In [1]:
```python
import numpy as np
```

In [2]:
```python
import qiskit
print(qiskit.__version__)
```

```
2.1.1
```

In [4]:
```python
import matplotlib.pyplot as plt
```

In [5]:
```python
from qiskit import QuantumCircuit
```

In [6]:
```python
from qiskit_ibm_runtime import sampler
```

In [7]:
```python
from qiskit import transpile
```

In [8]:
```python
from qiskit_aer import Aer
```

In [9]:
```python
from qiskit.visualization import plot_histogram
```

In [11]:
```python
from qiskit.quantum_info import Statevector
```

In [12]:
```python
from qiskit_aer import AerSimulator
```

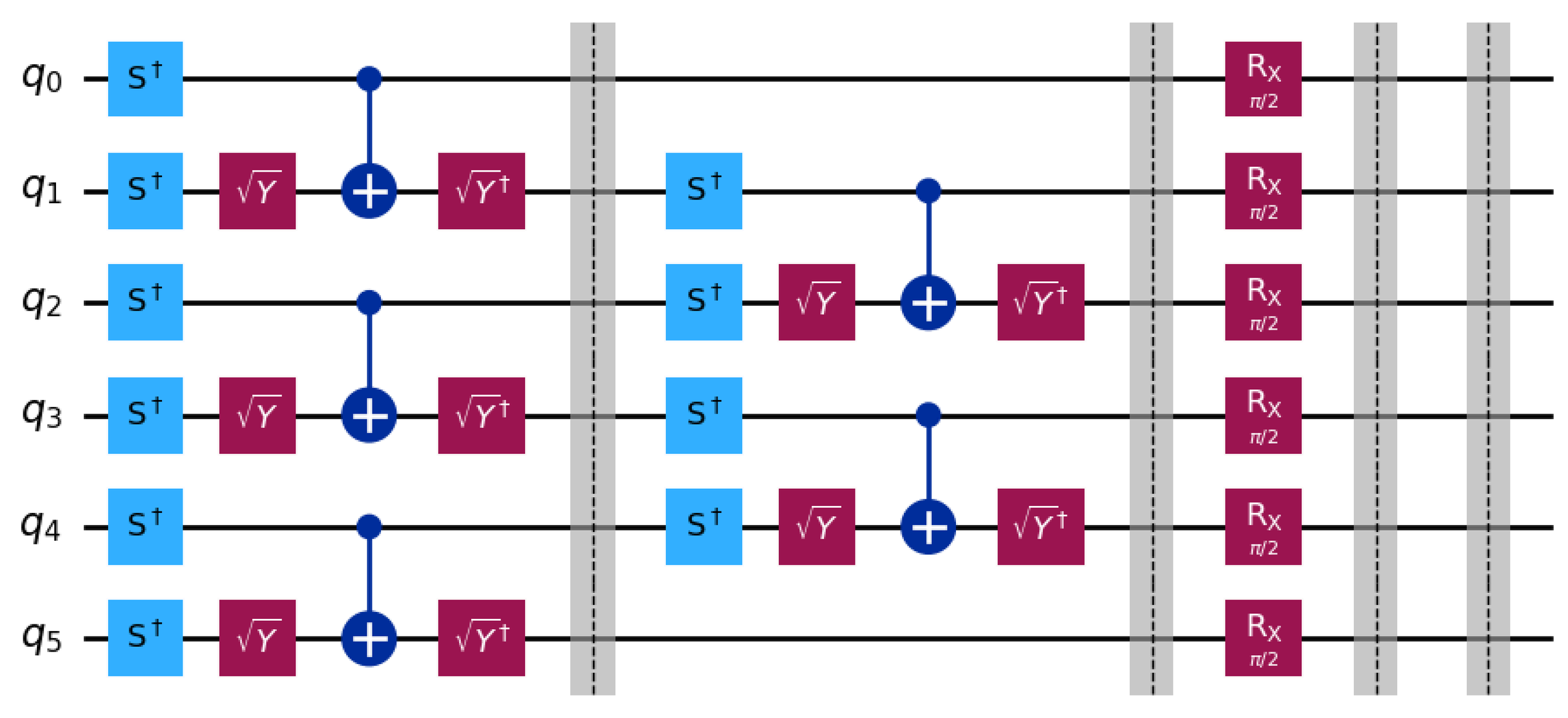
In [13]:
```python
from qiskit import QuantumCircuit
```

Transverse field Ising model to demo primitives

In [11]:
```python
from qiskit import QuantumCircuit
from qiskit.circuit.library import YGate, UnitaryGate
import numpy as np

SYGate = UnitaryGate(YGate().power(1/2), label=r"$\sqrt{Y}$")
SYdgGate = UnitaryGate(SYGate.inverse(), label=r"$\sqrt{Y}^\dag$")

def generate_1d_tfim_circuit(num_qubits, num_trotter_steps, rx_angle, num_cl_bits=0, trotter_barriers = False, layer_barriers = False):
    if num_cl_bits == 0:
        qc = QuantumCircuit(num_qubits)
    else:
        qc = QuantumCircuit(num_qubits, num_cl_bits)

    for trotter_step in range(num_trotter_steps):
        add_1d_tfim_trotter_layer(qc, rx_angle, layer_barriers)
        if trotter_barriers:
            qc.barrier()

    return qc

def add_1d_tfim_trotter_layer(qc, rx_angle, layer_barriers = False):
    # Adding Rzz in the even layers
    for i in range(0, qc.num_qubits-1, 2):
        qc.sdg([i, i+1])
        qc.append(SYGate, [i+1])
        qc.cx(i, i+1)
        qc.append(SYdgGate, [i+1])
    if layer_barriers:
        qc.barrier()
    # Adding Rzz in the odd layers
    for i in range(1, qc.num_qubits-1, 2):
        qc.sdg([i, i+1])
        qc.append(SYGate, [i+1])
        qc.cx(i, i+1)
        qc.append(SYdgGate, [i+1])
    if layer_barriers:
        qc.barrier()
    qc.rx(rx_angle, list(range(qc.num_qubits)))
    if layer_barriers:
        qc.barrier()
```

In [12]:
```python
num_qubits = 6
num_trotter_steps = 1
rx_angle = 0.5 * np.pi

qc = generate_1d_tfim_circuit(num_qubits, num_trotter_steps, rx_angle, trotter_barriers=True, layer_barriers=True)
qc.draw(output='mpl', fold=-1)
```

Out[12]:

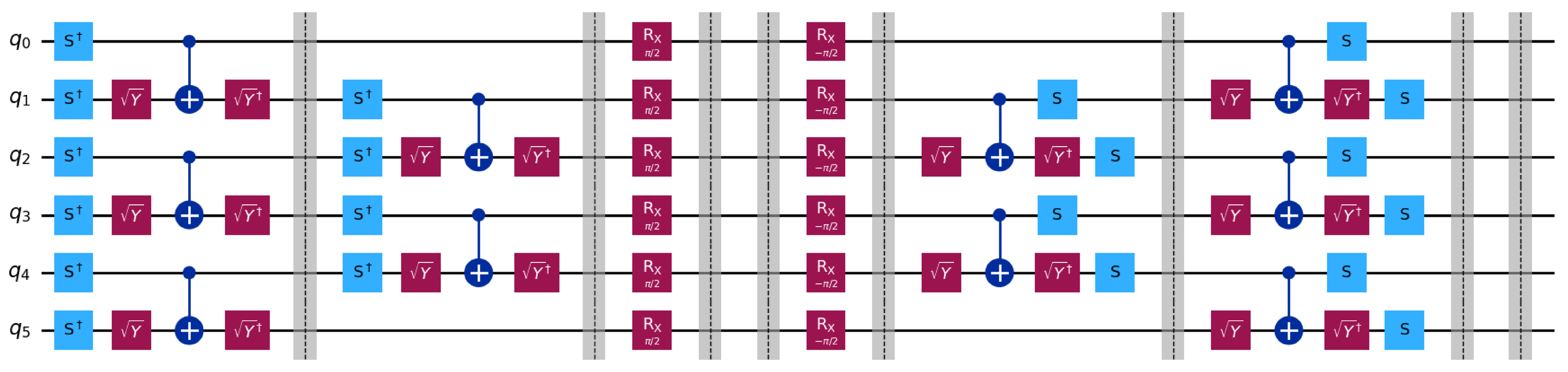


Demo: Sampler

In [13]:
```python
def append_mirrored_1d_tfim_circuit(qc, num_qubits, num_trotter_steps, rx_angle, trotter_barriers = False, layer_barriers = False):
    for trotter_step in range(num_trotter_steps):
        add_mirrored_1d_tfim_trotter_layer(qc, rx_angle, layer_barriers)
        if trotter_barriers:
            qc.barrier()

def add_mirrored_1d_tfim_trotter_layer(qc, rx_angle, layer_barriers = False):
    # Note after filming:
    # I constructed the inverse by hand here
    # But you could also use QuantumCircuit.inverse() to do this more efficiently
    qc.rx(-rx_angle, list(range(qc.num_qubits)))
    if layer_barriers:
        qc.barrier()
    # Adding Rzz in the odd layers
    for i in range(1, qc.num_qubits-1, 2):
        qc.append(SYGate, [i+1])
        qc.cx(i, i+1)
        qc.append(SYdgGate, [i+1])
        qc.s([i, i+1])
    if layer_barriers:
        qc.barrier()
    # Adding Rzz in the even layers
    for i in range(0, qc.num_qubits-1, 2):
        qc.append(SYGate, [i+1])
        qc.cx(i, i+1)
        qc.append(SYdgGate, [i+1])
        qc.s([i, i+1])
    if layer_barriers:
        qc.barrier()
```

In [14]:
```python
append_mirrored_1d_tfim_circuit(qc, num_qubits, num_trotter_steps, rx_angle, trotter_barriers=True, layer_barriers=True)
qc.draw(output='mpl', fold=-1)
```

Out[14]:



Step 1: Map the problem to circuits and observables

In [15]:
```python
max_trotter_steps = 10
num_qubits = 100
measured_qubits = [49, 50]

qc_list = []
for trotter_step in range(max_trotter_steps):
    qc = generate_1d_tfim_circuit(num_qubits, trotter_step, rx_angle, num_cl_bits=len(measured_qubits), trotter_barriers=True, layer_barriers=True)
    append_mirrored_1d_tfim_circuit(qc, num_qubits, trotter_step, rx_angle, trotter_barriers=True, layer_barriers=True)
    qc.measure(measured_qubits, list(range(len(measured_qubits))))
    qc_list.append(qc)
```

Step 2: Optimize

In [16]:
```python
from qiskit_ibm_runtime import QiskitRuntimeService  # ✅ correct class name

# Save your IBM Quantum account credentials (only needed once)
QiskitRuntimeService.save_account(
    channel="ibm_cloud",  # 🔄 'ibm_quantum' is deprecated — use 'ibm_cloud'
    token="4Uhb1rPkNtfpdU2g1aC9o-LgGW67nM8F5jN5WaTYUikj",
    overwrite=True
)

# Load the runtime service
service = QiskitRuntimeService()

# List available backends (optional for confirmation)
print(service.backends())

# Get the desired backend
backend = service.backend("ibm_brisbane")  # ✅ fixed typo: 'bsckend' → 'backend'
```

```
qiskit_runtime_service._resolve_cloud_instances:WARNING:2025-07-14 17:25:46,065: Default instance not set. Searching all available instances.
[<IBMBackend('ibm_brisbane')>, <IBMBackend('ibm_sherbrooke')>, <IBMBackend('ibm_torino')>]
```

In [17]:
```python
from qiskit import transpile
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService()
backend = service.backend(name='ibm_brisbane')
print("Done getting the backend")

# Note after filming:
# `transpile` will be deprecated soon
# so in the future, use `generate_preset_pass_manager` to achieve similar functionality
qc_list = []
qc_transpiled_list = transpile(qc_list, backend=backend, optimization_level=1)
```

```
qiskit_runtime_service._resolve_cloud_instances:WARNING:2025-07-14 17:25:58,474: Default instance not set. Searching all available instances.
Done getting the backend
```

Step 3: Execute on hardware

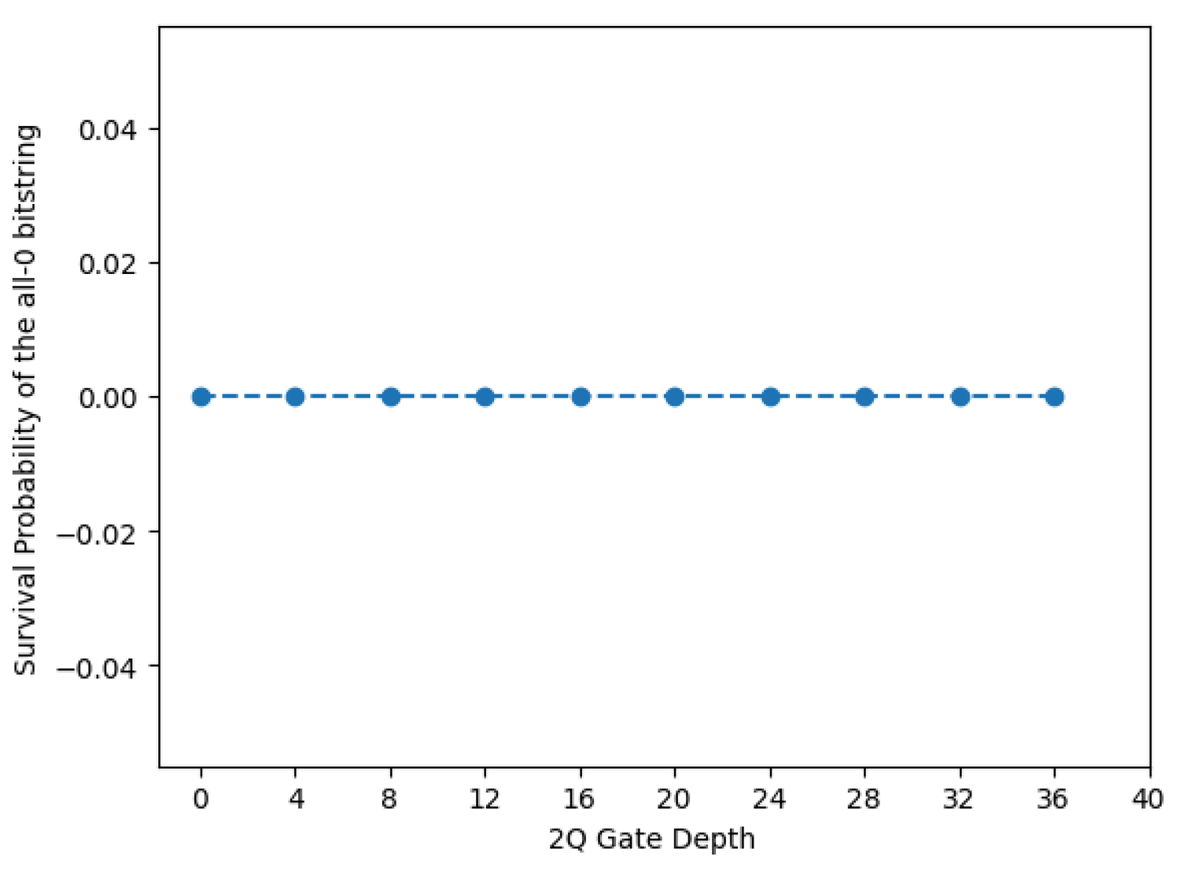
In [18]:
```python
from qiskit_ibm_runtime import SamplerV2 as Sampler

sampler = Sampler(backend)
sampler.options.dynamical_decoupling.enable = True
sampler.options.dynamical_decoupling.sequence_type = "XY4"

job = sampler.run(qc_transpiled_list)
print(job.job_id())
```

```
d1qff2v6d0hc73b9gc90
```

Step 4: Post-processing and plotting

In [19]:
```python
job_id = "d1qff2v6d0hc73b9gc90"
job = service.job(job_id)

max_trotter_steps = 10
survival_probability_list = []
for trotter_step in range(max_trotter_steps):
    try:
        data = job.result()[trotter_step].data
        survival_probability_list.append(data.c.get_counts()['0' * len(measured_qubits)] / data.c.num_shots)
    except:
        survival_probability_list.append(0)
```

In [21]:
```python
import matplotlib.pyplot as plt

plt.plot(list(range(0, 4 * max_trotter_steps, 4)), survival_probability_list, '--o')
plt.xlabel('2Q Gate Depth')
plt.ylabel('Survival Probability of the all-0 bitstring')
plt.xticks(np.arange(0, 44, 4))
plt.show()
```



Demo: Estimator

Step 1: Mapping the problem

In [22]:
```python
from qiskit.circuit import Parameter
rx_angle = Parameter("rx_angle")
trotter_steps = 2
qc = generate_1d_tfim_circuit(num_qubits, trotter_steps, rx_angle)

from qiskit.quantum_info import SparsePauliOp

middle_index = num_qubits // 2
observable = SparsePauliOp("I" * middle_index + "Z" + "I" * (middle_index-1))
```

Step 2: Optimize the circuit

In [23]:
```python
from qiskit import transpile
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService()
backend = service.backend(name='ibm_brisbane')

# Note after filming:
# `transpile` will be deprecated soon
# so in the future, use `generate_preset_pass_manager` to achieve similar functionality
qc_transpiled = transpile(qc, backend=backend, optimization_level=1)
observable = observable.apply_layout(qc_transpiled.layout)
```

```
qiskit_runtime_service._resolve_cloud_instances:WARNING:2025-07-14 17:28:40,668: Default instance not set. Searching all available instances.
```

Step 3: Execute on quantum hardware

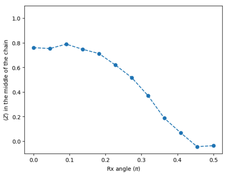
In [25]:
```python
from qiskit_ibm_runtime import EstimatorV2, EstimatorOptions

min_rx_angle = 0
max_rx_angle = np.pi/2
num_rx_angle = 12
rx_angle_list = np.linspace(min_rx_angle, max_rx_angle, num_rx_angle)

options = EstimatorOptions()
options.resilience_level = 1
options.dynamical_decoupling.enable = True
options.dynamical_decoupling.sequence_type = "XY4"

estimator = EstimatorV2(backend, options=options)

job = estimator.run([(qc_transpiled, observable, rx_angle_list)])
print(job.job_id())
```

```
d1qfgdf6d0hc73b9ge30
```

Step 4: Post-processing and plotting

In [26]:
```python
job_id = "d1qfgdf6d0hc73b9ge30"
job = service.job(job_id)

exp_val_list = job.result()[0].data.evs

plt.plot(rx_angle_list / np.pi, exp_val_list, '--o')
plt.xlabel(r'Rx angle ($\pi$)')
plt.ylabel(r'$\langle Z \rangle$ in the middle of the chain')
plt.ylim(-0.1, 1.1)
```

Out[26]:
```
(-0.1, 1.1)
```


1.0
0.8
0.6
0.4
0.2
0.0
$|Z|$ in the middle of the chain
0.0
0.1
0.2
0.3
0.4
0.5
Rx angle (π)

In [ ]: