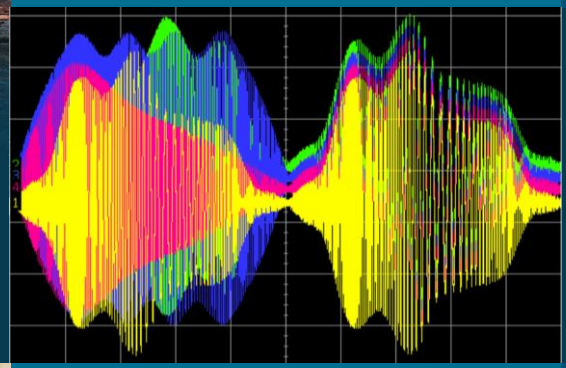
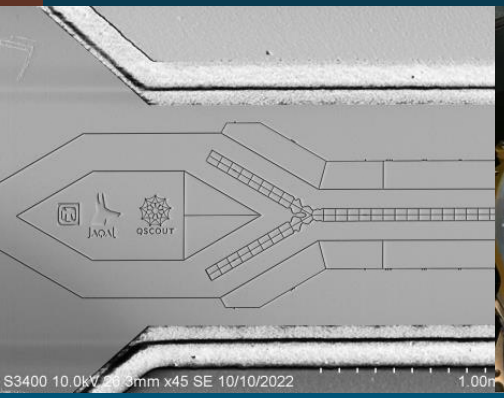
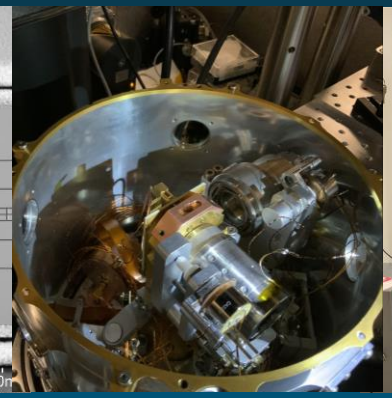
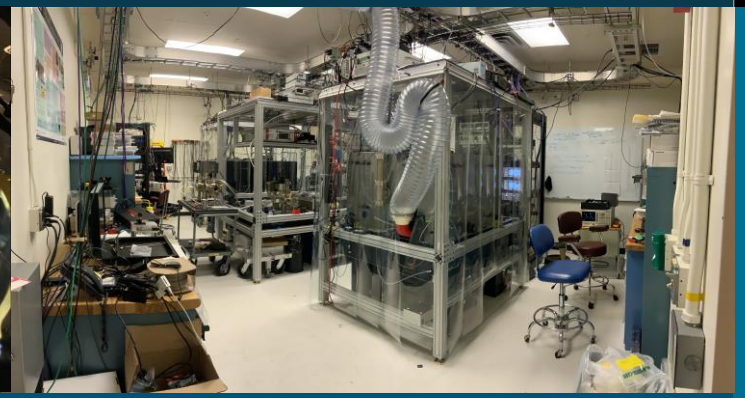
Efficient Implementation of Circuits on the QSCOUT Hardware through "Batching"









PRESENTED BY

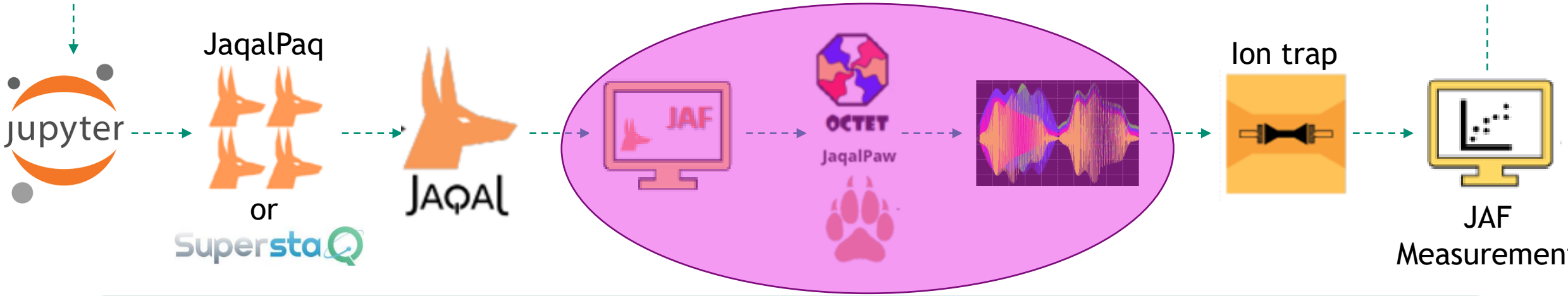
Christopher G. Yale

SAND# here

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Running Circuits on the Hardware

QSCOUT



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":

A. D. Burch, D. S. Lobser, C. G. Yale, J. W. Van Der Wall, O. G. Maupin, J. D. Goldberg, M. N. H. Chow, M. C. Revelle, S. M. Clark (2022) "Batching Circuits to Reduce Compilation in Quantum Control Hardware," *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 503-508.

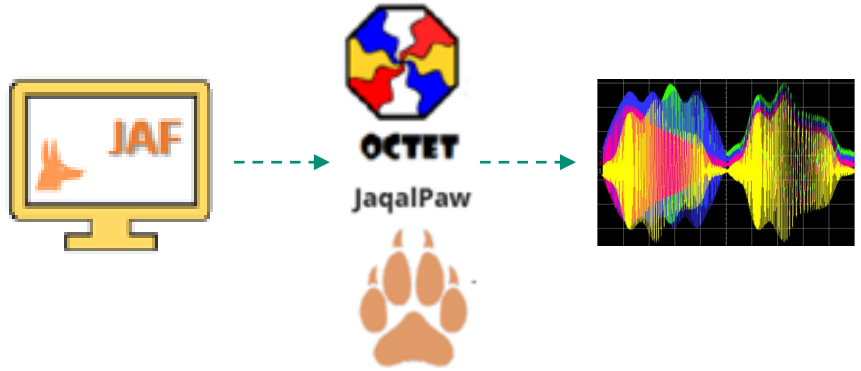
More on Octet and compiling:

D. Lobser, J. Van Der Wall, J. Goldberg (2022) "Performant coherent control: bridging the gap between high- and low-level operations on hardware," *2022 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 320-330.

Communication and Compile Time

QSCOUT

- 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

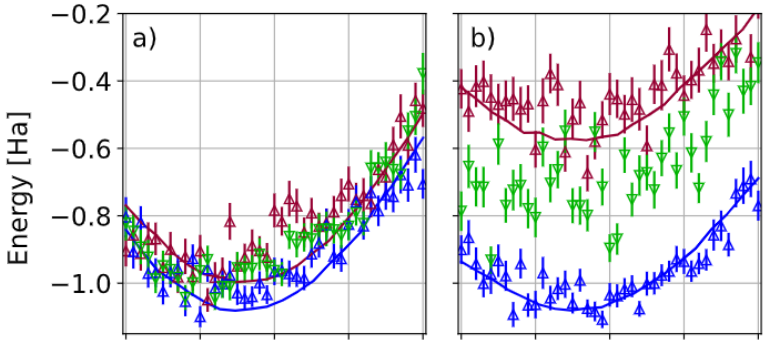


Example:

- 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

Example inspired by:
OAK RIDGE National Laboratory
Duke Quantum Center

- 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



arXiv:2205.14225

Solution: Batching!

QSCOUT

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!

Batching with Overrides

Batching with Subcircuits

Batching with Indexing

Mix them all together

Batching with Subcircuits

QSCOUT

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


```
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
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: 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)
```

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

Note: This already exists in the current Jaqal API!

Batching with Overrides

QSCOUT

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


```
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

```
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)
```

prepare_all Rx q[0] 0.1701 Ry q[2] 0.1701 MS q[2] q[3] 0 1.57079 MS q[3] q[0] 0 0.19634 measure_all	prepare_all Rx q[0] 0.1701 Ry q[2] 0.1701 MS q[2] q[3] 0 0.78539 MS q[3] q[0] 0 0.39269 measure_all	prepare_all Rx q[0] 0.1701 Ry q[2] 0.1701 MS q[2] q[3] 0 0.39269 MS q[3] q[0] 0 0.78539 measure_all	prepare_all Rx q[0] 0.1701 Ry q[2] 0.1701 MS q[2] q[3] 0 0.19634 MS q[3] q[0] 0 1.57079 measure_all

Note: Each dictionary entry must be an array of the same length or a scalar

Other Things to Override

QSCOUT

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

QSCOUT

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


```
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__"

```
override_dict = {"__index__": [[1,3,2,0]]}

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

prepare_all Sxd q[1] MS q[1] q[3] Sy q[3] measure_all	prepare_all MS q[0] q[2] measure_all	prepare_all Sx q[2] MS q[2] q[3] Sy q[3] measure_all	prepare_all Px q[0] MS q[0] q[1] Px q[1] measure_all

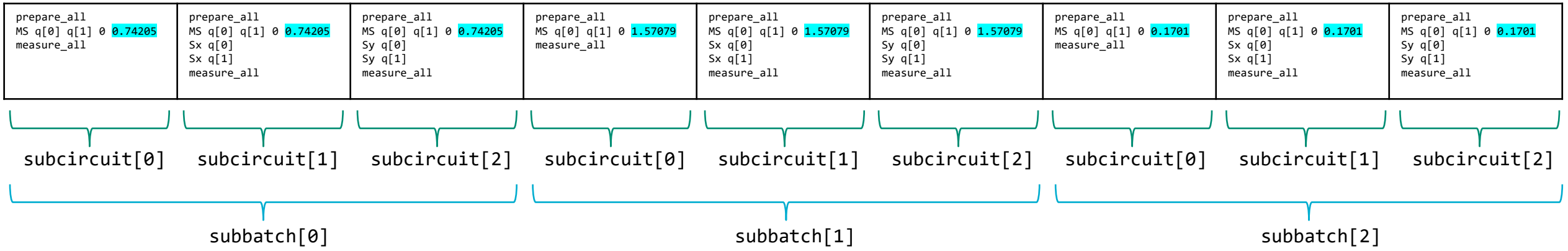
Combining Batching Approaches

QSCOUT

```
from qscout.v1.std usepulses*

let gamma 0.74205

register q[2]

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

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

prepare_all
MS q[0] q[1] 0 gamma
Sy q[0]
Sy q[1]
measure_all
```

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

QSCOUT

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

```
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	`result.by_subbatch[i]`
Select a subcircuit within the subbatch	`result.by_subbatch[i].by_subcircuit[j]`
(Emulator Only) Absolute multi-qubit state probabilities sorted by integer	`result.by_subbatch[i].by_subcircuit[j].simulated_probability_by_int`
(Emulator) state probs w/shot noise or (Experiment) actual data sorted by integer	`result.by_subbatch[i].by_subcircuit[j].relative_frequency_by_int`
(Emulator Only) Absolute multi-qubit state probabilities sorted by string	`result.by_subbatch[i].by_subcircuit[j].simulated_probability_by_str`
(Emulator) state probs w/shot noise or (Experiment) actual data sorted by string	`result.by_subbatch[i].by_subcircuit[j].relative_frequency_by_str`
(Emulator) absolute probabilities (Experiment) actual data	`result.by_subbatch[i].by_subcircuit[j].probability_by_int or _by_str`

Data Object Structure Continued

Instead of the data ordered by subbatches and subcircuits, we can also access the data in the time order it was taken

```
result.by_time[k]
        result.by_time[k].relative_frequency_by_int (and all the others)
```

Integer Ordering:

q[0] = 0
q[1] = 1
q[2] = 1

→ 0b110 = 7 →

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

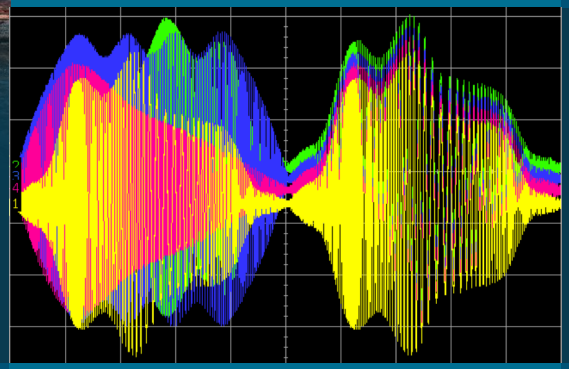
String Ordering:

q[0] = 0
q[1] = 1
q[2] = 1

→ $|011\rangle$ →

{‘000’: 0.017, ‘100’: 0.008, ‘010’: 0.021, ‘110’: 0.112, ‘001’: 0.231, ‘101’: 0.000, ‘011’: 0.541,’111’: 0.070}

Pulse-Level Control Using Jaqal Pulses and Waveforms (JaqalPaw)



QSCOUT

U.S. DEPARTMENT OF ENERGY
Office of Science

PRESENTED BY

Daniel Lobser

SAND# here

Pulse-Level Control Using Jaqal Pulses and Waveforms (JaqalPaw)

QSCOUT

References

Quantum Computing

IEEE Transactions on Quantum Engineering

Received March 30, 2021; revised June 11, 2021; accepted June 30, 2021; date of publication July 13, 2021; date of current version August 12, 2021.

Digital Object Identifier 10.1109/TQE.2021.3096480

Engineering the Quantum Scientific Computing Open User Testbed

SUSAN M. CLARK[1], DANIEL LOBSER[1], MELISSA C. REVELLE[1], CHRISTOPHER G. YALE[1], DAVID BOSSERT[1], ASHLYN D. BURCH[1], MATTHEW N. CHOW[1,2,3], CRAIG W. HOGLE[1], MEGAN IVORY[1], JESSICA PEHR[4], BRADLEY SALZBRENNER[1], DANIEL STICK[1], WILLIAM SWEATT[1], JOSHUA M. WILSON[1], EDWARD WINROW[1], AND PETER MAUNZ[4]

[1]Sandia National Laboratories, Albuquerque, NM 87123 USA
[2]Department of Physics and Astronomy, University of New Mexico, Albuquerque, NM 87131 USA
[3]Center for Quantum Information and Control, University of New Mexico, Albuquerque, NM 87131 USA
[4]IonQ, Inc., College Park, MD 20740 USA

JaqalPaw: A Guide to Defining Pulses and Waveforms for Jaqal

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. Clark[1]

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

(Dated: April 19, 2021)

https://qscout.sandia.gov

QSCOUT Info

- Submit a whitepaper for consideration to use the machine
- Jaqal Language Specs
- Download JaqalPaq
- JaqalPaw (Pulses and Waveforms)
- Jaqal Seminar Supplemental Materials:

Outline

QSCOUT

Experimental Details of Gate Implementation

Basic JaqalPaw: Simple waveforms

Advanced JaqalPaw: Experimentally meaningful waveforms

Realizing Quantum Gates

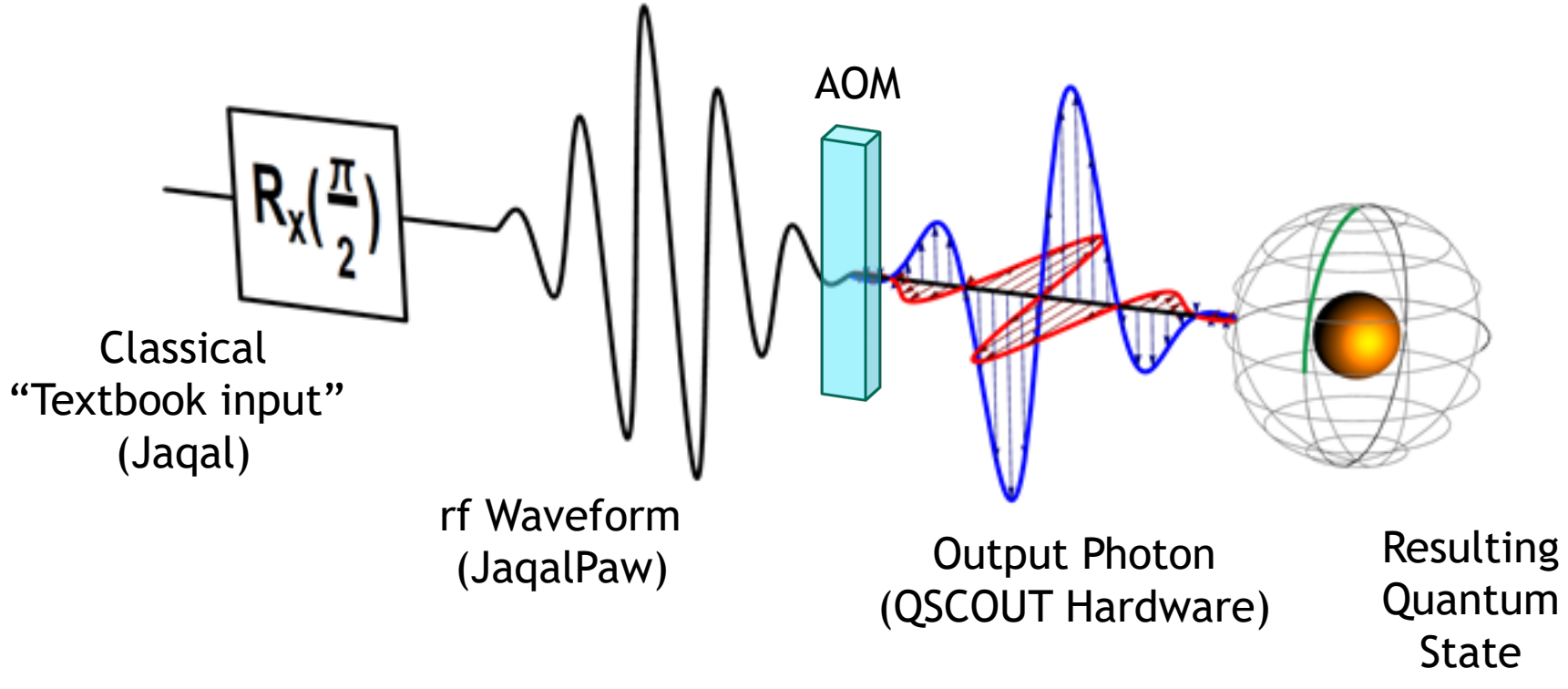
QSCOUT

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 a 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

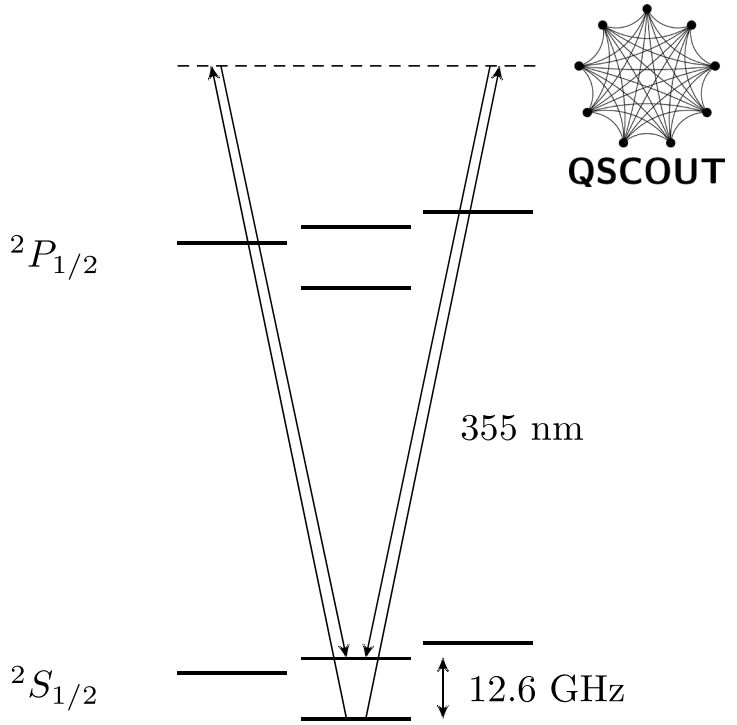
$^{171}Yb^{+}$ qubit, clock state 12.6 GHz

Individual addressing requires lasers

Optical frequency comb to bridges 12.6 GHz via Raman transitions

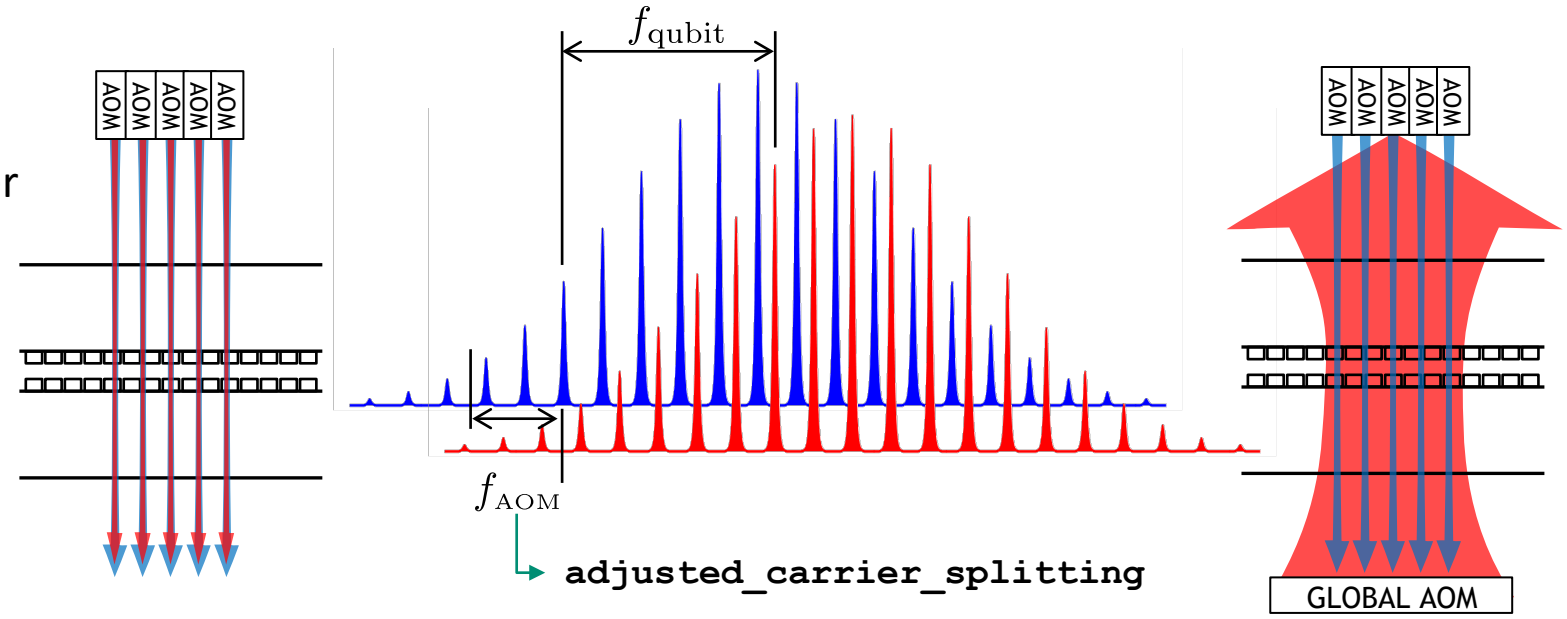
Frequency, phase, and amplitude control using RF signals applied to acousto-optic modulators (AOMs)

Two configurations: Co- and Counter-propagating



Co-propagating

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

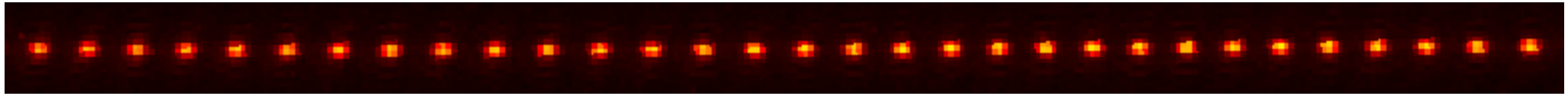


Counter-propagating

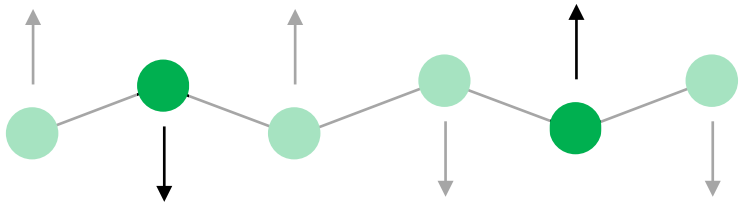
- Supports motional-state addressing and ground state cooling
- Affected by Doppler shifts
- Necessary for two-qubit gates

Two-qubit Mølmer-Sørensen Gates

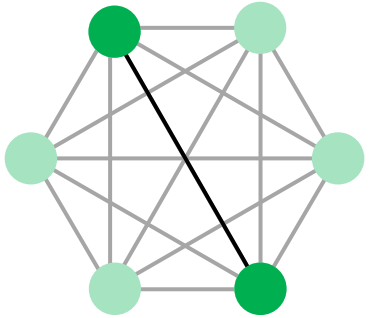
QSCOUT

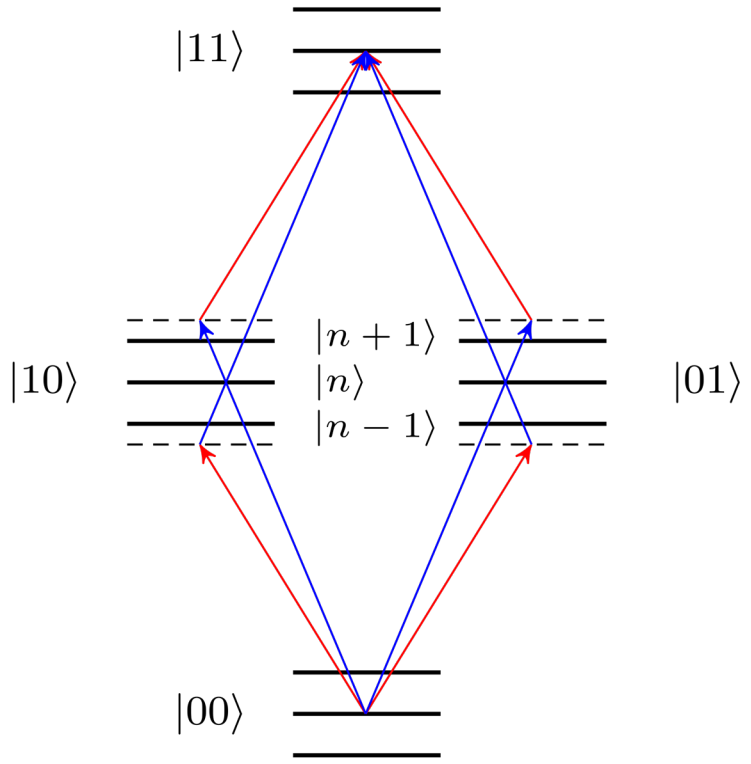


Common motional modes



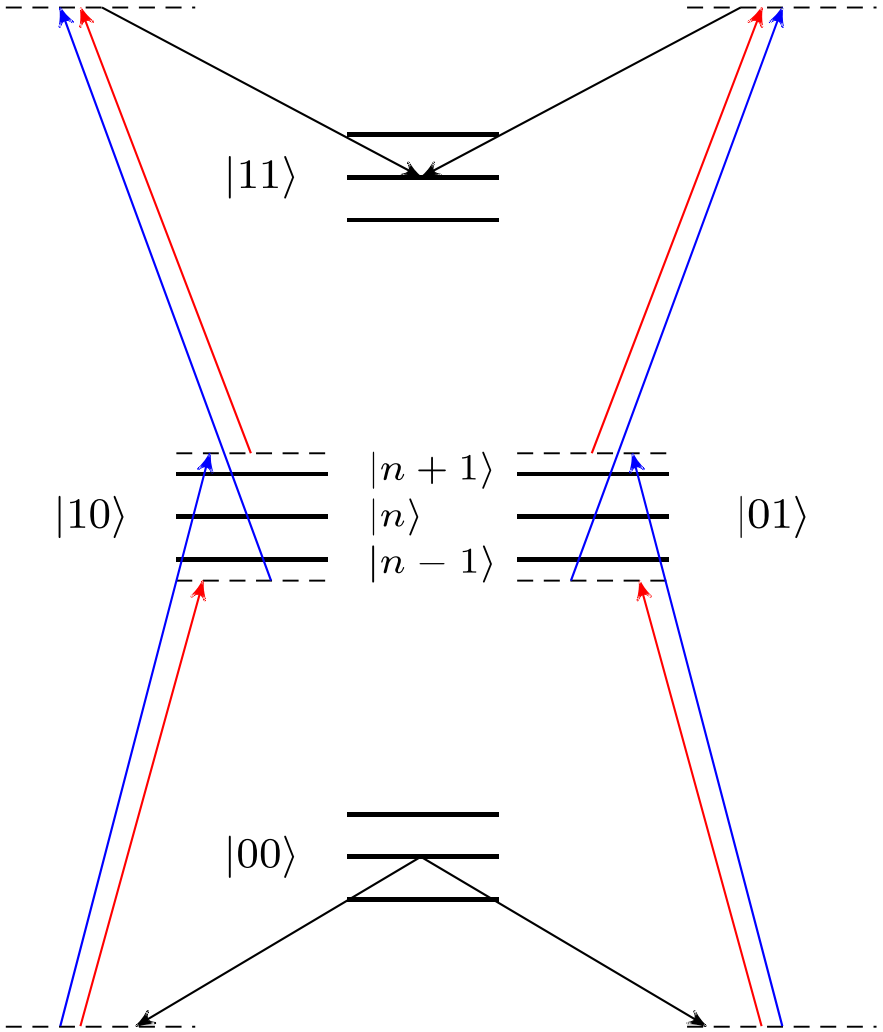
“Fully connected”

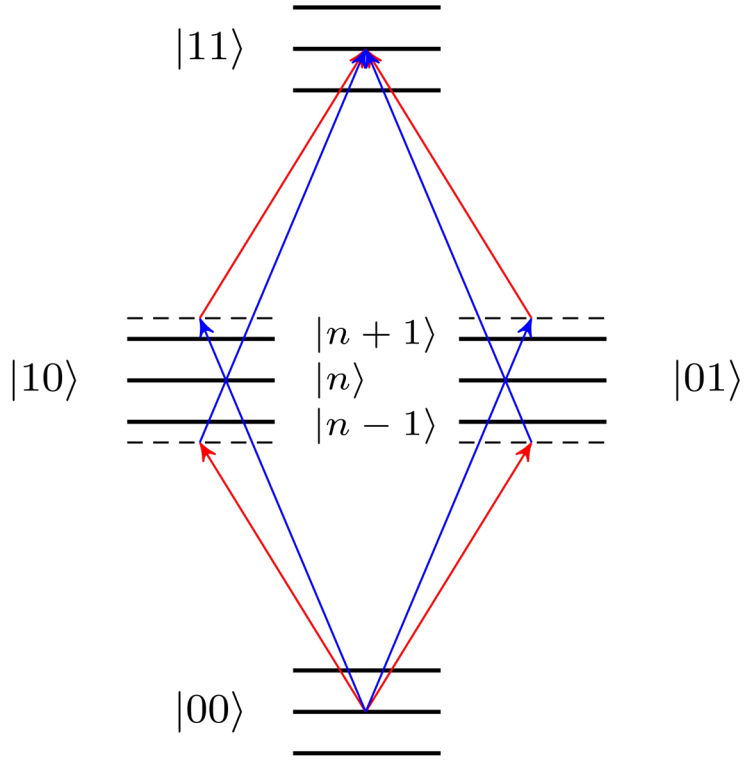




Two-qubit Mølmer-Sørensen Gates

QSCOUT





Outline

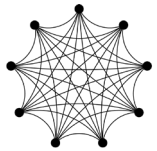
QSCOUT

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

QSCOUT

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

```
class MyGatePulses(QSCOUTBuiltins):
```

JaqalPaw in Broad Strokes

QSCOUT

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

```
class MyGatePulses(QSCOUTBuiltins):

    some_calibrated_parameter: float = 200e6  # Hz
```

JaqalPaw in Broad Strokes

QSCOUT

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

```
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)))
```

JaqalPaw in Broad Strokes

QSCOUT

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_”

```
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):
```

Arguments after “self” are passed in from Jaqal:

```
GaussPulse q[2] 3.8
```

JaqalPaw in Broad Strokes

QSCOUT

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_”

```
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
```

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

The PulseData Object

QSCOUT

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

```
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
          waittrig=False)           # wait for external trigger
```

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

The PulseData Object

QSCOUT

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

```
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
          waittrig=False)           # wait for external trigger
```

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

The PulseData Object

QSCOUT

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)

```
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
          waittrig=False)           # wait for external trigger
```

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

The PulseData Object

QSCOUT

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

```
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
          waittrig=False)           # wait for external trigger
```

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

The PulseData Object

QSCOUT

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,                  # 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
          waittrig=False)           # wait for external trigger
```

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

The PulseData Object

QSCOUT

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)

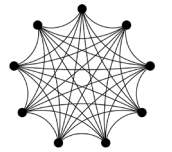
Z rotations are done virtually

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

```
PulseData(channel,                  # 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
          waittrig=False)           # wait for external trigger
```

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

Discrete and Spline Modulations



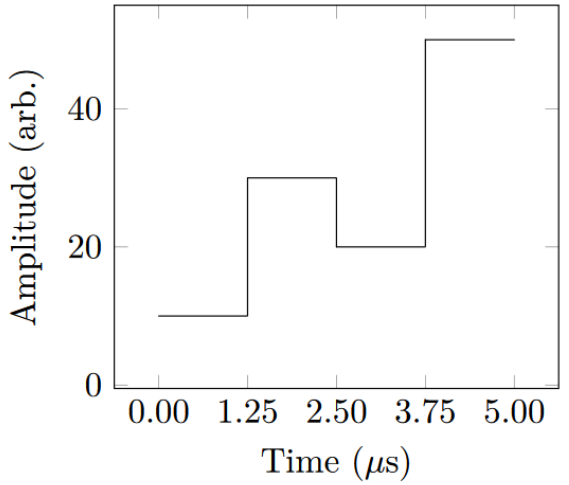
Discrete updates are represented as a list […], Spline updates are represented as a tuple (…)

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

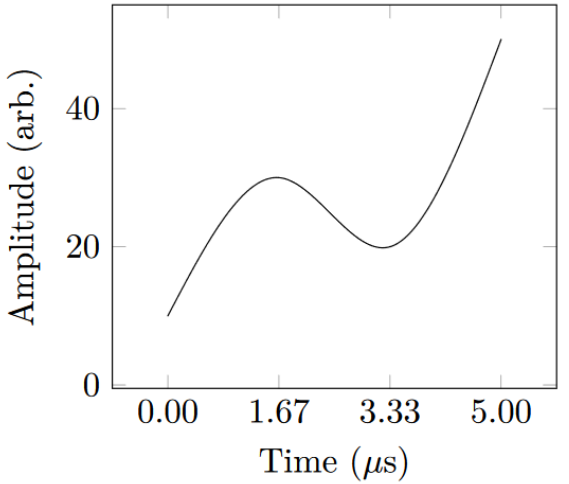
Type aliases "Spline" and "Discrete" will be provided

```
def gate_G(self, qubit):
    return [PulseData(qubit,
                      5e-6,
                      freq0=200e6,
                      amp0=[10,30,20,50],
                      phase0=0)]
```



EX. 5: Discrete updates are represented as a list of inputs and are equally distributed across the duration of the pulse.

```
def gate_G(self, qubit):
    return [PulseData(qubit,
                      5e-6,
                      freq0=200e6,
                      amp0=(10,30,20,50),
                      phase0=0)]
```



EX. 6: Smooth updates are represented as a tuple.

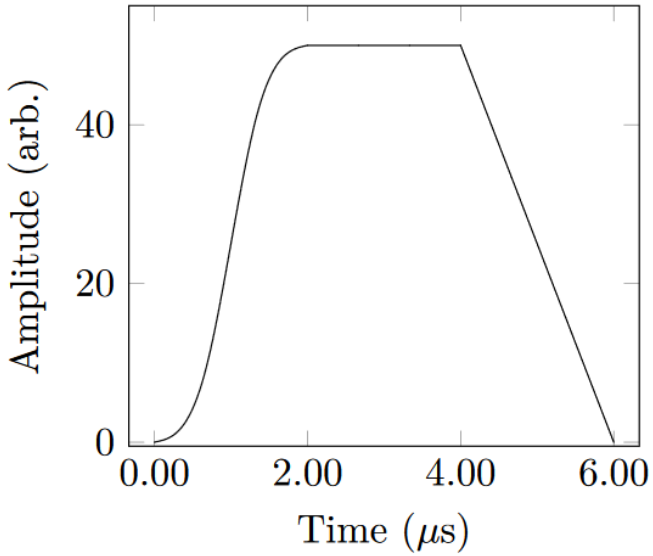
Piecewise Operations

QSCOUT

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

This also applies to gates in general

```
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

QSCOUT

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

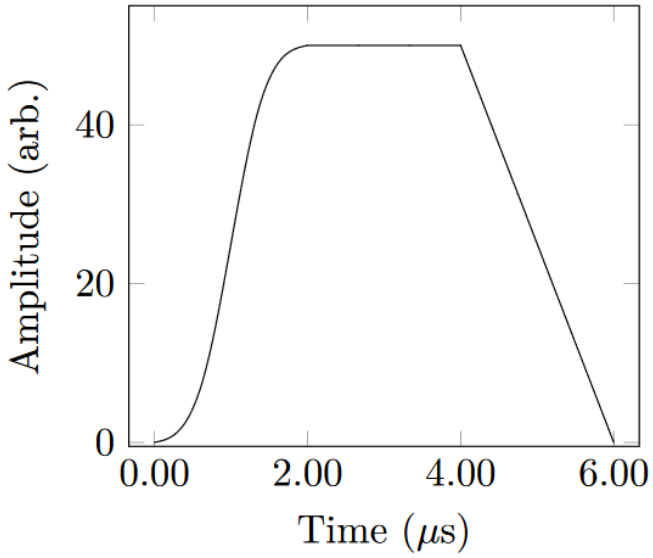
This also applies to gates in general

New feature has been implemented to simplify this notation:

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

```
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))]
```

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



Piecewise Operations

QSCOUT

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

This also applies to gates in general

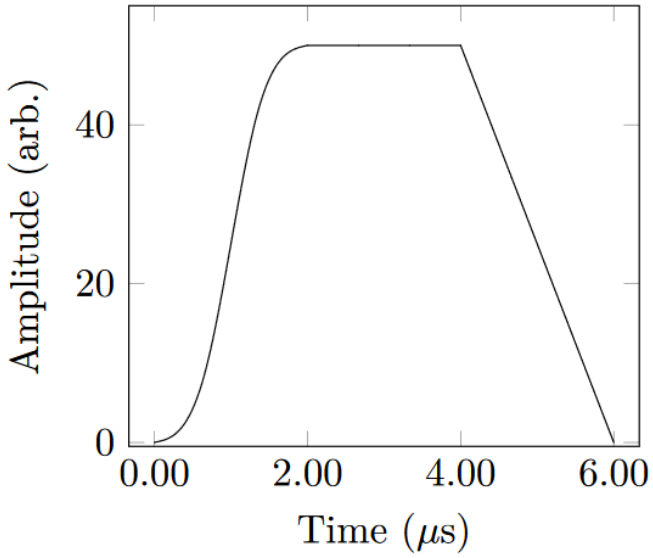
New feature has been implemented to simplify this notation:

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

Lists can contain scalar values, lists, or tuples

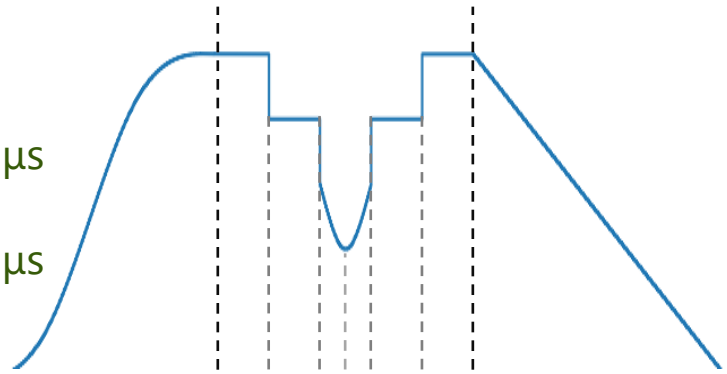
Tuples can only contain scalar values

```
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))]
```




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

```
def gate_G(self, qubit):
    return [PulseData(qubit, 6e-6,
                      amp0=[(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, 0.2 µs
                            (50, 0)])]    2 µs
```



Running a Gate Across Multiple Channels

QSCOUT

PulseData objects on different channels are run in parallel

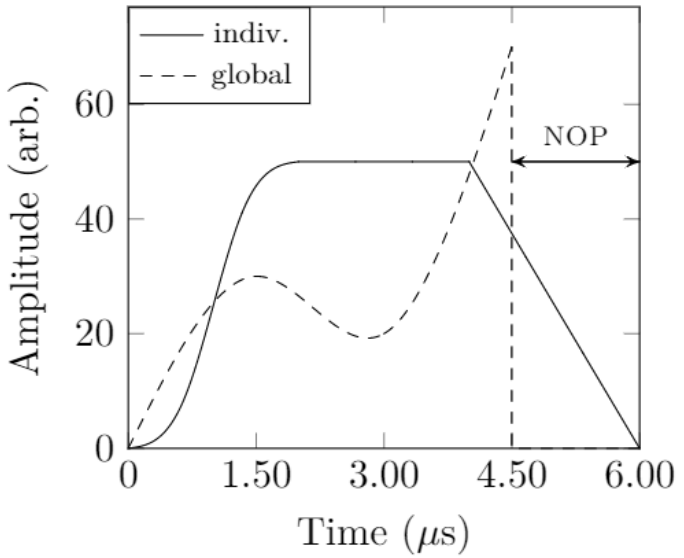
This always applies to PulseData objects in the same gate

This optionally applies to gates run in parallel if run on different channels, e.g. in Jaqal:

```
< G1 q[2] | G2 q[3] >
```

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

```
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.

Outline

QSCOUT

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:

SxSzSx → SySx

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

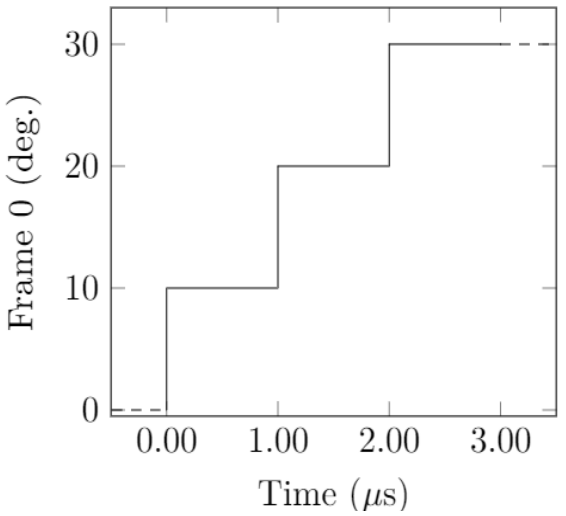
$$\sin(2\pi f t + \varphi + \varphi_z)$$

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

Frame rotations take scalar, discrete, and spline inputs

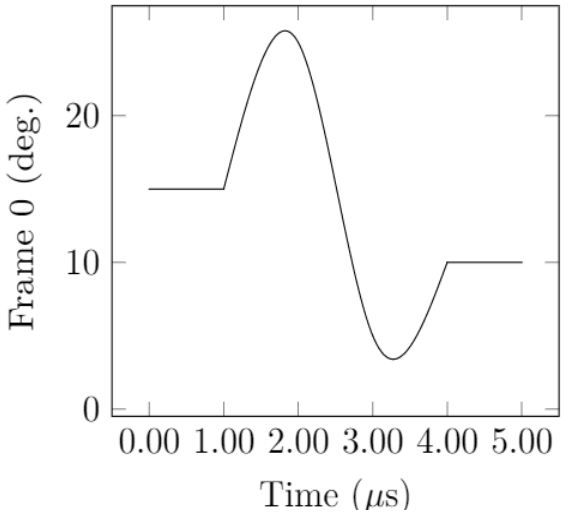
Spline inputs accumulate only the final value

```
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.

```
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

QSCOUT

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

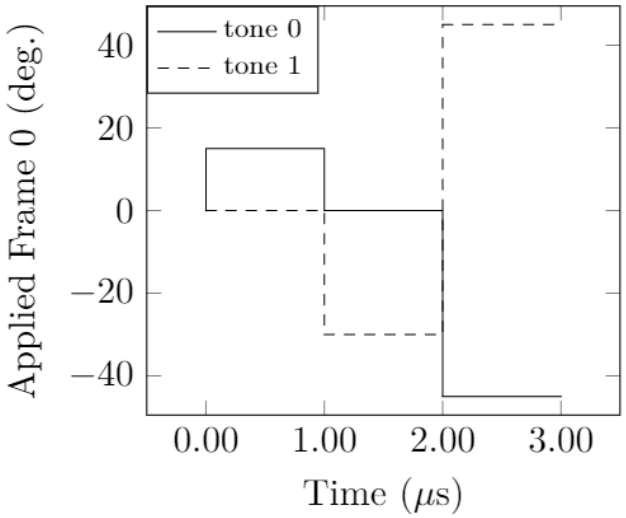
For a two-qubit Mølmer-Sørensen gate, both red and blue sideband inputs must have the frame forwarded

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

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

Input	Tone 1	Tone 0
0b00	-	-
0b01	-	✓
0b10	✓	-
0b11	✓	✓

```
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

QSCOUT

Frequency comb is not actively stabilized at the source!

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

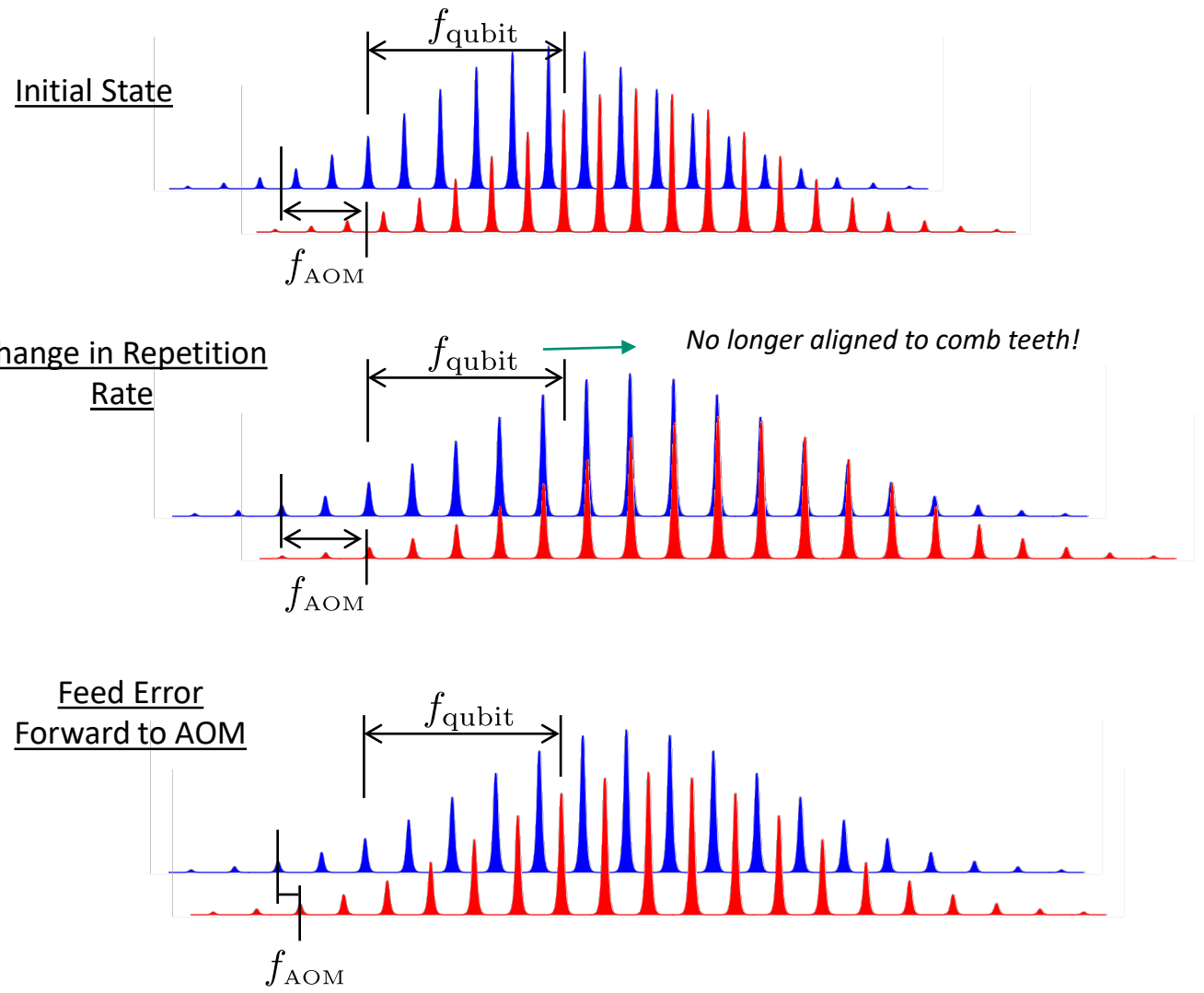
Beat note lock must be applied to only one of the two tones contributing to a Raman transition

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

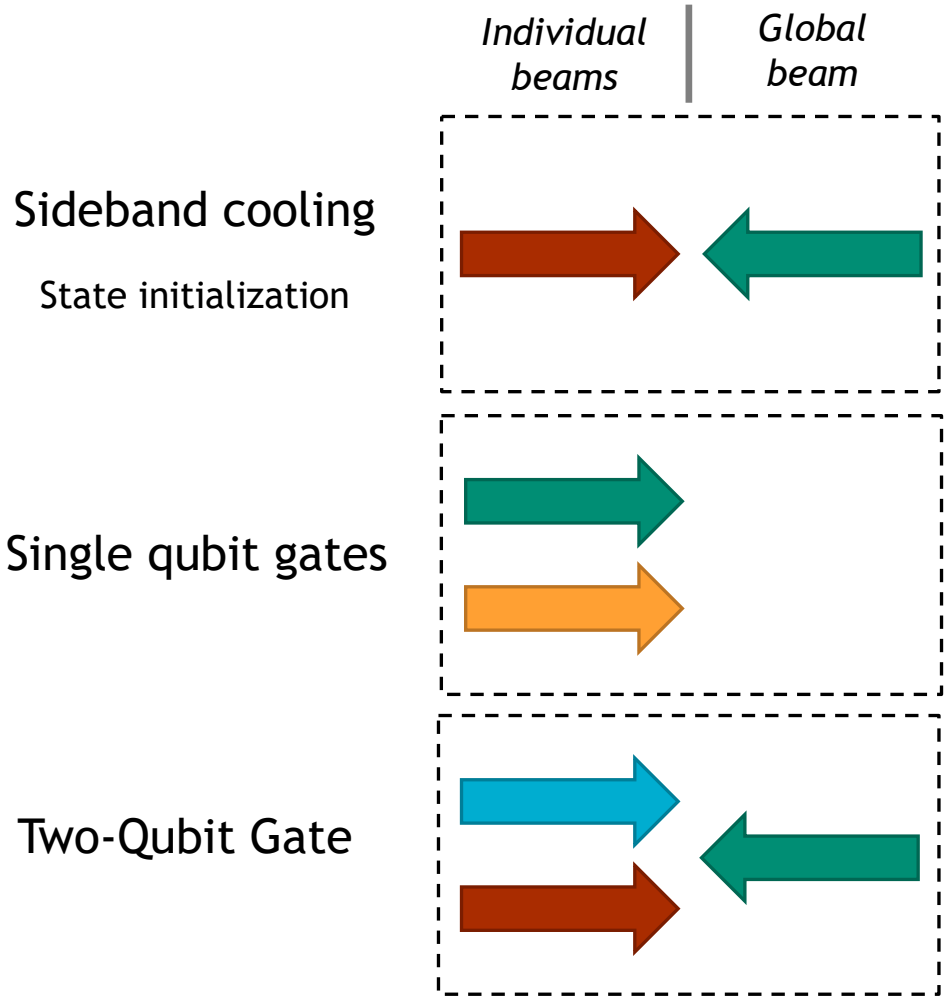
Comb Instability → Beat Note Lock



Challenges: RF Reproducibility and Agility

QSCOUT

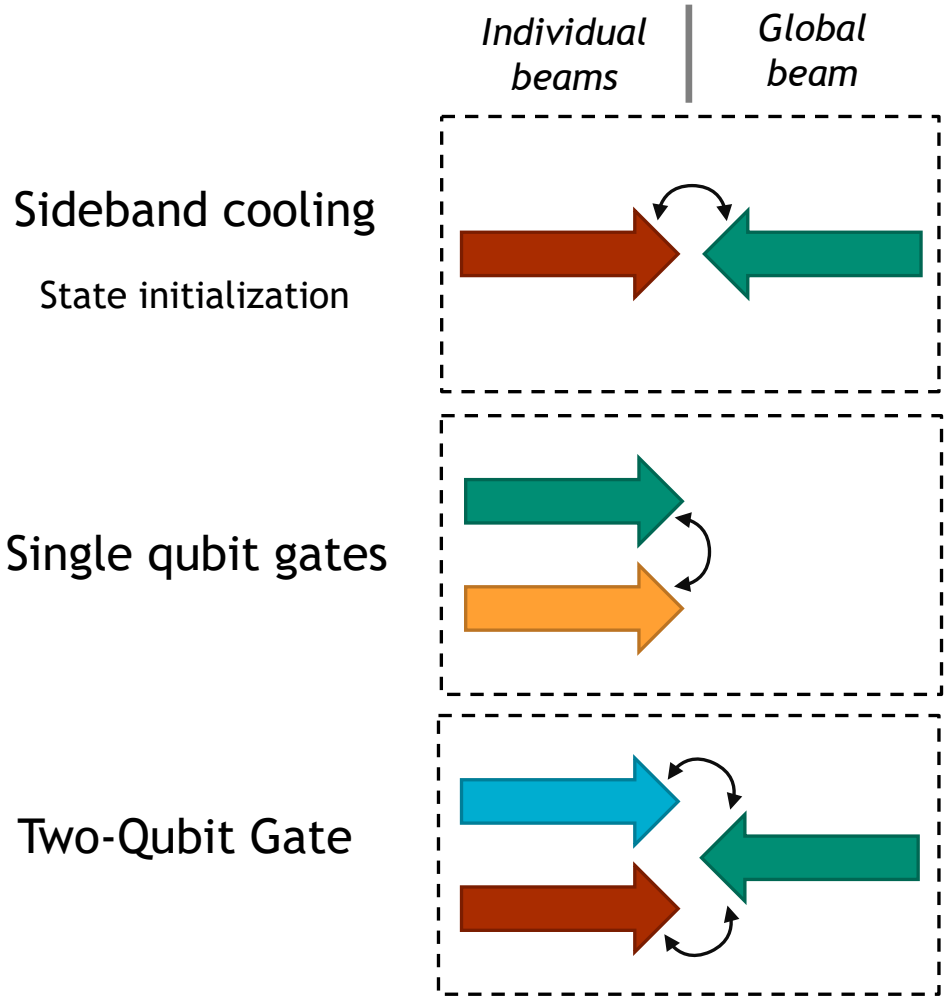
Three basic configurations



Challenges: RF Reproducibility and Agility

QSCOUT

Three basic configurations

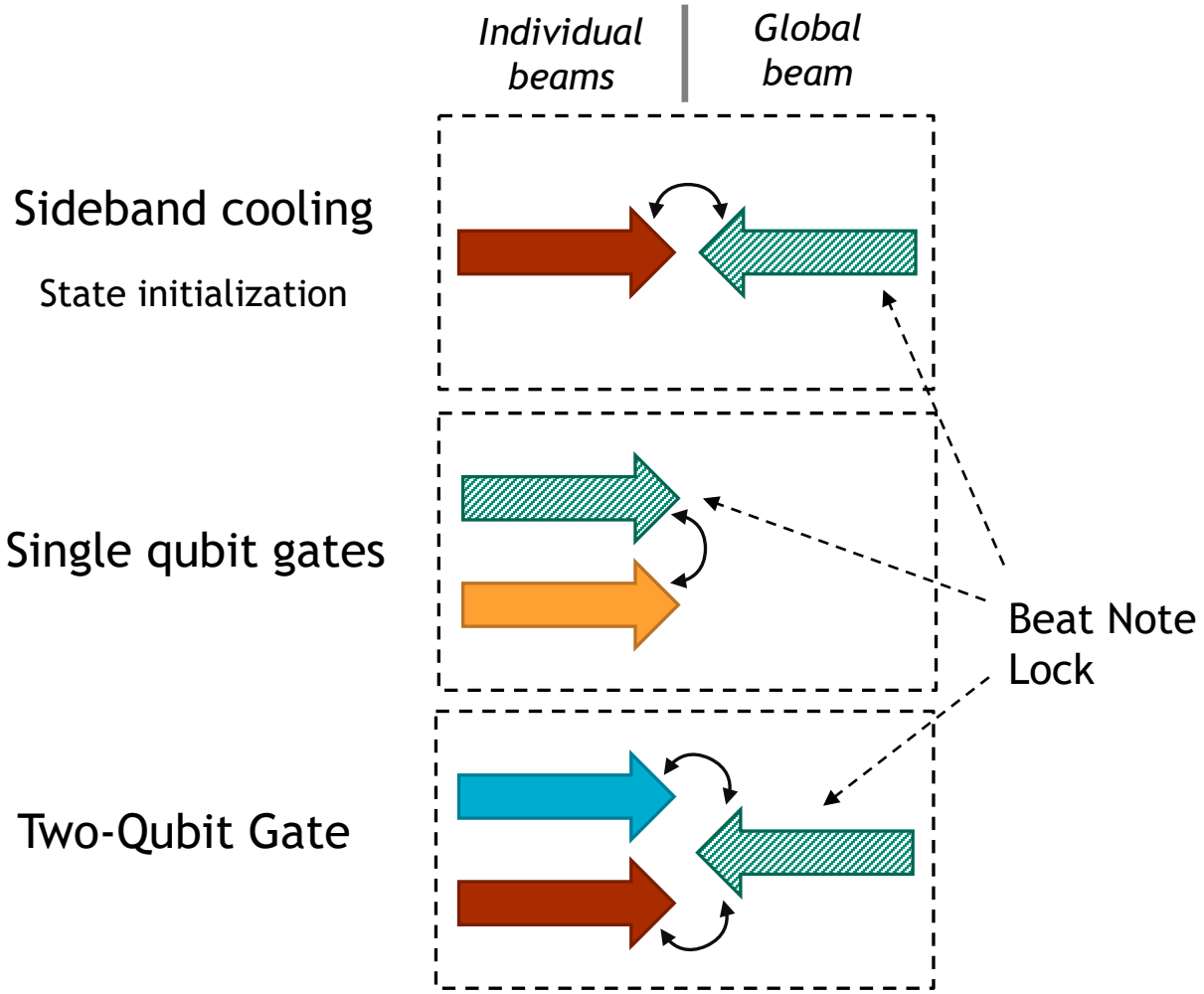


Lock must be applied to exactly one tone for each Raman pair!

Challenges: RF Reproducibility and Agility

QSCOUT

Three basic configurations

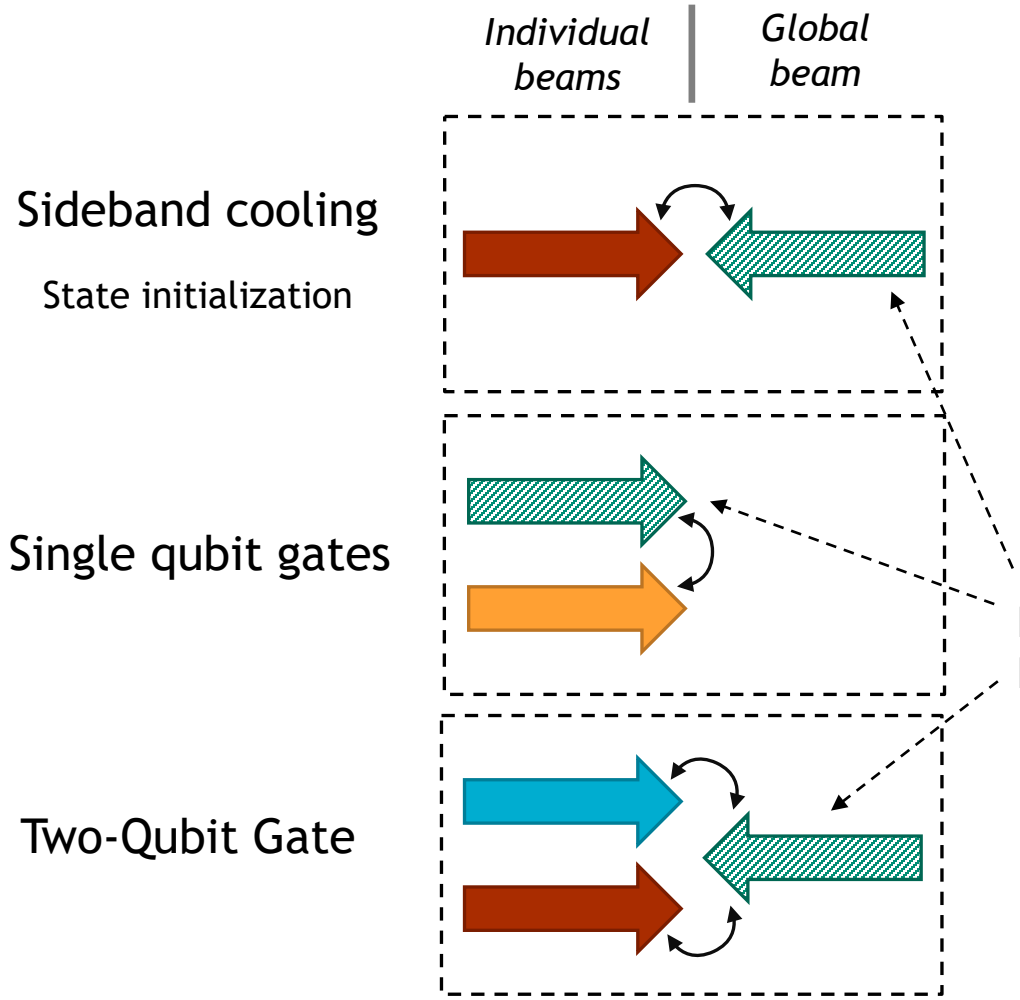


Lock must be applied to exactly one tone for each Raman pair!

Challenges: RF Reproducibility and Agility

QSCOUT

Three basic configurations




```
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
          waittrig=False)           # wait for external trigger
```

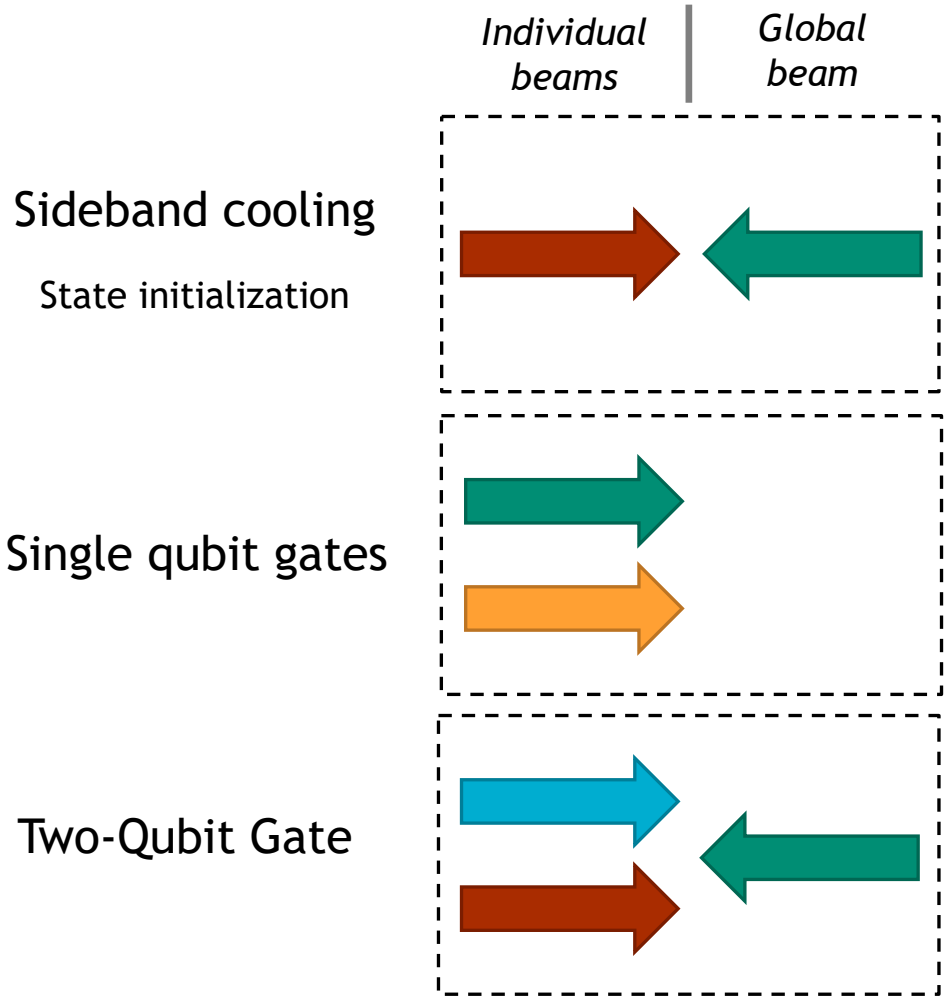
FIG. 1: Full argument signature of `PulseData` .

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

Challenges: RF Reproducibility and Agility

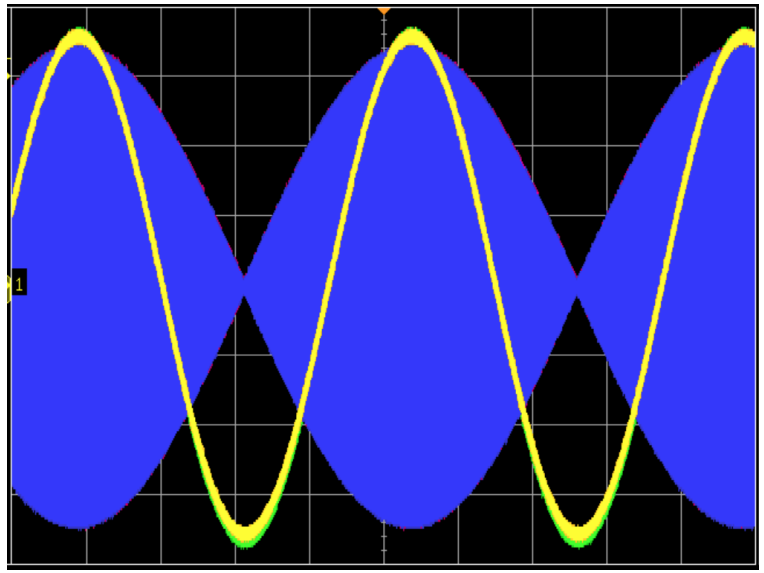
QSCOUT

Three basic configurations



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.

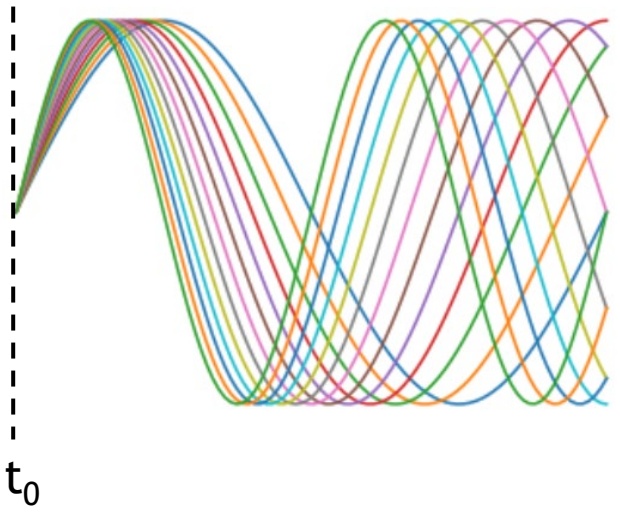
For absolute phase control, one must apply a synchronization trigger to a pulse by setting a non-zero value in the **sync_mask** argument of a PulseData object

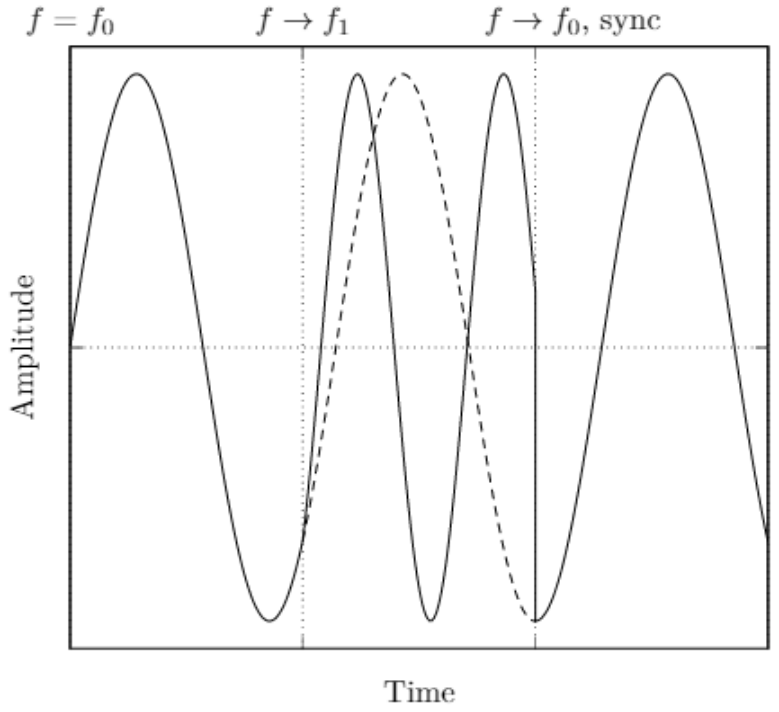
Synchronization will then set the internal oscillator phase to its free-running equivalent for a given frequency started from t_0

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

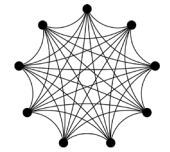
The sync_mask argument only applies to the beginning of the pulse

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

$$f_r + f_b = 2f_{qubit} \quad \text{where} \quad \begin{aligned} f_r &\equiv f_{qubit} - f_{\mathrm{SB}} \\ f_b &\equiv f_{qubit} + f_{\mathrm{SB}}. \end{aligned} \quad \text{but} \quad F_r + F_b \neq 2F_{qubit}$$

Best bet is to use the **discretize_frequency** helper function

```
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

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

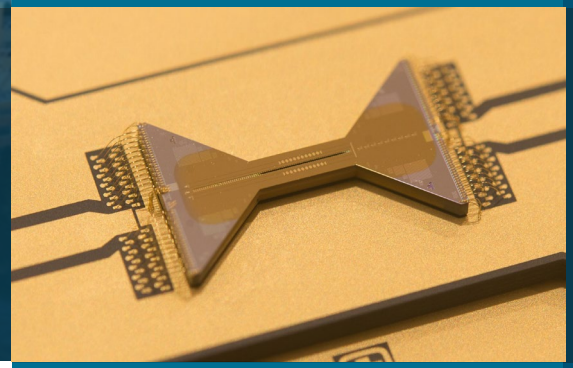
TABLE I: The fundamental input units for frequency, phase, and amplitude in `PulseData`. Note that the phase input is automatically converted modulo 360 such that $\theta \in [-180°, 180°)$. Amplitude is specified for a single tone, however the sum of the amplitude for two tones on the same channel must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

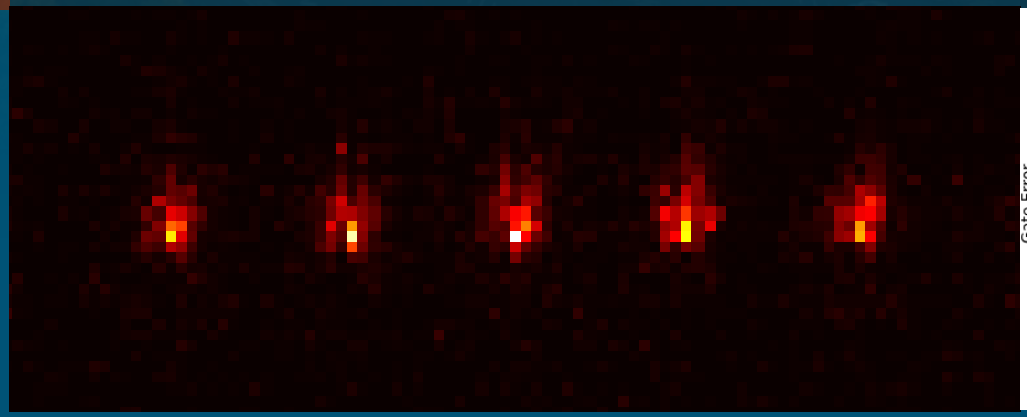
Spline input limits are a bit more subtle due to how spline coefficients are mapped to work with the on-chip interpolators. Best approach is to try running your code through jaqalpaw-emulate.

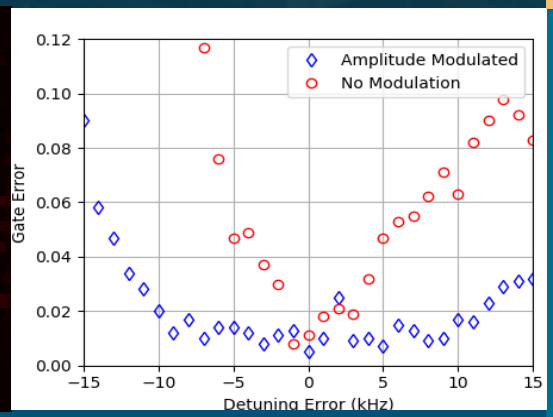
QSCOUT

Questions?

QSCOUT Webinar: JaqalPaw Example Programs







Presented by

Matthew Chow

March 1, 2023

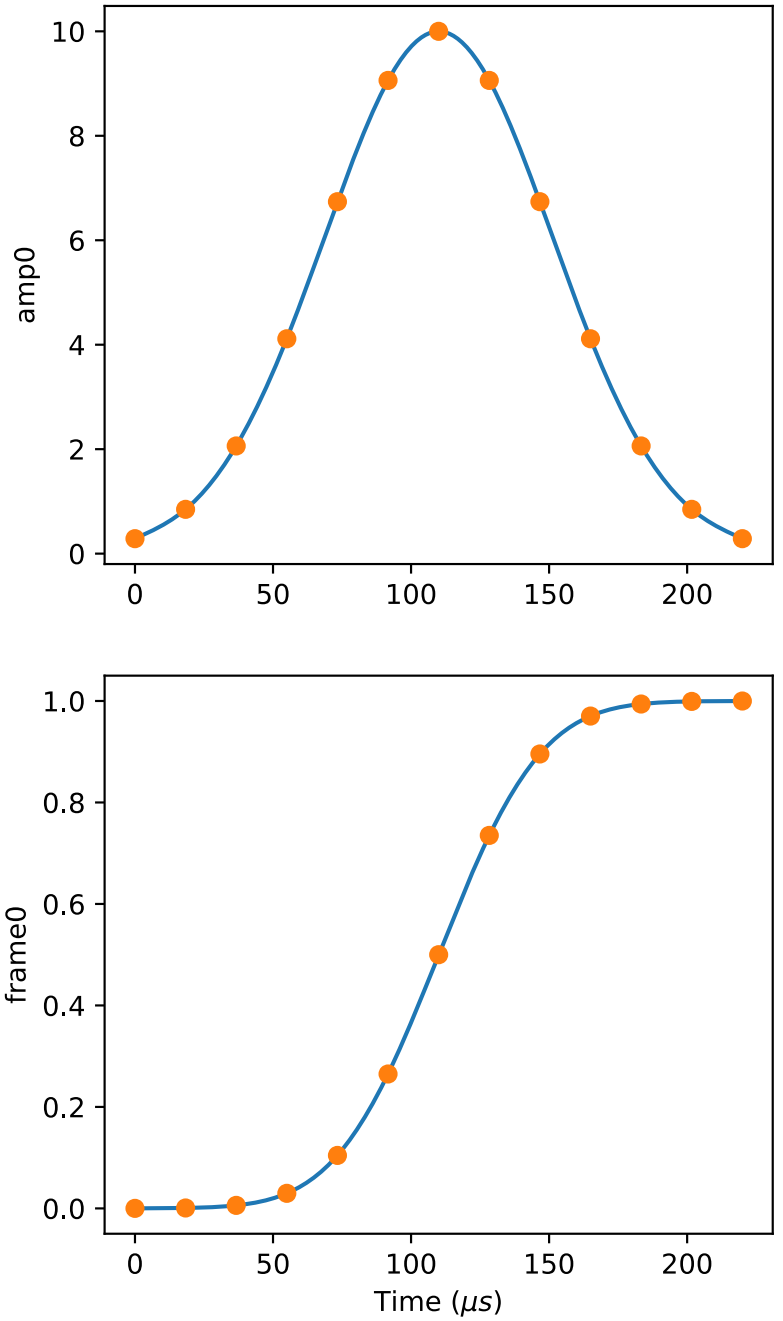
Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

SAND2023-12957PE

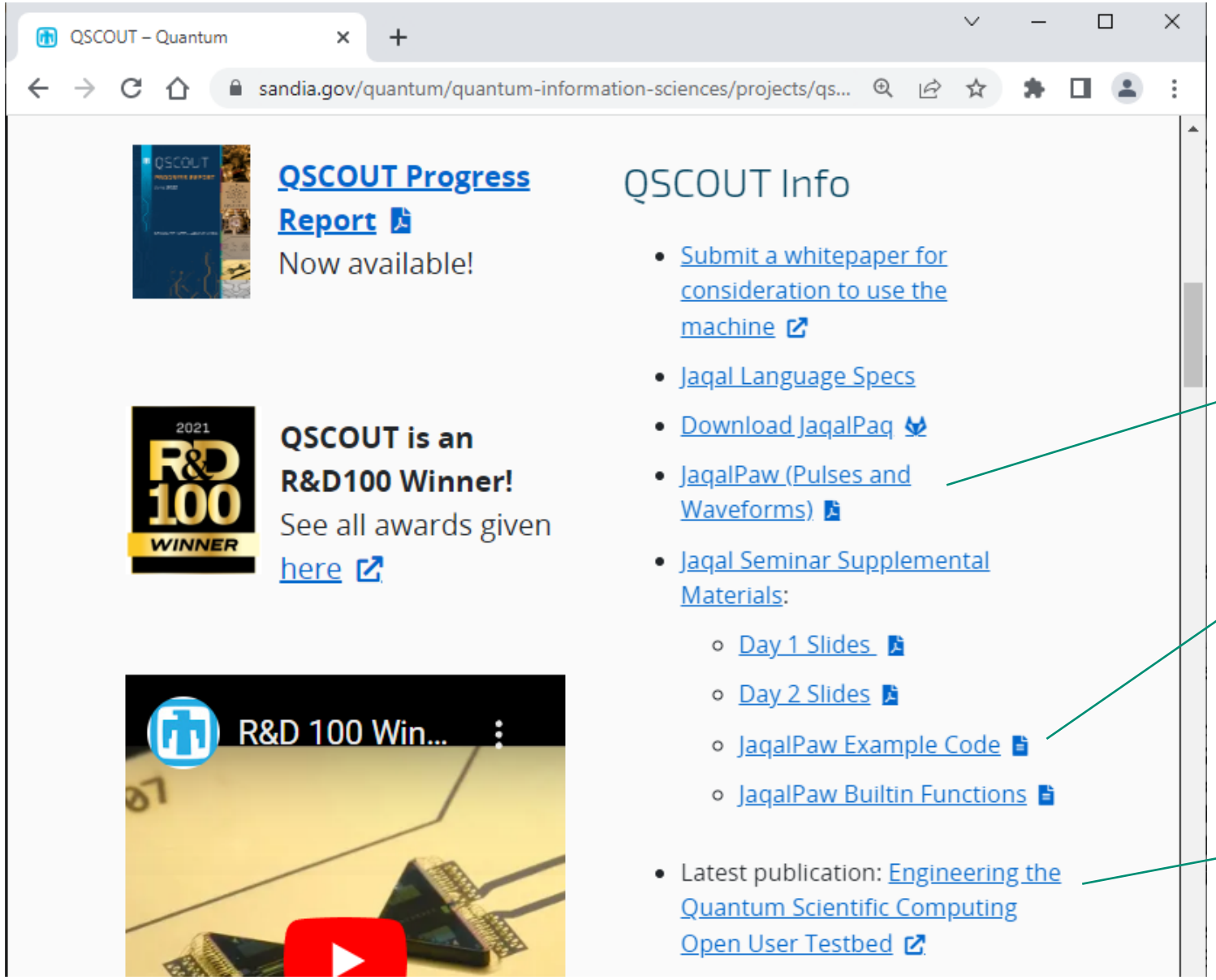
JaqalPaw Examples Tutorial

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



https://qscout.sandia.gov



Helpful guide for getting started

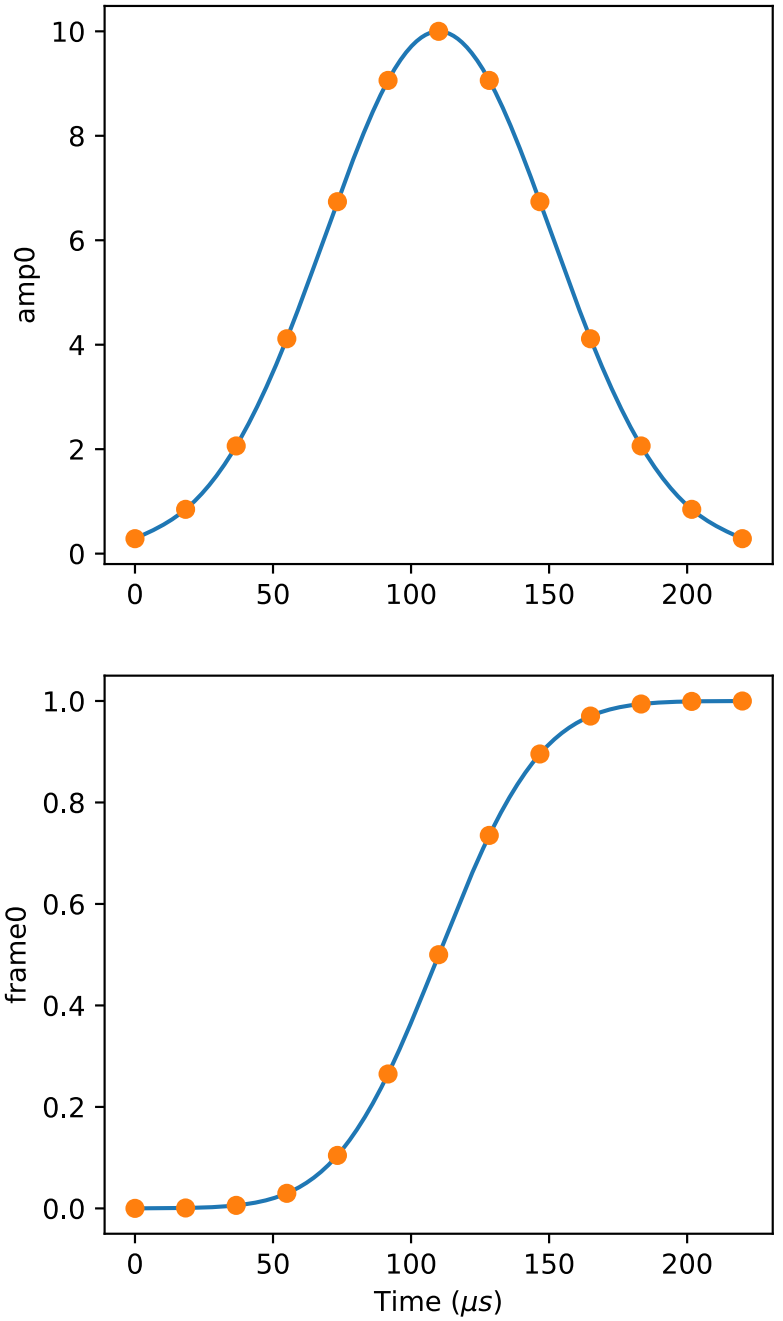
Example code (more to come here)

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

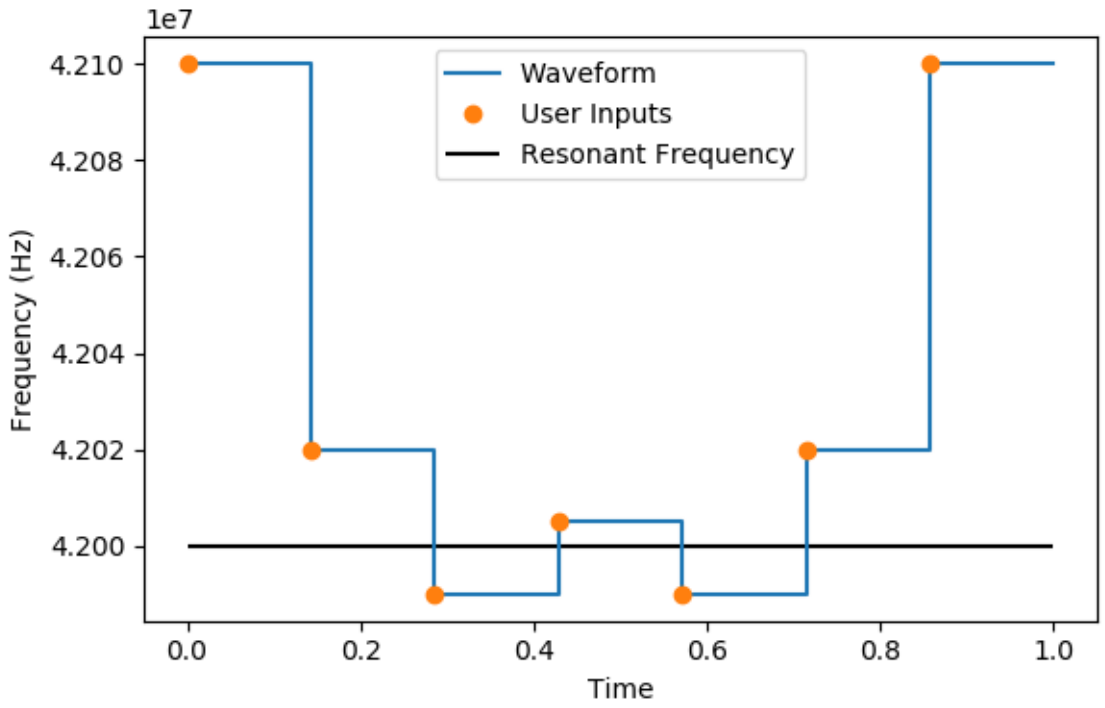


Example 1: Quick Review of Frequency Modulation

```
169  ## Frequency Modulation, simple example:
170  def gate_DiscreteFM_Microwave(self, channel_mw):
171      """ Frequency modulated microwave pulse. """
172
173      resonant_frequency = 42e6
174      detuning_knots = [100e3, 20e3, -10e3, 5e3, -10e3, 20e3, 100e3]
175      freq_fm0 = [resonant_frequency + d for d in detuning_knots]
176
177      return [PulseData(channel_mw, self.FM_pulse_duration,
178                  freq0 = freq_fm0,
179                  amp0 = 100,
180                  phase0=0
181                  )]
182
```

Frequency input is a list.
Gives frequency jumps.

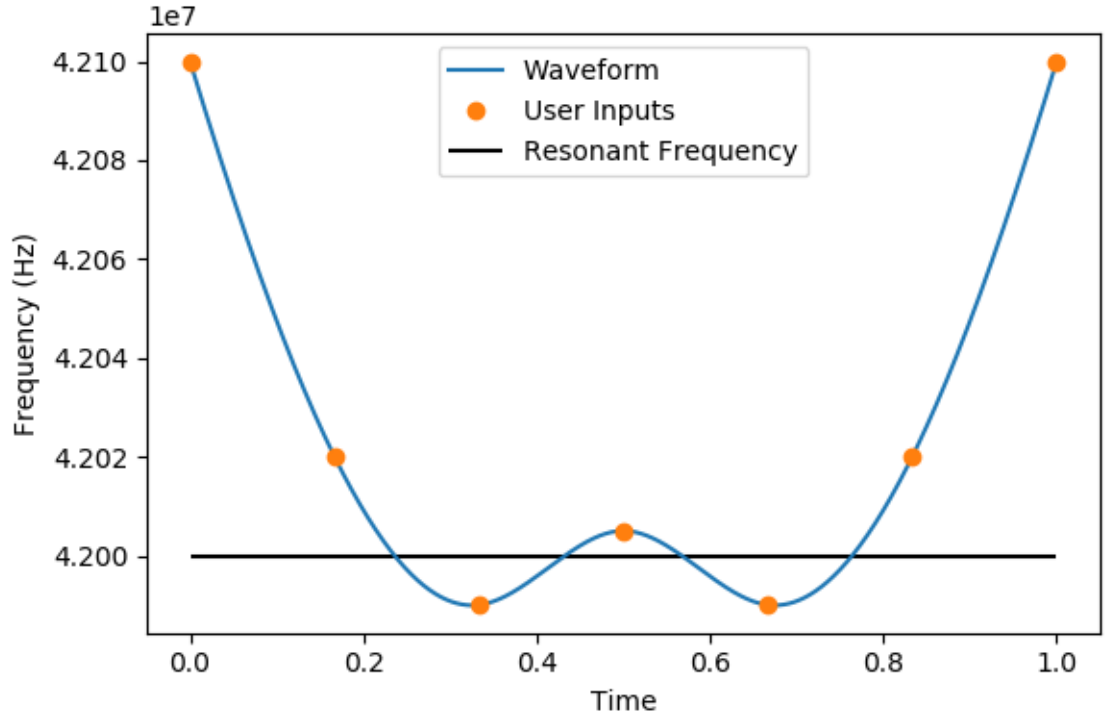
Frequency jumps applied at evenly spaced timing intervals.



Example 1b: Single Tone Continuous Frequency Modulation

```
183  ## Frequency Modulation, simple example:
184  def gate_ContinuousFM_Microwave(self, channel_mw):
185      """ Frequency modulated microwave pulse. """
186
187      resonant_frequency = 42e6
188      detuning_knots = [100e3, 20e3, -10e3, 5e3, -10e3, 20e3, 100e3]
189      freq_fm0 = tuple([resonant_frequency + d for d in detuning_knots])
190
191      return [PulseData(channel_mw, self.FM_pulse_duration,
192                  freq0 = freq_fm0,
193                  amp0 = 100,
194                  phase0=0
195                  )]
```

Tuple input gives a cubic spline interpolation.



Example 2: Simultaneous Amplitude and Phase (Gaussian Walsh)

```
198 class HelperFunctions:
199
200     @staticmethod
201     def gauss(npoints, A, freqwidth=300e3, total_duration=4e-6):
202         trange = np.linspace(-total_duration / 2, total_duration / 2, npoints)
203         sigma = 1 / (2 * np.pi * freqwidth)
204         return A * np.exp(-trange ** 2 / 2 / sigma ** 2)
205
```

```
210     gaussian_amps = list(self.gauss(npoints=13, A=100))
211     double_gauss = tuple(gaussian_amps * 2)
212
213     phase_steps = [0, 180]
214
215     return [PulseData(channel_global, self.MS_pulse_duration,
216                       freq0=self.freq0,
217                       amp0=double_gauss,
218                       phase0=phase_steps,
219                       sync_mask=3,
220                       fb_enable_mask=0),
```

Cast to list for convenient comprehension.

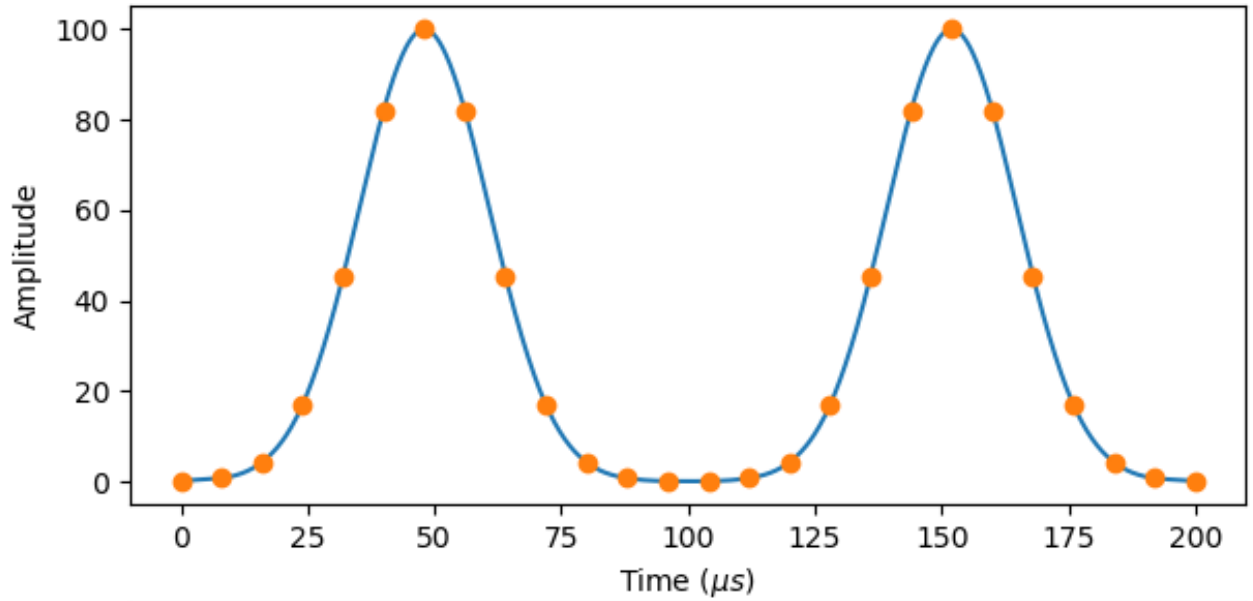
Cast to tuple for cubic spline.

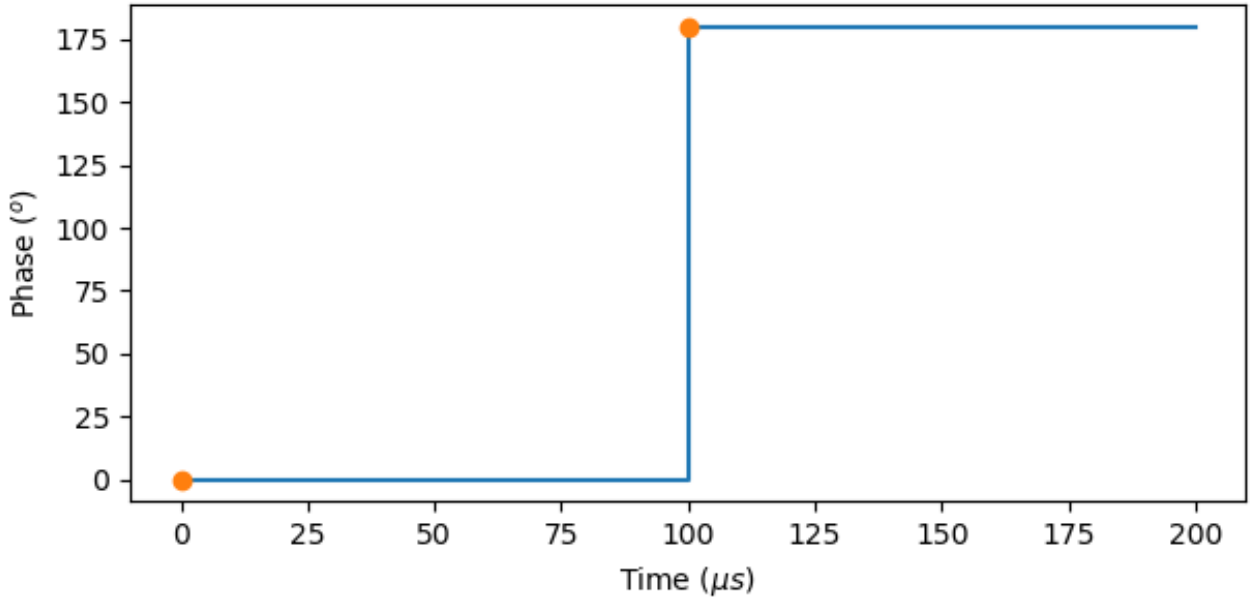
List of phase steps for discrete jump.

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

Walsh gate reference : Hayes et. al, 2012 : Phys. Rev. Lett. **109**, 020503

Example 2 Continued: Gaussian Walsh Gate Waveform

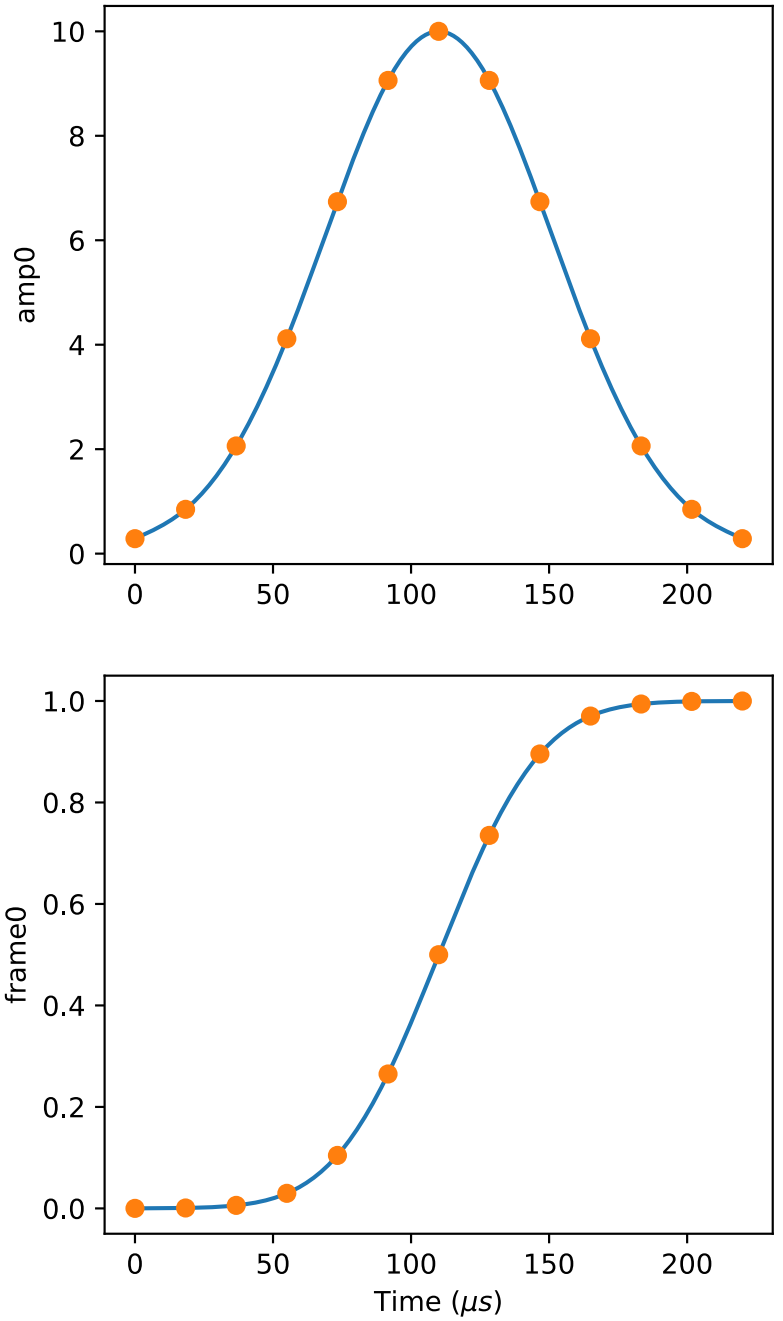




JaqalPaw Examples Tutorial

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.

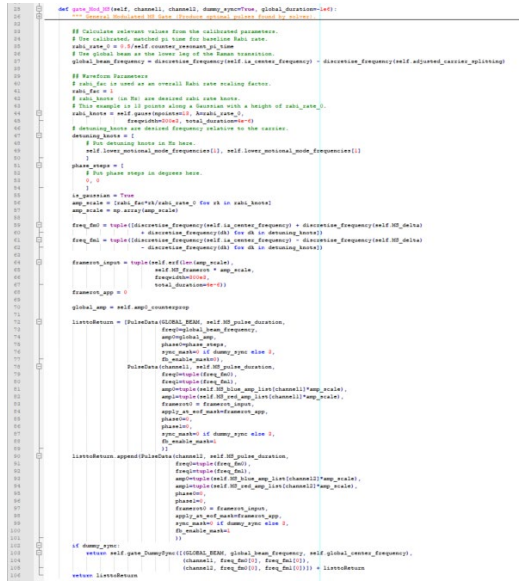
Jaqal code (10 substance lines)

```
// 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



This code available at qscout.sandia.gov

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.

Technical Details: Referencing Physical Calibrated Parameters

```
28  class CalibrationParameters:
29      """ Class that contains calibrated physical parameters and mapping
35      ## Raman carrier transition splitting and AOM center frequencies.
36      global_center_frequency: float = 200e6
37      ia_center_frequency: float = 230e6
38      adjusted_carrier_splitting: float = 28.6e6
39
40      ## Pricipal axis rotation (relative to Raman k_effective).
41      principal_axis_rotation: float = 45.0
42
43      ## Motional mode frequencies.
44      # Just 2 Ions in this example, list structure extends to N.
45      higher_motional_mode_frequencies: list = [-2.55e6, -2.45e6]
46      lower_motional_mode_frequencies: list = [-2.1e6, -2.05e6]
47
48      ## Matched pi time for single qubit gates.
49      co_ia_resonant_pi_time: float = 30e-6
50      counter_resonant_pi_time: float = 4e-6
51
52      ## Amplitudes to achieve matched pi times.
53      # Amplitude lists are indexed by RFSoC channel. [global,-,q0,q1,-,-,-,-]
54      amp0_coprop_list: list = [100, 0, 30, 30, 0, 0, 0, 30]
55      amp1_coprop_list: list = [100, 0, 30, 30, 0, 0, 0, 30]
56      amp0_counterprop: float = 100.0
57      amp1_counterprop_list: list = [0, 0, 30, 30, 0, 0, 0, 30]
58
59      ## Molmer Sorensen Gate Parameters
60      MS_pulse_duration: float = 1e-6
61      MS_delta: float = 0.0
62      MS_framerot: float = 0.0
63      MS_red_amp_list: list = [0, 0, 35, 30, 0, 0, 0, 35]
64      MS_blue_amp_list: list = [0, 0, 33, 27, 0, 0, 0, 33]
```

- 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!

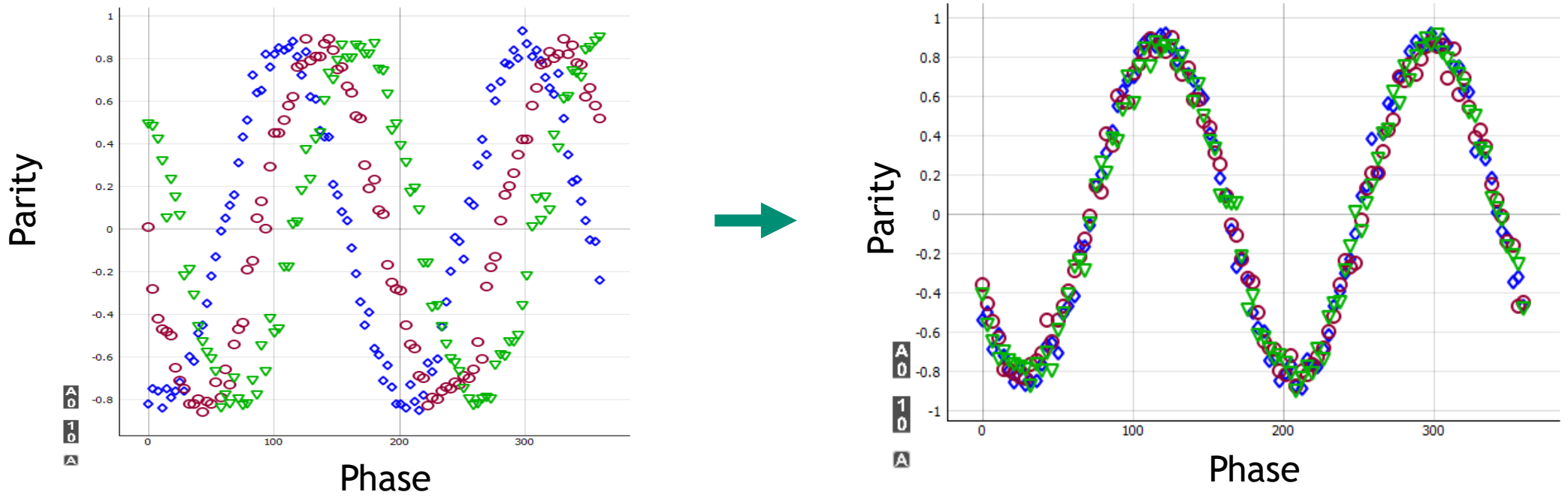
Technical Details: Math with Frequencies; Synchronization

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

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

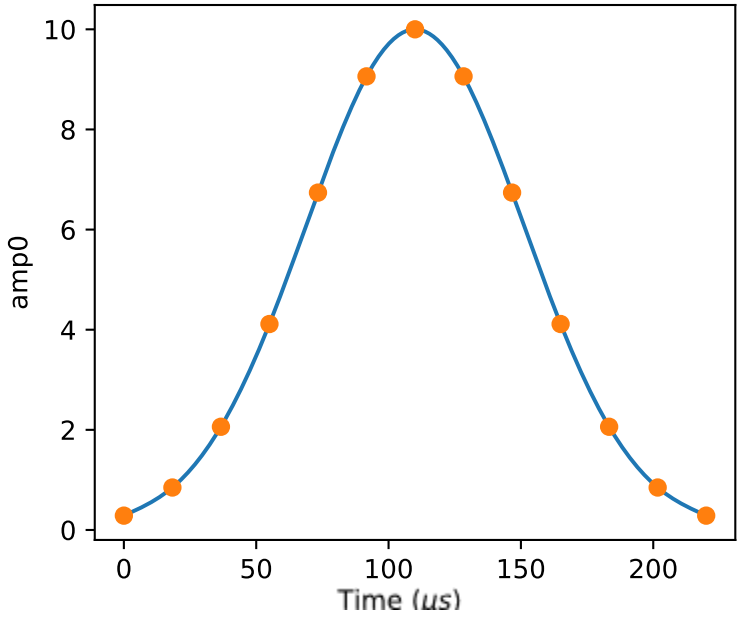
```
134   # Convert detuning knots to actual RF drive frequencies. Blue=fm0, Red=fm1
135   freq_fm0 = tuple([discretize_frequency(self.ia_center_frequency) + discretize_frequency(self.MS_delta)
136                     + discretize_frequency(dk) for dk in detuning_knots])
137   freq_fm1 = tuple([discretize_frequency(self.ia_center_frequency)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

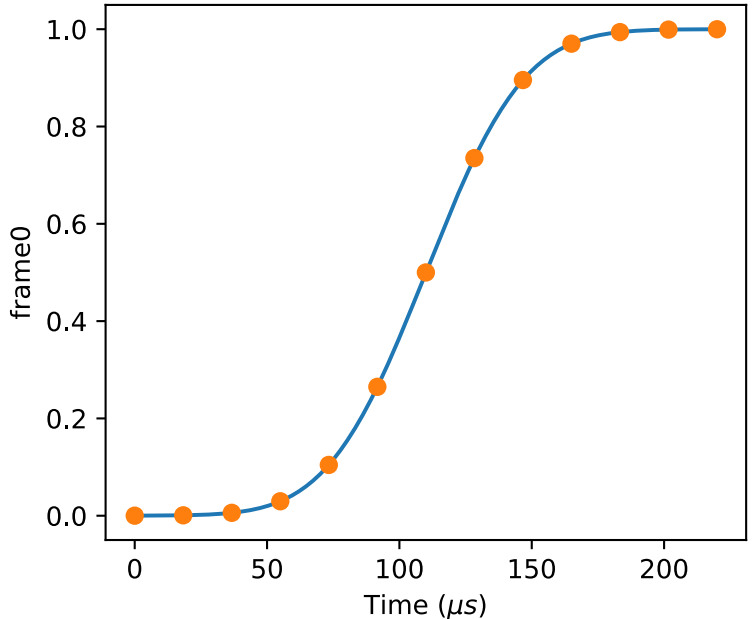
Synchronization: Usually, you should synchronize every tone in every pulse. (Syncmask = 3 or 0b11)



Technical Details: Frame rotation metadata

```
PulseData(channel1, self.MS_pulse_duration,
          freq0=blue_sb_freq,
          freq1=red_sb_freq,
          amp0=tuple(self.MS_blue_amp_list[channel1]*gaussian_amp),
          amp1=tuple(self.MS_red_amp_list[channel1]*gaussian_amp),
          framerot0 = tuple(erf_framerot),
          fwd_frame0_mask = 0b11,
          apply_at_eof_mask = ignored_value,
          phase0=self.MS_phi + axis,
          phase1=self.MS_phi + axis,
          sync_mask=both_tones,
          fb enable mask=no tones
```





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

Continuous framerot, natural cubic spline of an erf

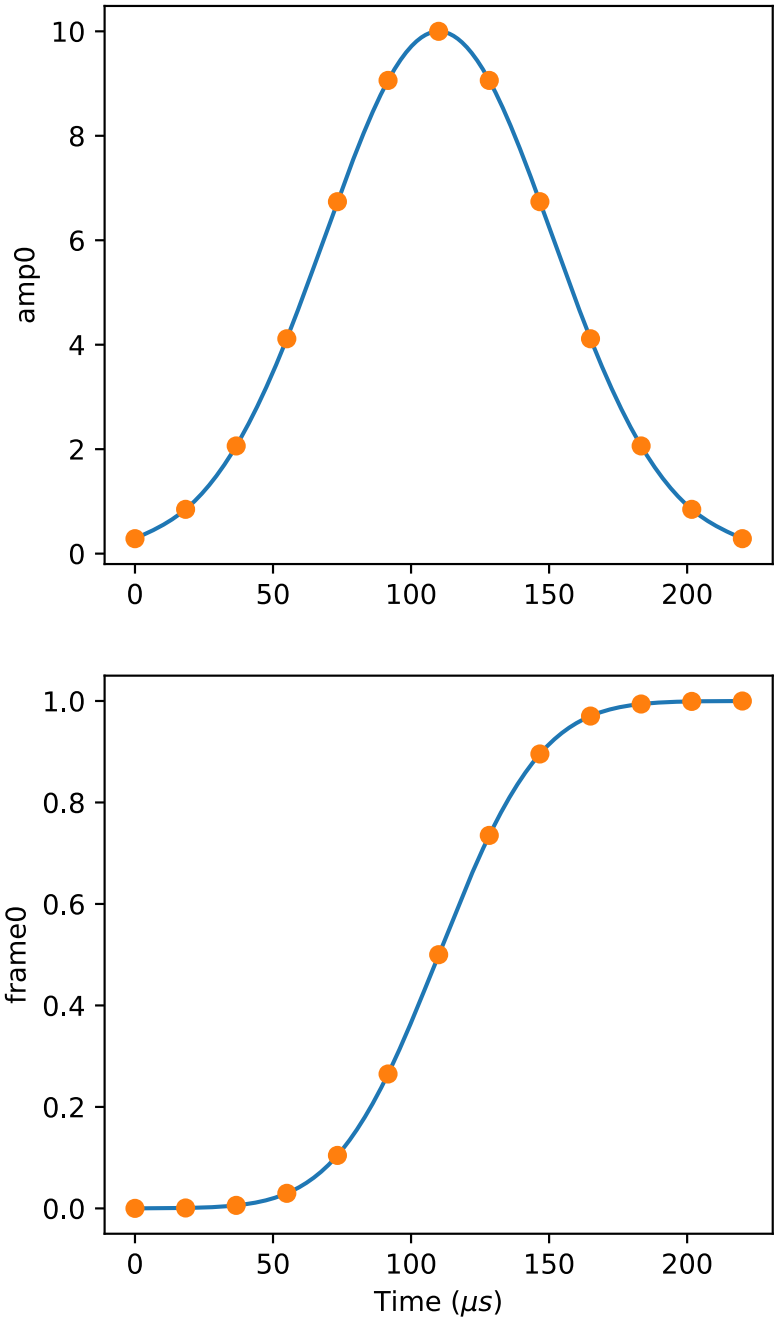
Forward to both tones, as red and blue sidebands need same phase for MS gate.

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



Build in Options by Parameterizing Pulses
Ex 3a: MS Amplitude Calibration – Simple

Simplest example of a parameterized pulse shape – just sweep the amplitude.

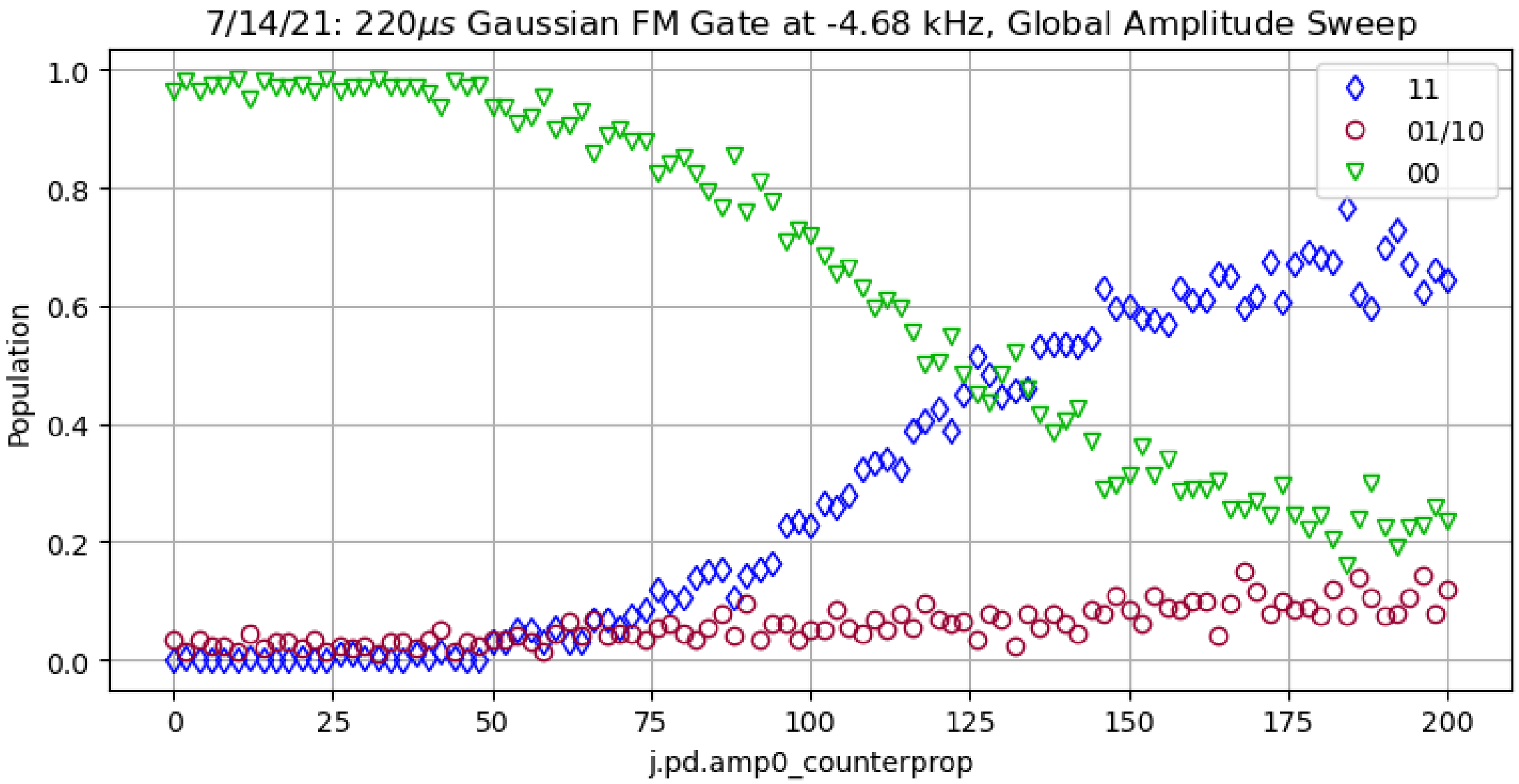
```
70  class ModulatedMSExemplar(QSCOUTBuiltins):
71
72      def gate_Mod_MS(self, channel1, channel2):
73
74          global_amp = self.amp0_counterprop
75
76          listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
77                                    freq0=global_beam_frequency,
78                                    amp0=global_amp,
79                                    phase0=phase_steps,
80                                    sync_mask=3,
81                                    fb_enable_mask=0),
```

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

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

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

Amplitude Sweep Data



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

```
124 class HelperFunctions:
125     @staticmethod
126     def gauss(npoints, A, freqwidth=300e3, total_duration=4e-6):
127         trange = np.linspace(-total_duration / 2, total_duration / 2, npoints)
128         sigma = 1 / (2 * np.pi * freqwidth)
129         return A * np.exp(-trange ** 2 / 2 / sigma ** 2)
130
131 class ModulatedMSExemplar(QSCOUTBuiltins, HelperFunctions):
132
133     def gate_Mod_MS(self, channel1, channel2):
134
135         global_amp = self.gauss(npoints=7, A=self.amp0_counterprop)
136
137         listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
138                                   freq0=global_beam_frequency,
139                                   amp0=global_amp,
140                                   phase0=phase_steps,
141                                   sync_mask=3,
142                                   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.

Build in Options by Parameterizing Pulses
Ex 3c: MS Pulse Shape

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.

```
147 class ModulatedMSExemplar(QSCOUTBuiltins, HelperFunctions):
148
149     def gate_Mod_MS(self, channel1, channel2, s):
150
151         global_amp = (1-s)*self.gauss(npoints=7, A=self.amp0_counterprop)
152                      + s*self.blackman(npoints=7, A=self.amp0_counterprop)
153
154         listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
155                                   freq0=global_beam_frequency,
156                                   amp0=global_amp,
157                                   phase0=phase_steps,
158                                   sync_mask=3,
159                                   fb_enable_mask=0),
160                         ]
```

The parameter “s” varies this pulse between Gaussian and Blackman.

Jaqal code modification:

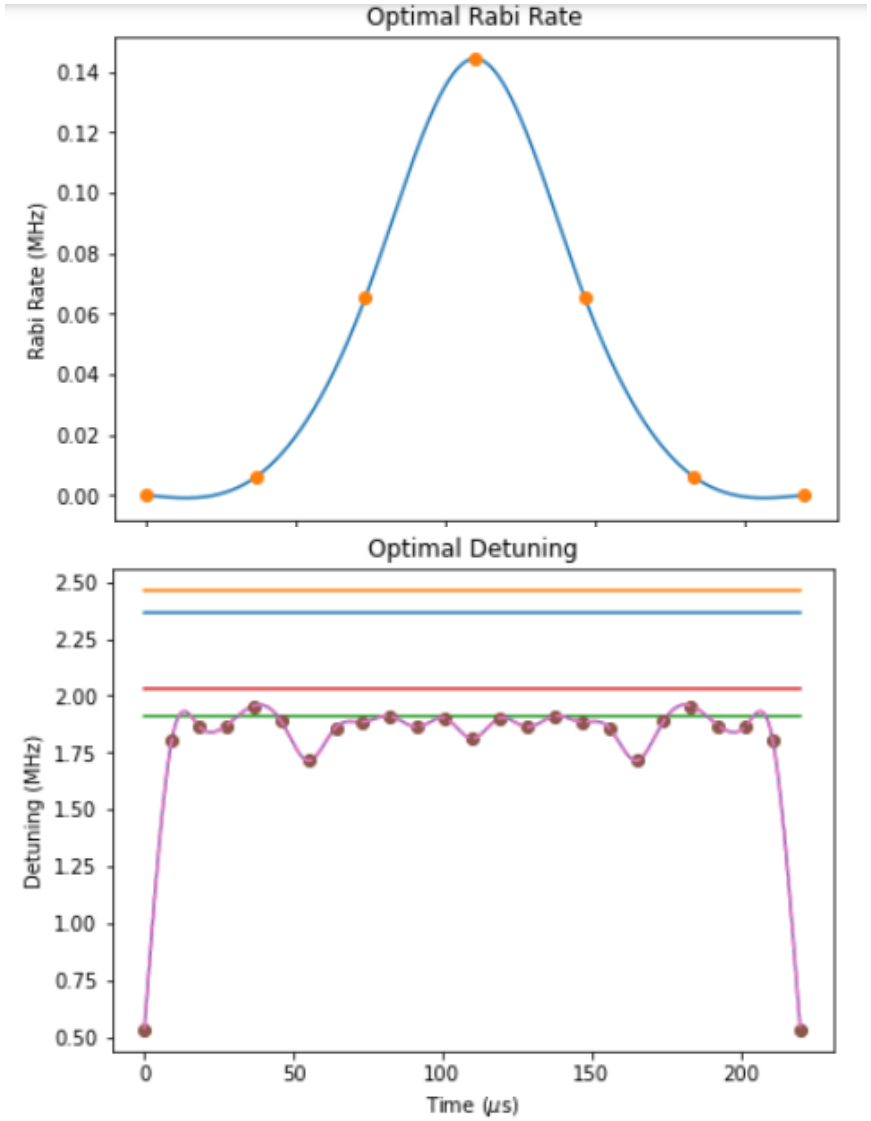
…

```
10 let s 0
```

…

```
16 loop ms_loops {
17     Mod_MS q[target1] q[target2] s
18 }
19
```

Use Both Tones to Generate More Complex Pulses
Ex 4: Track Spin State Dynamics Through a Gaussian FM Gate



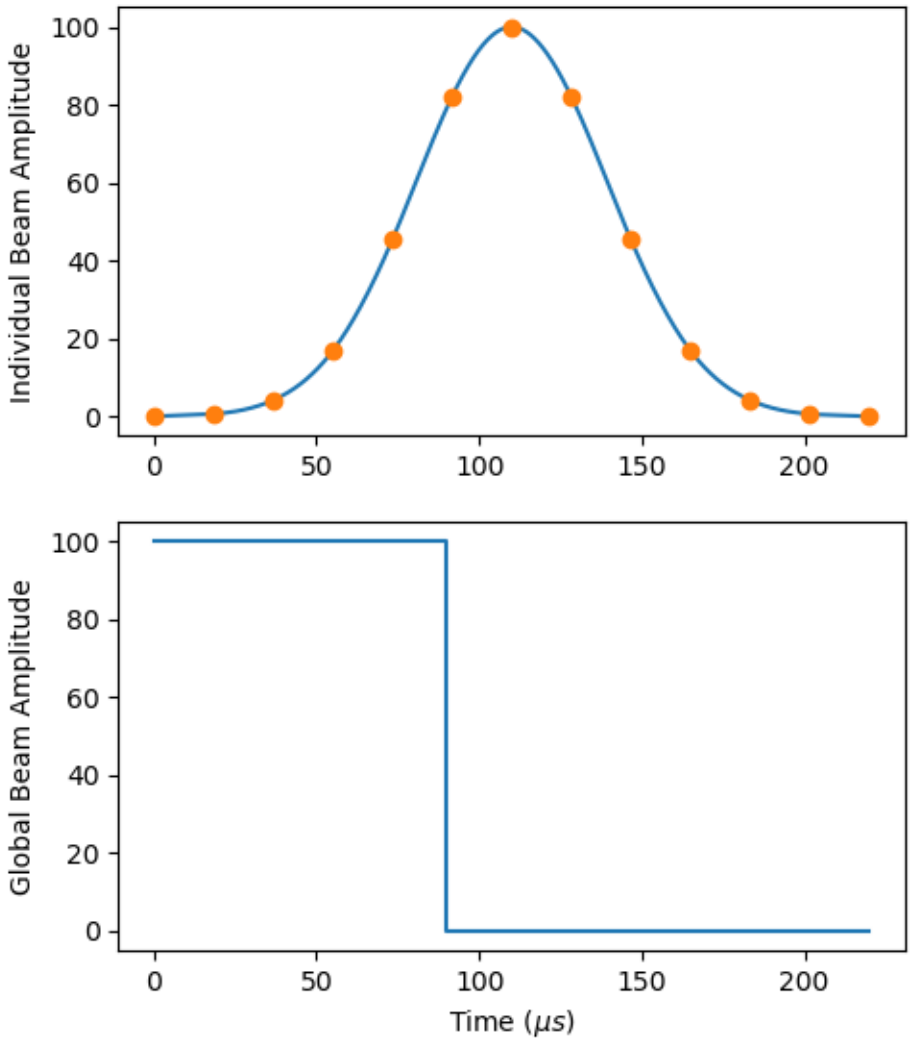
Goal: track the spin state during pulse

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

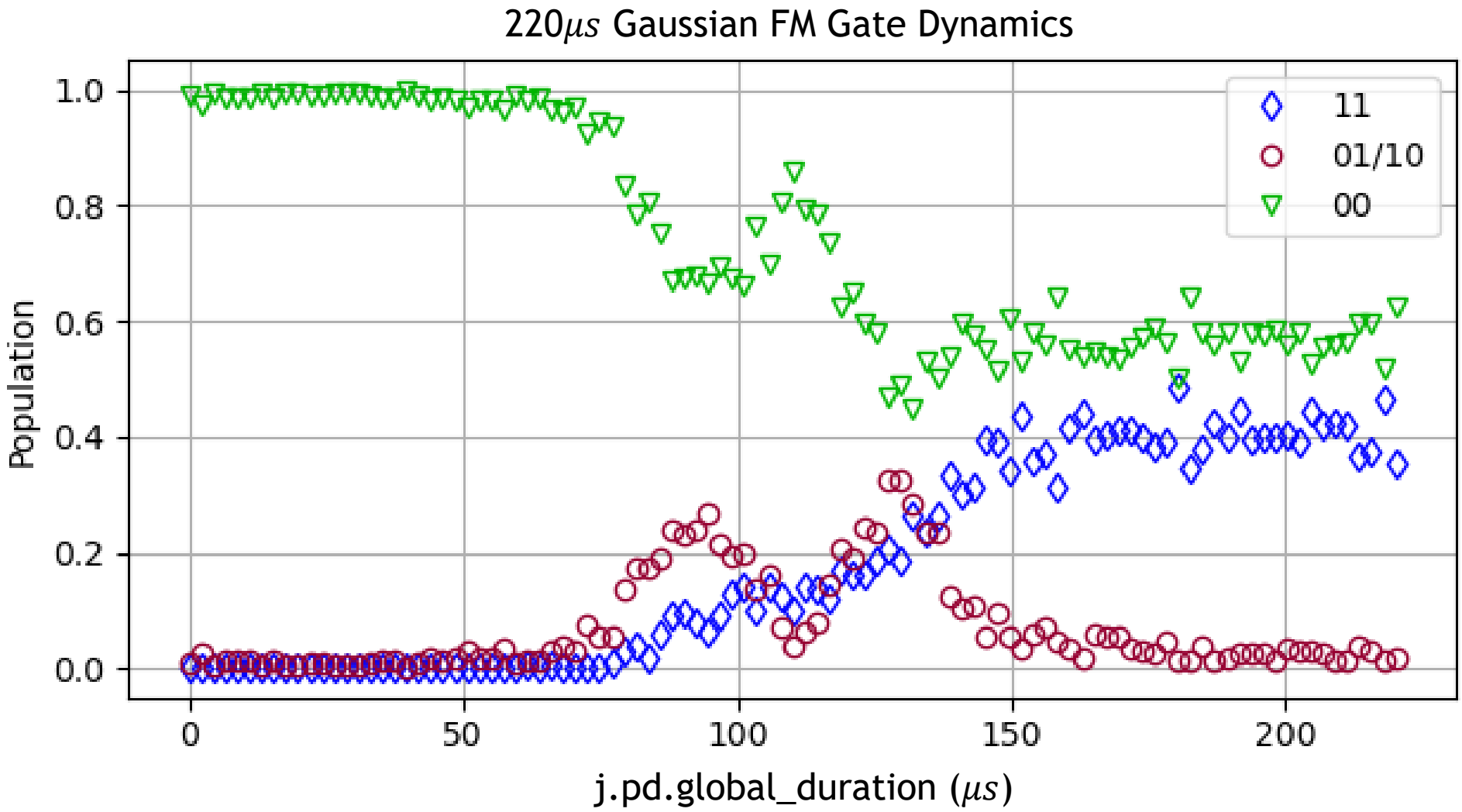
Sidestep the problem by putting discrete amplitude modulation on the other leg of the Raman transition.

Code for Spin State Tracking

```
240     def gate_Mod_MS(self, channel1, channel2, global_duration=-1e6):
241         """ General Modulated MS Gate (Produce optimal pulses found by solver).
247
248         ## Calculate gaussian knots to apply to IA beams.
249         rabi_rate_0 = 0.5/self.counter_resonant_pi_time
250         rabi_fac = 1
251         rabi_knots = self.gauss(npoints=13, A=rabi_rate_0,
252                                 freqwidth=300e3, total_duration=4e-6)
253         amp_scale = [rabi_fac*rk/rabi_rate_0 for rk in rabi_knots]
254         amp_scale = np.array(amp_scale)
255
256         ## If global duration is within valid range, shut off GLOBAL_BEAM after that time.
257         if global_duration >= 0 and global_duration < self.MS_pulse_duration:
258             global_amp = [self.amp0_counterprop if t <= global_duration
259                           else 0 for t in np.linspace(0, self.MS_pulse_duration, 1000)]
260         else:
261             global_amp = self.amp0_counterprop
262
263         ## Frequency knots and other waveform info here.
264
265         listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
266                                   freq0=global_beam_frequency,
267                                   freq1=self.global_center_frequency,
268                                   amp0=global_amp,
269                                   amp1=0,
270                                   phase0=phase_steps,
271                                   phase1=0,
272                                   sync_mask=0 if dummy_sync else 3,
273                                   fb_enable_mask=0),
274                         PulseData(channel1, self.MS_pulse_duration,
275                                   freq0=tuple(freq_fm0),
276                                   freq1=tuple(freq_fm1),
277                                   amp0=tuple(self.MS_blue_amp_list[channel1]*amp_scale),
278                                   amp1=tuple(self.MS_red_amp_list[channel1]*amp_scale),
279                                   framerot0 = framerot_input,
280                                   apply_at_eof_mask=framerot_app,
281                                   phase0=0,
282                                   phase1=0,
283                                   sync_mask=0 if dummy_sync else 3,
284                                   fb_enable_mask=1
285                                   ),
```



Data For Spin State Tracking



Programmatic Configurations

```
25  @staticmethod
26  def get_cfg(cfg_file, delimiter=':'):
27      """ Method for parsing simple cfguration files.
34      to_return = {}
35      if os.path.exists(cfg_file):
36          with open(cfg_file, 'r') as f:
37              for line in f:
38                  if len(line.strip()) > 0 and line.strip()[0] == "#":
39                      # Skip comment lines.
40                      continue
41                  if line.find("#") < 0:
42                      line = line[:line.find("#")] # Allow inline comments.
43                  if delimiter in line:
44                      line_list = line.split(delimiter)
45                      to_return[line_list[0]] = eval(line_list[1].strip())
46      return to_return
```

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.

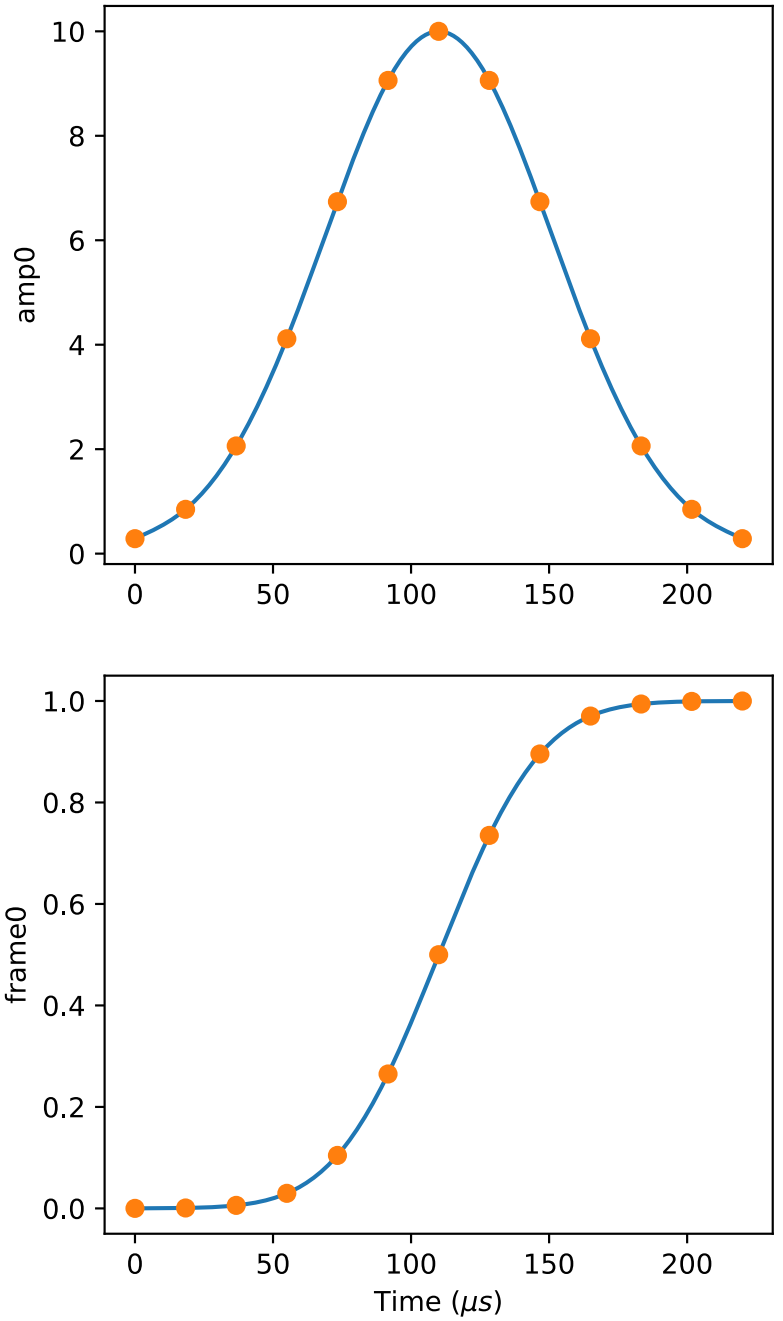
```
89          if from_file:
90              ## Use a cfguration file to read in waveform parameters.
91              cfg_file = r"D:\Users\Public\Documents\jaqalpaw_exemplar_modulated_ms\ExemplarConfig.txt"
92              cfg = self.get_cfg(cfg_file)
93
94              # Get amplitude knots from the cfguration file.
95              rabi_fac = cfg['rabi_fac']
96              rabi_knots = cfg['rabi_knots']
97
98              # Get detuning knots
99              detuning_knots = cfg['detuning_knots']
100
101             # Get phase steps (jumps, not spline knots) and convert to degrees.
102             phase_steps = [p*180/np.pi for p in cfg['phase_steps']]
103             if len(phase_steps) < 1:
104                 phase_steps = [0, 0]
```

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.

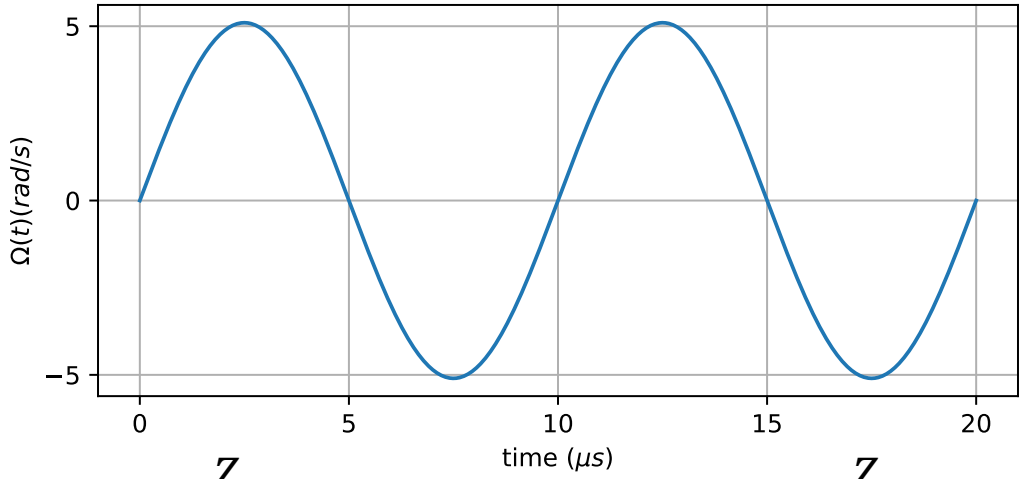
JaqalPaw Examples Tutorial

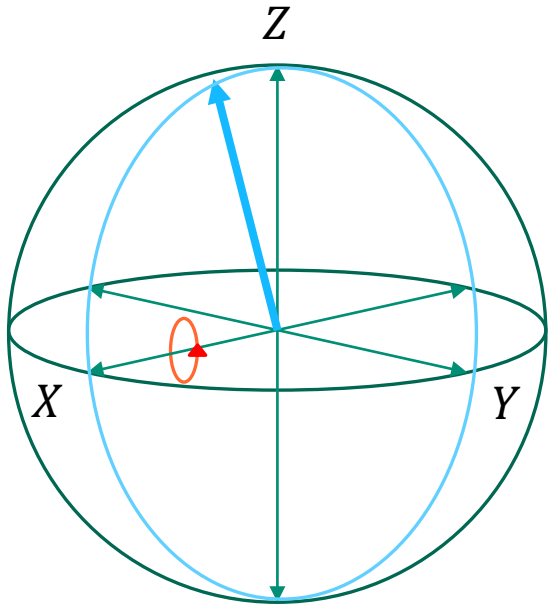
Topics:

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



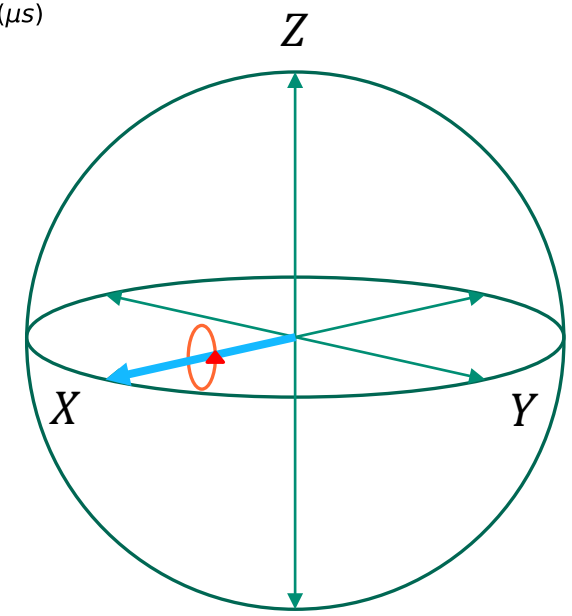
John's Hopkins Users (JHU) JaqalPaw 'SineLobe' Code





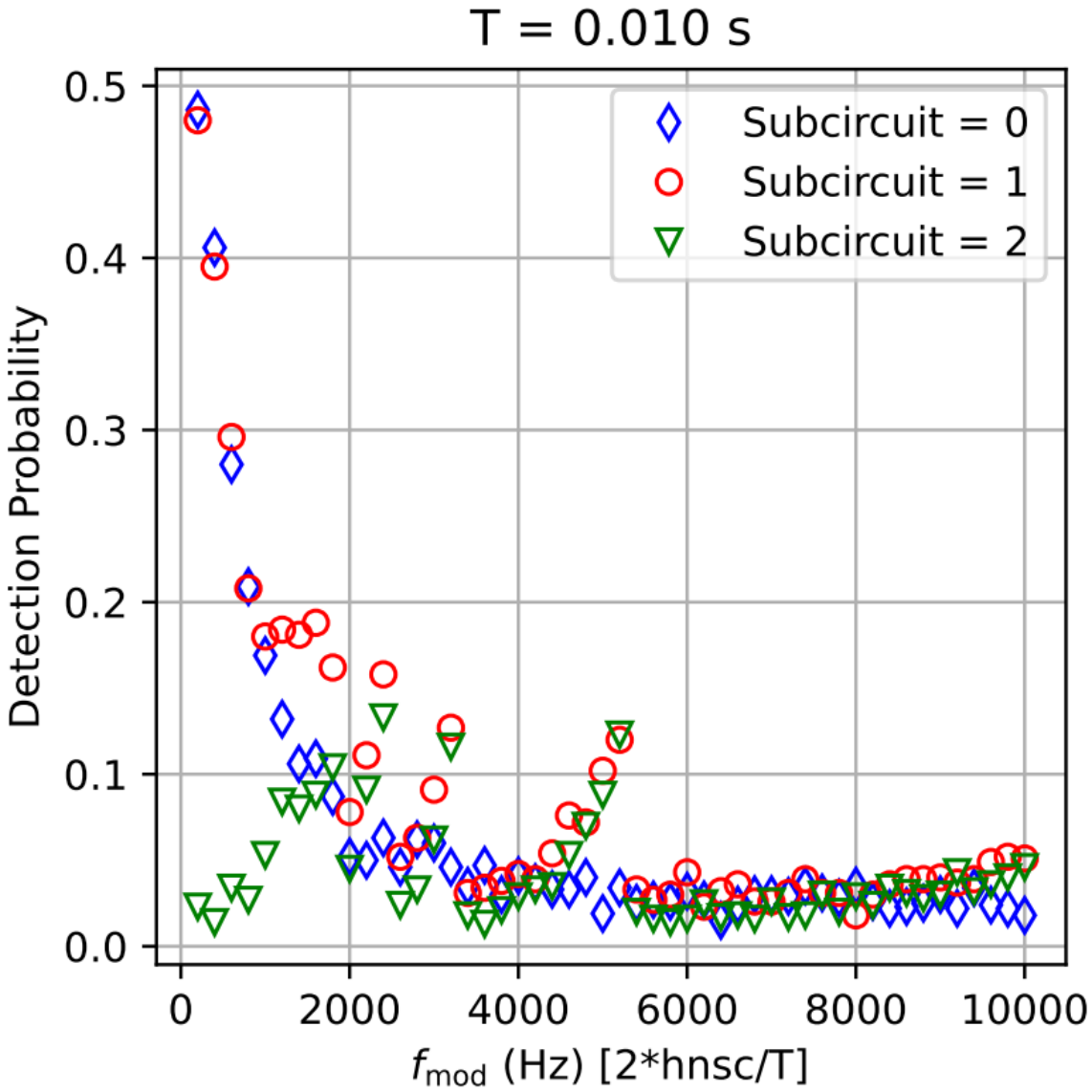
Subcircuit 0

$$R_Z\left(\frac{\pi}{2}\right) - R_x(t) - R_z\left(-\frac{\pi}{2}\right)$$



Subcircuit 2

$$R_y\left(\frac{\pi}{2}\right) - R_x(t) - R_y\left(-\frac{\pi}{2}\right)$$



JHU SineLobes – Calculate Relevant Parameters from Calibration Values

```
41      def gate_SineLobes(self, qubit, B, T, max_amplitude=60, correct_distortion=False, flip=False):
42          # max_amplitude = MAXAMP * 0.5 # don't consider amplitudes larger than this
43          num_spline_points = 25
44
45          """
46          pulse volume required to enact a pi pulse
47          """
48          amp0_scale = self.amp0_coprop_list[self.qubit_mapping[qubit]]
49          amp1_scale = self.amp1_coprop_list[self.qubit_mapping[qubit]]
50          t_scale = self.co_ia_resonant_pi_time
51          pi_pulse_volume = amp0_scale * amp1_scale * t_scale
52
53          """
54          This is specific to our protocol:
55              we require that the pulse volume of a single sine lobe corresponds
56              to a rotation which is 2x a root of the first Bessel function,
57              of which the first few roots are: 2.40, 5.51, 8.65, ...
58
59          This corresponds to rotations of: 1.53 pi, 3.51 pi, 5.51 pi, ...
60
61          We want to choose the largest such pulse volume which is less than `MAXAMP`
62          (or `max_amplitude = MAXAMP / 2` as set above if we want extra buffer room)
63          """
64
65          T_lobe = T / B
66
67          # RB change 20220915 - fixed formula for magical amplitude
68          bessel_roots = jn_zeros(0, 100) # first hundred roots
69          candidate_amp0s = bessel_roots * (B / T) * (amp0_scale * t_scale) * 2
```

JHU SineLobes – Validity checking and catching unphysical requirements

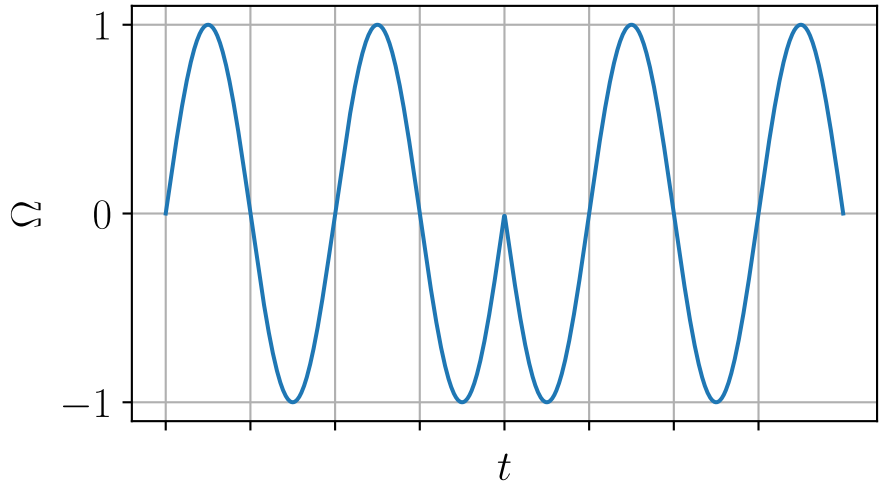
```
71          try:
72              # select the largest valid amplitude
73              if correct_distortion: # MC proposed change 20220810 - validity checking including distortion corre
74                  corrected_maxamps = amp0_scale*self.DiffractionEfficiencyLinMap(candidate_amp0s, qubit)/self.Di
75                  candidate_amp0s = corrected_maxamps[np.where(corrected_maxamps <= max_amplitude)]
76              else:
77                  candidate_amp0s = candidate_amp0s[np.where(candidate_amp0s <= max_amplitude)]
78              amp0 = np.max(candidate_amp0s)
79              print("Success, Bessel root amp0 found! amp0 = ", amp0) # 20220722 MC addition for debugging
80          except:
81              print("Minimum required amp0 for Bessel root = ", np.min(bessel_roots*(B/T) * (amp0_scale * t_scale)
82              raise ValueError("No suitable Dephasing-Robust amplitude found, check your parameters: T={}, B = {},
83
84          t = np.linspace(0, 2 * np.pi, num_spline_points)
85          sin_amps = np.sin(t) * amp0
```

JHU SineLobes – Parameterization allowing for options

```
41    def gate_SineLobes(self, qubit, B, T, max_amplitude=60, correct_distortion=False, flip=False):
42        # max_amplitude = MAXAMP * 0.5 # don't consider amplitudes larger than this
43        num_spline_points = 25
44
```

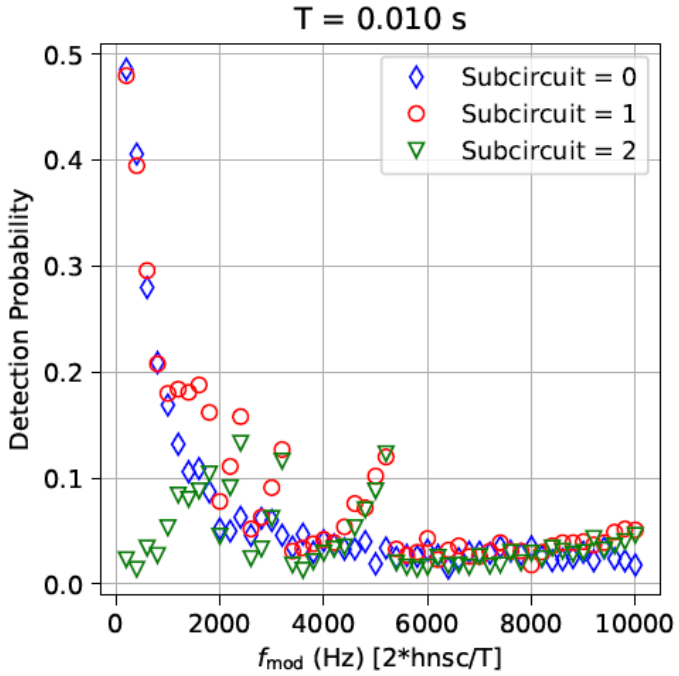
⋮

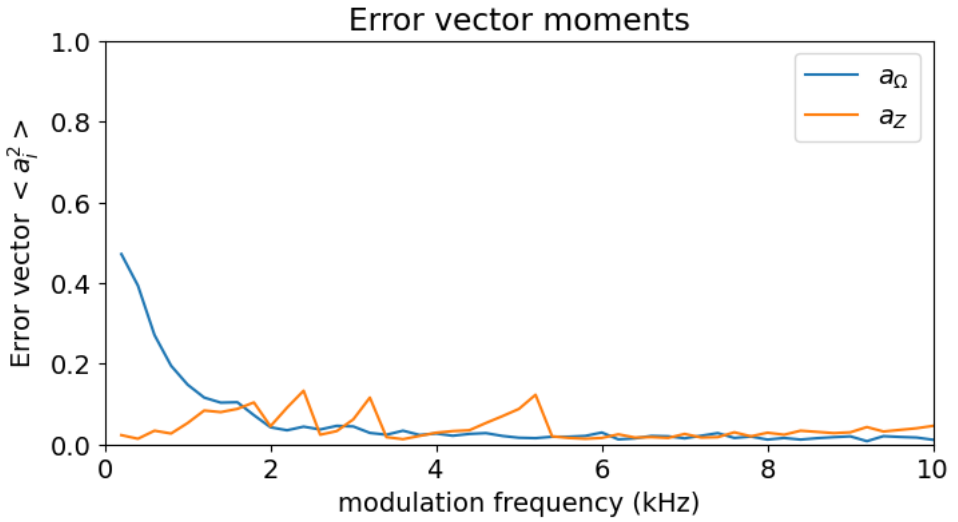
```
87        # MC proposed change 20220810 - apply distortion correction to amp0.
88        if correct_distortion:
89            sin_amps = amp0_scale*self.DiffractionEfficiencyLinMap(sin_amps, qubit)/self.DiffractionEfficiencyI
90
91        # RB proposed change 20220916 - apply waveform twice, positive then negative
92        if flip:
93            sin_amps = -sin_amps
```



JHU SineLobes – Final PulseData Object

```
95          sin_amps = tuple(sin_amps.tolist())
96
97          # the sine pulse is repeated `B` times, to fill up the total duration
98          return [PulseData(self.qubit_mapping[qubit],
99                            T_lobe,
100                           amp0=sin_amps,
101                           amp1=amp1_scale,
102                           freq0=self.ia_center_frequency-self.adjusted_carrier_splitting,
103                           freq1=self.ia_center_frequency,
104                           fb_enable_mask=tone0,
105                           fwd_frame0_mask=tone1, # This is mainly to account for any Stark shifts
106                           sync_mask=both_tones,
107                           )] # * B # uncomment this factor to go to a direct gate mode (no Jaqal
```





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()

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

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

Batching with Override Dictionary

In [32]:
```
#Create a jaqal code (string method)

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 diction
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, ju
#We can then call for the probability sorted by the integer of the binary represent
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) #absolu
    print(res.by_subbatch[i].by_subcircuit[0].relative_frequency_by_int) #with simu

#We can also sort the resulting probabilities by their string representation (matri
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) #absolu
    print(res.by_subbatch[i].by_subcircuit[0].relative_frequency_by_str) #with simu
```

```
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]]
[0.31000929 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.]
 [0.]
 [1.]]
```

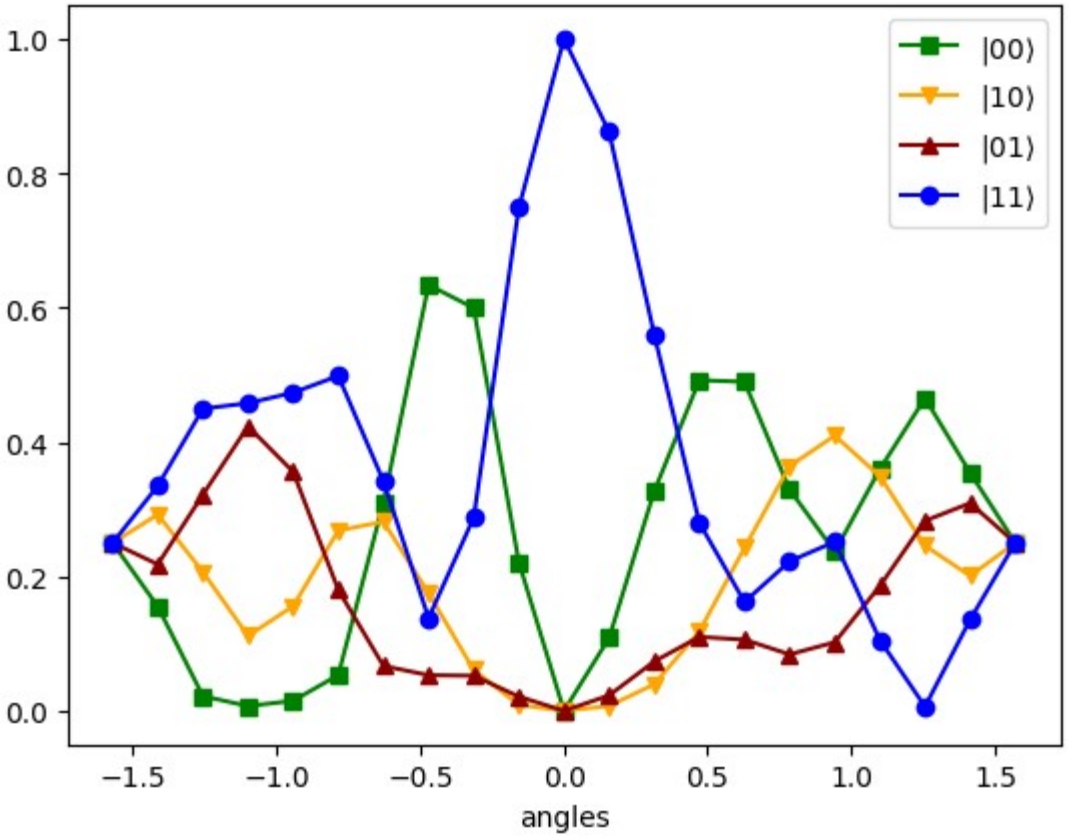
```
[0.108417   0.00719196 0.02298553 0.8614055 ]
[[0.112 ]
 [0.0065]
 [0.02  ]
 [0.8615]]
[0.32682367 0.03957536 0.07349173 0.56010924]
[[0.3235]
 [0.039 ]
 [0.076 ]
 [0.5615]]
[0.49195039 0.11809322 0.11003859 0.27991781]
[[0.496 ]
 [0.1135]
 [0.111 ]
 [0.2795]]
[0.48968353 0.24240519 0.10590413 0.16200716]
[[0.4825]
 [0.259 ]
 [0.1065]
 [0.152 ]]
[0.32923543 0.3642767  0.08394661 0.22254126]
[[0.335 ]
 [0.3645]
 [0.0845]
 [0.216 ]]
[0.23683399 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 0.28277381 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([0.2355]), '11': array([0.
2445])}
{'00': 0.15521142751307765, '10': 0.2920585780582002, '01': 0.21796374787072248, '1
1': 0.33476624655799975}
```

{'00': array([0.154]), '10': array([0.291]), '01': array([0.208]), '11': array([0.3
47])}
{'00': 0.02192059994387455, '10': 0.20659587429686832, '01': 0.3218363149609606, '1
1': 0.44964721079829656}
{'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': 0.4578703853537665}
{'00': array([0.008]), '10': array([0.1135]), '01': array([0.414]), '11': array([0.
4645])}
{'00': 0.014934251424144883, '10': 0.15583127627941548, '01': 0.35569488079623546,
'11': 0.47353959150020414}
{'00': array([0.0145]), '10': array([0.1555]), '01': array([0.346]), '11': array
([0.484])}
{'00': 0.05334708691207966, '10': 0.268305826175841, '01': 0.17991747852752235, '11
': 0.498429608384557}
{'00': array([0.0595]), '10': array([0.2625]), '01': array([0.1745]), '11': array
([0.5035])}
{'00': 0.31000928563221863, '10': 0.28146768785156573, '01': 0.0668416257031315, '1
1': 0.34168140081308407}
{'00': array([0.3075]), '10': array([0.278]), '01': array([0.066]), '11': array([0.
3485])}
{'00': 0.6338238629197582, '10': 0.17508098237206085, '01': 0.05305082507570646, '1
1': 0.1380443296324744}
{'00': array([0.64]), '10': array([0.1675]), '01': array([0.049]), '11': array([0.1
435])}
{'00': 0.5994921058869096, '10': 0.06079955040885978, '01': 0.05226753729546427, '1
1': 0.2874408064087663}
{'00': array([0.5975]), '10': array([0.065]), '01': array([0.059]), '11': array([0.
2785])}
{'00': 0.22137473690215836, '10': 0.008994782213862722, '01': 0.021182712390980435,
'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([0.]), '11': array([1.])}
{'00': 0.10841700433021274, '10': 0.007191964968329998, '01': 0.02298552963651315,
'11': 0.8614055010649442}
{'00': array([0.112]), '10': array([0.0065]), '01': array([0.02]), '11': array([0.8
615])}
{'00': 0.3268236684881671, '10': 0.03957536166136641, '01': 0.07349172604295771, '1
1': 0.5601092438075088}
{'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
': 0.27991780711110953}
{'00': array([0.496]), '10': array([0.1135]), '01': array([0.111]), '11': array([0.
2795])}
{'00': 0.4896835251099528, '10': 0.24240518785156598, '01': 0.1059041257031314, '11
': 0.16200716133534984}
{'00': array([0.4825]), '10': array([0.259]), '01': array([0.1065]), '11': array
([0.152])}
{'00': 0.3292354345603979, '10': 0.364276695296637, '01': 0.0839466094067262, '11':
0.22254126073623895}
{'00': array([0.335]), '10': array([0.3645]), '01': array([0.0845]), '11': array
([0.216])}

```
{'00': 0.23683398628649616, '10': 0.4094597995646406, '01': 0.10206635751101029, '1
1': 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
': 0.0062955999438744255}
{'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, '1
1': 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]:

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

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



Batching with Subcircuits

In [35]:
```python
#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] += ['Pz']
            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):
        equivalent_pauli = '' #No gates
        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 = "\n"+ equivalent_pauli + f" q[{index}]"
            gate_string += gate
    return gate_string
```

In [36]:
```
#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] 0 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>
<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] 0 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>
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>
Pz q[0]
MS q[0] q[1] 0 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] 0 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>
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]
Py q[1]
MS q[0] q[1] 0 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
```

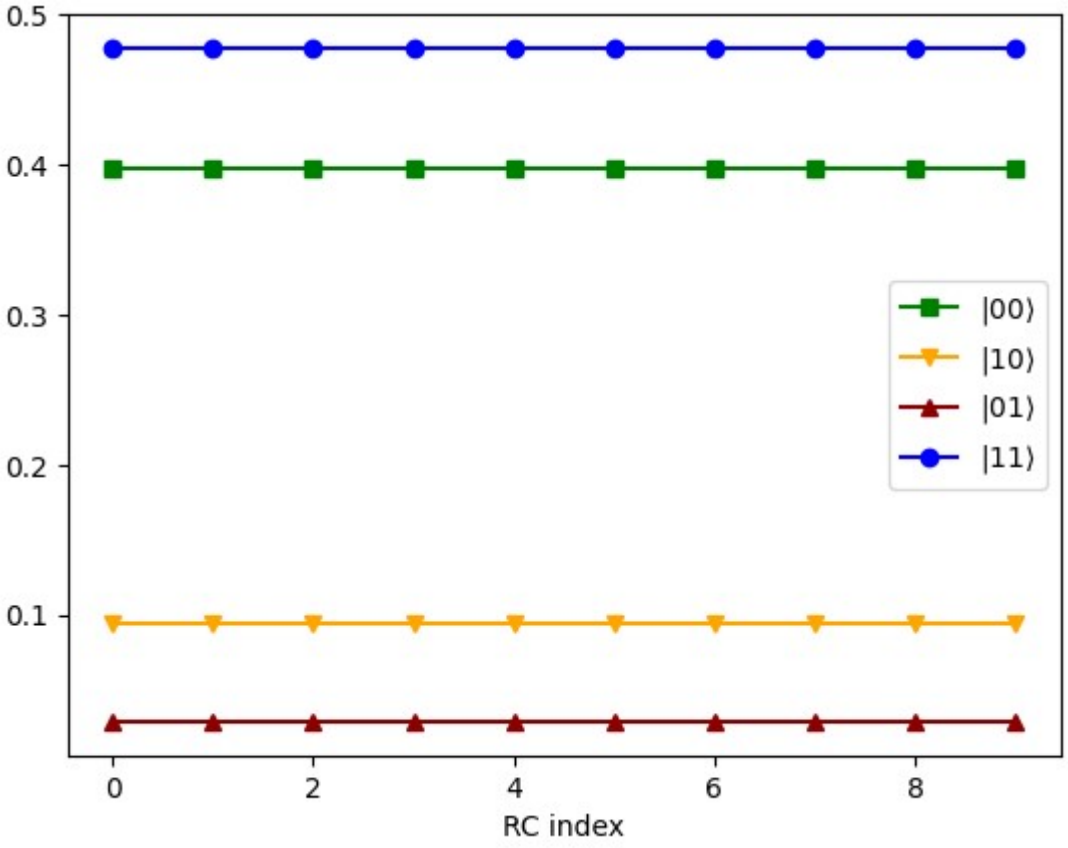
In [38]:
```
#Running the string

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 of the binary represent
print("PROBABILITIES ORDERED BY QUBIT INTEGER VALUE")
for i in range(num_RC):
    print(res.by_subbatch[0].by_subcircuit[i].simulated_probability_by_int) #absolu
    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 (matri
print("PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION")
for i in range(num_RC):
    print(res.by_subbatch[0].by_subcircuit[i].simulated_probability_by_str) #absolu
    print(res.by_subbatch[0].by_subcircuit[i].relative_frequency_by_str) #with simu
```

```
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': 0.3981868612330396, '10': 0.0948427056058017, '01': 0.0290400251628833, '11
': 0.4779304079982755}
{'00': array([0.44]), '10': array([0.09]), '01': array([0.02]), '11': array([0.4
5])}
```

```
{'00': 0.3981868612330396, '10': 0.0948427056058017, '01': 0.0290400251628833, '11
': 0.4779304079982755}
{'00': array([0.355]), '10': array([0.08]), '01': array([0.03]), '11': array([0.53
5])}
{'00': 0.3981868612330396, '10': 0.09484270560580167, '01': 0.02904002516288331, '1
1': 0.4779304079982755}
{'00': array([0.495]), '10': array([0.085]), '01': array([0.015]), '11': array([0.4
05])}
{'00': 0.3981868612330396, '10': 0.09484270560580167, '01': 0.02904002516288329, '1
1': 0.4779304079982755}
{'00': array([0.39]), '10': array([0.08]), '01': array([0.03]), '11': array([0.5])}
{'00': 0.3981868612330396, '10': 0.09484270560580167, '01': 0.02904002516288331, '1
1': 0.4779304079982755}
{'00': array([0.435]), '10': array([0.09]), '01': array([0.02]), '11': array([0.45
5])}
{'00': 0.3981868612330396, '10': 0.0948427056058017, '01': 0.029040025162883322, '1
1': 0.4779304079982755}
{'00': array([0.395]), '10': array([0.1]), '01': array([0.035]), '11': array([0.4
7])}
{'00': 0.3981868612330396, '10': 0.09484270560580167, '01': 0.02904002516288329, '1
1': 0.4779304079982755}
{'00': array([0.405]), '10': array([0.095]), '01': array([0.02]), '11': array([0.4
8])}
{'00': 0.3981868612330396, '10': 0.0948427056058017, '01': 0.0290400251628833, '11
': 0.4779304079982755}
{'00': array([0.38]), '10': array([0.115]), '01': array([0.01]), '11': array([0.49
5])}
{'00': 0.3981868612330397, '10': 0.0948427056058017, '01': 0.029040025162883322, '1
1': 0.47793040799827535}
{'00': array([0.4]), '10': array([0.115]), '01': array([0.03]), '11': array([0.45
5])}
{'00': 0.3981868612330397, '10': 0.0948427056058017, '01': 0.02904002516288334, '11
': 0.47793040799827535}
{'00': array([0.405]), '10': array([0.075]), '01': array([0.025]), '11': array([0.4
95])}
```

In [39]:
```
plt.figure(2)
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])
for n in range(num_states):
    plt.plot(range(num_RC),
             outcomes[n],
             label=f"$\\left\\vert{''.join(reversed(f'{n:02b}'))}\\right\\rangle$",
             marker="sv^o"[n%4])
plt.legend()
plt.xlabel("RC index")
```

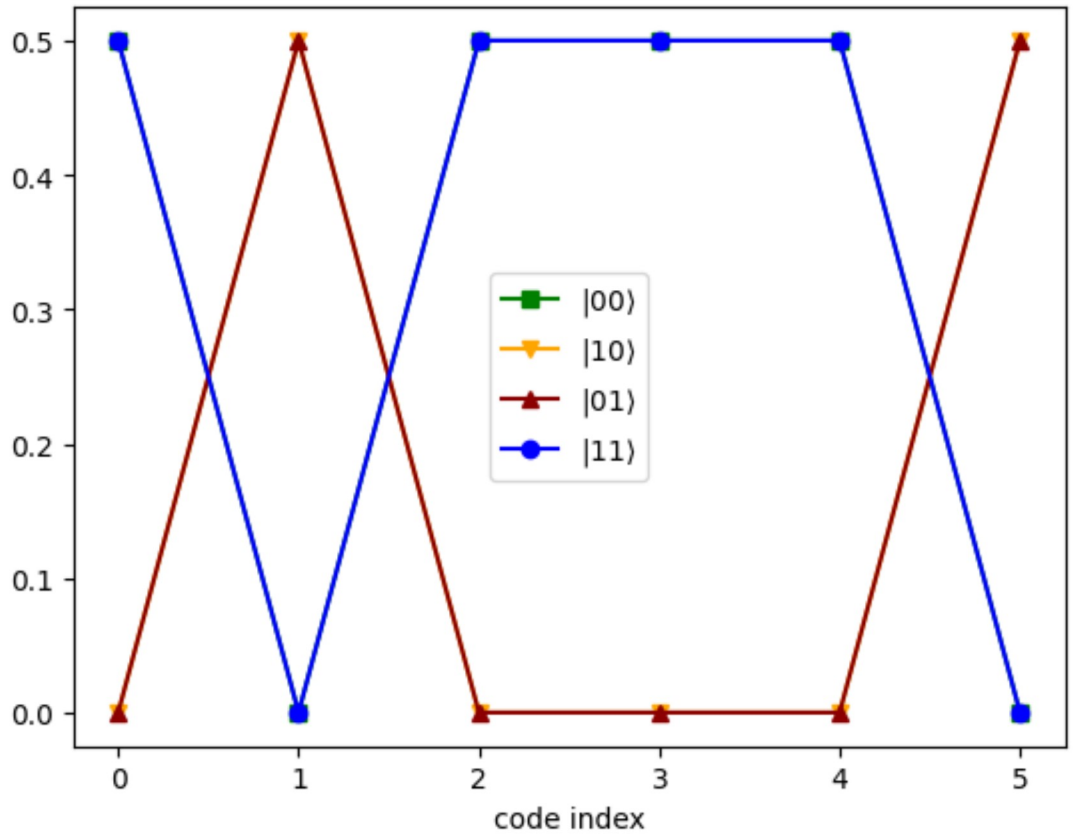
Out[39]:
```
Text(0.5, 0, 'RC index')
```



Batching with Indexing

In [40]:
```
#Nominally, the circuits are run in the order presented in the jaqal file, but if w

#For simplicity we will define another series of jaqal_code subcircuits


jaqal_code = f"""
from .custom_gates usepulses *

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

register q[{num_qubits}]

prepare_all
MS q[0] q[1] 0 pi_2
Rz q[0] pi_2
measure_all

prepare_all
MS q[0] q[1] 0 pi_2
Rz q[0] pi_2
<Sx q[0] | Sx q[1]>
measure_all

prepare_all
MS q[0] q[1] 0 pi_2
Rz q[0] pi_2
<Sy q[0] | Sy q[1]>
measure_all

prepare_all
MS q[0] q[1] pi_2 pi_2
Rz q[0] pi_2
measure_all

prepare_all
MS q[0] q[1] pi_2 pi_2
Rz q[0] pi_2
<Sx q[0] | Sx q[1]>
measure_all

prepare_all
MS q[0] q[1] pi_2 pi_2
Rz q[0] pi_2
<Sy q[0] | Sy q[1]>
measure_all
"""
```

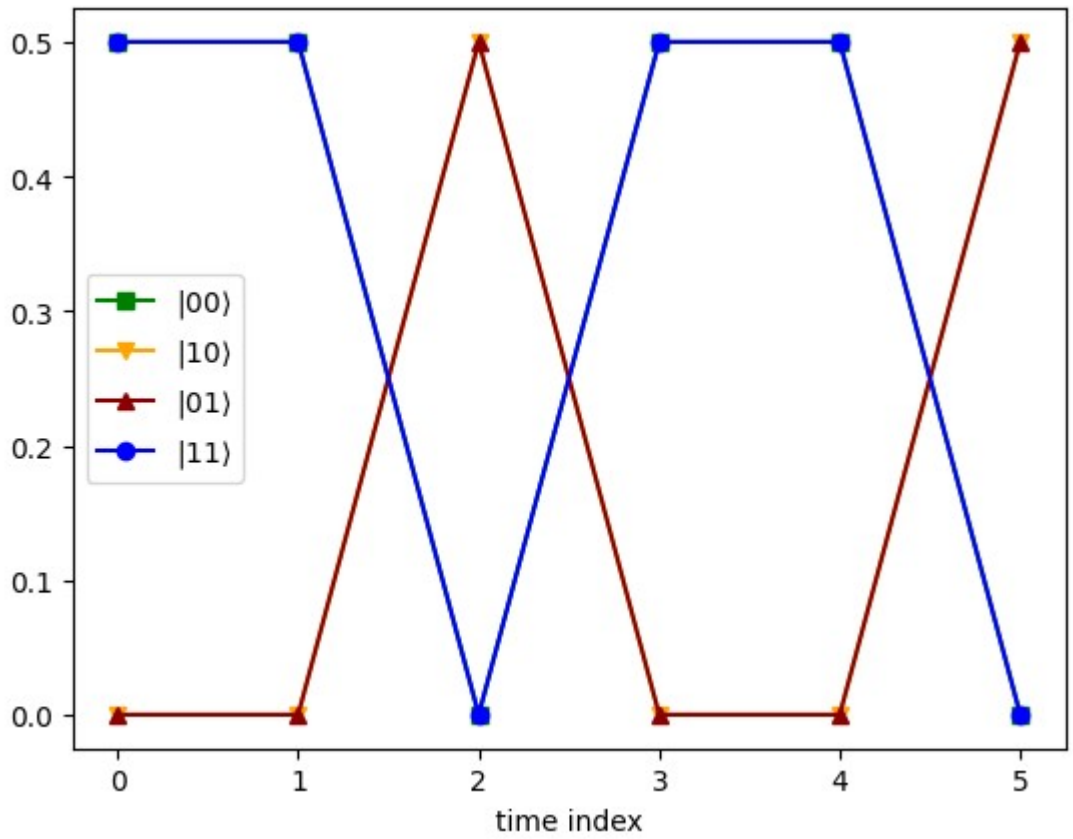
In [41]:
```
#Running it via standard subcircuit ordering

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) #absolu
    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 (matri
print("PROBABILITIES ORDERED BY QUBIT STRING REPRESENTATION")
for i in range(indices):
    print(res.by_subbatch[0].by_subcircuit[i].simulated_probability_by_str) #absolu
    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 h
# 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"$\\left\\vert{''.join(reversed(f'{n:02b}'))}\\right\\rangle$",
             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.   ]
 [0.505]]
[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': 0.4999999999999998, '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': 0.4999999999999998}
{'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': 0.5000000000000001, '10': 0.0, '01': 0.0, '11': 0.4999999999999998}
{'00': array([0.495]), '10': array([0.]), '01': array([0.]), '11': array([0.505])}
{'00': 0.0, '10': 0.4999999999999998, '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.  0.5 0.5 0. ]
[0.   0.48 0.52 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. ]
```

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



Putting it all together

In [43]:
```
#Create a jaqal code (string method)

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 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] 0 gamma
measure_all

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

prepare_all
MS q[0] q[1] 0 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],

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) #ab
        print(res.by_subbatch[i].by_subcircuit[j].relative_frequency_by_int) #with

#We can also sort the resulting probabilities by their string representation (matri
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) #ab
        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 h
# 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]
[[0.8585]
 [0.    ]
 [0.    ]
 [0.1415]]
[0.25 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': 0.8685423588862067, '10': 0.0, '01': 0.0, '11': 0.13145764111379327}
{'00': array([0.8585]), '10': array([0.]), '01': array([0.]), '11': array([0.141
5])}
{'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])}
```

```
{'00': 0.5000031633974483, '10': 0.0, '01': 0.0, '11': 0.4999968366025516}
{'00': array([0.4895]), '10': array([0.]), '01': array([0.]), '11': array([0.510
5])}
{'00': 0.24999999999999994, '10': 0.25000000000000006, '01': 0.24999999999999994, '
11': 0.25000000000000006}
{'00': array([0.258]), '10': array([0.2485]), '01': array([0.2595]), '11': array
([0.234])}
{'00': 0.24999999999999994, '10': 0.25000000000000006, '01': 0.24999999999999994, '
11': 0.25000000000000006}
{'00': array([0.259]), '10': array([0.2595]), '01': array([0.234]), '11': array([0.
2475])}
{'00': 0.9927839218733792, '10': 0.0, '01': 0.0, '11': 0.007216078126620813}
{'00': array([0.993]), '10': array([0.]), '01': array([0.]), '11': array([0.007])}
{'00': 0.2500000000000001, '10': 0.25, '01': 0.25, '11': 0.2499999999999999}
{'00': array([0.2485]), '10': array([0.2545]), '01': array([0.236]), '11': array
([0.261])}
{'00': 0.2500000000000001, '10': 0.25, '01': 0.25, '11': 0.2499999999999999}
{'00': array([0.251]), '10': array([0.2305]), '01': array([0.265]), '11': array([0.
2535])}
RESULTS ORDERED BY TIME
[0.86854236 0.         0.         0.13145764]
[0.8585 0.     0.     0.1415]
[0.25 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]
```

In [ ]: