Efficient Implementation of Circuits on the QSCOUT Hardware through “Batching”

Presented by
Christopher G. Yale
Running Circuits on the Hardware

1) User builds their desired Jaqal circuits via JaqalPaq and/or SuperstaQ in an *.ipynb
2) (Users can also construct new gates and pulses using JaqalPaw)
3) The code is uploaded to the JAF network service running through a Docker container
4) These circuits are sent to the hardware Octet, interpreted and then translated into laser pulses
5) Laser pulses are streamed out and quantum circuits are performed on the ions in the trap
6) Measurements are recorded in the Jaqal Application Framework and returned to the user’s notebook

More on “batching”:

More on Octet and compiling:
Communication and Compile Time

- Communicating and compiling a single circuit takes about 1-2 s
- However, when we consider a large number of circuits, this adds a lot of overhead

Example:
- Let’s say we want to do a variational quantum eigensolver (VQE) with some randomized compiling!
  - Circuit runtime per shot: 4 ms
  - 200 shots
  - 10 RCs per variational parameter
  - 3 Hamiltonian projections
  - Sweep 41 variational parameters

- If we sent a single circuit over the channel at a time...
  - Runtime = 41 param*3 proj*10 RCs*200 s*4 ms = 984 s = 16:24 min
  - Com-time = 41 param*3 proj*10 RCs*1 s = 1230 s = 20:30 min

Example inspired by: arXiv:2205.14225
Basic premise: Send a batch of circuits over the pipeline instead of one at a time

- In practice, this has previously involved our experimental team rewriting users notebooks to take advantage of this

- Coming soon → a new Jaqal API that allows you to mimic the behavior of batching circuits to the experiment, but on the emulator!

- 3 “flavors” of batching available!

`.run_jaqal_string(code_string,...)`
`.run_jaqal_circuit(code_circuit,...)`

These calls works on both the emulator & experiment!
Use Case: You have a series of circuits which vary in the types of gates and angles, but they’re on the “shorter” side and/or consist of a “reduced” set of gates.

Solution: Place multiple “prepare_all/gates/measure_all” blocks in your code. The experiment and emulator will sort through them in order:

```
result = run_jaqal_string(code_string)
```

Note: This already exists in the current Jaqal API!
Batching with Overrides

**Use Case:** You have a circuit which will always consist of the same set of gates, but you want to vary all of the different phases and rotation angles.

```python
from qscout.v1.std usepulses

let alpha 0.1701
let beta 0.1701
let gamma 0.74205
let delta 0.74656

register q[4]

prepare_all
Rx q[0] alpha
Ry q[2] beta
MS q[2] q[3] 0 gamma
MS q[3] q[0] 0 delta
measure_all

Solution: Create an override dictionary which gets sent along with your circuit to the emulator/experiment

```python
override_dict = {
    "gamma": [1.57079, 0.78539, 0.39269, 0.19634],
    "delta": [0.19634, 0.39269, 0.78539, 1.57079]
}

result = run_jaqal_string(code_string, overrides = override_dict)
```

Note: Each dictionary entry must be an array of the same length or a scalar.
Other Things to Override

The override dictionary has a couple of other powerful tools that might be of interest! (Talk to us to learn even more!)

Not only can you override your “let” parameters in your Jaqal file, but you can also override:

**JaqalPaw a.k.a. pulse definition parameters:**

```{"pd.amp0": [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100]}
```

*Why? Users interested in crafting their own gates via JaqalPaw may want to see gates’ performance under a variety of conditions.*

**Number of shots per circuit:**

```{"__repeats__": [100, 200, 500, 1000, 2000]}
```

*Why? To give more weight to certain types of circuits.*

**Running subcircuits in a certain order:**

```{"__index__": [[1, 3, 0, 2, 4]]} ← NESTED LIST...BEWARE!
```

*Why? Randomizing the circuit’s operational order. Also allows for repeated calls to a particular subcircuit (see next slides for use).*
Batching with Indexing

**Use Case:** Randomizing the order in which circuits are performed and/or repeating certain circuits

```python
from qscout.v1.std usepulses

register q[4]

prepare_all
Px q[0]
MS q[0] q[1]
Px q[1]
measure_all

prepare_all
Sxd q[1]
MS q[1] q[3]
Sy q[3]
measure_all

prepare_all
Sx q[2]
MS q[2] q[3]
Sy q[3]
measure_all

prepare_all
MS q[0] q[2]
measure_all

Solution: Provide a nested list inside your override dictionary with the key “__index__”

```python
override_dict = {"__index__": [[1,3,2,0]]}
```
Combining Batching Approaches

Overrides + Subcircuits:
Runs all subcircuits for a particular set of overrides, then runs all subcircuits for the next set of overrides...

override_dict = {“gamma”: [.74205,1.57079,0.1701]}

result = run_jaqal_string(code_string,overrides = override_dict)

Note: Overrides + Indexing acts similarly, running all indexed subcircuits before moving onto next set of overrides
Data Object Structure

When I call “run_jaqal_string” on the emulator/experiment, what does the return look like?

```python
result = run_jaqal_string(code_string, overrides = override_dict)
```

**Important Caveat! This structure applies to the new Jaqal API, coming soon!**

All circuits within a subbatch

```python
result.by_subbatch[i]
```

Select a subcircuit within the subbatch

```python
result.by_subbatch[i].by_subcircuit[j]
```

(Emulator Only) Absolute multi-qubit state probabilities sorted by integer

```python
result.by_subbatch[i].by_subcircuit[j].simulated_probability_by_int
```

(Emulator) state probs w/shot noise or (Experiment) actual data sorted by integer

```python
result.by_subbatch[i].by_subcircuit[j].relative_frequency_by_int
```

(Emulator Only) Absolute multi-qubit state probabilities sorted by string

```python
result.by_subbatch[i].by_subcircuit[j].simulated_probability_by_str
```

(Emulator) state probs w/shot noise or (Experiment) actual data sorted by string

```python
result.by_subbatch[i].by_subcircuit[j].relative_frequency_by_str
```

(Emulator) absolute probabilities (Experiment) actual data

```python
result.by_subbatch[i].by_subcircuit[j].probability_by_int or _by_str
```
Instead of the data ordered by subbatches and subcircuits, we can also access the data in the time order it was taken.

String Ordering:

\[
\begin{align*}
q[0] &= 0 \\
q[1] &= 1 \\
q[2] &= 1
\end{align*}
\]

\[
|011\rangle
\]

\[
\{'000': 0.017, '100': 0.008, '010': 0.021, '110': 0.112, '001': 0.231, '101': 0.000, '011': 0.541, '111': 0.070\}
\]

Integer Ordering:

\[
\begin{align*}
q[0] &= 0 \\
q[1] &= 1 \\
q[2] &= 1
\end{align*}
\]

\[
0b110 = 7
\]

\[
[0.017, 0.008, 0.021, 0.112, 0.231, 0.000, 0.541, 0.070]
\]

\[
result.by_time[k].relative_frequency_by_int (and all the others)
\]
Pulse-Level Control Using Jaqal Pulses and Waveforms (JaqalPaw)

Daniel Lobser
References


Daniel Lobser,1,* Joshua Goldberg,1 Andrew J. Landahl,1 Peter Maunz,1,2
Benjamin C. A. Morrison,1 Kenneth Rudinger,1 Antonio Russo,1 Brandon
Ruzic,1 Daniel Stick,1 Jay Van Der Wall,1 and Susan M. Clark1

1Sandia National Laboratories, Albuquerque, New Mexico 87123, USA
2Currently at IonQ, College Park, Maryland 20740, USA

(Dated: April 19, 2021)

https://qscout.sandia.gov
Experimental Details of Gate Implementation

**Basic JaqalPaw:** Simple waveforms

**Advanced JaqalPaw:** Experimentally meaningful waveforms
Realizing Quantum Gates

Gates specified in Jaqal must be converted to a form that is experimentally realizable.

The internal quantum states of individually-addressed ions are manipulated via laser light passed through acousto-optic modulators (AOMs).

Each AOM is modulated with an rf waveform to precisely tune the frequency, phase, and amplitude of the light. These waveforms are specified using JaqalPaw (“Jaqal Pulses and Waveforms”).
Gate Implementation at the Pulse Level

$^{171}$Yb$^+$ qubit, clock state 12.6 GHz

Individual addressing requires lasers

Optical frequency comb to bridges 12.6 GHz via Raman transitions

Frequency, phase, and amplitude control using RF signals applied to acousto-optic modulators (AOMs)

Two configurations: Co- and Counter-propagating

Co-propagating

- Immune to Doppler shifts
- Not affected by timing errors and pulse overlap

Counter-propagating

- Supports motional-state addressing and ground state cooling
- Affected by Doppler shifts
- Necessary for two-qubit gates
Two-qubit Mølmer-Sørensen Gates

Common motional modes

“Fully connected”
Two-qubit Mølmer-Sørensen Gates
Experimental Details of Gate Implementation

Basic JaqalPaw: Simple Waveforms

Advanced JaqalPaw: Experimentally meaningful waveforms
JaqalPaw in Broad Strokes

JaqalPaw is a package that relies on a small set of conventions using pure Python.
JaqalPaw in Broad Strokes

JaqalPaw is a package that relies on a small set of conventions using pure Python.

Gate definitions are defined in a class.
They can derive from other gate definition classes.
JaqalPaw is a package that relies on a small set of conventions using pure Python.

Gate definitions are defined in a class.

They can derive from other gate definition classes.

Calibration data will be exposed as annotated class variables.
JaqalPaw is a package that relies on a small set of conventions using pure Python.

Gate definitions are defined in a class.

They can derive from other gate definition classes.

Calibration data will be exposed as annotated class variables.

Arbitrary helper functions allowed.
JaqalPaw is a package that relies on a small set of conventions using pure Python.

```
class MyGatePulses(QSCOUTBuiltins):
    some_calibrated_parameter: float = 200e6  # Hz

    @staticmethod
    def gauss(A, sigma, num_points):
        x = np.linspace(-1, 1, num_points)
        return tuple(np.sqrt(A*np.exp(-x**2/2/sigma**2)))

    def gate_GaussPulse(self, qubit, sigma):
        return [PulseData(...), ...]
```

Calibration data will be exposed as annotated class variables.

Gate definitions are defined in a class.

They can derive from other gate definition classes.

Arguments after “self” are passed in from Jaqal:

GaussPulse q[2] 3.8
JaqalPaw in Broad Strokes

JaqalPaw is a package that relies on a small set of conventions using pure Python.

Gate definitions are defined in a class.

They can derive from other gate definition classes.

Calibration data will be exposed as annotated class variables.

Arbitrary helper functions allowed.

Gates exposed at to Jaqal must have names that start with “gate_”.

Gates must return a list of “PulseData” objects. Objects targeting the same qubit are run back to back and objects targeting different qubits are run in parallel.

PulseData objects are simply a collection of parameters that define the shape and behavior of a waveform.

Class MyGatePulses(QSCOUTBuiltins):

some_calibrated_parameter: float = 200e6  # Hz

@staticmethod
def gauss(A, sigma, num_points):
    x = np.linspace(-1, 1, num_points)
    return tuple(np.sqrt(A*np.exp(-x**2/2/sigma**2)))

def gate_GaussPulse(self, qubit, sigma):
    return [PulseData(...), ...]

Arguments after “self” are passed in from Jaqal:

GaussPulse q[2] 3.8
**The PulseData Object**

PulseData objects are the primary building blocks for constructing gates. They are specific to output channels on hardware, addressing either an individual qubit, or all qubits if the global beam is specified.

```python
PulseData(channel,   # output channel
dur,               # total duration to apply parameters (s)
freq0=0,           # tone 0 frequency (Hz)
phase0=0,          # tone 0 phase (deg.)
amp0=0,            # tone 0 amplitude (arb.)
freq1=0,           # tone 1 frequency (Hz)
phase1=0,          # tone 1 phase (deg.)
amp1=0,            # tone 1 amplitude (arb.)
framerot0=0,       # frame 0 virtual rotation (deg.)
framerot1=0,       # frame 1 virtual rotation (deg.)
# metadata parameters (XXX_mask indicates per-tone settings)
sync_mask=0b00,    # synchronize phase for current frequency
enable_mask=0b00,  # toggle the output enable state
fb_enable_mask=0b00, # enable frequency correction
apply_at_end_mask=0b00, # apply frame rotation at end of pulse
rst_frame_mask=0b00, # reset accumulated frame rotation
fwd_frame0_mask=0b00, # forward frame 0
fwd_framel_mask=0b00, # forward frame 1
inv_frame0_mask=0b00, # invert frame 0 sign
inv_framel_mask=0b00, # invert frame 1 sign
waitrig=False)      # wait for external trigger
```

**FIG. 1:** Full argument signature of PulseData.
The PulseData Object

PulseData objects are the primary building blocks for constructing gates. They are specific to output channels on hardware, addressing either an individual qubit, or all qubits if the global beam is specified.

Always requires channel and duration.

```python
PulseData(channel, dur,
          freq=0, phase=0, amp=0,
          freq1=0, phase1=0, amp1=0,
          framerot0=0, framerot1=0,
          # metadata parameters (XXX_mask indicates per-tone settings)
          sync_mask=0b00, enable_mask=0b00, fb_enable_mask=0b00,
          apply_at_end_mask=0b00, rst_frame_mask=0b00,
          fwd_frame0_mask=0b00, fwd_frame1_mask=0b00, inv_frame0_mask=0b00,
          inv_frame1_mask=0b00, waittrig=False)
```

FIG. 1: Full argument signature of `PulseData`.
The PulseData Object

PulseData objects are the primary building blocks for constructing gates. They are specific to output channels on hardware, addressing either an individual qubit, or all qubits if the global beam is specified. Always requires channel and duration.

Frequency, phase, amplitude can be constant-valued, have multiple discrete updates (lists), or continuous spline modulation (tuples).

```python
PulseData(channel,  # output channel
dur,  # total duration to apply parameters (s)
freq0=0,  # tone 0 frequency (Hz)
phase0=0,  # tone 0 phase (deg.)
amp0=0,  # tone 0 amplitude (arb.)
freq1=0,  # tone 1 frequency (Hz)
phase1=0,  # tone 1 phase (deg.)
amp1=0,  # tone 1 amplitude (arb.)
framerot0=0,  # frame 0 virtual rotation (deg.)
framerot1=0,  # frame 1 virtual rotation (deg.)
# metadata parameters (XXX_mask indicates per-tone settings)
sync_mask=0b00,  # synchronize phase for current frequency
enable_mask=0b00,  # toggle the output enable state
fb_enable_mask=0b00,  # enable frequency correction
apply_at_end_mask=0b00,  # apply frame rotation at end of pulse
rst_frame_mask=0b00,  # reset accumulated frame rotation
fwd_frame0_mask=0b00,  # forward frame 0
fwd_frame1_mask=0b00,  # forward frame 1
inv_frame0_mask=0b00,  # invert frame 0 sign
inv_frame1_mask=0b00,  # invert frame 1 sign
waitrig=False)  # wait for external trigger
```

FIG. 1: Full argument signature of `PulseData`.
The PulseData Object

PulseData objects are the primary building blocks for constructing gates

They are specific to output channels on hardware, addressing either an individual qubit, or all qubits if the global beam is specified

Always requires channel and duration

Frequency, phase, amplitude can be constant-valued, have multiple discrete updates (lists), or continuous spline modulation (tuples)

Z rotations are done virtually

FIG. 1: Full argument signature of PulseData.
PulseData objects are the primary building blocks for constructing gates.

They are specific to output channels on hardware, addressing either an individual qubit, or all qubits if the global beam is specified.

Always requires channel and duration.

Frequency, phase, amplitude can be constant-valued, have multiple discrete updates (lists), or continuous spline modulation (tuples).

Z rotations are done virtually.

Metadata inputs are tied to the PulseData object and can only be single-valued.

```
PulseData(channel, dur, freq0=0, phase0=0, amp0=0, freq1=0, phase1=0, amp1=0, framerot0=0, framerot1=0),

# metadata parameters (XXX_mask indicates per-tone settings)
sync_mask=0b00, enable_mask=0b00, fb_enable_mask=0b00, apply_at_end_mask=0b00, rst_frame_mask=0b00,
fwd_frame0_mask=0b00, fwd_frame1_mask=0b00, inv_frame0_mask=0b00, inv_frame1_mask=0b00,
waittrig=False)
```

FIG. 1: Full argument signature of `PulseData`.
The PulseData Object

PulseData objects are the primary building blocks for constructing gates. They are specific to output channels on hardware, addressing either an individual qubit, or all qubits if the global beam is specified. Always requires channel and duration.

Frequency, phase, amplitude can be constant-valued, have multiple discrete updates (lists), or continuous spline modulation (tuples).

Z rotations are done virtually.

Metadata inputs are tied to the PulseData object and can only be single-valued.

```
PulseData(channel, dur, # output channel
dur,
freq0=0, # total duration to apply parameters (s)
phase0=0,
amp0=0,
freq1=0,
phase1=0,
amp1=0,
framerot0=0,
framerot1=0,
# metadata parameters (XXX_mask indicates per-tone settings)
sync_mask=0b00,
enable_mask=0b00,
fb_enable_mask=0b00,
apply_at_end_mask=0b00,
rst_frame_mask=0b00,
fwd_frame0_mask=0b00,
fwd_frame1_mask=0b00,
inv_frame0_mask=0b00,
inv_frame1_mask=0b00,
waittrig=False)
```

FIG. 1: Full argument signature of PulseData.
Discrete and Spline Modulations

Discrete updates are represented as a list [...], Spline updates are represented as a tuple (...)

Updates are equally distributed over the duration of the pulse (non-uniform time distribution of spline/discrete updates is not currently supported)

Note that N-1 segments are used in a spline, while N segments are used for discrete updates

Type aliases “Spline” and “Discrete” will be provided
Piecewise Operations

PulseData objects are run back to back when on the same channel. This also applies to gates in general.

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
                      amp0=(0,9,41,50)),
            PulseData(qubit, 2e-6,
                      amp0=50),
            PulseData(qubit, 2e-6,
                      amp0=(50,0))]
```

EX. 8: Piecewise functions can be constructed by chaining PulseData objects together.
Piecewise Operations

PulseData objects are run back to back when on the same channel.

This also applies to gates in general.

**New feature has been implemented** to simplify this notation:

- Different modulation types are nested in a list.
- Each list entry is subdivided in time.

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
            ampθ=(0,9,41,50)),
            PulseData(qubit, 2e-6,
            ampθ=50),
            PulseData(qubit, 2e-6,
            ampθ=(50,0))]
```

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 6e-6,
            ampθ=[(0,9,41,50), 2 μs
            50, 2 μs
            (50, 0)]), 2 μs
```
PulseData objects are run back to back when on the same channel.

This also applies to gates in general.

**New feature has been implemented** to simplify this notation:

- Different modulation types are nested in a list
- Each list entry is subdivided in time

Lists can contain scalar values, lists, or tuples.

Tuples can only contain scalar values.

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
                      ampθ=(0,9,41,50)),
            PulseData(qubit, 2e-6,
                      ampθ=50),
            PulseData(qubit, 2e-6,
                      ampθ=(50,0))]
```

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 6e-6,
                      ampθ=[(0,9,41,50), 2 μs
                             50, 2 μs
                             (50, 0)]) 2 μs]
```

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 6e-6,
                      ampθ=[(0,9,41,50), 2 μs
                             [50, 40, 0.2 μs, 0.2 μs
                             (30,20,30), 0.2 μs
                             40, 50], 0.2 μs
                             (50, 0)]) 2 μs]
```
Running a Gate Across Multiple Channels

PulseData objects on different channels are run in parallel. This always applies to PulseData objects in the same gate. This optionally applies to gates run in parallel if run on different channels, e.g. in Jaqal:

\[ \langle G1 \ q[2] \ | \ G2 \ q[3] \rangle \]

Mismatched durations are automatically padded with NOPs at the end of the pulse.

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
                       amp0=(0,9,41,50)),
            PulseData(qubit, 2e-6,
                       amp0=50),
            PulseData(qubit, 2e-6,
                       amp0=(50,0)),
            PulseData(GLOBAL_BEAM, 4.5e-6,
                       amp0=(0,30,20,70))]
```

EX. 9: Chaining `PulseData` objects on different channels results in parallel execution. Differences in cumulative duration will be padded with a NOP pulse.
Experimental Details of Gate Implementation

Basic JaqalPaw: Simple Waveforms

Advanced JaqalPaw: Experimentally meaningful waveforms
Frame Rotations

Sometimes referred to as “Virtual Rotations” or “Z Rotations”

QSCOUT doesn’t support direct Z rotations, but gate sequences can reflect effective Z rotations:

\[ S_x S_z S_x \rightarrow S_y S_x \]

The Octet hardware used by QSCOUT implements virtual rotations natively by tracking the qubit frame with a separate phase:

\[ \sin(2\pi ft + \varphi + \varphi_z) \]

Frame rotations are cumulative and apply to subsequent gates until the frame is explicitly reset.

Frame rotations take scalar, discrete, and spline inputs.

Spline inputs accumulate only the final value.

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 1e-6,
                      framerot0=10)
            for _ in range(3)]

def gate_G(self, qubit):
    return [PulseData(qubit, 3e-6,
                      framerot0=[10,10,10]
                      )]

EX. 10: Frame rotation inputs are equivalent to phase, but their values accumulate.
```

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 1e-6,
                      framerot0=15),
            PulseData(qubit, 3e-6,
                      framerot0=(0,10,-10,-5)),
            PulseData(qubit, 1e-6)] # NOP

EX. 13: Frame rotations support spline inputs. Only the final value of the spline is added to the accumulator.
```
Frame Forwarding and Inversion

Two frames are supplied, but each frame is common to the qubit and must be forwarded to tones as needed.

Single-qubit co-propagating gates must have the frame forwarded to a single tone.

For a two-qubit Mølmer-Sørensen gate, both red and blue sideband inputs must have the frame forwarded.

Optionally, sign of phase can be inverted for special gate configurations.

Frame forwarding and inversion is controlled via metadata, which uses a bitmask convention.

<table>
<thead>
<tr>
<th>Input</th>
<th>Tone 1</th>
<th>Tone 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0b00</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0b01</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>0b10</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>0b11</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 1e-6,
                      framerot0=15,
                      fwd_frame0_mask=0b01,
                      inv_frame0_mask=0b00),
            PulseData(qubit, 1e-6,
                      framerot0=15,
                      fwd_frame0_mask=0b10,
                      inv_frame0_mask=0b10),
            PulseData(qubit, 1e-6,
                      framerot0=15,
                      fwd_frame0_mask=0b11,
                      inv_frame0_mask=0b01)]
```
Challenges: Shimming Out Errors

Frequency comb is not actively stabilized at the source!

Small variations in the comb spacing require dynamic corrections to stay on resonance.

Beat note lock must be applied to only one of the two tones contributing to a Raman transition.

Lock is set using the fb_enable_mask input in PulseData.

Lock should be applied to the lower frequency tones.

This parameter will be different in certain cases, for example single-qubit co-propagating gates and two-qubit gates.
Three basic configurations

<table>
<thead>
<tr>
<th>Individual beams</th>
<th>Global beam</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sideband cooling</strong></td>
<td></td>
</tr>
<tr>
<td>State initialization</td>
<td></td>
</tr>
<tr>
<td><strong>Single qubit gates</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Two-Qubit Gate</strong></td>
<td></td>
</tr>
</tbody>
</table>
Challenges: RF Reproducibility and Agility

Three basic configurations

- **Sideband cooling**
  - State initialization

- **Single qubit gates**

- **Two-Qubit Gate**

*Lock must be applied to exactly one tone for each Raman pair!*
Challenges: RF Reproducibility and Agility

Three basic configurations

- **Sideband cooling**
  - State initialization

- **Single qubit gates**
  - Beat Note
  - Lock

- **Two-Qubit Gate**
  - Global beam

*Lock must be applied to exactly one tone for each Raman pair!*
Challenges: RF Reproducibility and Agility

Three basic configurations

- **Sideband cooling**
- **State initialization**
- **Single qubit gates**
- **Two-Qubit Gate**

**Locked must be applied to exactly one tone for each Raman pair!**

```python
PulseData(channel, # output channel
dur, # total duration to apply parameters (s)
freq0=0, # tone 0 frequency (Hz)
phase0=0, # tone 0 phase (deg.)
amp0=0, # tone 0 amplitude (arb.)
freq1=0, # tone 1 frequency (Hz)
phase1=0, # tone 1 phase (deg.)

fb_enable_mask=0b00, # enable frequency correction
framerot1=0, # frame 1 virtual rotation (deg.)
# metadata parameters (XXX_mask indicates per-tone settings)
sync_mask=0b00, # synchronize phase for current frequency
enable_mask=0b00, # toggle the output enable state

fb_enable_mask=0b00, # enable frequency correction
apply_at_end_mask=0b00, # apply frame rotation at end of pulse
rst_frame_mask=0b00, # reset accumulated frame rotation
fwd_frame0_mask=0b00, # forward frame 0
fwd_frame1_mask=0b00, # forward frame 1
inv.frame0_mask=0b00, # invert frame 0 sign
inv.frame1_mask=0b00, # invert frame 1 sign
wait trig=False) # wait for external trigger
```

**FIG. 1:** Full argument signature of PulseData.
Challenges: RF Reproducibility and Agility

Three basic configurations

<table>
<thead>
<tr>
<th>Individual beams</th>
<th>Global beam</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sideband cooling</strong></td>
<td></td>
</tr>
</tbody>
</table>
| State initialization
|                  |
| **Single qubit gates** |
| **Two-Qubit Gate** |

*Absolute phase control is imperative!*

- Each configuration requires different frequencies
- Beat note lock needs to be applied to different tones
- Phase of beat note produced by red and blue sideband tones determines global phase of Mølmer-Sørensen gate
Our Approach to Synchronization

Our synchronization approach assumes all frequencies start at the same time, \( t_0 \), at an arbitrary point in the past.

For absolute phase control, one must apply a synchronization trigger to a pulse by setting a non-zero value in the `sync_mask` argument of a PulseData object.

Synchronization will then set the internal oscillator phase to its free-running equivalent for a given frequency started from \( t_0 \).

Synchronization pulses must be applied for all pulses where phase must be aligned to each other.

The `sync_mask` argument only applies to the beginning of the pulse.

For cases where explicit phase accumulation is desired for a frequency modulated pulse, `sync_mask` might need to be set to 0.
Synchronization Caveats

Complex phase/frequency relationships are subject to rounding errors when converting from floating point values to the 40-bit representations used by the Octet hardware.

\[ f_r + f_b = 2f_{\text{qubit}} \quad \text{where} \quad f_r = f_{\text{qubit}} - f_{\text{SB}} \]
\[ f_b = f_{\text{qubit}} + f_{\text{SB}}. \]

but \[ F_r + F_b \neq 2F_{\text{qubit}} \]

Best bet is to use the \texttt{discretize_frequency} helper function:

```python
sb_freq = discretize_frequency(motional_mode_frequencies[0])
qubit_freq = discretize_frequency(global_aom_frequency)
rsb_freq = qubit_freq - sb_freq
bsb_freq = qubit_freq + sb_freq
```
Input Limits

Standard input limits

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Units</th>
<th>Allowed Range</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>s</td>
<td>$t \in [9.77\text{ ns}, 2684.35456\text{ s}]$</td>
<td>2.4414 ns</td>
</tr>
<tr>
<td>Frequency</td>
<td>Hz</td>
<td>$f \in [-409.6\text{ MHz}, 409.6\text{ MHz}]$</td>
<td>745.0581 $\mu\text{Hz}$</td>
</tr>
<tr>
<td>Phase</td>
<td>Degrees</td>
<td>$\theta \in [-\infty, \infty]$</td>
<td>3.2742e-10 deg.</td>
</tr>
<tr>
<td>Amplitude</td>
<td>Arb.</td>
<td>$\mathbb{R} \in [-100, 100]$</td>
<td>6.1035e-3</td>
</tr>
</tbody>
</table>

**TABLE I**: The fundamental input units for frequency, phase, and amplitude in **PulseData**. Note that the phase input is automatically converted modulo 360 such that $\theta \in [-180^\circ, 180^\circ)$. Amplitude is specified for a single tone, however the sum of the amplitude for two tones on the same channel must obey this range.

Spline input limits are a bit more subtle due to how spline coefficients are mapped to work with the on-chip interpolators. Best approach is to try running your code through jaqalpaw-emulate.
Questions?
QSCOUT Webinar: 
JaqalPaw Example Programs

Presented by
Matthew Chow
March 1, 2023
JaqalPaw Examples Tutorial

Topics:

1. Review of basic modulation.

2. Technical details of writing pulse definitions.
   a) Calling code from Jaqal
   b) Referencing calibration parameters
   c) Frequency discretization and synchronization
   d) Frame rotation metadata settings

3. Handy features
   a) Parameterized pulses
   b) Making use of both Raman beams
   c) Programmatic configuration

4. JHU user code with amplitude modulation
Helpful guide for getting started

Example code (more to come here)

IEEE QSCOUT Manual
JaqalPaw Examples Tutorial

Topics:

1. Review of basic modulation.
2. Technical details of writing pulse definitions.
   a) Calling code from Jaqal
   b) Referencing calibration parameters
   c) Frequency discretization and synchronization
   d) Frame rotation metadata settings
3. Handy features
   a) Parameterized pulses
   b) Making use of both Raman beams
   c) Programmatic configuration
4. JHU user code with amplitude modulation
Example 1: Quick Review of Frequency Modulation

Frequency input is a list. Gives frequency jumps.

Frequency jumps applied at evenly spaced timing intervals.
Example 1b: Single Tone Continuous Frequency Modulation

# Frequency Modulation, simple example:

```python
def gate_ContinuousFM_Microwave(self, channel_mw):
    """ Frequency modulated microwave pulse. """

    resonant_frequency = 42e6
    detuning_knots = [100e3, 20e3, -10e3, 5e3, -10e3, 20e3, 100e3]
    freq_fm0 = tuple([resonant_frequency + d for d in detuning_knots])

    return [PulseData(channel_mw, self.FM_pulse_duration,
        freq0 = freq_fm0,
        amp0 = 100,
        phase0=0
    )]
```

Tuple input gives a cubic spline interpolation.
Example 2: Simultaneous Amplitude and Phase (Gaussian Walsh)

```python
class HelperFunctions:

    @staticmethod
    def gauss(npoints, A, freqwidth=300e3, total_duration=4e-6):
        trange = np.linspace(-total_duration / 2, total_duration / 2, npoints)
        sigma = 1 / (2 * np.pi * freqwidth)
        return A * np.exp(-trange ** 2 / 2 / sigma ** 2)

    gaussian_amps = list(self.gauss(npoints=13, A=100))
    double_gauss = tuple(gaussian_amps * 2)
    phase_steps = [0, 180]

    return [PulseData(channel_global, self.MS_pulse_duration,
                        freq0=self.freq0,
                        amp0=double_gauss,
                        phase0=phase_steps,
                        sync_mask=3,
                        fb_enable_mask=0),
```

... the rest of the pulse data objects just put square pulses on the IA beams for q0 and q1.

Cast to list for convenient comprehension.

Cast to tuple for cubic spline.

List of phase steps for discrete jump.

Example 2 Continued: Gaussian Walsh Gate Waveform
JaqalPaw Examples Tutorial

Topics:

1. Review of basic modulation.

2. Technical details of writing pulse definitions.
   a) Calling code from Jaqal
   b) Referencing calibration parameters
   c) Frequency discretization and synchronization
   d) Frame rotation metadata settings

3. Handy features
   a) Parameterized pulses
   b) Making use of both Raman beams
   c) Programmatic configuration

4. JHU user code with amplitude modulation
Technical Details: Calling Your JaqalPaw Code

I keep my Jaqal code simple, and essentially the same for all the programs I write.

Jaqal code (10 substance lines)

```python
// Import JaqalPaw Code.
from ExemplarPulseDefinitions.ModulatedMSExemplar usepulses *

// Declare variables.
let target1 2
let target2 3
let ms_loops 1

register q[8]

// Prepare - Sideband cool, then Pump to F=0
prepare_all

loop ms_loops {
    Mod_MS q[target1] q[target2]
}

measure_all
```

Responsible for:
- Calling JaqalPaw code within a framework that will run on the experiment apparatus.

JaqalPaw has all the substance

Responsible for:
- Calculating waveform (using calibration parameters if desired).
- Create PulseData objects with waveform information for each channel of pulse.
- Synchronization, feedback, and other technical details.

This code available at qscout.sandia.gov
**Technical Details: Referencing Physical Calibrated Parameters**

- Calibrated parameters are currently contained within QSCOUTBuiltins, which you can import into your jaqalpaw code.

- Note, the values here are **overwritten with calibrated values at run time**. Therefore, you should call parameters by name, not copy the number.

- Disclaimer: This structure of calibration parameters is subject to change.

- If you need access to other parameters, just talk to us and we’ll work with you!

```
class CalibrationParameters:
    """ Class that contains calibrated physical parameters and mapping
    
    # Raman carrier transition splitting and AOM center frequencies.
    global_center_frequency: float = 200e6
    ia_center_frequency: float = 230e6
    adjusted_carrier_splitting: float = 28.6e6

    # Principal axis rotation (relative to Raman k_effective).
    principal_axis_rotation: float = 45.0

    # Motional mode frequencies.
    # Just 2 Ions in this example, list structure extends to N.
    higher_motional_mode_frequencies: list = [-2.556, -2.45e6]
    lower_motional_mode_frequencies: list = [-2.1e6, -2.05e6]

    # Matched pi time for single qubit gates.
    co_ia_resonant_pi_time: float = 30e-6
    counter_resonant_pi_time: float = 4e-6

    # Amplitudes to achieve matched pi times.
    # Amplitude lists are indexed by RFSoC channel. [global,-,q0,q1,-,-,-,-,-]
    amp0_coprop_list: list = [100, 0, 30, 30, 0, 0, 0, 0, 30]
    amp1_coprop_list: list = [100, 0, 30, 30, 0, 0, 0, 0, 30]
    amp0_counterprop: float = 100.0
    amp1_counterprop_list: list = [0, 0, 30, 30, 0, 0, 0, 0, 30]

    # Mølmer Sørensen Gate Parameters
    MS_pulse_duration: float = 1e-6
    MS_delta: float = 0.0
    MS_framerot: float = 0.0
    MS_red_amp_list: list = [0, 0, 35, 30, 0, 0, 0, 0, 35]
    MS_blue_amp_list: list = [0, 0, 33, 27, 0, 0, 0, 0, 33]
```
Technical Details: Math with Frequencies; Synchronization

Math with frequencies: To avoid detrimental rounding errors from discretization, use the jaqalpaw utility function, “discretize_frequency.”

```python
from jaqalpaw.utilities.helper_functions import discretize_frequency

# Convert detuning knots to actual RF drive frequencies. Blue=fm0, Red=fm1
freq_fm0 = tuple([discretize_frequency(self.ia_center_frequency) + discretize_frequency(self.MS_delta) + discretize_frequency(dk) for dk in detuning_knots])
freq_fm1 = tuple([discretize_frequency(self.ia_center_frequency) - discretize_frequency(self.MS_delta) - discretize_frequency(dk) for dk in detuning_knots])
```

Synchronization: Usually, you should synchronize every tone in every pulse. (Syncmask = 3 or 0b11)
Technical Details: Frame rotation metadata

Gaussian MS gate is example of framerot for continuous ACS compensation.

Continuous framerot, natural cubic spline of an erf

Forward to both tones, as red and blue sidebands need same phase for MS gate.

Apply at end mask is ignored for spline framerot
JaqalPaw Examples Tutorial

Topics:

1. Review of basic modulation.

2. Technical details of writing pulse definitions.
   a) Calling code from Jaqal
   b) Referencing calibration parameters
   c) Frequency discretization and synchronization
   d) Frame rotation metadata settings

3. Handy features
   a) Parameterized pulses
   b) Making use of both Raman beams
   c) Programmatic configuration

4. JHU user code with amplitude modulation
Simplest example of a parameterized pulse shape - just sweep the amplitude.

This gate definition is within a class that inherits QSCOUTBuiltins, so the CalibrationParameters are accessible.

CalibrationParameter for the amplitude on the global beam is referenced with self.amp0_counterprop and passed in as a parameter to the pulse definition.

...The rest of the pulse definition defines what happens with the IA beam. (Applies frequency and amplitude modulation to make an FM Gaussian Gate)
Amplitude Sweep Data

7/14/21: 220μs Gaussian FM Gate at -4.68 kHz, Global Amplitude Sweep

Population

0.0 0.2 0.4 0.6 0.8 1.0

0 25 50 75 100 125 150 175 200

j.pd.amp0_counterprop

11 01/10 00
Build in Options by Parameterizing Pulses
Ex 3b: MS Gaussian Peak Height

Slightly more complex example. The same parameter is now passed in as an input to a function defining

```python
class HelperFunctions:
    @staticmethod
def gauss(npoints, A, freqwidth=300e3, total_duration=4e-6):
    t_range = np.linspace(-total_duration/2, total_duration/2, npoints)
    sigma = 1 / (2 * np.pi * freqwidth)
    return A * np.exp(-t_range ** 2 / 2 / sigma ** 2)

class ModulatedMSExemplar(QSCOUTBuiltins, HelperFunctions):
    def gate_Mod_MS(self, channel1, channel2):
        global_amp = self.gauss(npoints=7, A=self.amp0_counterprop)

        listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
                                  freq0=global_beam_frequency,
                                  amp0=global_amp,
                                  phase0=phase_steps,
                                  sync_mask=3,
                                  fb_enable_mask=0),
```

Now we inherit HelperFunctions in addition to QSCOUTBuiltins.

The CalibrationParameter is now passed as an argument to a function that returns amplitude spline knots.
Adding a non-standard parameter can be done by adding an argument to your gate and passing it as a let parameter from your jaqal code.

Jaqal code modification:

```jaqal
let s 0
```

```jaqal
loop ms_loops {
    Mod_MS q[target1] q[target2] s
}
```
Use Both Tones to Generate More Complex Pulses

Ex 4: Track Spin State Dynamics Through a Gaussian FM Gate

Goal: track the spin state during pulse

Since there is already continuous modulation, stopping the pulse at arbitrary time requires recalculation.

Sidestep the problem by putting discrete amplitude modulation on the other leg of the Raman transition.
def gate_Mod_MS(self, channel1, channel2, global_duration=1e6):
    """ General Modulated MS Gate (Produce optimal pulses found by solver). """

    rabi_rate_0 = 0.5 / self.counter_resonant_pi_time
    rabi_fac = 1
    rabi_knots = self.gauss(np.array([13], A=rabi_rate_0, npoints=13, A=rabi_rate_0, npoints=13)
    freqwidth=300e3, total_duration=4e-6)
    amp_scale = [rabi_fac * r for r in rabi_knots]
    amp_scale = np.array(amp_scale)

    if global_duration >= 0 and global_duration < self.MS_pulse_duration:
        global_amp = [self.amp0_counterprop if t <= global_duration
                      else 0 for t in np.linspace(0, self.MS_pulse_duration, 1000)]
    else:
        global_amp = self.amp0_counterprop

    listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
       freq0=global_beam_frequency,
       freq1=self.global_center_frequency,
       amp0=global_amp,
       amp1=0,
       phase0=phase_steps,
       phase1=0,
       sync_mask=0 if dummy_sync else 3,
       fb_enable_mask=0),
       PulseData(channel1, self.MS_pulse_duration,
       freq0=tuple(freq_fm0),
       freq1=tuple(freq_fm1),
       amp0=tuple(self.MS_blue_amp_list[channel1] * amp_scale),
       amp1=tuple(self.MS_red_amp_list[channel1] * amp_scale),
       framerot0 = framerot_input,
       apply_at_eot_mask=framerot_app,
       phase0=0,
       phase1=0,
       sync_mask=0 if dummy_sync else 3,
       fb_enable_mask=1),
       ]
Data For Spin State Tracking

220\(\mu s\) Gaussian FM Gate Dynamics

![Graph showing the population over time for different states.

- Green triangles represent state 00.
- Blue circles represent state 01/10.
- Red diamonds represent state 11.

The x-axis represents j.pd.global_duration (\(\mu s\)) ranging from 0 to 200.
The y-axis represents population ranging from 0.0 to 1.0.](image)
You can write a simple text file parser (or use this one) and include it as a static method.

Nice for leaving a record of what you’ve tried without writing a bunch of JaqalPaw files.

For my gates, I like to let my optimal pulse solver code actually write the config files along with some information about how they got there.
JaqalPaw Examples Tutorial

Topics:

1. Review of basic modulation.

2. Technical details of writing pulse definitions.
   a) Calling code from Jaqal
   b) Referencing calibration parameters
   c) Frequency discretization and synchronization
   d) Frame rotation metadata settings

3. Handy features
   a) Parameterized pulses
   b) Making use of both Raman beams
   c) Programmatic configuration

4. JHU user code with amplitude modulation
John’s Hopkins Users (JHU) JaqalPaw ‘SineLobe’ Code

\[
Z(t) = R_Z \left( \frac{\pi}{2} \right) - R_x(t) - R_z \left( -\frac{\pi}{2} \right)
\]

\[
Y(t) = R_y \left( \frac{\pi}{2} \right) - R_x(t) - R_y \left( -\frac{\pi}{2} \right)
\]

T = 0.010 s

Subcircuit 0

Subcircuit 2
```python
def gate_SineLobes(self, qubit, B, T, max_amplitude=60, correct_distortion=False, flip=False):
    # max_amplitude = MAXAMP * 0.5 # don't consider amplitudes larger than this
    num_spline_points = 25

    """
    pulse volume required to enact a pi pulse
    """
    amp0_scale = self.amp0_coprop_list[self.qubit_mapping[qubit]]
    ampl1_scale = self.amp1_coprop_list[self.qubit_mapping[qubit]]
    t_scale = self.co_ia_resonant_pi_time
    pi_pulse_volume = amp0_scale * ampl1_scale * t_scale

    """
    This is specific to our protocol:
    we require that the pulse volume of a single sine lobe corresponds
    to a rotation which is 2x a root of the first Bessel function,
    of which the first few roots are: 2.40, 5.51, 8.65, ...
    This corresponds to rotations of: 1.53 pi, 3.51 pi, 5.51 pi, ...
    We want to choose the largest such pulse volume which is less than `MAXAMP`
    (or `max_amplitude = MAXAMP / 2` as set above if we want extra buffer room)
    """

    T_lobe = T / B

    # RB change 20220915 - fixed formula for magical amplitude
    bessel_roots = jn_zeros(0, 100) # first hundred roots
    candidate_amp0s = bessel_roots * (B / T) * (amp0_scale * t_scale) * 2
```
JHU SineLobes – Validity checking and catching unphysical requirements

```python
try:
    # select the largest valid amplitude
    if correct_distortion: # MC proposed change 20220810 - validity checking including distortion correction
        corrected_maxamps = amp0_scale*self.DiffractionEfficiencyLinMap(candidate_amp0s, qubit)/self.DiffractiveMax.
        candidate_amp0s = corrected_maxamps[np.where(corrected_maxamps <= max_amplitude)]
    else:
        candidate_amp0s = candidate_amp0s[np.where(candidate_amp0s <= max_amplitude)]
    amp0 = np.max(candidate_amp0s)
    print("Success, Bessel root amp0 found! amp0 = ", amp0) # 20220722 MC addition for debugging
except:
    print("Minimum required amp0 for Bessel root = ", np.min(bessel_roots*(B/T) * (amp0_scale * t_scale;
    raise ValueError("No suitable Dephasing-Robust amplitude found, check your parameters: T={}, B={}

    t = np.linspace(0, 2 * np.pi, num_spline_points)
    sin_amps = np.sin(t) * amp0
```
def gate_SineLobes(self, qubit, B, T, max_amplitude=60, correct_distortion=False, flip=False):
    """ max_amplitude = MAXAMP * 0.5 # don't consider amplitudes larger than this 
    num_spline_points = 25 
"

# MC proposed change 20220810 - apply distortion correction to amp0.
if correct_distortion:
    sin_amps = amp0_scale*self.DiffractionEfficiencyLinMap(sin_amps, qubit)/self.DiffractionEfficiency

# RB proposed change 20220916 - apply waveform twice, positive then negative
if flip:
    sin_amps = -sin_amps
```python
sin_amps = tuple(sin_amps.tolist())

# the sine pulse is repeated `B` times, to fill up the total duration
return [PulseData(self.qubit_mapping[qubit],
                    T_lobes,
                    amp0=sin_amps,
                    ampl=amp1_scale,
                    freq0=self.ia_center_frequency-self.adjusted_carrier_splitting,
                    freq1=self.ia_center_frequency,
                    fb_enable_mask=ton0,
                    fwd_frame0_mask=ton1,  # This is mainly to account for any Stark shifts
                    sync_mask=both_tones,
                    )]  # * B # uncomment this factor to go to a direct gate mode (no Jagal
```

$T = 0.010 \text{ s}$

![DetectionProbability](image1)

![Errorvectormoments](image2)
Questions?
In [30]:
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl
import itertools
from jaqalpaq import run
from jaqalpaq import emulator
from jaqalpaq.run import run_jaqal_file, run_jaqal_string, run_jaqal_batch, run_jaq
# from jaqalpaq.run import * (identical to from jaqalpaq import run aside from need
# from jaqalpaq.run import frontend (needed to switch between emulator and experime
from jaqalpaq.parser import parse_jaqal_string

from jaqalpaq.emulator.unitary import UnitarySerializedEmulator
emulator_backend = UnitarySerializedEmulator()

cmplrcParams['axes.prop_cycle'] = mpl.cycler(color=['g', 'orange', 'darkred', 'b'])

In [31]:
num_qubits = 2
num_states = 1 << num_qubits

Batching with Override Dictionary
# Create a jaqal code (string method)

```python
jaqal_code = f""
// Comment via a double forward slash in jaqal strings

// Pulse Definitions Import Statement
from qscout.v1.std usepulses *

// Define let parameters
let alpha 0.1701
let beta 0.1701
let gamma 0.72405
let delta 0.74656
let epsilon 0.01
let zeta 0.1031
let eta 0.82893
let theta 0.75567
let iota 0.76884
let kappa 0.1701

let num_loops 0
let pi_4 {np.pi/4}

// Select your register of qubits
register q[\{num_qubits\}]

// Create jaqal circuit, starting with prepare_all, ending with measure_all
prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
MS q[0] q[1] eta theta
R q[0] iota kappa
loop num_loops {{
    MS q[0] q[1] 0 pi_4}}
measure_all
"
```
In [33]:

    # Define series of arrays for an 'override' dictionary
    angles = list(np.linspace(-np.pi/2, np.pi/2, 21))
    thetas = [np.pi/2-abs(a) for a in angles]
    loops = list(range(0, 21))

    # Define a python dictionary to be the override dictionary
    # We can override any and/or all let parameters. Caveat: All elements of the dictionary should be set to a value.
    override_dict = {
        "alpha": angles,
        "beta": angles,
        "gamma": angles,
        "delta": angles,
        "epsilon": angles,
        "zeta": angles,
        "eta": angles,
        "theta": thetas,
        "iota": angles,
        "kappa": angles,
        "num_loops": loops,
        "__repeats__": 2000
    }

    # Run the circuit with the parameters being overwritten
    jaqal_circuit = parse_jaqal_string(jaqal_code)

    res = run_jaqal_circuit(jaqal_circuit, overrides=override_dict)

    # The result object that is returned has quite a lot of information stored in it.
    # This object contains the data sorted by each instance of the override dictionary,
    # This subbatch is then further divided into subcircuits (if we had any). If not, just print("PROBABILITIES ORDERED BY QUBIT INTEGER VALUE")
    for i in range(len(angles)):
        print(res.by_subbatch[i].by_subcircuit[0].simulated_probability_by_int) # absolute
        print(res.by_subbatch[i].by_subcircuit[0].relative_frequency_by_int) # with simulation

    # We can also sort the resulting probabilities by their string representation (matrix
    print("PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION")
    for i in range(len(angles)):
        print(res.by_subbatch[i].by_subcircuit[0].simulated_probability_by_str) # absolute
        print(res.by_subbatch[i].by_subcircuit[0].relative_frequency_by_str) # with simulation
PROBABILITIES ORDERED BY QUBIT INTEGER VALUE

[[0.25 0.25 0.25 0.25]
 [0.252 0.268 0.2355 0.2445]
 [0.15521143 0.29205858 0.21796375 0.33476625]
 [0.154 0.291 0.208 0.347]
 [0.0219206 0.20659587 0.32183631 0.44964721]
 [0.0235 0.2205 0.3245 0.4315]
 [0.00710145 0.11161837 0.42340979 0.45787039]
 [0.008 0.1135 0.414 0.4645]
 [0.01493425 0.15583128 0.35569488 0.47353959]
 [0.0145 0.1555 0.346 0.484]
 [0.05334709 0.26830583 0.17991748 0.49842961]
 [0.0595 0.2625 0.1745 0.5035]
 [0.31009929 0.28146769 0.06684163 0.3416814]
 [0.3075 0.278 0.066 0.3485]
 [0.63382386 0.17508098 0.05305083 0.13804433]
 [0.64 0.1675 0.049 0.1435]
 [0.59949211 0.06079955 0.05226754 0.28744081]
 [0.5975 0.065 0.059 0.2785]
 [0.22137474 0.00899478 0.02118271 0.74844777]
 [0.2265 0.009 0.026 0.7385]
 [0. 0. 0. 1.]
 [0.]
 [0.]
 [1.]]
[[0.108417  0.00719196  0.02298553  0.8614055 ]
 [[0.112   ]
 [0.0065   ]
 [0.02   ]
 [0.8615   ]
 [0.32682367  0.03957536  0.07349173  0.56010924]
 [0.3235   ]
 [0.039   ]
 [0.076   ]
 [0.5615   ]
 [0.49195039  0.11809322  0.11003859  0.27993471]
 [0.496   ]
 [0.1135  ]
 [0.111   ]
 [0.2795   ]
 [0.48968353  0.24240519  0.10590413  0.16200716]
 [0.4825   ]
 [0.259   ]
 [0.1865   ]
 [0.152   ]
 [0.32923543  0.3642767  0.08394661  0.22254126]
 [0.335   ]
 [0.3645   ]
 [0.0845   ]
 [0.216   ]
 [0.2368399  0.4094598  0.10206636  0.25163986]
 [0.226   ]
 [0.4015   ]
 [0.1055   ]
 [0.267   ]
 [0.35965133  0.34917237  0.1858558  0.1053205   ]
 [0.3735   ]
 [0.3605   ]
 [0.164   ]
 [0.102   ]
 [0.46527221  0.24565837  0.2827381  0.0062956   ]
 [0.4675   ]
 [0.245   ]
 [0.281   ]
 [0.0065   ]
 [0.3542318  0.20092072  0.3091016  0.13574587]
 [0.374   ]
 [0.1895   ]
 [0.306   ]
 [0.1305   ]
 [0.25  0.25  0.25  0.25   ]
 [0.2305   ]
 [0.2525   ]
 [0.261   ]
 [0.256   ]

PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION

{"00": 0.25, "10": 0.2500000000000001, "01": 0.2499999999999999, "11": 0.25}
{"00": array([0.252]), "10": array([0.268]), "01": array([0.235]), "11": array([0.2445])}
{"00": 0.15521142751307765, "10": 0.2920585780582002, "01": 0.21796374787072248, "11": 0.3347662465579975}
[00': array([0.154]), '10': array([0.291]), '01': array([0.208]), '11': array([0.347])
'00': 0.02192059994387455, '10': 0.20659587429686832, '01': 0.32183631496096606, '11': 0.4496472107829656
'00': array([0.0235]), '10': array([0.2205]), '01': array([0.3245]), '11': array([0.4315])
'00': 0.0071014473257372274, '10': 0.11161837371274047, '01': 0.42340979360775577, '11': 0.457870385337665
'00': array([0.008]), '10': array([0.1135]), '01': array([0.414]), '11': array([0.4645])
'00': 0.14934251424144883, '10': 0.15583127627941548, '01': 0.35569488079623546, '11': 0.47353951950602841
'00': array([0.0145]), '10': array([0.1555]), '01': array([0.346]), '11': array([0.484])
'00': 0.0533478691267966, '10': 0.268305826175841, '01': 0.17991747852752235, '11': 0.49842968384557
'00': array([0.0595]), '10': array([0.2625]), '01': array([0.1745]), '11': array([0.5035])
'00': 0.3100928563221863, '10': 0.2814676878515673, '01': 0.0668416257031315, '11': 0.34168140081308407
'00': array([0.3075]), '10': array([0.278]), '01': array([0.066]), '11': array([0.3485])
'00': 0.6338238629197582, '10': 0.17508089232726085, '01': 0.05305082507570646, '11': 0.1388443296324744
'00': array([0.64]), '10': array([0.1675]), '01': array([0.049]), '11': array([0.1435])
'00': 0.599492105889096, '10': 0.0607995504085978, '01': 0.05226753729546427, '11': 0.287440864087663
'00': array([0.5975]), '10': array([0.065]), '01': array([0.059]), '11': array([0.2785])
'00': 0.22137473696215836, '10': 0.008994782213862722, '01': 0.02118721390980435, '11': 0.7484477684929984
'00': array([0.2265]), '10': array([0.009]), '01': array([0.026]), '11': array([0.7385])
'00': 0.0, '10': 0.0, '01': 0.0, '11': 1.0
'00': array([0.]), '10': array([0.]), '01': array([1.])
'00': 0.10841700433021274, '10': 0.00719164968329998, '01': 0.02298552963651315, '11': 0.8614055801649442
'00': array([0.112]), '10': array([0.0065]), '01': array([0.02]), '11': array([0.8615])
'00': 0.3268236684881671, '10': 0.03957536166136641, '01': 0.07349172604295771, '11': 0.5601092430705888
'00': array([0.3235]), '10': array([0.039]), '01': array([0.076]), '11': array([0.5615])
'00': 0.4919503854411232, '10': 0.11809322170802049, '01': 0.1100385857397468, '11': 0.27991780711180953
'00': array([0.496]), '10': array([0.1135]), '01': array([0.111]), '11': array([0.2795])
'00': 0.4896835251899528, '10': 0.2424051875156598, '01': 0.1059041257031314, '11': 0.16208716133534984
'00': array([0.4825]), '10': array([0.259]), '01': array([0.1065]), '11': array([0.152])
'00': 0.3292354345603979, '10': 0.364276695296637, '01': 0.0839466894867262, '11': 0.22254126073623895
'00': array([0.335]), '10': array([0.345]), '01': array([0.0845]), '11': array([0.216])
{'00': 0.23683398628649616, '10': 0.4094597995646406, '01': 0.10206635751101029, '11': 0.2516398566378528}
{'00': array([0.226]), '10': array([0.4015]), '01': array([0.1055]), '11': array([0.267])}
{'00': 0.35965133071561206, '10': 0.3491723686766947, '01': 0.1858557986438019, '11': 0.10532050196389135}
{'00': array([0.3735]), '10': array([0.3605]), '01': array([0.164]), '11': array([0.102])}
{'00': 0.4652722107982965, '10': 0.24565837429686838, '01': 0.2827738149609607, '11': 0.006295599438744255}
{'00': array([0.4675]), '10': array([0.245]), '01': array([0.281]), '11': array([0.0065])}
{'00': 0.3542317991371078, '10': 0.20092072228347876, '01': 0.30910160364544376, '11': 0.1357458749339696}
{'00': array([0.374]), '10': array([0.1895]), '01': array([0.306]), '11': array([0.1305])}
{'00': 0.25, '10': 0.25, '01': 0.25, '11': 0.25}
{'00': array([0.2305]), '10': array([0.2525]), '01': array([0.261]), '11': array([0.256])}

In [34]:

```python
In [34]:

```
Batching with Subcircuits
# Define some functions to do Pauli twirling about the MS gate

def random_gate_insert():
gates_added = [[], []]
for qubit in range(2):  # For each qubit in the circuit
twirling_gate = np.random.choice(['I', 'Px', 'Py', 'Pz'])
    if twirling_gate != 'I':
        gates_added[qubit] += [twirling_gate]
return gates_added

def inverse_gate_insert(gates_added):
    inverse_added = [[], []]
    for qubit in range(2):
        if gates_added[qubit] == ['Px']:
            inverse_added[qubit] += ['Px']
        elif gates_added[qubit] == ['Py']:
            inverse_added[qubit] += ['Py']
            inverse_added[(qubit+1)%2] += ['Px']
        elif gates_added[qubit] == ['Pz']:
            inverse_added[qubit] += ['Py']
            inverse_added[(qubit+1)%2] += ['Px']
    return inverse_added

def two_pauli_multiplication(pauli1, pauli2):
    new_pauli = ''
    if ((pauli1 == 'Px') & (pauli2 == 'Py')) | ((pauli1 == 'Py') & (pauli2 == 'Px'))
        new_pauli = 'Pz'
    elif ((pauli1 == 'Px') & (pauli2 == 'Pz')) | ((pauli1 == 'Pz') & (pauli2 == 'Px'))
        new_pauli = 'Py'
    elif ((pauli1 == 'Pz') & (pauli2 == 'Py')) | ((pauli1 == 'Py') & (pauli2 == 'Pz'))
        new_pauli = 'Px'
    return new_pauli

def pauli_compile(inverse_added, gates_added):
gate_string = ''
for index in range(2):  # No gates
    equivalent_pauli = ''
    if len(inverse_added[index]) == 2:  # Two gates
        equivalent_pauli = two_pauli_multiplication(inverse_added[index][0], in
    elif len(inverse_added[index]) == 1:  # One gate
        equivalent_pauli = inverse_added[index][0]
    if gates_added[index] != []:
        if equivalent_pauli != '':  # Combine gates before and after
            equivalent_pauli = two_pauli_multiplication(equivalent_pauli, gates
        else:
            equivalent_pauli = gates_added[index][0]
    if equivalent_pauli != '':
        gate = '
    gate_string += gate
return gate_string
```python
# Create the elements of the jaqal file

jaqal_header = f""
// Comment via a double forward slash in jaqal strings

// Pulse Definitions Import Statement
from qscout.v1.std usepulses *

// Define let parameters
let alpha 0.1701
let beta 0.1701
let gamma 0.72405
let delta 0.74656
let epsilon 0.01
let zeta 0.1031
let eta 0.82893
let theta 0.75567
let iota 0.76884
let kappa 0.1701

let num_loops 0
let pi_4 {np.pi/4}
let pi_2 {np.pi/2}

// Select your register of qubits
register q[{num_qubits}]"

jaqal_prep = ""
prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>"

jaqal_MS = ""
MS q[0] q[1] @ pi_2"

jaqal_measure = ""
R q[0] iota kappa
measure_all"
```
In [37]: #Create the jaqal file with the Pauli twirls

jaqal_code = jaqal_header
num_RC = 10

for n in range(num_RC):
    gates_added = [[], []]
    inverse_added = [[], []]
    jaqal_code += jaqal_prep
    gates_added = random_gate_insert()
    jaqal_code += pauli_compile(inverse_added, gates_added)
    jaqal_code += jaqal_MS
    inverse_added = inverse_gate_insert(gates_added)
    jaqal_code += pauli_compile(inverse_added, [[], []])
    jaqal_code += jaqal_measure
    jaqal_code += "\n"

print(jaqal_code)
//Comment via a double forward slash in jaqal strings

//Pulse Definitions Import Statement
from qscout.v1.std usepulses *

//Define let parameters
let alpha 0.1701
let beta 0.1701
let gamma 0.72405
let delta 0.74656
let epsilon 0.01
let zeta 0.1031
let eta 0.82893
let theta 0.75567
let iota 0.76884
let kappa 0.1701

let num_loops 0
let pi_4 0.7853981633974483
let pi_2 1.5707963267948966

//Select your register of qubits
register q[2]
prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Py q[0]
MS q[0] q[1] 0 pi_2
Pz q[0]
Px q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Py q[0]
MS q[0] q[1] 0 pi_2
Pz q[0]
Px q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Px q[0]>
Py q[1]
MS q[0] q[1] 0 pi_2
Pz q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Px q[0]
Py q[1]
MS q[0] q[1] 0 pi_2
Pz q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Py q[0]
Px q[1]
MS q[0] q[1] θ pi_2
Pz q[0]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Pz q[0]
Py q[1]
MS q[0] q[1] θ pi_2
Pz q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Pz q[0]
MS q[0] q[1] θ pi_2
Py q[0]
Px q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Py q[0]
Px q[1]
MS q[0] q[1] θ pi_2
Pz q[0]
Px q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
Py q[0]
Py q[1]
MS q[0] q[1] θ pi_2
Py q[0]
Py q[1]
R q[0] iota kappa
measure_all

prepare_all
<R q[0] alpha beta | R q[1] gamma delta>
<Rz q[0] epsilon | Rz q[1] zeta>
MS q[0] q[1] 0 pi_2
R q[0] iota kappa
measure_all

In [38]:

#Running the string
res = run_jaqal_string(jaqal_code, overrides = {"__repeats__": 200}, backend = emul

#The result object that is returned has quite a lot of information stored in it.
#In this case we have a single subbatch, since we're not providing an override dict.
#This subbatch is then further divided into our 10 subcircuits.
#We can then call for the probability sorted by the integer representation of the binary result.
print("PROBABILITIES ORDERED BY QUBIT INTEGER VALUE")
for i in res
    print(res.by_subbatch[i].by_subcircuit[i].simulated_probability_by_int) #absolute
    print(res.by_subbatch[i].by_subcircuit[i].relative_frequency_by_int) #with simulated

#We can also sort the resulting probabilities by their string representation (matrix)
print("PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION")
for i in range(num_RC):
    print(res.by_subbatch[i].by_subcircuit[i].simulated_probability_by_str) #absolute
    print(res.by_subbatch[i].by_subcircuit[i].relative_frequency_by_str) #with simulated
PROBABILITIES ORDERED BY QUBIT INTEGER VALUE

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.44]
 [0.09]
 [0.02]
 [0.45]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.355]
 [0.08]
 [0.03]
 [0.535]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.495]
 [0.085]
 [0.015]
 [0.405]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.39]
 [0.08]
 [0.03]
 [0.5]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.435]
 [0.09]
 [0.02]
 [0.455]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.395]
 [0.1]
 [0.035]
 [0.47]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.405]
 [0.095]
 [0.02]
 [0.48]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.38]
 [0.115]
 [0.01]
 [0.495]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.4]
 [0.115]
 [0.03]
 [0.455]]
```

```
[0.39818686 0.09484271 0.02904003 0.47793041]
[[0.405]
 [0.075]
 [0.025]
 [0.495]]
```

PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION

```
{ '00': 0.39818686, '10': 0.09484271, '01': 0.02904003, '11': 0.47793041 }
```

```
{ '00': array([0.44]), '10': array([0.09]), '01': array([0.02]), '11': array([0.45]) }
```
Out[39]: Text(0.5, 0, 'RC index')
Batching with Indexing
In [41]:

# Running it via standard subcircuit ordering

outcomes = [[] for _ in range(num_states)]

res = run_jaqal_string(jaqal_code, overrides = "__repeats__": 200, backend = emul

# Plot the results

outcomes = [[] for _ in range(num_states)]
for r in res.by_subbatch[0].by_subcircuit:
    for n in range(num_states):
        outcomes[n].append(r.probability_by_int[n])

plt.figure(1)
for n in range(num_states):
    plt.plot([0,1,2,3,4,5],
              outcomes[n],
              label=f$\left\vert .join(reversed(f'{n:02b}'))\right\rangle$, marker="sv^o"[n%4])

plt.legend()
plt.xlabel("code index");
In [42]:
    # We can also run this by indexing different codes, so if we wanted all Z, X, or Y
    indices = 6
    res = run_jaqal_string(jaqal_code, overrides = {"__index__": [0,3,1,4,2,5],
                                                        "__repeats__": 200}, backend = emul
    #Note that the indexes are contained within a nested list!

    print("PROBABILITIES ORDERED BY QUBIT INTEGER VALUE")
    for i in range(indices):
        print(res.by_subbatch[0].by_subcircuit[i].simulated_probability_by_int) #absolute
        print(res.by_subbatch[0].by_subcircuit[i].relative_frequency_by_int) #with simu

    #We can also sort the resulting probabilities by their string representation (matrix
    print("PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION")
    for i in range(indices):
        print(res.by_subbatch[0].by_subcircuit[i].simulated_probability_by_str) #absolute
        print(res.by_subbatch[0].by_subcircuit[i].relative_frequency_by_str) #with simu

    #Note, these are still plotting based on subcircuit number! But if we want to see how
    # on the experiment or the emulator, we can use the by_time command:

    print("RESULTS ORDERED BY TIME")
    for i in range(indices):
        print(res.by_time[i].simulated_probability_by_int) #absolute probability
        print(res.by_time[i].relative_frequency_by_int) #with simulated shot noise

    #Plot the results by time
    outcomes = [[] for _ in range(num_states)]
    for r in res.by_time:
        for n in range(num_states):
            outcomes[n].append(r.probability_by_int[n])

    plt.figure(1)
    for n in range(num_states):
        plt.plot([0,1,2,3,4,5],
                 outcomes[n],
                 label=f"$\left\{\cdot{\cdot{2b}}\right\}$\right\rangle\rangle";
                 marker="sv^o"[n%4])
    plt.legend()
    plt.xlabel("time index");
PROBABILITIES ORDERED BY QUBIT INTEGER VALUE

[0.5 0. 0. 0.5]

[[0.47]
 [0. ]
 [0. ]
 [0.53]]

[0. 0.5 0.5 0.]

[[0. ]
 [0.48]
 [0.52]
 [0. ]]

[0.5 0. 0. 0.5]

[[0.46]
 [0. ]
 [0. ]
 [0.54]]

[0.5 0. 0. 0.5]

[[0.435]
 [0. ]
 [0. ]
 [0.565]]

[0.5 0. 0. 0.5]

[[0.495]
 [0. ]
 [0. ]
 [0.585]]

[0. 0.5 0.5 0.]

[[0. ]
 [0.51]
 [0.49]
 [0. ]]

PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION

{ '00': 0.5, '10': 0.0, '01': 0.0, '11': 0.4999999999999999
 { '00': array([0.47]), '10': array([0.]), '01': array([0.]), '11': array([0.53])
 { '00': 0.0, '10': 0.4999999999999999, '01': 0.5000000000000001, '11': 0.0
 { '00': array([0.]), '10': array([0.48]), '01': array([0.52]), '11': array([0.])
 { '00': 0.5000000000000001, '10': 0.0, '01': 0.0, '11': 0.4999999999999999
 { '00': array([0.46]), '10': array([0.]), '01': array([0.]), '11': array([0.54])
 { '00': 0.5, '10': 0.0, '01': 0.0, '11': 0.4999999999999999
 { '00': array([0.435]), '10': array([0.]), '01': array([0.]), '11': array([0.565])
 { '00': 0.5, '10': 0.0, '01': 0.0, '11': 0.4999999999999999
 { '00': array([0.495]), '10': array([0.]), '01': array([0.]), '11': array([0.505])
 { '00': 0.0, '10': 0.4999999999999999, '01': 0.5000000000000001, '11': 0.0
 { '00': array([0.]), '10': array([0.51]), '01': array([0.49]), '11': array([0.])

RESULTS ORDERED BY TIME

[0.5 0. 0. 0.5]

[0.47 0. 0. 0.53]

[0.5 0. 0. 0.5]

[0.435 0. 0. 0.565]

[0.5 0. 0.5 0.]

[0. 0.5 0.5 0.]

[0.5 0. 0. 0.5]

[0.495 0. 0. 0.505]

[0.5 0. 0. 0.5]

[0.46 0. 0. 0.54]

[0. 0.5 0.5 0.]
Putting it all together
#Create a jaqal code (string method)

```python
jaqal_code = f'""
//Comment via a double forward slash in jaqal strings

//Pulse Definitions Import Statement
from qscout.v1.std use pulses *

//Define let parameters
let gamma 0.72405

//Select your register of qubits
register q[[num_qubits]]

//Create jaqal circuit, starting with prepare_all, ending with measure_all
prepare_all
MS q[0] q[1] @ gamma
measure_all

prepare_all
MS q[0] q[1] @ gamma
Sx q[0]
Sx q[1]
measure_all

prepare_all
MS q[0] q[1] @ gamma
Sy q[0]
Sy q[1]
measure_all
""
```
In [49]:
res = run_jaqal_string(jaqal_code, overrides = {"gamma": [0.74205, 1.57079, 0.1701],
ind indices_SB = 3
indices_SC = 3

print("PROBABILITIES ORDERED BY QUBIT INTEGER VALUE")
for i in range(indices_SB):
    for j in range(indices_SC):
        print(res.by_subbatch[i].by_subcircuit[j].simulated_probability_by_int) #absolute probability
        print(res.by_subbatch[i].by_subcircuit[j].relative_frequency_by_int) #with

#We can also sort the resulting probabilities by their string representation (matrix)
print("PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION")
for i in range(indices_SB):
    for j in range(indices_SC):
        print(res.by_subbatch[i].by_subcircuit[j].simulated_probability_by_str) #absolute probability
        print(res.by_subbatch[i].by_subcircuit[j].relative_frequency_by_str) #with

#Note, these are still plotting based on subcircuit number! But if we want to see how
# on the experiment or the emulator, we can use the by_time command:

print("RESULTS ORDERED BY TIME")
for k in range(indices):
    print(res.by_time[k].simulated_probability_by_int) #absolute probability
    print(res.by_time[k].relative_frequency_by_int) #with simulated shot noise
PROBABILITIES ORDERED BY QUBIT INTEGER VALUE

```
[[0.86854236 0. 0. 0.13145764]
 [0. 0. 0. 0.]
 [0.1415 0.25 0.25 0.25]
 [0.2355 0.256 0.246 0.2625]
 [0.25 0.25 0.25 0.25]
 [0.2275 0.26 0.2675 0.245]
 [0.50000316 0. 0. 0.49999684]
 [0.4895 0. 0. 0.5105]
 [0.25 0.25 0.25 0.25]
 [0.258 0.2485 0.2595 0.234]
 [0.25 0.25 0.25 0.25]
 [0.259 0.2595 0.234 0.2475]
 [0.99278392 0. 0. 0.00721608]
 [0.993 0. 0. 0.007]
 [0.25 0.25 0.25 0.25]
 [0.2485 0.2545 0.236 0.261]
 [0.25 0.25 0.25 0.25]
 [0.251 0.2305 0.265 0.2535]]
```

PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION

```
{"00": 0.868542358862067, '10': 0.0, '01': 0.0, '11': 0.13145764111379327}
{"00": array([0.8585]), '10': array([0.]), '01': array([0.]), '11': array([0.1415])}
{"00": 0.2500000000000001, '10': 0.25, '01': 0.25, '11': 0.25}
{"00": array([0.2355]), '10': array([0.256]), '01': array([0.246]), '11': array([0.2625])}
{"00": 0.2500000000000001, '10': 0.25, '01': 0.25, '11': 0.25}
{"00": array([0.2275]), '10': array([0.26]), '01': array([0.2675]), '11': array([0.245])}
```
In [ ]: