Strawberry fields + pennylane + qiskit

Wednesday, July 30, 2025
4:18 PM

🍓 Strawberry Fields (SF)

Purpose: Quantum photonics simulation and programming using **continuous-variable (CV) quantum computing**.

🔧 Developed by:

Xanadu

🌟 Features:

- Built for **CV quantum computing** (qumodes instead of qubits).
- Uses **Blackbird** language for circuit definition.
- Can simulate **Gaussian and Fock backends** (classical simulators for photonic circuits).
- Supports **quantum optical gates**, **interferometers**, and **measurement operations** like `homodyne`, `heterodyne`, `Fock`.

🧪 Example: Simple quantum photonic circuit

python

```python
import strawberryfields as sf
from strawberryfields.ops import Sgate, Dgate, MeasureFock

eng = sf.Engine("fock", backend_options={"cutoff_dim": 5})
prog = sf.Program(1)

with prog.context as q:
    Dgate(0.5) | q[0]
    Sgate(0.3) | q[0]
    MeasureFock() | q[0]

result = eng.run(prog)
print(result.samples)
```

□ PennyLane

Purpose: Framework for **hybrid quantum-classical computing**, especially for **machine learning**, compatible with many backends.

🔧 Developed by:

Also by Xanadu

🌟 Features:

- Focused on **quantum differentiable programming**.
- Integrates well with **PyTorch**, **TensorFlow**, **JAX**.
- Supports **both qubit and continuous-variable models.**
- Can **interface with Strawberry Fields**, as well as IBM Qiskit, Rigetti Forest, etc.

🧠 Used for:

- Variational Quantum Circuits (VQCs)
- Quantum Neural Networks (QNNs)
- Quantum optimization & quantum chemistry
- Quantum machine learning

🔗 Using Strawberry Fields inside PennyLane

PennyLane has a plugin for Strawberry Fields, so you can define a photonic circuit and optimize it with gradient descent!

📌 Example: Photonic circuit with PennyLane

```python
import pennylane as qml
from pennylane import numpy as np

dev = qml.device("strawberryfields.fock", wires=1, cutoff_dim=10)

@qml.qnode(dev)
def circuit(x):
    qml.Displacement(x, 0.0, wires=0)
    qml.Squeezing(0.1, 0.0, wires=0)
    return qml.expval(qml.NumberOperator(0))
```

```
x = np.array(0.1, requires_grad=True)
print(circuit(x))  # Output expectation value

grad = qml.grad(circuit)(x)
print(grad)        # Gradient
```

🧭 Summary Table

Feature	Strawberry Fields	PennyLane
Type	Photonic circuit simulator	Hybrid QML framework
Focus	CV quantum computing	Differentiable programming (QML)
Supports qubits?	No	Yes
Supports CV (qumodes)?	Yes	Yes
Integration	Native	Via plugin (e.g. `strawberryfields.fock`)
Optimization support	Limited	Full (autograd, pytorch, jax etc.)

🧪 Project: Simulating Continuous-Variable (CV) Quantum Teleportation

---

🔭 What You'll Learn:

- How to simulate a **CV teleportation protocol**
- How to use **Strawberry Fields**' Fock backend
- Concepts like **entangled squeezed states**, **Bell measurements**, and **displacement corrections**

🧰 Requirements

Install Strawberry Fields (if not done yet):

```bash
pip install strawberryfields
```

🧠 Theoretical Outline

In CV teleportation:

1. An entangled **two-mode squeezed vacuum** state is shared between Alice and Bob.
2. Alice wants to teleport an unknown state (e.g., a coherent state).
3. Alice performs **Bell-type homodyne measurements** on her two modes.
4. Bob applies **displacement operations** based on Alice's measurement results to reconstruct the state.

🧬 Circuit Setup

- Mode 0: The input quantum state (e.g., a coherent state).
- Mode 1: One half of the entangled resource (Alice's side).
- Mode 2: The other half of the entangled resource (Bob's side).

📜 Code Walkthrough

python

```python
import strawberryfields as sf
from strawberryfields.ops import *

# Set up a 3-mode Fock backend
eng = sf.Engine("fock", backend_options={"cutoff_dim": 10})
prog = sf.Program(3)

# Coherent state amplitude to be teleported
alpha = 0.5
r = 1.0  # Squeezing parameter
```

```python
# Coherent state amplitude to be teleported
alpha = 0.5
r = 1.0  # Squeezing parameter

with prog.context as q:
    # Step 1: Prepare the input coherent state
    Coherent(alpha, 0) | q[0]
```

```python
# Step 2: Create the entangled two-mode squeezed vacuum (modes 1 and 2)
Squeezed(r)          | q[1]
Squeezed(-r)         | q[2]
BSgate()             | (q[1], q[2])  # 50:50 beam splitter

# Step 3: Alice performs Bell measurement (homodyne) on modes 0 and 1
BSgate()             | (q[0], q[1])
MeasureHomodyne(0.0) | q[0]  # x-quadrature
MeasureHomodyne(0.0) | q[1]  # p-quadrature
```

```python
# Run the engine
result = eng.run(prog)

# Retrieve measurement results (classical communication)
x, p = result.samples

# Step 4: Bob applies corrective displacement on mode 2
# (This is done in a second program using conditional operations)

prog2 = sf.Program(1)
with prog2.context as q2:
    # Apply correction
    Dgate(x + 1j*p) | q2[0]
```

```python
# Run correction on Bob's mode (2)
state = eng.run(prog2, args={"q2": result.state.reduced_dm(2)}).state

# Visualize the teleported state (optional)
import matplotlib.pyplot as plt
state.plot_wigner(0)
plt.title("Wigner function of the teleported state")
plt.show()
```

📊 Expected Outcome

If teleportation works well:

- The final state on Bob's side (mode 2) should resemble the original coherent state.
- The **Wigner function** will show a Gaussian peak centered near `alpha`.

🔄 Optional Enhancements

- Try different values of `alpha` and `r`.
- Use **Fidelity calculation** to check teleportation quality:

```python
from strawberryfields.utils import fidelity

fidel = fidelity(state.reduced_dm(0), sf.fock_coherent_state(alpha, cutoff_dim=10))
print(f"Fidelity with original state: {fidel:.4f}")
```

- Animate Wigner functions before and after teleportation.

Absolutely, Damodar! Here's a **comprehensive Python program** that **demonstrates most of the key commands and operations** in the **Strawberry Fields SDK**, including:

- State preparation (vacuum, coherent, squeezed, thermal)
- Gaussian and non-Gaussian gates
- Interferometers
- Measurements
- Backend settings (Fock and Gaussian)
- Accessing simulation results

🧪 Python Program: Full Demonstration of Strawberry Fields SDK

python Copy Edit

```python
import strawberryfields as sf
from strawberryfields.ops import *
from strawberryfields.utils import random_interferometer
import numpy as np
import matplotlib.pyplot as plt
```

```python
# Setup: use 4 modes and Fock backend
cutoff = 10
eng = sf.Engine("fock", backend_options={"cutoff_dim": cutoff})
prog = sf.Program(4)

with prog.context as q:
    # --- 1. State Preparation ---
    Vacuum()          | q[0]                    # Explicit vacuum
    Coherent(0.5, 0) | q[1]                    # Coherent state
    Squeezed(1.0)     | q[2]                    # Squeezed vacuum
    Thermal(1.0)      | q[3]                    # Thermal state
```

```python
   # --- 2. Gaussian Gates ---
   Dgate(0.3, np.pi/4)      | q[0]              # Displacement
   Sgate(0.4, np.pi/2)      | q[1]              # Squeezing
   Rgate(np.pi/3)           | q[2]              # Rotation
   BSgate(np.pi/4, np.pi/2)| (q[0], q[1])      # Beamsplitter
   MZgate(np.pi/4, np.pi/4)| (q[1], q[2])      # Mach-Zehnder interferometer
   Interferometer(random_interferometer(3)) | [q[1], q[2], q[3]]

   # --- 3. Non-Gaussian Gates ---
   Kgate(0.1)        | q[0]                     # Kerr interaction
   Vgate(0.2)        | q[1]                     # Cubic phase gate
   Zgate(0.3)        | q[2]                     # Quadratic phase gate
```

```python
   # --- 4. Non-Gaussian State Preparation ---
   Catstate(1.0, 0)  | q[0]                     # Schrödinger cat state
   Fock(2)           | q[3]                     # Fock state |2⟩

   # --- 5. Measurements ---
   # Note: comment out one type at a time, as measurements collapse the state
   MeasureFock()      | q[0]
   MeasureHomodyne(0.0) | q[1]                  # x quadrature
   MeasureHomodyne(np.pi/2) | q[2]              # p quadrature
   MeasureHeterodyne() | q[3]
```

```python
# Run simulation
result = eng.run(prog)

# Access samples
print("Measurement samples:\n", result.samples)

# Access state object
state = result.state

# Print expectation value of photon number in mode 2
mean_n2 = state.mean_photon(2)
print(f"⟨n⟩ in mode 2: {mean_n2:.4f}")
```

```python
# Plot Wigner function for mode 2 (if not measured)
if not isinstance(result.samples[2], int):
    state.plot_wigner(2)
    plt.title("Wigner function of mode 2")
    plt.show()
```

✅ Covered Commands and Features

Category	Examples Used
States	`Vacuum`, `Coherent`, `Squeezed`, `Thermal`, `Fock`, `Catstate`
Gaussian Gates	`Dgate`, `Sgate`, `Rgate`, `BSgate`, `MZgate`, `Interferometer`
Non-Gaussian Gates	`Kgate`, `Vgate`, `Zgate`
Measurements	`MeasureFock`, `MeasureHomodyne`, `MeasureHeterodyne`
Utilities	`random_interferometer`, `state.mean_photon`, `state.plot_wigner`

🧠 PennyLane SDK Command Categories

Category	Commands/Functions	Use Case
Devices	`qml.device()`	Connect to backend (default.qubit, strawberryfields.fock, etc.)
QNodes	`@qml.qnode`, `qml.QNode`	Define quantum circuits
Operations	`qml.RX`, `qml.CNOT`, `qml.Squeezing`, `qml.Displacement`, etc.	Apply gates
Measurements	`qml.expval()`, `qml.sample()`, `qml.var()`, `qml.probs()`	Extract quantum results

Gradients	`qml.grad`, `qml.jacobian`, `qml.qml.metric_tensor`	Compute gradients
Templates	`qml.templates.*`	Use pre-built circuits (e.g., QAOA, AmplitudeEmbedding)
Classical Functions	`qml.math.*`, `qml.numpy.*`	JAX-like math support
Transforms	`qml.transforms.*`	Modify circuits (e.g., layerize, defer measurements)
Compilation	`qml.transforms.compile`, `qml.optimize()`	Optimize circuit layout
Optimizers	`qml.GradientDescentOptimizer`, `qml.AdamOptimizer`	Train parameters
QChem	`qml.qchem.*`	Quantum chemistry module
ML Integration	`qml.qnn.TorchLayer`, `qml.qnn.KerasLayer`	For hybrid ML models

🧪 Full Python Program Demonstrating All Major Commands

This single program covers devices, gates, measurements, optimization, templates, gradients, and visualization.

python Co

```python
import pennylane as qml
from pennylane import numpy as np
import matplotlib.pyplot as plt
```

```python
# Step 1: Device setup
dev = qml.device("default.qubit", wires=2, shots=None)

# Step 2: Define a parameterized quantum circuit
@qml.qnode(dev)
def circuit(params):
    # Rotation gates
    qml.RX(params[0], wires=0)
    qml.RY(params[1], wires=1)

    # Entanglement
    qml.CNOT(wires=[0, 1])

    # Return measurements
    return [qml.expval(qml.PauliZ(0)), qml.probs(wires=[0, 1])]
```

```
# Step 3: Evaluate circuit
params = np.array([0.1, 0.2], requires_grad=True)
out = circuit(params)
print("Expectations and probabilities:", out)

# Step 4: Compute gradient
grad_fn = qml.grad(circuit)
gradient = grad_fn(params)
print("Gradient:", gradient)
```

```
# Step 5: Optimization loop
opt = qml.GradientDescentOptimizer(stepsize=0.2)
for i in range(10):
    params = opt.step(lambda p: circuit(p)[0], params)
    print(f"Step {i+1}, params = {params}, cost = {circuit(params)[0]}")

# Step 6: Using templates (Amplitude Embedding + StronglyEntanglingLayers)
dev2 = qml.device("default.qubit", wires=3)
@qml.qnode(dev2)
def template_circuit(x, weights):
    qml.templates.AmplitudeEmbedding(x, wires=[0, 1, 2], normalize=True)
    qml.templates.StronglyEntanglingLayers(weights, wires=[0, 1, 2])
    return qml.expval(qml.PauliZ(0))
```

```
features = np.array([1, 0, 0, 0, 0, 0, 0, 0], requires_grad=True)
weights = np.random.randn(2, 3, 3)

print("Template circuit output:", template_circuit(features, weights))
```

✅ What This Covers

Feature	Demonstrated
Device setup	✅ `qml.device()`
QNode declaration	✅ `@qml.qnode`
Gates	✅ `RX`, `RY`, `CNOT`
Measurements	✅ `expval()`, `probs()`
Gradients	✅ `qml.grad`
Optimizers	✅ `GradientDescentOptimizer`

Templates	✅ `AmplitudeEmbedding`, `StronglyEntanglingLayers`
Classical Interface	✅ `qml.numpy`
ML-ready circuit	✅ Parameters with `requires_grad=True`

📚 Bonus: List of Common Gates & Operations in PennyLane

🔶 Qubit Gates:

- `qml.RX`, `RY`, `RZ`, `Rot`
- `qml.PauliX`, `PauliY`, `PauliZ`, `Hadamard`
- `qml.CNOT`, `CZ`, `SWAP`, `Toffoli`, `CSWAP`

🔶 CV Gates:

- `qml.Displacement`, `Squeezing`, `Beamsplitter`, `Kerr`

🔶 Measurements:

- `expval`, `var`, `probs`, `sample`

🔶 Math/Transform/Optimize:

- `qml.math.exp`, `qml.transforms.merge_rotations`, `qml.compile`
- `qml.GradientDescentOptimizer`, `qml.AdamOptimizer`, `qml.QNGOptimizer`

🧠 Qiskit SDK Command Categories

Category	Commands	Use Case
Core Modules	`QuantumCircuit`, `Aer`, `execute`	Define & simulate quantum circuits
Gates	`h`, `x`, `cx`, `rx`, `ry`, `rz`, `u3`	Apply quantum gates
Measurements	`measure_all`, `measure`, `save_statevector`, etc.	Extract classical data
Simulation	`AerSimulator`, `qasm_simulator`, `statevector_simulator`	Simulate circuits
Visualization	`circuit.draw()`, `plot_histogram`, `plot_bloch_vector`	Visualize circuits and results

Transpilation	`transpile()`, `assemble()`	Optimize and compile circuits
Execution	`execute()`	Run circuits on simulator or real hardware
Results	`job.result()`, `result.get_counts()`	Analyze outputs
Quantum Info	`Statevector`, `Operator`, `Pauli`, `random_unitary()`	Represent and manipulate states/operators
Machine Learning & Chemistry	`qiskit_machine_learning`, `qiskit_nature`	Specialized domains
Hardware Access	`IBMQ`, `IBMQBackend`, `IBMQ.save_account()`	Connect to real devices

🧪 Python Program: All Major Qiskit Commands in Action

python

```python
from qiskit import QuantumCircuit, Aer, transpile, assemble, execute
from qiskit.visualization import plot_histogram
from qiskit.quantum_info import Statevector, Pauli
import matplotlib.pyplot as plt

# Step 1: Create a quantum circuit with 2 qubits and 2 classical bits
qc = QuantumCircuit(2, 2)
```

```python
# Step 2: Apply basic gates
qc.h(0)          # Hadamard on qubit 0
qc.cx(0, 1)      # CNOT from qubit 0 to 1
qc.rx(0.5, 0)
qc.ry(1.0, 1)
qc.barrier()

# Step 3: Measurement
qc.measure([0, 1], [0, 1])

# Step 4: Draw circuit
qc.draw('mpl')
plt.title("Quantum Circuit")
plt.show()
```

```python
# Step 5: Simulate
sim = Aer.get_backend('qasm_simulator')
compiled = transpile(qc, sim)
job = execute(compiled, backend=sim, shots=1024)
result = job.result()

# Step 6: Analyze results
counts = result.get_counts()
print("Measurement results:", counts)
plot_histogram(counts)
plt.show()
```

```python
# Step 7: Statevector simulation (before measurement)
qc_sv = QuantumCircuit(2)
qc_sv.h(0)
qc_sv.cx(0, 1)
state = Statevector.from_instruction(qc_sv)
print("Statevector:", state)

# Step 8: Apply an operator (Pauli Z ⊗ I)
op = Pauli("ZI")
new_state = state.evolve(op)
print("After applying Pauli Z:", new_state)
```

✅ Features Demonstrated

Feature	Command
Quantum circuit creation	`QuantumCircuit()`
Gate application	`h` , `cx` , `rx` , `ry`
Measurement	`measure()`
Simulation	`Aer.get_backend()` , `execute()`
Visualization	`draw()` , `plot_histogram()`
Statevector & operators	`Statevector.from_instruction()` , `Pauli()`

📚 Common Gate Reference

🔶 Single Qubit Gates:

- `x`, `y`, `z` – Pauli gates
- `h` – Hadamard
- `s`, `sdg`, `t`, `tdg` – Phase gates
- `rx(θ)`, `ry(θ)`, `rz(θ)` – Rotations
- `u3(θ, ϕ, λ)` – Universal gate

🔶 Multi-Qubit Gates:

- `cx`, `cz`, `swap`, `ccx` – CNOT, Toffoli
- `cry`, `crz`, `cu3` – Controlled rotations

🔶 Measurement:

- `measure(qubit, classical_bit)`
- `measure_all()`

🔌 Optional Modules

- **Machine Learning**: `qiskit_machine_learning.neural_networks.EstimatorQNN`
- **Quantum Chemistry**: `qiskit_nature`, `qiskit_chemistry`
- **Finance**: `qiskit_finance` for option pricing and portfolio optimization

🍓 Strawberry Fields SDK: Command Categories & Use Cases

Category	Command / Function	Use Case
Engine & Program	`sf.Engine`, `sf.Program`	Define quantum photonic circuits
State Prep (Gaussian)	`Vacuum()`, `Coherent()`, `Squeezed()`, `Thermal()`	Prepare input photonic states
State Prep (Non-Gaussian)	`Fock(n)`, `Catstate()`	Use non-Gaussian resource states
Gaussian Gates	`Dgate`, `Sgate`, `Rgate`, `BSgate`, `MZgate`, `Interferometer`	Gaussian transformations
Non-Gaussian Gates	`Kgate`, `Vgate`, `Zgate`, `CubicPhase`	Useful for quantum computation beyond Gaussian models

Measurement	`MeasureFock`, `MeasureHomodyne`, `MeasureHeterodyne`	Collapse state and extract info
Utilities	`sf.utils.fidelity`, `sf.utils.random_interferometer`	Compare states, generate gates
Backends	`"fock"`, `"gaussian"`, `"tf"`	Choose simulation precision & type
State Access	`result.state`, `state.dm()`, `state.fock_prob`, `state.plot_wigner()`	Get info about the quantum state

🧪 Comprehensive Python Program

This program covers **most commands** in a single execution:

```python
import strawberryfields as sf
from strawberryfields.ops import *
from strawberryfields.utils import fidelity, random_interferometer
import numpy as np
import matplotlib.pyplot as plt

# Create a 4-mode program
prog = sf.Program(4)
```

```
# Create a 4-mode program
prog = sf.Program(4)

# Parameters
alpha = 0.5
r = 0.7
cutoff = 10

with prog.context as q:
    # --- State preparation (Gaussian & non-Gaussian) ---
    Coherent(alpha, 0) | q[0]
    Squeezed(r)        | q[1]
    Thermal(1.2)       | q[2]
    Fock(2)            | q[3]
```

```python
    # --- Gaussian gates ---
    Dgate(0.2, np.pi/4) | q[0]
    Sgate(0.5, np.pi/2) | q[1]
    Rgate(np.pi/3)      | q[2]
    BSgate(np.pi/4)     | (q[0], q[1])
    MZgate(np.pi/4, np.pi/2) | (q[1], q[2])
    Interferometer(random_interferometer(3)) | [q[1], q[2], q[3]]

    # --- Non-Gaussian gates ---
    Kgate(0.01)         | q[0]
    Vgate(0.05)         | q[1]
    Zgate(0.03)         | q[2]
```

```python
    # --- Measurement (choose one group) ---
    MeasureHomodyne(0.0) | q[0]
    MeasureHeterodyne()  | q[1]
    MeasureFock()        | q[2]
    MeasureFock()        | q[3]

# Run on Fock backend
eng = sf.Engine("fock", backend_options={"cutoff_dim": cutoff})
result = eng.run(prog)

# --- State Access and Analysis ---
state = result.state
```

```python
# Print Fock probabilities for mode 2
probs = state.fock_prob([0, 0, 0, 0])
print("Fock probability of |0000>:", probs)

# Wigner function of mode 1
state.plot_wigner(mode=1)
plt.title("Wigner function of mode 1")
plt.show()
```

✅ Commands Demonstrated

Feature	Commands Used
Program & Engine	`sf.Program`, `sf.Engine`
State Prep	`Coherent`, `Squeezed`, `Thermal`, `Fock`
Gaussian Gates	`Dgate`, `Sgate`, `Rgate`, `BSgate`, `MZgate`, `Interferometer`
Non-Gaussian Gates	`Kgate`, `Vgate`, `Zgate`
Measurements	`MeasureFock`, `MeasureHomodyne`, `MeasureHeterodyne`
Utilities	`random_interferometer`
State Access	`state.fock_prob`, `state.plot_wigner`

🔧 Optional Enhancements

- Add `Catstate()` to include Schrödinger cat state
- Add fidelity comparison:

```python
from strawberryfields.ops import Dgate

# Create ideal coherent state
ref_prog = sf.Program(1)
with ref_prog.context as r:
    Dgate(alpha) | r[0]
```

```
ref_state = sf.Engine("fock", backend_options={"cutoff_dim": cutoff}).run(ref_prog).state
tele_state = result.state.reduced_dm(0)
print("Fidelity:", fidelity(tele_state, ref_state.dm()))
```

🔗 Integration: PennyLane + Qiskit

PennyLane provides a plugin:

📦 `pennylane-qiskit`

Install it:

```bash
pip install pennylane qiskit pennylane-qiskit
```

🧪 Example: Optimize a Qiskit-style circuit using PennyLane

✅ What this does:

- Creates a PennyLane QNode using a **Qiskit simulator**
- Applies **Qiskit-style gates**
- Computes expectation values and gradients

```python
import pennylane as qml
from pennylane import numpy as np

# Step 1: Device using Qiskit backend (default.qubit from Qiskit)
dev = qml.device("qiskit.aer", wires=2, backend="aer_simulator_statevector")

# Step 2: Define hybrid quantum node using PennyLane + Qiskit
@qml.qnode(dev)
def qiskit_qnode(params):
    qml.RX(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(1))
```

```python
# Step 3: Evaluate circuit
params = np.array([0.1, 0.2], requires_grad=True)
print("Circuit output:", qiskit_qnode(params))

# Step 4: Gradient and optimization
grad_fn = qml.grad(qiskit_qnode)
print("Gradient:", grad_fn(params))

opt = qml.GradientDescentOptimizer(stepsize=0.2)
for i in range(5):
    params = opt.step(qiskit_qnode, params)
    print(f"Step {i+1}, cost = {qiskit_qnode(params):.4f}")
```

🔍 Device Options in `pennylane-qiskit`

Device ID	Description
`qiskit.aer`	Local simulator from Qiskit Aer
`qiskit.basicaer`	Legacy simulator (use only if needed)
`qiskit.ibmq`	Real IBM quantum devices (requires IBMQ account)

🛠 Using Real IBM Hardware

```python
from qiskit import IBMQ

IBMQ.save_account('MY_IBM_TOKEN')
IBMQ.load_account()
provider = IBMQ.get_provider()

# Get a real backend
backend = provider.get_backend('ibmq_quito')
```

```
# Register with PennyLane
dev = qml.device("qiskit.ibmq", wires=2, backend=backend)
```

> ⚠ Note: IBM backends take longer and require job queue handling.

✅ Summary of What You Can Do

Goal	Method
Use Qiskit-style circuits in PennyLane	Use `qml.device("qiskit.aer")` or `qiskit.ibmq`
Optimize Qiskit circuits	Use `qml.grad`, `qml.optimize`
Interface Qiskit and PyTorch	Use `qml.qnn.TorchLayer` with Qiskit-based QNodes
Access real IBM hardware	Use `qiskit.ibmq` with IBM credentials
Visualize gates	Use `qml.draw(qnode)` or `qnode.qtape.to_openqasm()`

Now you're exploring the **triple integration** of:

✅ Qiskit + □ PennyLane + 🍓 Strawberry Fields

This powerful stack allows you to:

🧩 Why Integrate All Three?

Platform	Purpose
🧠 PennyLane	Auto-differentiation, ML integration, hybrid quantum-classical circuits
✅ Qiskit	Gate-based circuits (qubits), IBMQ hardware access
🍓 Strawberry Fields	Photonic quantum computing (CV: continuous variables)

You can:

- Use **Qiskit's devices** inside PennyLane
- Use **Strawberry Fields' CV devices** inside PennyLane
- Optimize **both qubit and photonic circuits** in one unified framework via PennyLane

🔧 Required Installation

Install all 3:

```bash
pip install pennylane qiskit pennylane-qiskit strawberryfields pennylane-sf
```

🧪 Full Example: One Program Using All Three

We'll define:

1. A **Qiskit-based QNode** (qubit model)
2. A **Strawberry Fields QNode** (CV model)
3. Optimize both circuits using PennyLane

```python
import pennylane as qml
from pennylane import numpy as np

# ---------- Qiskit-based QNode (qubits) ----------
dev_qubit = qml.device("qiskit.aer", wires=2, backend="aer_simulator_statevector")

@qml.qnode(dev_qubit)
def qiskit_circuit(params):
    qml.RX(params[0], wires=0)
    qml.RY(params[1], wires=1)
    qml.CNOT(wires=[0, 1])
    return qml.expval(qml.PauliZ(1))
```

```python
# ---------- Strawberry Fields-based QNode (CV/photonic) ----------
dev_cv = qml.device("strawberryfields.fock", wires=1, cutoff_dim=10)

@qml.qnode(dev_cv)
def sf_circuit(x):
    qml.Displacement(x, 0.0, wires=0)
    qml.Squeezing(0.1, 0.0, wires=0)
    return qml.expval(qml.NumberOperator(0))
```

```python
# ---------- Unified Objective Function ----------
def combined_cost(params):
    q_part = qiskit_circuit(params)
    cv_part = sf_circuit(params[0])  # use only one param for SF
    return q_part + cv_part  # simple sum of expectations

# ---------- Optimization ----------
params = np.array([0.1, 0.2], requires_grad=True)
opt = qml.GradientDescentOptimizer(stepsize=0.2)
```

```python
for i in range(10):
    params = opt.step(combined_cost, params)
    print(f"Step {i+1}, Combined Cost = {combined_cost(params):.4f}")
```

✅ What You Just Did:

Part	Used
Qiskit circuit	via `qiskit.aer` device
SF photonic circuit	via `strawberryfields.fock`
PennyLane QNodes	to unify & differentiate both
Optimization	over parameters from both domains