oLogic – Fragility–Emergence Layer (FEL) Source Code

Author: Evgeny Ovchinnikov
Independent Researcher, Moscow
savoir-vivre@mail.ru

PRIOR ART AND LEGAL PROTECTION STATEMENT

This document contains the complete, executable source code for the Fragility–Emergence Layer (FEL) circuit, the SPSA optimizer setup, and all experimental configurations described in the oLogic architecture paper (Ovchinnikov, 2026). It is a standalone Prior Art publication deposited on Zenodo (CERN, Switzerland) with a persistent DOI under the MIT License (MIT).

By this publication, the Author:

1. Fixes the date of creation and public disclosure of this software.
2. Asserts his exclusive authorship of this code under the Berne Convention.
3. Establishes this document as a defensive publication that destroys the novelty of any patent application seeking to claim the FEL circuit, its specific gate sequences, the Fragility and Emergence blocks, or the variational optimization protocol for quantum search under depolarizing noise.
4. Grants to any person the rights, free of charge, to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, subject to the condition that the above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
5. Provides the software "AS IS", without warranty of any kind. In no event shall the Author be liable for any claim, damages, or other liability arising from its use.
6. Affirms that the Author is of sound mind, acting with full legal capacity, and that this publication constitutes a deliberate, conscious creative act.
7. States that all persons who have made a creative contribution to this work are listed as co-authors; any other discussions were consultative in nature.

FUNCTIONAL PRIORITY STATEMENTS

The functional priority statements from the main article (oLogic_main_article.pdf) apply fully to this code. Any implementation, in any programming language or hardware description language, that realizes the functional descriptions set forth therein infringes the priority established by this publication.

Complete FEL Source Code

```
import numpy as np

from qiskit import QuantumCircuit, transpile

from qiskit_aer import AerSimulator

from qiskit_aer.noise import NoiseModel, depolarizing_error



# ============================================================

# 1. Oracle for 4 palindromic strings: 0110, 1001, 1111, 0000

# ============================================================

def palindrome_oracle():

    """Creates a quantum circuit that marks the 4 palindromic 4-bit strings."""

    qc = QuantumCircuit(5)  # 4 work qubits + 1 ancilla

    # ancilla in |-> state

    qc.x(4)

    qc.h(4)
```

```
# Mark 0110: flip qubits 0 and 3 to turn 0110 into 1111

qc.x(0)

qc.x(3)

qc.h(3)

qc.mcx([0, 1, 2], 3)  # multi-controlled X on qubit 3 with controls 0,1,2

qc.h(3)

qc.x(0)

qc.x(3)



# Mark 1001: flip qubits 1 and 2 to turn 1001 into 1111

qc.x(1)

qc.x(2)

qc.h(3)

qc.mcx([0, 1, 2], 3)

qc.h(3)

qc.x(1)

qc.x(2)



# Mark 1111: already matches
```

```
    qc.h(3)
    qc.mcx([0, 1, 2], 3)
    qc.h(3)

    # Mark 0000: flip all work qubits to turn 0000 into 1111
    qc.x([0, 1, 2, 3])
    qc.h(3)
    qc.mcx([0, 1, 2], 3)
    qc.h(3)
    qc.x([0, 1, 2, 3])

    # ancilla back to |0>
    qc.h(4)
    qc.x(4)
    return qc

# ==============================
# 2. Grover diffusion operator
# ==============================
```

```
def grover_diffusion(n_qubits=4):
    """Standard Grover diffusion operator for n_qubits."""
    qc = QuantumCircuit(n_qubits)
    qc.h(range(n_qubits))
    qc.x(range(n_qubits))
    qc.h(n_qubits - 1)
    qc.mcx(list(range(n_qubits - 1)), n_qubits - 1)
    qc.h(n_qubits - 1)
    qc.x(range(n_qubits))
    qc.h(range(n_qubits))
    return qc


# ========================================
# 3. Build one iteration of the FEL circuit
# ========================================
def fel_iteration(theta):
    """
    One iteration of the FEL.
    Uses 4 work qubits (0-3) and 3 ancilla qubits (4,5,6).
    """
```

```
n_work = 4

n_anc = 3

total = n_work + n_anc

qc = QuantumCircuit(total)



# --- Oracle ---

qc.append(palindrome_oracle(), range(5))  # oracle uses qubits 0-4



# --- Grover diffusion ---

qc.append(grover_diffusion(n_work), range(n_work))



# --- Fragility block ---

# rotate ancillas by theta

for i in range(n_anc):

    qc.ry(theta, n_work + i)

# CNOT from each ancilla to a distinct work qubit

for i in range(n_anc):

    qc.cx(n_work + i, i)

# CNOTs between work qubits
```

```
for i in range(n_work):
    for j in range(i + 1, n_work):
        qc.cx(i, j)


# --- Emergence block ---
# rotate ancillas again
for i in range(n_anc):
    qc.ry(theta, n_work + i)
# CNOTs from ancillas to work qubits
for i in range(n_anc):
    qc.cx(n_work + i, (i + 1) % n_work)
# cyclic shift among work qubits
for i in range(n_work - 1):
    qc.swap(i, i + 1)
# full pairwise CNOTs among work qubits
for i in range(n_work):
    for j in range(i + 1, n_work):
        qc.cx(i, j)


return qc
```

```
# ====================================

# 4. Build full FEL with K iterations

# ====================================

def build_fel_circuit(theta, iterations=3):

    """Full FEL circuit with K iterations."""

    n_work = 4

    n_anc = 3

    total = n_work + n_anc

    qc = QuantumCircuit(total, n_work)  # measure work qubits



    for _ in range(iterations):

        qc.append(fel_iteration(theta), range(total))



    qc.measure(range(n_work), range(n_work))

    return qc



# ============================

# 5. Standard Grover (K=1)
```

```
# ============================

def build_grover_k1():

    """Standard Grover search with 1 iteration for 4 qubits, 4 targets."""

    qc = QuantumCircuit(5, 4)

    qc.append(palindrome_oracle(), range(5))

    qc.append(grover_diffusion(4), range(4))

    qc.measure(range(4), range(4))

    return qc



# ====================

# 6. Noise model

# ====================

def get_noise_model(p=0.05):

    """Depolarizing noise model with single-qubit error prob p."""

    noise_model = NoiseModel()

    error = depolarizing_error(p, 1)

    noise_model.add_all_qubit_quantum_error(error, ['u1', 'u2', 'u3', 'rx', 'ry', 'rz

    return noise_model



# ====================
```

```
# 7. Simulation runner
# ====================
def run_circuit(qc, shots=100000, noise_model=None):
    """Executes a circuit and returns counts."""
    simulator = AerSimulator()
    if noise_model:
        circ = transpile(qc, simulator)
        job = simulator.run(circ, shots=shots, noise_model=noise_model)
    else:
        circ = transpile(qc, simulator)
        job = simulator.run(circ, shots=shots)
    return job.result().get_counts()


# ======================================
# 8. Success probability calculator
# ======================================
TARGETS = {'0110', '1001', '1111', '0000'}


def success_probability(counts):
```

```
    """Fraction of shots yielding any target string."""
    total = sum(counts.values())
    hits = sum(counts.get(t, 0) for t in TARGETS)
    return hits / total


# ====================================
# 9. SPSA optimizer (minimal version)
# ====================================
def spsa_minimize(cost_func, initial_theta, max_iter=100, a=0.1, c=0.01):
    """
    Simple SPSA optimizer.
    Returns optimal theta and history of cost values.
    """
    theta = initial_theta
    history = []
    for k in range(1, max_iter + 1):
        ak = a / (k + 1) ** 0.602
        ck = c / k ** 0.101
        delta = np.random.choice([-1, 1])
        theta_plus = theta + ck * delta
```

```
        theta_minus = theta - ck * delta
        cost_plus = cost_func(theta_plus)
        cost_minus = cost_func(theta_minus)
        grad = (cost_plus - cost_minus) / (2 * ck * delta)
        theta = theta - ak * grad
        theta = np.clip(theta, 0.0, np.pi)
        history.append(cost_func(theta))
    return theta, history


# ==============================
# 10. Main experiment runner
# ==============================
def run_experiment():
    noise_model = get_noise_model(p=0.05)
    shots = 100000

    print("Running Grover K=1...")
    qc_grover = build_grover_k1()
    counts_grover = run_circuit(qc_grover, shots, noise_model)
```

```
    succ_grover = success_probability(counts_grover)
    print(f"Grover K=1 success: {succ_grover:.4f}")

    def cost(theta):
        qc = build_fel_circuit(theta, iterations=3)
        counts = run_circuit(qc, shots=shots, noise_model=noise_model)
        return -success_probability(counts)  # minimize negative success

    print("Running SPSA optimization for FEL...")
    opt_theta, history = spsa_minimize(cost, 0.5, max_iter=100)

    qc_fel = build_fel_circuit(opt_theta, iterations=3)
    counts_fel = run_circuit(qc_fel, shots, noise_model)
    succ_fel = success_probability(counts_fel)
    print(f"FEL optimal theta: {opt_theta:.4f}")
    print(f"FEL success: {succ_fel:.4f}")
    print(f"Delta (FEL - Grover): {succ_fel - succ_grover:.4f}")
if __name__ == "__main__":
    run_experiment()
```