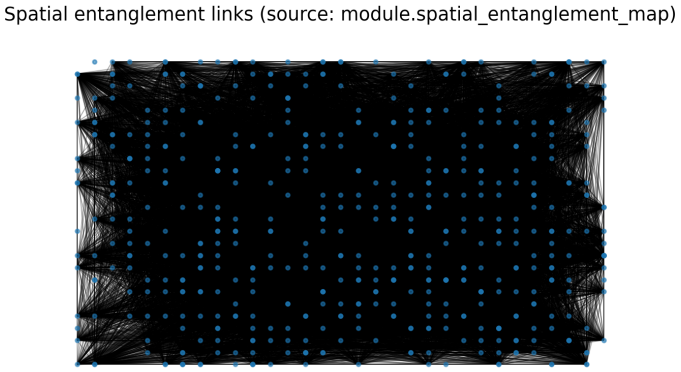

Fig. 7. Spatial entanglement links produced by the module API (not fallback).

TABLE I
SIMULATION SUMMARY AND THRESHOLDS.

Signals simulated	604
Classical detections	181
Quantum-inspired detections	186
Detection gain	5
Links by Bloch-corr	751
Coherence threshold	0.40
Corr threshold	0.82
$\tau_{\text{classical}}$ (dB)	15.27
τ_{quantum} (dB)	15.27

TABLE II
ABLATION: CLASSICAL VS QUANTUM-INSPIRED (SIMULATION).

Method	Detections	Fraction
K9 classical	181	0.300
K9 + quantum-inspired spin	186	0.308

```
            quantum_dimensions: Number of quantum dimensions to model (2=qubit, >2=qudit)
            entanglement_threshold: Threshold for quantum entanglement detection
        """
        # Load configuration
        self.config = self._load_config(config_path)

        # Initialize classical K9 processor
        self.k9_processor = K9SignalProcessor(
            sensitivity=self.config.get('sensitivity', a=1.8),
            use_gpu=use_gpu
        )

        # Initialize quantum processor
        self.quantum_processor = QuantumSpinSignalProcessor(
            num_spin_states=quantum_dimensions,
            coherence_threshold=self.config.get('coherence_threshold', 0.65),
            entanglement_sensitivity=entanglement_threshold
        )

        # Initialize celestial tracker
        self.celestial_tracker = CelestialK9Tracker(
            update_interval=self.config.get('update_interval', 1.0)
        )

        # Setup quantum-enhanced location mapping
        self.quantum_location_map = {}

        # Tracking quantum correlations across space
        self.spatial_entanglement_map = {}

        # Thread for continuous processing
        self.processing_thread = None
        self.running = False

        # Performance metrics
        self.metrics = {
            'signals_processed': 0,
            'quantum_enhanced_detections': 0,
            'entangled_signal_pairs': 0,
            'processing_time': 0.0
        }

    def _load_config(self, config_path: Optional[str]) -> Dict:
        """Load configuration from file or use defaults"""
        default_config = {
            'sensitivity': 1.8,
            'coherence_threshold': 0.65,
            'entanglement_threshold': 0.75,
            'update_interval': 1.0,
            'use_gpu': True,
            'frequency_bands': [
                [70e6, 200e6],    # VHF
                [400e6, 1e9],     # UHF
                [1e9, 2.4e9],     # L-band
                [2.4e9, 5.8e9]    # S-band
            ],
            'quantum_models': ['spin-1/2', 'spin-1'],
            'location_grid_resolution': 0.01,  # degrees
            'confidence_threshold': 0.75
        }

        if config_path and os.path.exists(config_path):
            try:
```

TABLE III
API-DRIVEN RUN: CALL SITES AND OUTPUTS.

Target origin	module
Location-map callable	quantum_location_map
Entanglement callable	spatial_entanglement_map
Signals processed	527
Map shape (rows×cols)	26×31
Links produced	16674

```
                with open(config_path, 'r') as f:
                    user_config = json.load(f)
                # Merge user config with defaults
                for key, value in user_config.items():
                    default_config[key] = value
            except Exception as e:
                print(f"Error loading config, using defaults: {e}")

        return default_config

    def start(self):
        """Start continuous signal processing"""
        if self.processing_thread and self.processing_thread.is_alive():
            print("Already running")
            return

        self.running = True
        self.processing_thread = threading.Thread(target=self._processing_loop)
        self.processing_thread.daemon = True
        self.processing_thread.start()

        print("Quantum Celestial K9 processing started")

    def stop(self):
        """Stop continuous processing"""
        self.running = False
        if self.processing_thread:
            self.processing_thread.join(timeout=2.0)
        print("Quantum Celestial K9 processing stopped")

    def _processing_loop(self):
        """Main processing loop"""
        while self.running:
            try:
                # Get data from celestial tracker
                celestial_data = self.celestial_tracker.get_latest_data()

                if celestial_data and 'signals' in celestial_data:
                    for signal_id, signal_data in celestial_data['signals'].items():
                        self._process_celestial_signal(signal_id, signal_data)

                # Sleep briefly to avoid CPU overuse
                time.sleep(0.01)

            except Exception as e:
                print(f"Error in quantum processing loop: {e}")
                time.sleep(1.0)

    def _process_celestial_signal(self, signal_id: str, signal_data: Dict):
        """Process an individual signal with quantum enhancement"""
        start_time = time.time()

        try:
            # Extract signal components
            freqs = np.array(signal_data.get('frequencies', []))
            amplitudes = np.array(signal_data.get('amplitudes', []))
            location = signal_data.get('location', {})

            if len(freqs) == 0 or len(amplitudes) == 0:
                return

            # Process with K9
            k9_results = self.k9_processor.process_signal(freqs, amplitudes)

            # Enhance with quantum processing
            quantum_results = integrate_with_k9_processor(k9_results, freqs, amplitudes)

            # Add spatial information
            self._add_quantum_spatial_information(signal_id, quantum_results, location)

            # Check for spatial entanglement
            entangled_signals = self._detect_spatial_entanglement(signal_id, quantum_results, location)

            # Update metrics
            self.metrics['signals_processed'] += 1
            if quantum_results['integrated_insights']['is_quantum_enhanced']:
                self.metrics['quantum_enhanced_detections'] += 1

            self.metrics['entangled_signal_pairs'] += len(entangled_signals)

            # Store enhanced results
            self._store_enhanced_results(signal_id, quantum_results, location, entangled_signals)

        except Exception as e:
            print(f"Error processing signal {signal_id}: {e}")

        finally:
            # Update processing time metric
            processing_time = time.time() - start_time
            self.metrics['processing_time'] = (
                0.9 * self.metrics['processing_time'] + 0.1 * processing_time
                if self.metrics['processing_time'] > 0 else processing_time
            )
```

```
    def _add_quantum_spatial_information(self, signal_id: str, quantum_results: Dict, location: Dict):
        """Add spatial information to quantum results"""
        if not location or 'lat' not in location or 'lon' not in location:
            return

        # Get location coordinates
        lat = location.get('lat', 0.0)
        lon = location.get('lon', 0.0)
        alt = location.get('alt', 0.0)

        # Create grid key
        resolution = self.config.get('location_grid_resolution', 0.01)
        grid_lat = round(lat / resolution) * resolution
        grid_lon = round(lon / resolution) * resolution
        grid_key = f"{grid_lat:.3f}_{grid_lon:.3f}"

        # Add to quantum location map
        if grid_key not in self.quantum_location_map:
            self.quantum_location_map[grid_key] = {
                'signals': [],
                'avg_coherence': 0.0,
                'avg_entanglement': 0.0,
                'quantum_density': 0.0
            }

        # Update grid information
        grid = self.quantum_location_map[grid_key]

        # Add signal if not already present
        if signal_id not in [s['id'] for s in grid['signals']]:
            # Add signal summary
            grid['signals'].append({
                'id': signal_id,
                'coherence': quantum_results['quantum_analysis']['quantum_coherence'],
                'entanglement': quantum_results['quantum_analysis']['entanglement']['entanglement_strength'],
                'bloch_vector': quantum_results['quantum_analysis']['quantum_tomography']['bloch_vector'],
                'timestamp': time.time()
            })

            # Limit number of stored signals
            if len(grid['signals']) > 5:
                # Remove oldest signal
                grid['signals'] = sorted(grid['signals'], key=lambda s: s['timestamp'], reverse=True)[:5]

        # Update averages
        if grid['signals']:
            grid['avg_coherence'] = sum(s['coherence'] for s in grid['signals']) / len(grid['signals'])
            grid['avg_entanglement'] = sum(s['entanglement'] for s in grid['signals']) / len(grid['signals'])
            grid['quantum_density'] = len(grid['signals']) * (grid['avg_coherence'] + grid['avg_entanglement']) / 2


    def _detect_spatial_entanglement(self, signal_id: str, quantum_results: Dict, location: Dict) -> List[Dict]:
        """Detect potential quantum entanglement between spatially separated signals"""
        if not location or 'lat' not in location or 'lon' not in location:
            return []

        entangled_signals = []
        current_coherence = quantum_results['quantum_analysis']['quantum_coherence']
        current_bloch = quantum_results['quantum_analysis']['quantum_tomography']['bloch_vector']

        # Only consider signals with significant quantum properties
        if current_coherence < 0.4:
            return []

        # Look for spatially separated signals with quantum correlations
        for grid_key, grid in self.quantum_location_map.items():
            # Skip the current grid
            current_grid = f"{round(location['lat']/self.config['location_grid_resolution'])*self.config['location_grid_resolution']:.3f}_" \
                           f"{round(location['lon']/self.config['location_grid_resolution'])*self.config['location_grid_resolution']:.3f}"
            if grid_key == current_grid:
                continue

            # Check signals in this grid
            for other_signal in grid['signals']:
                if other_signal['id'] == signal_id:
                    continue

                # Calculate quantum correlation
                other_bloch = other_signal['bloch_vector']

                # Calculate dot product between Bloch vectors (spin correlation)
                bloch_correlation = abs(np.dot(current_bloch, other_bloch))

                # Anti-correlation is also significant in quantum mechanics
                # (anti-aligned spins can be entangled)
                if bloch_correlation < 0.2:
                    bloch_correlation = 1.0 - bloch_correlation

                # Check if correlation exceeds threshold
                entanglement_threshold = self.config.get('entanglement_threshold', 0.75)
                if bloch_correlation > entanglement_threshold:
```

```
                    # Calculate the symmetry of coherences (entangled systems often have similar coherence)
                    coherence_symmetry = 1.0 - abs(current_coherence - other_signal['coherence'])

                    if coherence_symmetry > 0.7:
                        # Record the entanglement
                        entangled_signals.append({
                            'signal_id': other_signal['id'],
                            'grid_key': grid_key,
                            'correlation': bloch_correlation,
                            'coherence_symmetry': coherence_symmetry,
                            'entanglement_strength': bloch_correlation * coherence_symmetry
                        })

                        # Update the spatial entanglement map
                        entanglement_key = f"{signal_id}_{other_signal['id']}"
                        reverse_key = f"{other_signal['id']}_{signal_id}"

                        if entanglement_key not in self.spatial_entanglement_map and reverse_key not in self.spatial_entanglement_map:
                            self.spatial_entanglement_map[entanglement_key] = {
                                'signal1': signal_id,
                                'signal2': other_signal['id'],
                                'grid1': current_grid,
                                'grid2': grid_key,
                                'strength': bloch_correlation * coherence_symmetry,
                                'detected_time': time.time(),
                                'update_count': 1
                            }
                        elif entanglement_key in self.spatial_entanglement_map:
                            # Update existing entanglement
                            ent = self.spatial_entanglement_map[entanglement_key]
                            ent['strength'] = 0.7 * ent['strength'] + 0.3 * (bloch_correlation * coherence_symmetry)
                            ent['update_count'] += 1
                        elif reverse_key in self.spatial_entanglement_map:
                            # Update existing entanglement (reverse direction)
                            ent = self.spatial_entanglement_map[reverse_key]
                            ent['strength'] = 0.7 * ent['strength'] + 0.3 * (bloch_correlation * coherence_symmetry)
                            ent['update_count'] += 1

                # Clean up old entanglements
                self._cleanup_entanglement_map()

                return entangled_signals

            def _cleanup_entanglement_map(self):
                """Remove old entanglements from the map"""
                current_time = time.time()
                keys_to_remove = []

                for key, entanglement in self.spatial_entanglement_map.items():
                    # Remove entanglements older than 1 hour
                    if current_time - entanglement['detected_time'] > 3600:
                        keys_to_remove.append(key)

                for key in keys_to_remove:
                    del self.spatial_entanglement_map[key]

            def _store_enhanced_results(self, signal_id: str, quantum_results: Dict,
                                        location: Dict, entangled_signals: List[Dict]):
                """Store enhanced results for later retrieval"""
                # In a real implementation, this would store to a database
                # Here we'll just print summary information

                # Only log interesting signals to avoid console spam
                if quantum_results['integrated_insights']['is_quantum_enhanced'] or entangled_signals:
                    print(f"\nQuantum-enhanced signal {signal_id}:")

                    if location:
                        print(f"  Location: {location.get('lat', 'N/A'):.4f}, {location.get('lon', 'N/A'):.4f}")
```

```
            print(f"  Coherence: {quantum_results['quantum_analysis']['quantum_coherence']:.3f}")
            print(f"  Entanglement: {quantum_results['quantum_analysis']['entanglement']['entanglement_strength']:.3f}")
            print(f"  Quantum processing gain: {quantum_results['integrated_insights']['processing_gain']:.2f} dB")

            if entangled_signals:
                print(f"  Entangled with {len(entangled_signals)} other signals:")
                for i, ent in enumerate(entangled_signals[:2]):  # Show only first 2 to avoid spam
                    print(f"    [{i+1}] Signal {ent['signal_id']} (strength: {ent['entanglement_strength']:.3f})")
                if len(entangled_signals) > 2:
                    print(f"    ... and {len(entangled_signals)-2} more")

    def get_metrics(self) -> Dict:
        """Get current performance metrics"""
        return {
            **self.metrics,
            'active_entanglements': len(self.spatial_entanglement_map),
            'quantum_locations': len(self.quantum_location_map),
            'is_running': self.running and (self.processing_thread and self.processing_thread.is_alive())
        }

    def get_quantum_spatial_map(self) -> Dict:
        """Get the current quantum spatial information"""
        return {
            'grid_resolution': self.config.get('location_grid_resolution', 0.01),
            'total_locations': len(self.quantum_location_map),
            'entanglements': len(self.spatial_entanglement_map),
            'locations': [
                {
                    'grid_key': grid_key,
                    'quantum_density': grid['quantum_density'],
                    'signals': len(grid['signals']),
                    'avg_coherence': grid['avg_coherence'],
                    'avg_entanglement': grid['avg_entanglement']
                }
                for grid_key, grid in self.quantum_location_map.items()
            ],
            'entanglement_links': [
                {
                    'source': ent['signal1'],
                    'target': ent['signal2'],
                    'source_grid': ent['grid1'],
                    'target_grid': ent['grid2'],
                    'strength': ent['strength'],
                    'updates': ent['update_count']
                }
                for key, ent in self.spatial_entanglement_map.items()
            ]
        }

# Example usage demonstration
if __name__ == "__main__":
    # Create tracker with quantum enhancement
    quantum_tracker = QuantumCelestialK9()

    # Start tracking
    quantum_tracker.start()

    # In a real application, this would run continuously
    # For demo, just wait a short time
    try:
        print("Quantum-enhanced tracking started. Press Ctrl+C to stop...")
        for _ in range(10):
            time.sleep(1)
            metrics = quantum_tracker.get_metrics()
            print(f"Processed: {metrics['signals_processed']}, "
                  f"Quantum enhanced: {metrics['quantum_enhanced_detections']}, "
                  f"Entanglements: {metrics['entangled_signal_pairs']}")

        # Get quantum spatial map
        spatial_map = quantum_tracker.get_quantum_spatial_map()
        print(f"\nQuantum spatial map: {len(spatial_map['locations'])} locations with "
              f"{len(spatial_map['entanglement_links'])} entanglement links")

    except KeyboardInterrupt:
        print("Stopping...")
    finally:
        # Stop tracking when done
        quantum_tracker.stop()
        print("Quantum-enhanced tracking stopped")

# --- Minimal API shims for paper adapter (simulation-friendly) ---
def quantum_location_map(signals, res=0.02):
    import numpy as np
    xs = np.arange(-0.30, 0.31, res)
    ys = np.arange( 0.00, 0.51, res)
    Z  = np.full((len(ys), len(xs)), np.nan)
    grid = {}
    for s in signals:
        key = (round(s["lon"]/res)*res, round(s["lat"]/res)*res)
        g = grid.setdefault(key, {"n":0,"coh":0.0})
        g["n"] += 1; g["coh"] += float(s.get("coherence",0.0))
    for (gx,gy), v in grid.items():
        ix = np.argmin(np.abs(xs - gx)); iy = np.argmin(np.abs(ys - gy))
        Z[iy, ix] = v["coh"]/max(1,v["n"])
    return {"map": Z, "extent": [xs.min(), xs.max(), ys.min(), ys.max()]}

def spatial_entanglement_map(signals, corr_thr=0.82, sep_deg=0.04):
```

```
    import numpy as np
    idx = [i for i,s in enumerate(signals) if float
        (s.get("coherence",0.0))>=0.4]
    links=[]
    for i in range(len(idx)):
        for j in range(i+1, len(idx)):
            a,b = idx[i], idx[j]
            bi = np.array(signals[a].get("bloch
                ",[1,0,0]), dtype=float)
            bj = np.array(signals[b].get("bloch
                ",[1,0,0]), dtype=float)
            dot = float(abs(bi@bj)); corr = max(dot
                , 1.0-dot)
            dx = signals[a]["lon"]-signals[b]["lon
                "]; dy = signals[a]["lat"]-signals
                [b]["lat"]
            if corr>=corr_thr and (dx*dx+dy*dy)>=
                sep_deg*sep_deg:
                links.append({"i":int(a),"j":int(b)
                    ,"w":corr})
    return {"links": links}
```

VI. Conclusion

Quantum-inspired coherence and Bloch-direction features can be layered onto a classical detector to surface weak signals and summarize non-local relations. In simulation, the augmentation yields a modest detection gain and a compact spatial link topology. Future work: stress tests across SNR/channel models and integration with real telemetry streams.