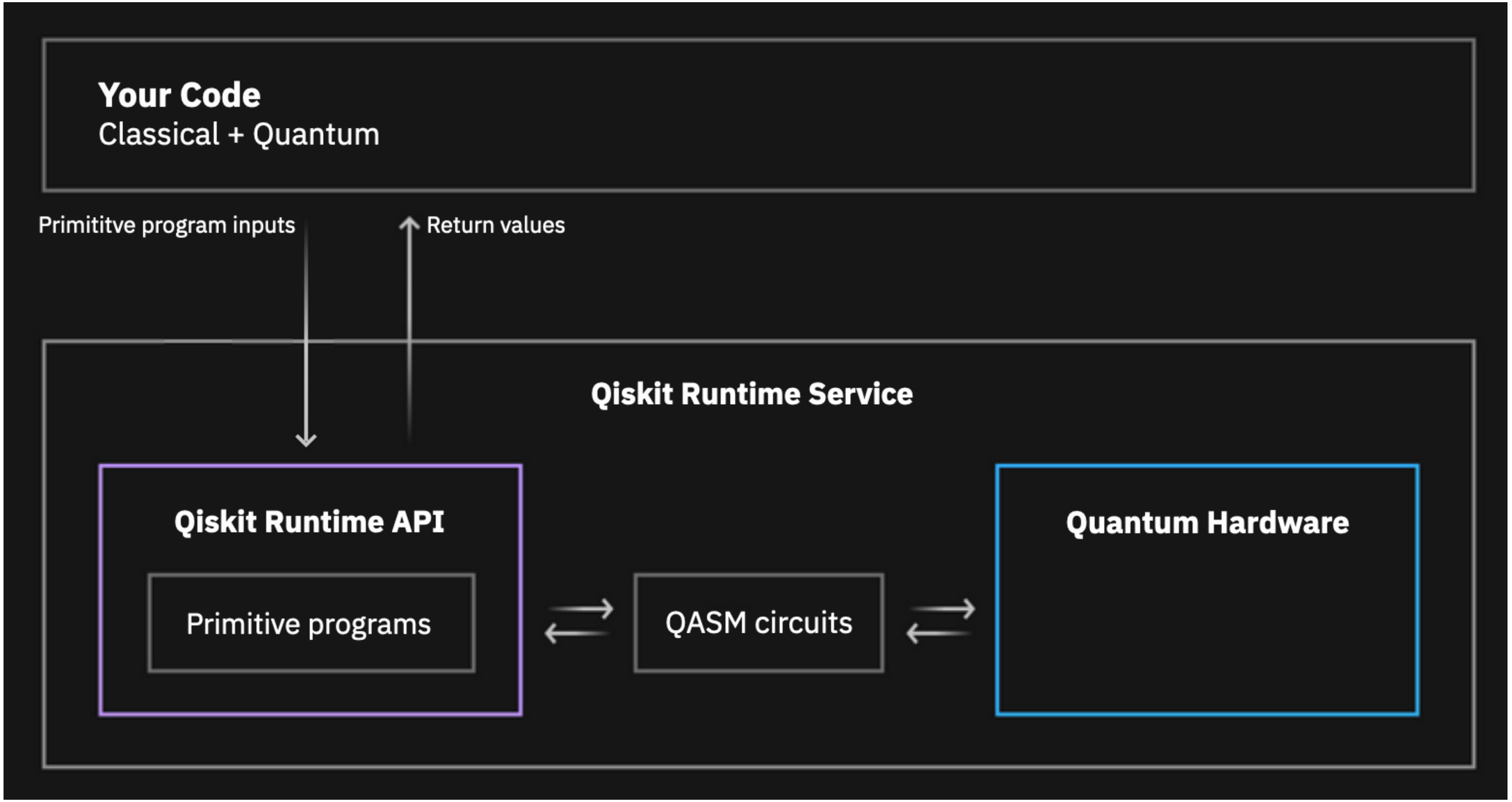
Qiskit Global Summer School 2024

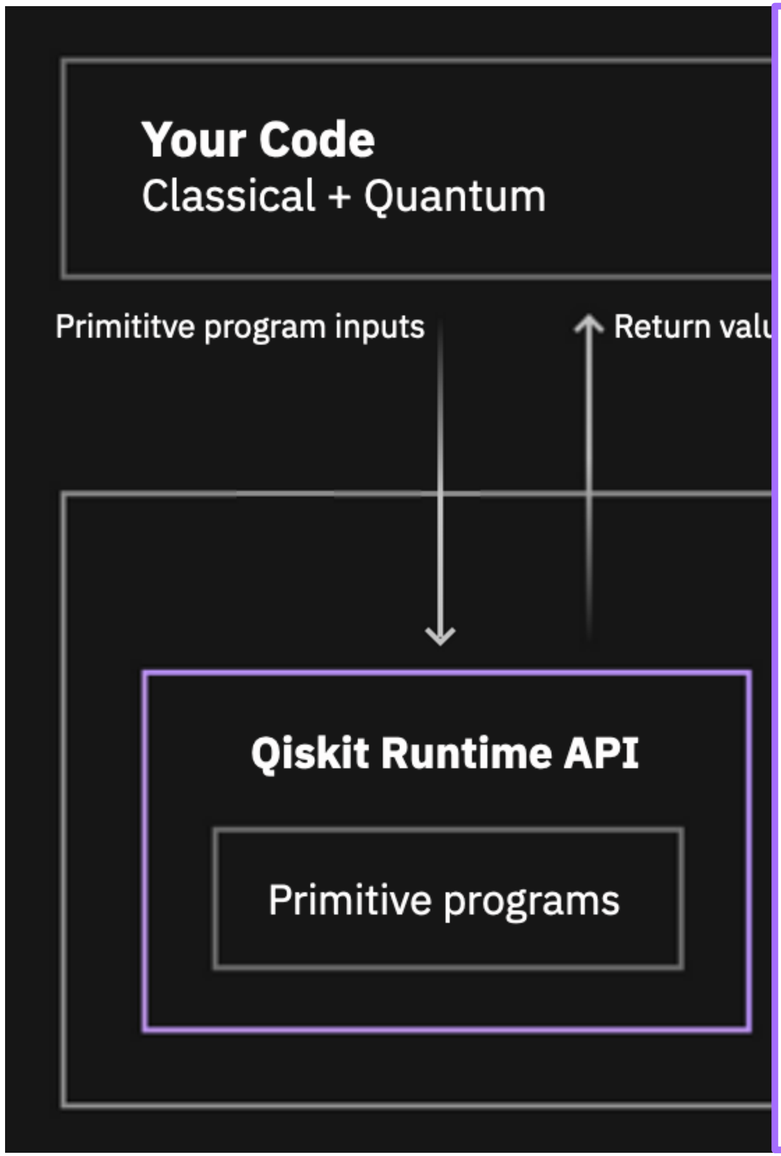
Qiskit Runtime Primitives V2 Introduction

Christopher J Wood
Senior Research Scientist
IBM

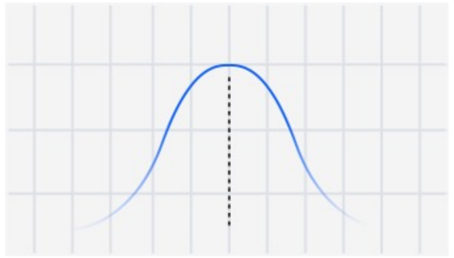
Qiskit Runtime



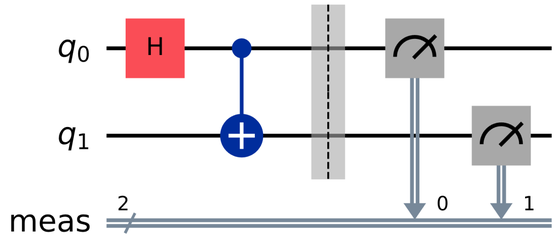
Qiskit Runtime



Sampler primitive

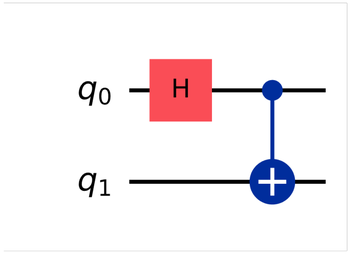


- Low-level execution of circuits.
- Returns *single-shot* measurement outcomes.
- Circuits should include measurements



Estimator primitive

- Higher level execution for algorithms and applications
- Returns expectation values of observables.
- Circuit should not include measurements.



V2 Primitives

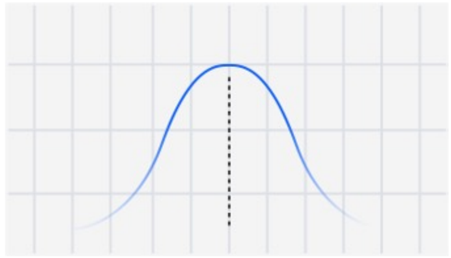
Qiskit V2 Primitives

- New primitive API introduce in Qiskit 1.0
- Efficient specification of *parametric* programs

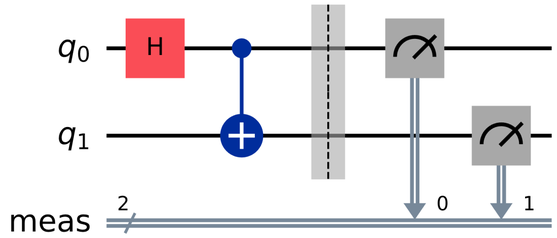
IBM V2 Primitives

- Introduced in Qiskit IBM Runtime 0.21
- Efficient execution of *parametric* programs on IBM hardware
- Built in error *suppression* and *mitigation* techniques
- Intended for utility scale quantum computing

Sampler primitive

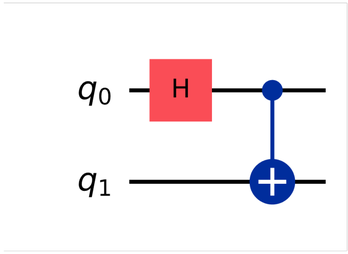


- Low-level execution of circuits.
- Returns *single-shot* measurement outcomes.
- Circuits should include measurements



Estimator primitive

- Higher level execution for algorithms and applications
- Returns expectation values of observables.
- Circuit should not include measurements.



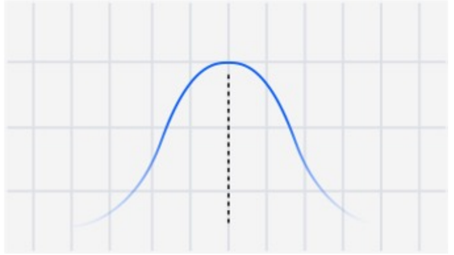
Sampler V2

- The Sampler primitive is a low-level primitive for the execution of circuits and obtaining raw measurement outcomes for each shot.
- The Sampler primitive API is defined by its run method

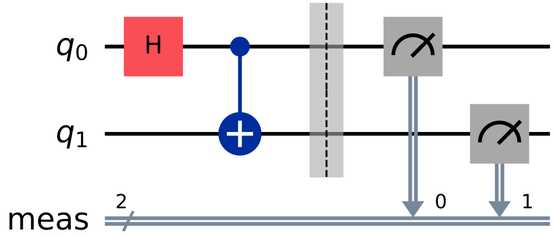
```
Sampler.run(
    pubs: list[SamplerPubLike],
    shots: int | None = None,
) -> Job[SamplerResult]
```

- It can accept a list of 1 or more input programs represented as sampler Primitive Unified Blocs (pubs)
- Optional specification of the number of shots to run the pubs for

Sampler primitive



- Low-level execution of circuits
- Returns *single-shot* measurement outcomes
- Circuits should include measurements



Sampler Pubs

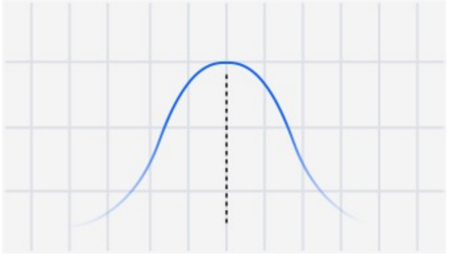
Primitive Unified Blocs

The Primitive Unified Bloc (pub) program input for the sampler is in general a tuple

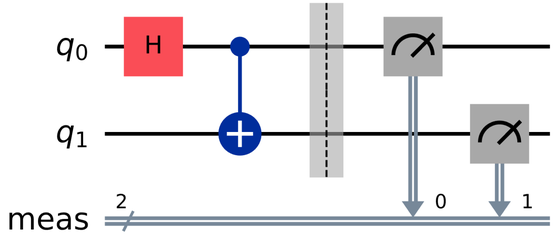
```
pub = (circuit [required],
       parameter_values [optional],
       shots [optional])
```

- Circuit: An ISA `QuantumCircuit` containing 1 or more `ClassicalRegister` and `measure` instructions.
- Parameter Values: An tensor (ND-array) of sets of parameter values to evaluate a parametric circuit with.
- Shots: The number of samples or repetitions to measure the circuit for each set of parameter values.

Sampler primitive



- Low-level execution of circuits
- Returns *single-shot* measurement outcomes
- Circuits should include measurements



Sampler Pubs

Primitive Unified Blocs

The Primitive Unified Bloc (pub) program input for the sampler is in general a tuple

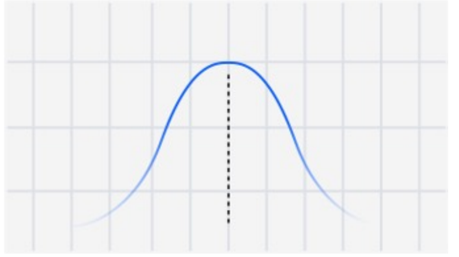
```
pub = (circuit [required],
       parameter_values [optional],
       shots [optional])
```

Allowed pub-Like inputs

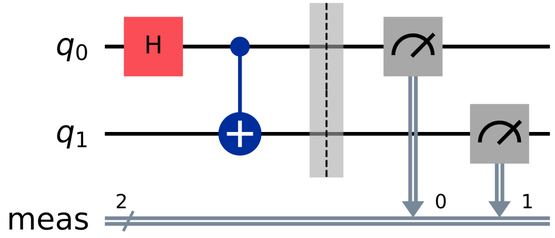
Because of the optional arguments the following are all valid input pubs for the sampler

1. (parametric_isa_circuit, parameter_values, shots)
2. (parametric_isa_circuit, parameter_values)
3. (non_parametric_isa_circuit, None, shots)
4. (non_parametric_isa_circuit,)

Sampler primitive



- Low-level execution of circuits
- Returns *single-shot* measurement outcomes
- Circuits should include measurements



Sampler V2 Example

Example

Run a simple Bell-state measurement circuit

- Load runtime service account and select a backend

```
from qiskit_ibm_runtime import QiskitRuntimeService

# Load saved runtime account
service = QiskitRuntimeService()

# Select a backend
backend = service.get_backend("ibm_auckland")
```

- Create a circuit and transpile to a backend ISA circuit

```
from qiskit import QuantumCircuit, transpile

# create a Bell circuit
bell_meas = QuantumCircuit(2)
bell_meas.h(0)
bell_meas.cx(0, 1)
bell_meas.measure_all()

# Transpile to an ISA Circuit for the intended backend
isa_bell_meas = transpile(bell_meas, backend)

# Construct pub and run
pub_bell_meas = (isa_bell_meas,)
```

- Initialize a Sampler primitive and run pub

```
from qiskit_ibm_runtime import SamplerV2

# Initalize a sampler for the backend and run
sampler = SamplerV2(backend)
job_bell_meas = sampler.run([pub_bell_meas], shots=10)

# Extract results
result_bell_meas = job_bell_meas.result()
data_bell_meas = result_bell_meas[0].data
data_bell_meas
```

```
DataBin(meas=BitArray(<shape=(), num_shots=10, num_bits=2>))
```

Sampler V2 Results

`PubResult` Class

When run a Sampler Job will return a **`PrimitiveResult`** object containing an ordered list of **`PubResult`**`s corresponding to each of the input pubs

- `PubResult`: A data container of a single input Pub's execution results
- `PubResult.data`: Contains the measurement outcome data for all classical registers in the input pubs circuit
- `PubResult.metadata`: Contains any additional metadata that a primitive might record

`DataBin` Class

The **`PubResult.data`** is a container called a **`DataBin`**

- A **`DataBin`** which stores the outcomes of measurements in **`ClassicalRegister`** in the PUB circuit
- Register results are accessed via name either as attributes, or as a mapping
- Each register’s data is stored in a **`BitArray`** container

Sampler V2 BitArrays

Accessing Measurement Data

- The `BitArray` class stores the single-shot outcomes of all classical registers
- `BitArray` is an ND-array like object with the following attributes:
  - `shape`: given by the input PUB parameter shape
  - `num_bits`: the size of the input circuits classical register
  - `num_shots`: the number of shots per parameterization
  - `array`: the internal result data as a packed arrays of uint8 (byte) values

Example

```
bits = data_bell_meas.meas
print("shape:", bits.shape)
print("num_bits:", bits.num_bits)
print("num_shots:", bits.num_shots)
print("array:\n", bits.array)
```

```
shape: ()
num_bits: 2
num_shots: 10
array:
 [[0]
 [0]
 [0]
 [3]
 [0]
 [0]
 [3]
 [3]
 [3]
 [0]]
```

Helper methods

Convert to bitstring formats

```
bitstrings = bits.get_bitstrings()
print(bitstrings)
```

```
['00', '00', '00', '11', '00', '00', '11', '11', '11', '00']
```

```
counts = bits.get_counts()
print(counts)
```

```
{'00': 6, '11': 4}
```

ISA Circuits

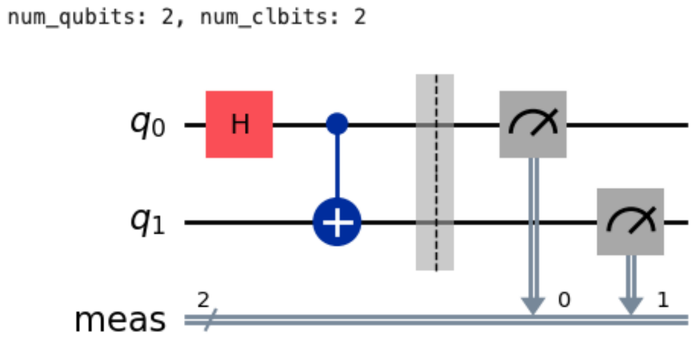
What is an ISA Circuit?

An ISA circuit is a a **`QuantumCircuit`** satisfying

1. The same *number of qubits* as a device
2. Only contains *basis gates* for a device
3. Satisfies the *connectivity* of a device

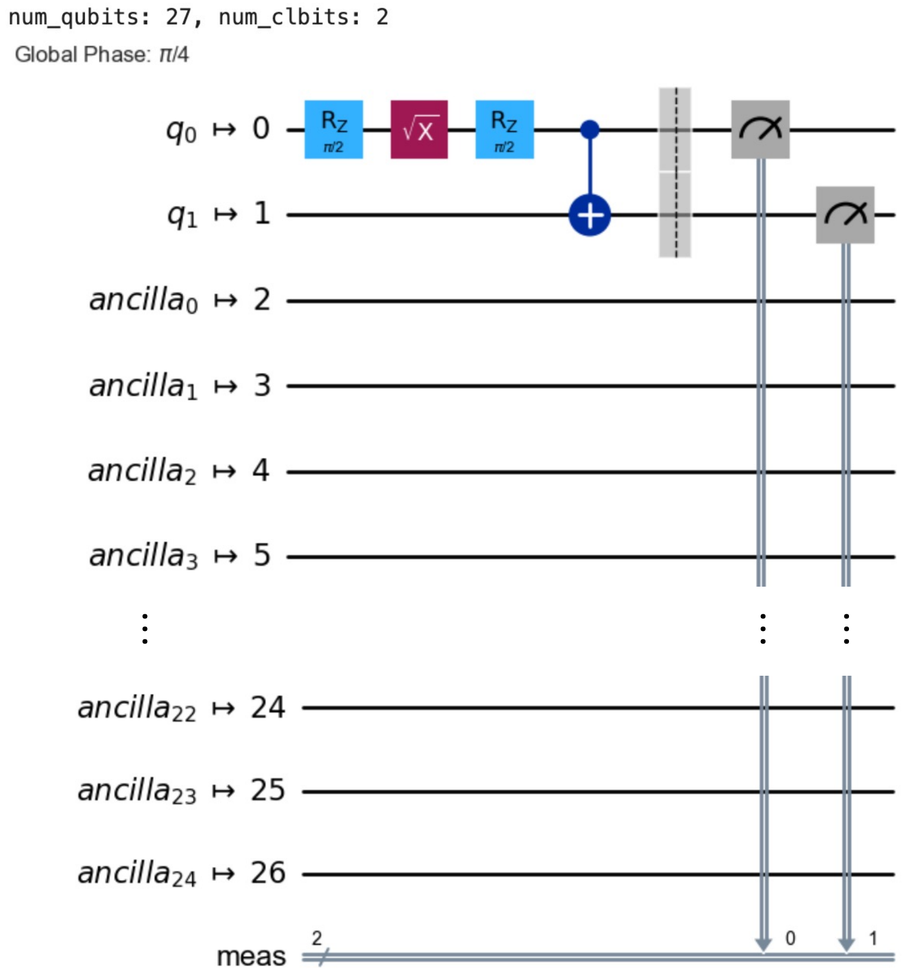
If you are used to working with abstract or logical circuits, you will typically obtain an ISA circuit via *transpilation* to a target or backend.

2-qubit Circuit



transpile

27-qubit ISA Circuit



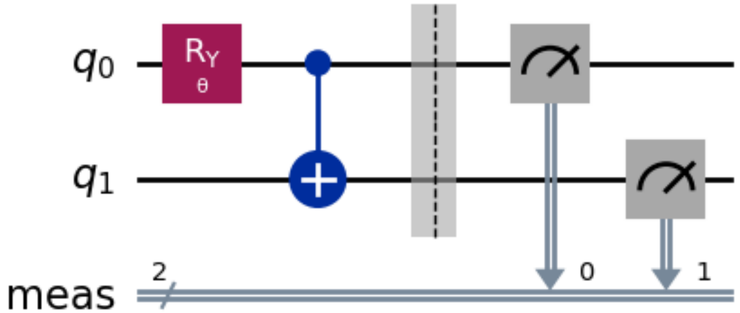
Parametric Circuits

What is a parametric circuit?

- A parametric circuit (ISA or abstract) is a circuit that contains *unbound parameter* values.

Example

Control 2-qubit entanglement via parameter



The shape of a Pub and PubResult

A sampler Pub shape is given the tensor shape its parameterizations to be evaluated

- A non-parametric Pub has shape ()
- If the ISA circuit with K parameters and (N, K) shaped parameter values array has shape (N,)
- A parametric Pub can be converted to a list of non-parametric pubs by binding all parameter values

Sampler V2 Parametric Example

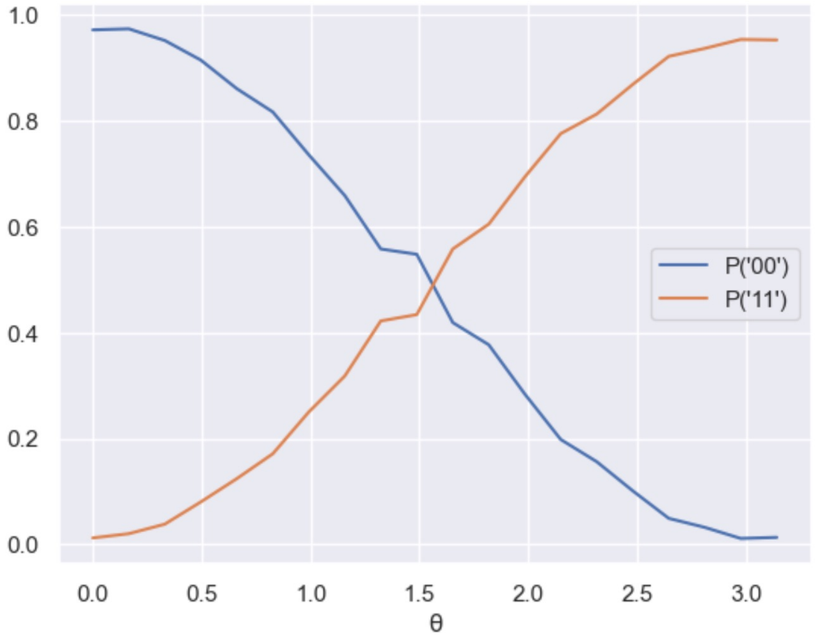
Example

Control 2-qubit entanglement via parameter

```
import numpy as np

# Parameter values to evalute with 20 theta values
param_vals = np.linspace(0, np.pi, 20)

# Transpile to an ISA Circuit for the intended backend
isa_par_bell_meas = transpile(par_bell_meas, backend)

# Construct pub and run
# Pub and result shape is (20,)
pub_par_bell_meas = (isa_par_bell_meas, param_vals)
job_par_bell_meas = sampler.run([pub_par_bell_meas], shots=1000)
result_par_bell_meas = job_par_bell_meas.result()

# Extract creg data
bits = result_par_bell_meas[0].data.meas
bits
```

```
BitArray(<shape=(20,), num_shots=1000, num_bits=2>)
```

```
import matplotlib.pyplot as plt
import seaborn
seaborn.set()

# Compute the probability of "00" outcome and "11" outcome
p0s = np.sum(bits.array == 0, axis=1) / bits.num_shots
p1s = np.sum(bits.array == 3, axis=1) / bits.num_shots

plt.plot(param_vals, p0s, label="P('00')")
plt.plot(param_vals, p1s, label="P('11')")
plt.xlabel("θ")
plt.legend()
plt.show()
```



Sampler V2 Shots

Specifying the number of shots

There are multiple ways to specify shots when running the Sampler

The order of resolution for these values is given by:

Sampler V2 Shots

Specifying the number of shots

There are multiple ways to specify shots when running the Sampler

The order of resolution for these values is given by:

1. If a pub specifies shots it will be used

```
# 1. All shots in pub
pub1a = (isa_bell_meas, None, 100)
pub1b = (isa_bell_meas, None, 200)

result1 = sampler.run([pub1a, pub1b], shots=1000).result()

for i, res in enumerate(result1):
    print(f"Pub {i}: num_shots = {res.data.meas.num_shots}")
```

```
Pub 0: num_shots = 100
Pub 1: num_shots = 200
```

Sampler V2 Shots

Specifying the number of shots

There are multiple ways to specify shots when running the Sampler

The order of resolution for these values is given by:

1. If a pub specifies shots it will be used
2. If run kwarg specifies shots, it will be used in all pubs *that do not specify shots*

```
# 1. All shots in pub
pub1a = (isa_bell_meas, None, 100)
pub1b = (isa_bell_meas, None, 200)

result1 = sampler.run([pub1a, pub1b], shots=1000).result()

for i, res in enumerate(result1):
    print(f"Pub {i}: num_shots = {res.data.meas.num_shots}")
```

```
Pub 0: num_shots = 100
Pub 1: num_shots = 200
```

```
# 2. Fall back to `run` shots
pub2a = (isa_bell_meas,)
pub2b = (isa_bell_meas, None, 200)

result2 = sampler.run([pub2a, pub2b], shots=1000).result()

for i, res in enumerate(result2):
    print(f"Pub {i}: num_shots = {res.data.meas.num_shots}")
```

```
Pub 0: num_shots = 1000
Pub 1: num_shots = 200
```

Sampler V2 Shots

Specifying the number of shots

There are multiple ways to specify shots when running the Sampler

The order of resolution for these values is given by:

1. If a pub specifies shots it will be used
2. If run kwarg specifies shots, it will be used in all pubs *that do not specify shots*
3. If no run kwarg is provided all Pubs without shots will use a default value chosen by the Sampler

For The IBM runtime Sampler the default value is 4096.

```python
# 1. All shots in pub
pub1a = (isa_bell_meas, None, 100)
pub1b = (isa_bell_meas, None, 200)

result1 = sampler.run([pub1a, pub1b], shots=1000).result()

for i, res in enumerate(result1):
    print(f"Pub {i}: num_shots = {res.data.meas.num_shots}")
```

```
Pub 0: num_shots = 100
Pub 1: num_shots = 200
```

```python
# 2. Fall back to `run` shots
pub2a = (isa_bell_meas,)
pub2b = (isa_bell_meas, None, 200)

result2 = sampler.run([pub2a, pub2b], shots=1000).result()

for i, res in enumerate(result2):
    print(f"Pub {i}: num_shots = {res.data.meas.num_shots}")
```

```
Pub 0: num_shots = 1000
Pub 1: num_shots = 200
```

```python
# 3. Fallback to default shots
pub3a = (isa_bell_meas,)
pub3b = (isa_bell_meas, None, 200)

result3 = sampler.run([pub3a, pub3b]).result()

for i, res in enumerate(result3):
    print(f"Pub {i}: num_shots = {res.data.meas.num_shots}")
```

```
Pub 0: num_shots = 4096
Pub 1: num_shots = 200
```

Estimator V2

The Estimator primitives is a higher-level primitive than the Sampler.

- Used for evaluating estimates of *expectation values* on the state prepared by a circuit

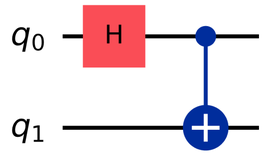
- The Estimator primitive API is defined by its run method

```
Estimator.run(
    pubs: list[EstimatorPub],
    precision: float | None = None,
) -> Job[EstimatorResult]
```

- It can accept a list of 1 or more input programs represented as estimator pubs
- Optional specification of the desired *precision* of expectation value estimates.

Estimator primitive

- Higher level execution for algorithms and applications
- Returns expectation values of observables.
- Circuit should not include measurements.



Estimator Pubs

Primitive Unified Blocs

The Primitive Unified Bloc (pub) program input for the estimator is in general a tuple

```
pub = (circuit [required],
       observables [required],
       parameter_values [optional],
       precision [optional],)
```

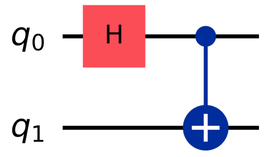
Allowed pub-like inputs

Because of the optional arguments the following are all valid input pubs for the sampler

1. (parametric_isa_circuit, isa_observables, parameter_values, precision)
2. (parametric_isa_circuit, isa_observables, parameter_values)
3. (non_parametric_isa_circuit, isa_observables, None, precision)
4. (non_parametric_isa_circuit, isa_observables)

Estimator primitive

- Higher level execution for algorithms and applications
- Returns expectation values of observables.
- Circuit should not include measurements.



Estimator Observables

Representing Observables

- An observable is a *Hermitian operator* represented as linear combinations of Pauli operators

- *Single Pauli* operators can be represented as:
  - A string containing I, X, Y, Z terms
  - A `quantum_info.Pauli` object

- A *list of Pauli* operators can be represented as:
  - `list[Pauli]`
  - `quantum_info.PauliList`

- A *Hamiltonian* or other operator can be represented as a `quantum_info.SparsePauliOp`

Example

Create an ISA circuit

```
from qiskit_ibm_runtime import EstimatorV2

# Create a Bell circuit
bell = QuantumCircuit(2)
bell.h(0)
bell.cx(0, 1)

# Transpile to an ISA Circuit for the intended backend
isa_bell = transpile(bell, backend)
```

Create an ISA observable

```
import qiskit.quantum_info as qi

# Create an observable
obs = qi.SparsePauliOp(["ZZ"])

# Transpile to ISA observable for intended ISA circuit
isa_obs = obs.apply_layout(isa_bell.layout)
```

Estimator V2 Results

Estimator DataBin

Like a sampler, an Estimator Job will contain a `PrimitiveResult` with a list of `PubResult`s for each input pub.

Instead of measure outcomes its `DataBin` contains fields:

- `evs`: The mean expectation value estimates for all input pub parameter values and observables.
- `stds`: The standard error of the mean of expectation value estimates.

Example

Run an Estimator pub

```
# Construct Estimator
estimator = EstimatorV2(backend)

# Construct pub and run
pub = (isa_bell, isa_obs)
est_job_bell = estimator.run([pub])
est_result_bell = est_job_bell.result()
```

View result data

```
data = est_result_bell[0].data
evs = data.evs
stds = data.stds
print(f"<ZZ> = {evs:.3f} ± {stds:.3f}")
```

```
<ZZ> = 1.002 ± 0.008
```

Estimator V2 Shape

An Estimator Pub has a *shape* that depends on both the *parameter values shape* and the *observables shape*

Trivial Shaped Pubs

- Trivial shaped pubs have `shape==()`
- These return a single float as the `evs` result

Examples of `shape==()` pubs

- A *non-parametric* pub with a *single observable*
- A *parametric pub* with a *single parameter values set* and a *single observable*

For general tensor observables and parameter values sets the shape will be the *broadcasted shape* of the two tensors

Estimator V2 Broadcasting

ND-Array Broadcasting Rules

Broadcasting follows the same rules a NumPy ND-array broadcasting:

- Input arrays do not need to have the same number of dimensions
- The resulting array will have the same number of dimensions as the largest
- The size of each dimension is the largest size of the corresponding dimension
- Missing dimensions are assumed to have size one
- It starts with the trailing (i.e. rightmost) dimension and works its way left
- Two dimensions are compatible when their sizes are equal or one of them is 1

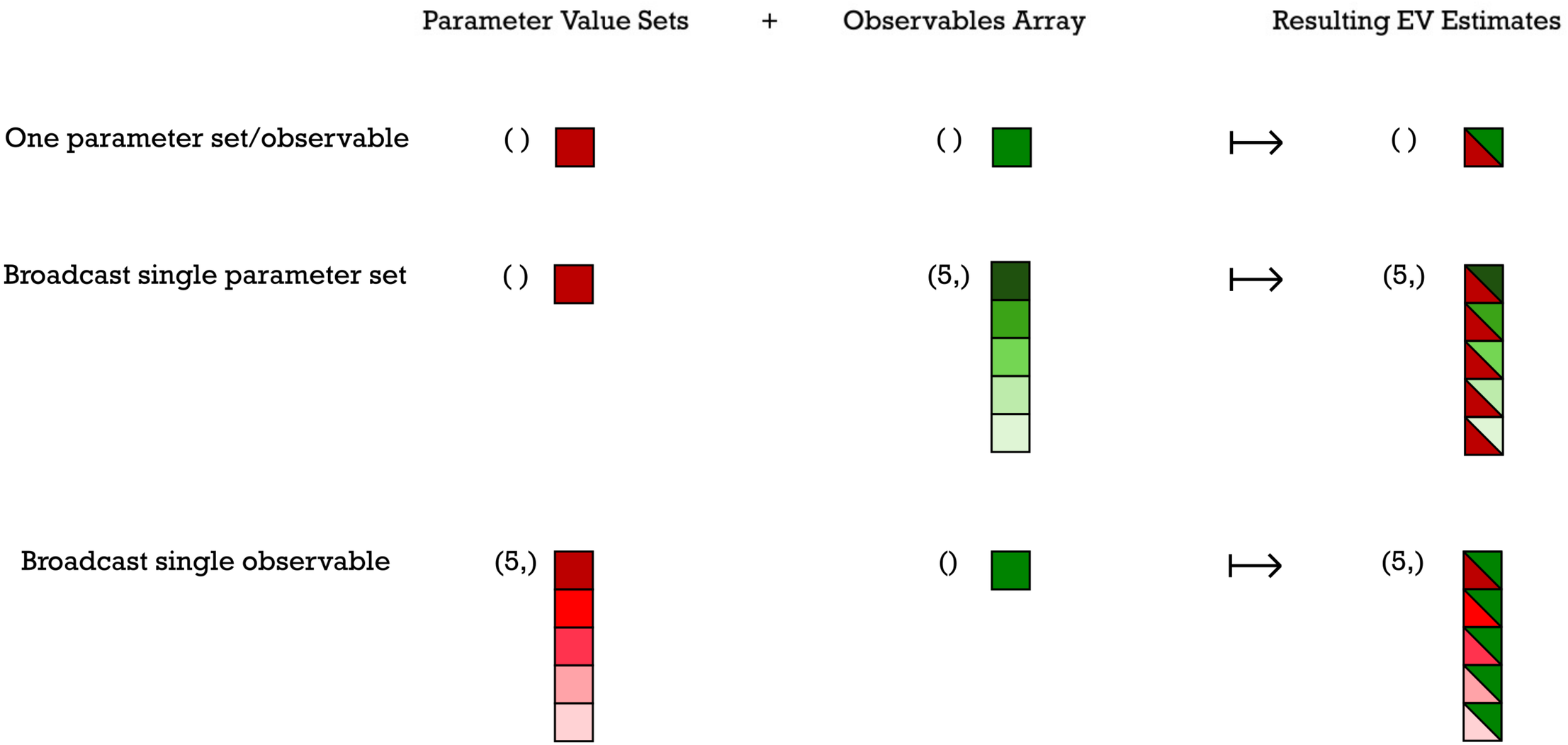
Example

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4
```

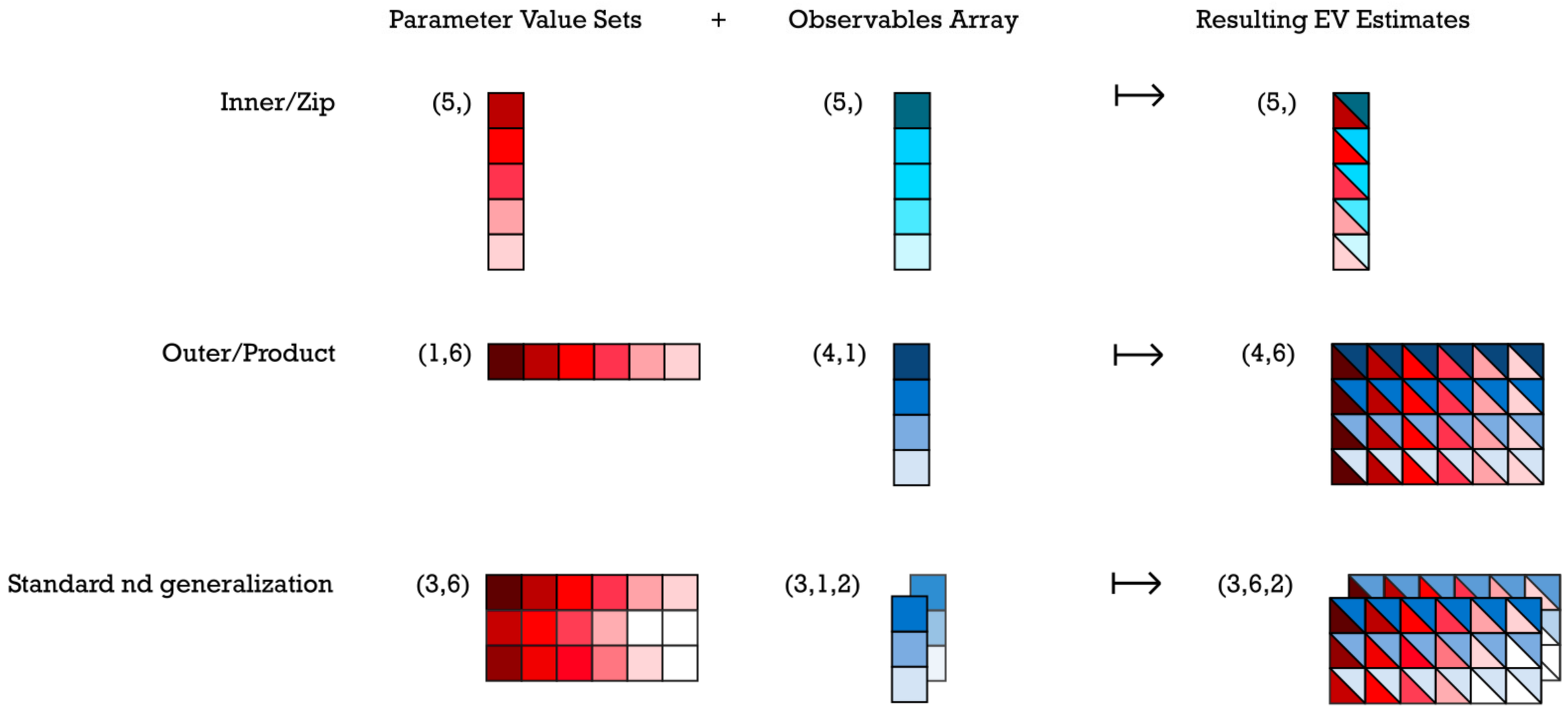
Example

```
A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5
```

Estimator V2 Broadcasting



Estimator V2 Broadcasting



Estimator V2 Broadcasting

Example

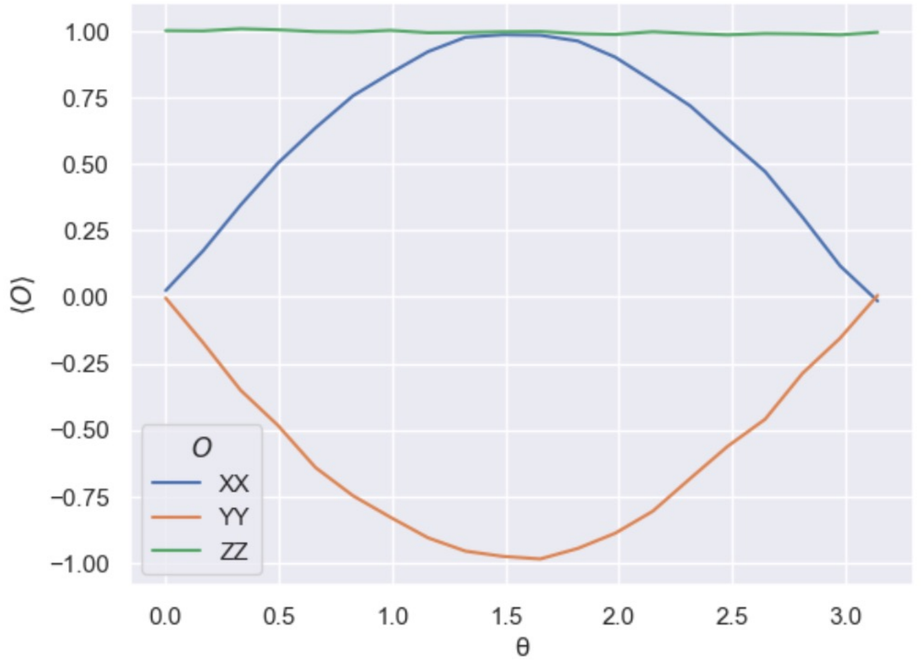
Parameterized bell circuit with multiple observables

```
# Parameterized bell circuit
theta = Parameter("θ")
par_bell = QuantumCircuit(2)
par_bell.ry(theta, 0)
par_bell.cx(0, 1)

# Transpile to an ISA Circuit for the intended backend
isa_par_bell = transpile(par_bell, backend)

# Parameter values to evalute with 20 theta values
param_vals = np.linspace(0, np.pi, 20)

# Create an observable array with 3 observables
par_bell_obs = [
    qi.SparsePauliOp(["XX"]),
    qi.SparsePauliOp(["YY"]),
    qi.SparsePauliOp(["ZZ"]),
]
# Transpile to ISA observable and reshape to (3, 1) array
isa_par_bell_obs = [
    [op.apply_layout(isa_par_bell.layout)]
    for op in par_bell_obs
]

# Construct pub and run
# Pub and result shape is (20,)
pub = (isa_par_bell, isa_par_bell_obs, param_vals)
```

Run on estimator

```
# Run estimator
est_job_par_bell = estimator.run([pub])
est_result_par_bell = est_job_par_bell.result()
evs = est_result_par_bell[0].data.evs
stds = est_result_par_bell[0].data.stds
```

Plot results



ISA Observables

What is an ISA Observable?

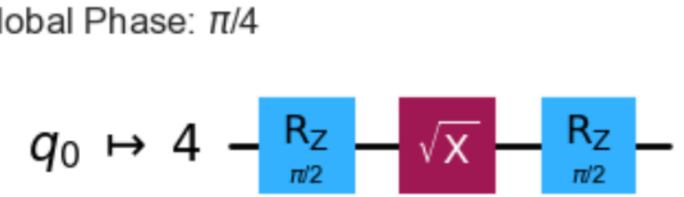
1. A linear combination of Pauli's with real coefficients
2. Each Pauli is defined on the same number of qubits as an ISA circuit

- If you are used to working with abstract observables you will need to *apply the layout* of a transpiled circuit to map the observable qubits

- Several `SparsePauliOp` and `Pauli` operator classes have an `apply_layout` method to simplify this process

Example

```
abstract_example = QuantumCircuit(1)
abstract_example.h(0)
isa_example = transpile(
    abstract_example, backend, initial_layout=[4]
)
display(isa_example.draw('mpl', idle_wires=False))
```




```
abstract_obs = qi.SparsePauliOp(["X", "Y", "Z"])
abstract_obs.apply_layout(isa_example.layout)
```

```
SparsePauliOp(['IIIIIIIIIIIIIIIIIIIIXIIII', 'IIIIIIIIIIIIIII
IIIIIIYIIII', 'IIIIIIIIIIIIIIIIIIIIZIIII'],
              coeffs=[1.+0.j, 1.+0.j, 1.+0.j])
```

Estimator Containers

Helper container classes

- `ObservablesArray` and `BindingsArray` are new helper classes for manipulating these arrays
  - `ObservablesArray` is like a NumPy array of `SparsePauliOp` observables
  - `BindingsArray` is like a NumPy array for named parameter value sets
  - Have convenience methods like (eg reshape)

Disclaimer

These classes are *experimental*, so their API is not guaranteed to be stable

Example

```
from qiskit.primitives.containers.observables_array import (
    ObservablesArray
)

# Shape (3, 1) list of Pauli array
obs = ObservablesArray(["XX", "YY", "ZZ"]).reshape(3,1)

# Apply ISA circuit layout
isa_obs = obs.apply_layout(isa_par_bell.layout)
```

```
ObservablesArray([[{'IIIIIIIIIIIIIIIIIIIIIIIIXX': 1}]
                  [{'IIIIIIIIIIIIIIIIIIIIIIIIYY': 1}]
                  [{'IIIIIIIIIIIIIIIIIIIIIIIIZZ': 1}]], shape=(3, 1))
```

```
from qiskit.primitives.containers.bindings_array import (
    BindingsArray
)

BindingsArray({tuple(isa_par_bell.parameters): param_vals})
```

```
BindingsArray(<shape=(20,), num_parameters=1, parameters=['θ']>)
```

Measurement Observables

Decomposing Measurement Observables

- IBM primitives apply Pauli grouping to minimize the number of measurement circuits
- Groups Pauli's that can be computed from marginals of a single measurement
- Grouping is done via the `PauliList.group_commuting(qubit_wise=True)`

```
paulis = qi.PauliList([
    "ZII", "ZII", "IIZ",
    "XII", "IXI", "IIX",
    "YII", "IYI", "IIY"]
)
paulis.group_commuting(qubit_wise=True)
```

```
[PauliList(['XII', 'IIZ', 'IXI']),
 PauliList(['YII', 'IIX', 'IYI']),
 PauliList(['ZII', 'ZII', 'IIY'])]
```

Component Expectation Values

- When evaluating a Hamiltonian $H = \sum_i \alpha_i P_i$ you can also include the individual P_i term expectation values.
- This will incur no additional QPU time, and a minor increase in classical post-processing time

```
# Want to estimate expval of
H = qi.SparsePauliOp(["ZII", "IZI", "IIZ"], 1/3)

# To get access to component terms as well we should set
obs = ObservablesArray([H, *H.paulis])
obs
```

```
ObservablesArray([{'ZII': 0.3333333333333333, 'IZI': 0.333333
3333333333, 'IIZ': 0.3333333333333333}
                  {'ZII': 1} {'IZI': 1} {'IIZ': 1}], shape=
(4,))
```

IBM Runtime Primitives Options

- The Sampler and Estimator API described allows creation of portable programs
- The IBM runtime primitives support additional functionality which can be configured by *options*
- Enables automatic error *suppression* and *mitigation* method

Sampler Options

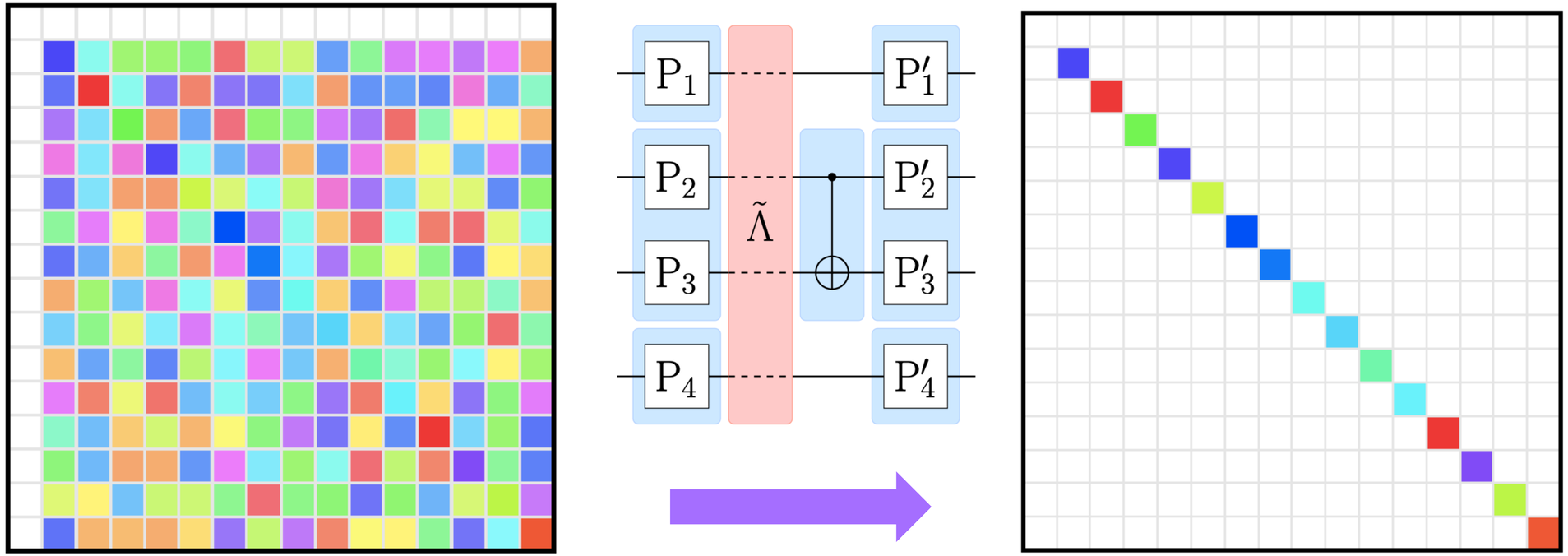
- `default_shots`: the default shots when running pubs
- `dynamical_decoupling`: sub-options for enabling and configuring automatic dynamical decoupling (DD)
- `twirling`: sub-options for enabling and configuring automatic Pauli-twirling

Estimator Options

- `default_precision`: The default precision when running pubs
- `default_shots`: A proxy for setting default precision in terms of shots
- `resilience_level`: Enable preset resilience options.
- `dynamical_decoupling`: sub-options for enabling and configuring automatic dynamical decoupling (DD)
- `twirling`: sub-options for enabling and configuring automatic Pauli-twirling
- `resilience`: sub-options for enabling and configuring error mitigation methods.

Pauli Twirling

Pauli twirling converts a general noise channel into a Pauli error channel.



General error channel

Pauli error channel

- Requires *transpiling* circuit to identify layers of twirled 2-qubit gates or measurements.
- *Reparameterizes* the circuit to allow *parametrically inserting* random Paulis
- Samples randomly Pauli's to insert into the circuit in a specific way
- Averages (Estimator) or concatenates (Sampler) the results over samples

Twirling Options

Twirling is controlled via the following twirling options

- `enable_gates`: Whether to apply 2-qubit gate twirling
- `enable_measure`: Whether to enable twirling of measurements
- `num_randomizations`: The number of random samples to use when twirling or peforming sampled mitigation
- `shots_per_randomization`: The number of shots to run for each random sample
- `strategy`: Specify the strategy of twirling qubits in identified layers of 2-qubit twirled gates

```python
# Set options after initializing
twirl_sampler = SamplerV2(backend)
twirl_sampler.options.twirling.enable_gates = True
twirl_sampler.options.twirling.enable_measure = True
twirl_sampler.options.twirling.strategy = "active-circuit"
```

```python
# Pass-options during initialization
options = {
    "twirling": {
        "enable_gates": True,
        "enable_measure": True,
        "strategy": "active-circuit",
    },
}
twirl_sampler = SamplerV2(backend, options=options)
```

Default Values

If default values the number of shots for the pub will be divided across 32 randomizations.

If values are set for `num_randomizations` and `shots_per_randomization` these values must be >= the shots for all executed pubs.

Twirling Strategy

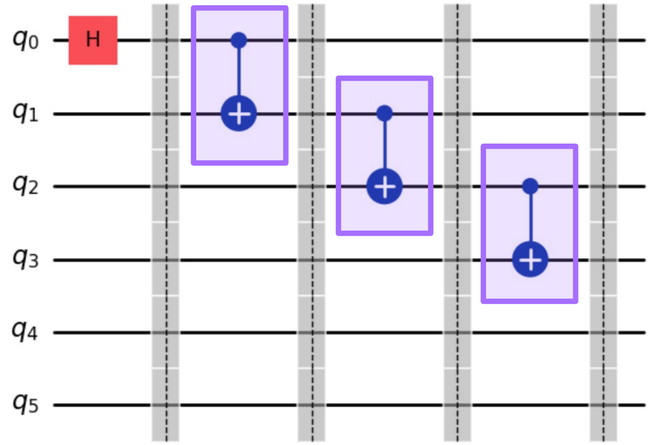
The following twirling strategies are supported

- `active`: only twirl gate qubits in each layer
- `active-accum`: twirl the accumulated gate qubits up to and including the current layer
- `active-circuit`: in each layer twirl the union of all gate qubits in the whole circuit
- `all`: in each layer twirl all qubits in the circuit, including idle qubits

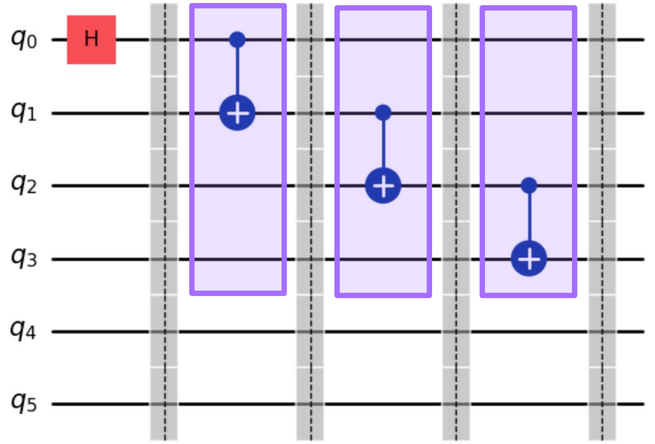
```python
# Set options after initializing
twirl_sampler = SamplerV2(backend)
twirl_sampler.options.twirling.enable_gates = True
twirl_sampler.options.twirling.enable_measure = True
twirl_sampler.options.twirling.strategy = "active-circuit"
```

```python
# Pass-options during initialization
options = {
    "twirling": {
        "enable_gates": True,
        "enable_measure": True,
        "strategy": "active-circuit",
    },
}
twirl_sampler = SamplerV2(backend, options=options)
```

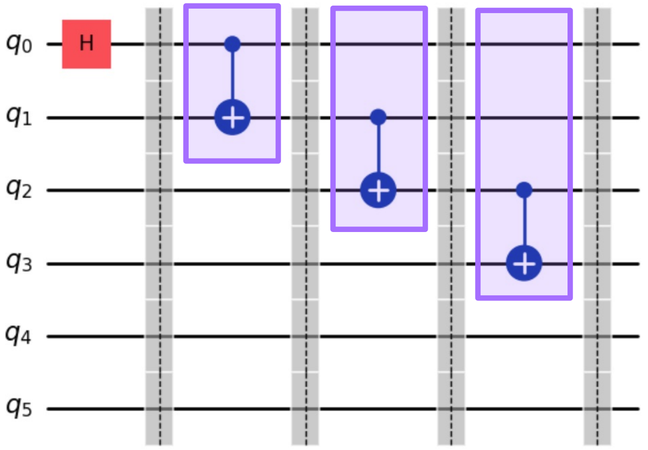
active



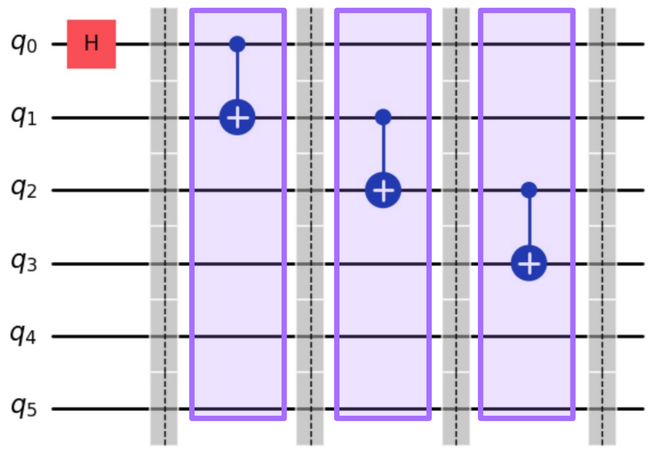
active-circuit



active-accum



all



Estimator Resilience Sub-Options

The IBM Runtime Estimator primitives supports built-in error mitigation methods

These can be enabled via the following `resilience` options

- `measure_mitigation`: Whether to enable measurement error mitigation method
- `zne_mitigation`: Whether to turn on Zero Noise Extrapolation error mitigation method
- `pec_mitigation`: Whether to turn on Probabilistic Error Cancellation error mitigation method

These will be covered more in later sessions.

There are also sub-options for additional configuration of error mitigation:

- `zne`: Additional options for configuring ZNE mitigation
- `pec`: Additional options for configuring PEC mitigation
- `measure_noise_learning`: Additional options for noise learning for measure mitigation
- `layer_noise_learning`: Additional options for noise learning for PEC mitigation

Estimator Resilience Levels

The runtime Estimator has built in *resilience levels* to automate option configuration

Option	0	1	2
Measure Twirling	False	True	True
Measure Mitigation	False	True	True
Gate Twirling	False	False	True
ZNE Mitigation	False	False	True

Resilience Level Definitions

Default resilience levels can further be customized using the individual options

```
# Initialize estimator for ZNE
estimator = EstimatorV2(
backend, options={"resilience_level": 2})
```

```
# Initialize and customize an estimator for ZNE
estimator = EstimatorV2(backend)
estimator.options.resilience_level = 2
estimator.options.dynamical_decoupling.enable = True
estimator.options.dynamical_decoupling.sequence_type = "XY4"
estimator.options.resilience.zne.extrapolator = "linear"
```

Additional Resources

For additional details see the Qiskit Runtime documentation

Qiskit Docs and Tutorials
https://docs.quantum.ibm.com/

Qiskit Primitives Docs
https://docs.quantum.ibm.com/run/primitives

Qiskit Runtime API Docs
https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/runtime_service