Qiskit Runtime Lab - Complete Guide

Core Concepts & Implementation

1. Environment Setup and Prerequisites

Conceptual Background:

- Lab consists of seven progressive exercises focused on Qiskit Runtime
- Each exercise builds upon previous concepts
- Requires quantum computing knowledge and Python experience
- IBM Quantum account with API token needed

Implementation Setup:

```
# Install required packages

%pip install qiskit[visualization]

%pip install qiskit_aer

%pip install qiskit_ibm_runtime

%pip install matplotlib

%pip install pylatexenc

%pip install qiskit-transpiler-service

%pip install
git+https://github.com/qiskit-community/Quantum-Challenge-Grade
r.git


# Core imports

import numpy as np

from typing import List, Callable

from scipy.optimize import minimize

import matplotlib.pyplot as plt
```

```python
# Qiskit imports

from qiskit import QuantumCircuit

from qiskit.quantum_info import Statevector, Operator,
SparsePauliOp

from qiskit.primitives import StatevectorSampler

from qiskit.circuit.library import TwoLocal

from qiskit.transpiler.preset_passmanagers import
generate_preset_pass_manager

from qiskit.visualization import plot_histogram

from qiskit_ibm_runtime import Session, EstimatorV2 as
Estimator

from qiskit_aer import AerSimulator
```

Programming Decisions:

- Complete package installation upfront
- Structured imports for clarity
- Essential modules grouped by functionality

2. Bell State Circuit Creation (Exercise 1)

Quantum Concepts:

- Bell states are fundamental entangled quantum states
- $|\psi^-\rangle$ state demonstrates quantum entanglement
- Requires precise gate sequence for state preparation

Implementation:

```python
# Create Bell state circuit

qc = QuantumCircuit(2)

qc.h(0)  # Hadamard gate

qc.cx(0,1)  # CNOT gate
```

```python
qc.z(1)  # Z gate

qc.x(1)  # X gate

qc.measure_all()

qc.draw('mpl')
```

Programming Decisions:

- Sequential gate application for clarity
- Clear qubit assignment structure
- Visual circuit verification

3. StatevectorSampler Usage (Exercise 2)

Quantum Concepts: ● Quantum state sampling requires multiple measurements ● StatevectorSampler provides efficient sampling mechanism ● Results analysis through histogram visualization

Implementation:

```python
# Create sampler instance

sampler = StatevectorSampler()


# Prepare and run circuit

pub = (qc)

job_sampler = sampler.run([pub], shots=10000)


# Get results

result_sampler = job_sampler.result()

counts_sampler = result_sampler[0].data.meas.get_counts()


# Visualize
```

```
print(counts_sampler)

plot_histogram(counts_sampler)
```

Programming Decisions:

- High shot count for statistical significance
- Structured result collection
- Immediate visualization for verification

4. W-State Circuit Implementation (Exercise 3)

Quantum Concepts: ● W-states provide equal superposition of specific states ● Used to represent three-state system (chocolates) ● Requires precise rotation and entanglement operations

Implementation:

```
# Create W-state circuit

qc = QuantumCircuit(3)

qc.ry(1.91063324, 0)  # Specific rotation

qc.ch(0,1)  # Controlled Hadamard

qc.cx(1,2)  # First CNOT

qc.cx(0,1)  # Second CNOT

qc.x(0)  # Final X gate

qc.measure_all()

qc.draw('mpl')


# Sample and visualize

sampler = StatevectorSampler()

pub = (qc)

job_sampler = sampler.run([pub], shots=10000)
```

```
result_sampler = job_sampler.result()

counts_sampler = result_sampler[0].data.meas.get_counts()

plot_histogram(counts_sampler)
```

Programming Decisions:

- Precise rotation angle for equal distribution
- Clear state mapping (001, 010, 100)
- Immediate result verification

5. Parameterized Circuit Development (Exercise 4)

Quantum Concepts:

- TwoLocal circuits for variational algorithms
- Parameterized gates for optimization
- Entanglement patterns for quantum correlations

Implementation:

```
# TwoLocal circuit parameters

num_qubits = 3

rotation_blocks = ['ry', 'rz']

entanglement_blocks = 'cz'

entanglement = 'full'


# Create ansatz

ansatz = TwoLocal(

    num_qubits,

    rotation_blocks,

    entanglement_blocks,

    entanglement=entanglement,
```

```
    reps=1,
    insert_barriers=True
)
ansatz.decompose().draw('mpl')
```

Programming Decisions:

- Clear parameter definition
- Structured circuit construction
- Visual circuit verification

6. Circuit Transpilation (Exercise 5)

Quantum Concepts:

- ISA compliance for backend execution
- Optimization levels affect circuit efficiency
- Backend-specific gate set conversion

Implementation:

```
from qiskit_ibm_runtime.fake_provider import FakeSherbrooke
from qiskit import transpile

# Transpile circuit
isa_circuit = transpile(
    circuits=ansatz,
    backend=FakeSherbrooke(),
    optimization_level=2,
    seed_transpiler=0
)
```

```python
isa_circuit.draw('mpl', idle_wires=False)


# Define Hamiltonian

hamilton_isa = pauli_op.apply_layout(layout=isa_circuit.layout)
```

Programming Decisions:

- Fixed seed for reproducibility
- Optimal visualization settings
- Clear layout application

7. VQE Cost Function (Exercise 6)

Quantum Concepts:

- Energy expectation calculation
- Parameter optimization
- Progress tracking for convergence

Implementation:

```python
def cost_func(params, ansatz, hamiltonian, estimator,
callback_dict):

    """Return estimate of energy from estimator"""

    # Run estimator

    job = estimator.run([(ansatz, hamiltonian, params)])

    result = job.result()


    # Get energy value

    energy = result[0].data.evs


    # Update callback tracking

    callback_dict["iters"] += 1
```

```
        callback_dict["prev_vector"] = params
        callback_dict["cost_history"].append(energy)

        print(energy)
        return energy, result

    callback_dict = {
        "prev_vector": None,
        "iters": 0,
        "cost_history": [],
    }
```

Programming Decisions:

- Comprehensive tracking system
- Clear parameter handling
- Organized result structure

8. Qiskit Runtime V2 Execution (Exercise 7)

Quantum Concepts:

- V2 primitives for improved performance
- Local testing methodology
- Optimization convergence monitoring

Implementation:

```
    # Initialize backend
    backend = AerSimulator()

    # Create estimator
```

```
estimator = Estimator(backend)

# Modified cost function for scipy
def cost_func_2(params, *args):
    energy_and_result = cost_func(params, *args)
    return energy_and_result[0]

# Initial parameters
x0 = 2 * np.pi * np.random.random(num_params)

# Run optimization
result = minimize(
    cost_func_2,
    x0,
    args=(isa_circuit, hamiltonian_isa, estimator,
callback_dict),
    method="cobyla",
    options={'maxiter': 30}
)

# Plot convergence
fig, ax = plt.subplots()
plt.plot(range(callback_dict["iters"]),
callback_dict["cost_history"])
plt.xlabel("Iteration")
plt.ylabel("Cost")
```

```
plt.draw()
```

Programming Decisions:

- Local testing configuration
- Structured optimization setup
- Clear visualization of results

Troubleshooting and Best Practices

Common Issues:

1. Environment Setup: ○ Solution: Install packages sequentially ○ Reason: Dependencies need proper ordering
2. Circuit Execution: ○ Solution: Verify gate compatibility ○ Reason: Backend limitations
3. Optimization: ○ Solution: Adjust parameters and iterations ○ Reason: Convergence sensitivity

Best Practices:

1. Regular circuit verification
2. Monitor optimization progress
3. Start with local testing
4. Document parameter choices
5. Use appropriate shot counts
6. Maintain consistent naming conventions

This guide provides a complete workflow from setup to execution, with each section building upon the previous ones. Follow the implementations sequentially and verify results at each step