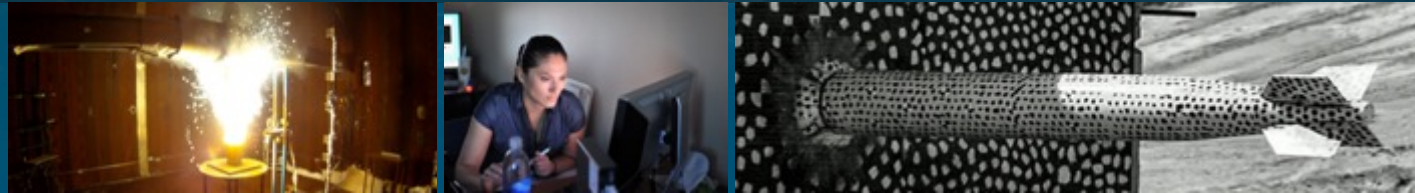


# QSCOUT Webinar: JaqalPaw Overview



*Presented by*

Daniel Lobser

September 15, 2021

# Experimental Details of Gate Implementation

**Basic JaqalPaw:** Simple Waveforms

**Advanced JaqalPaw:** Experimentally meaningful waveforms

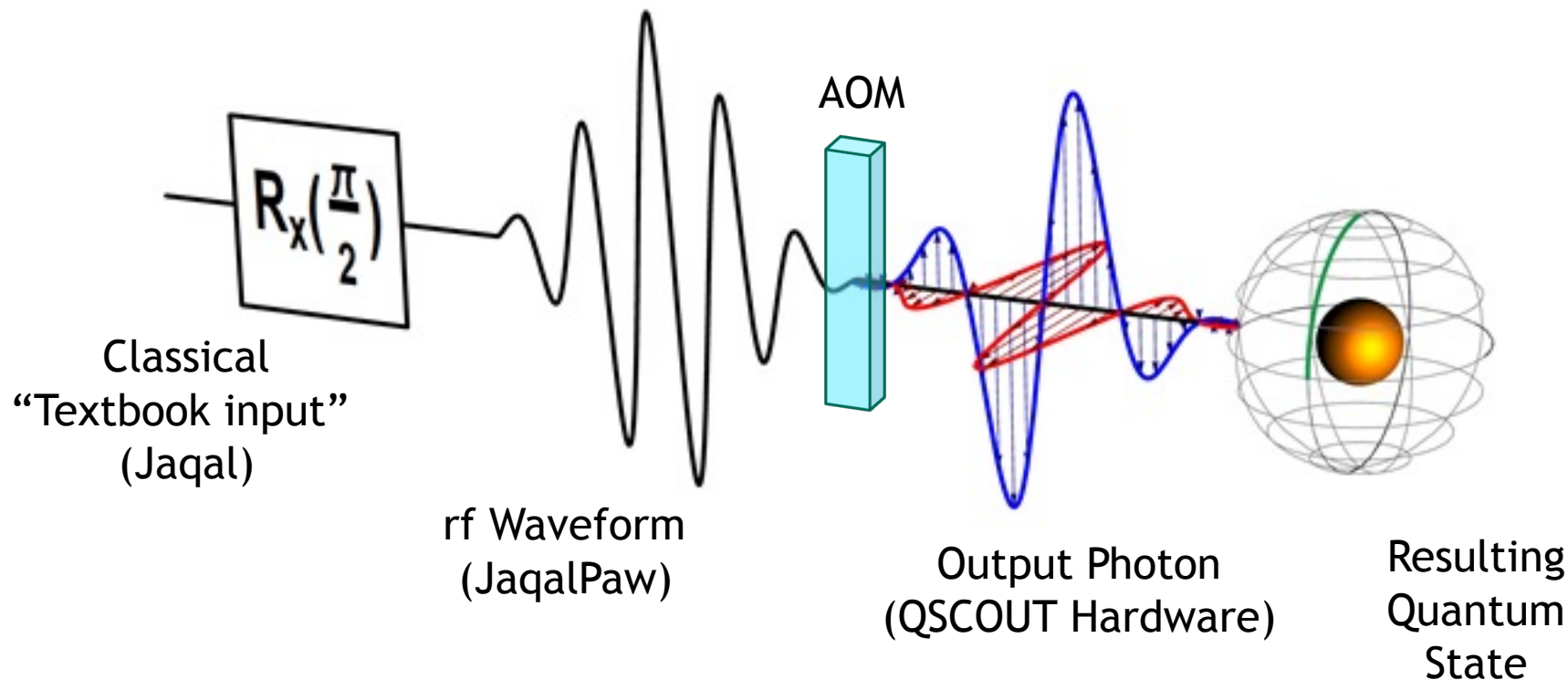
### 3 Realizing Quantum Gates

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

The internal quantum states of individually-addressed ions are manipulated via laser light passed through 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

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

Individual addressing requires lasers

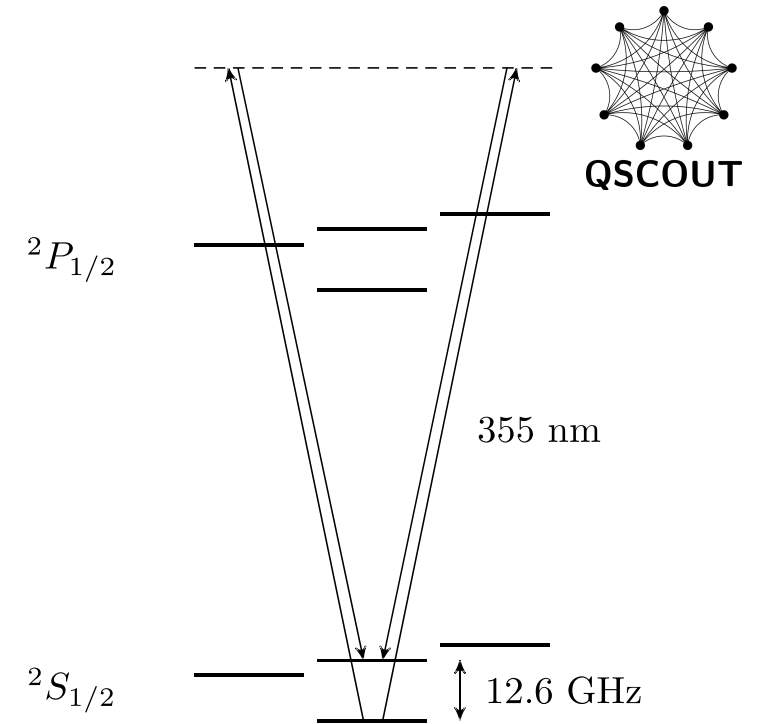
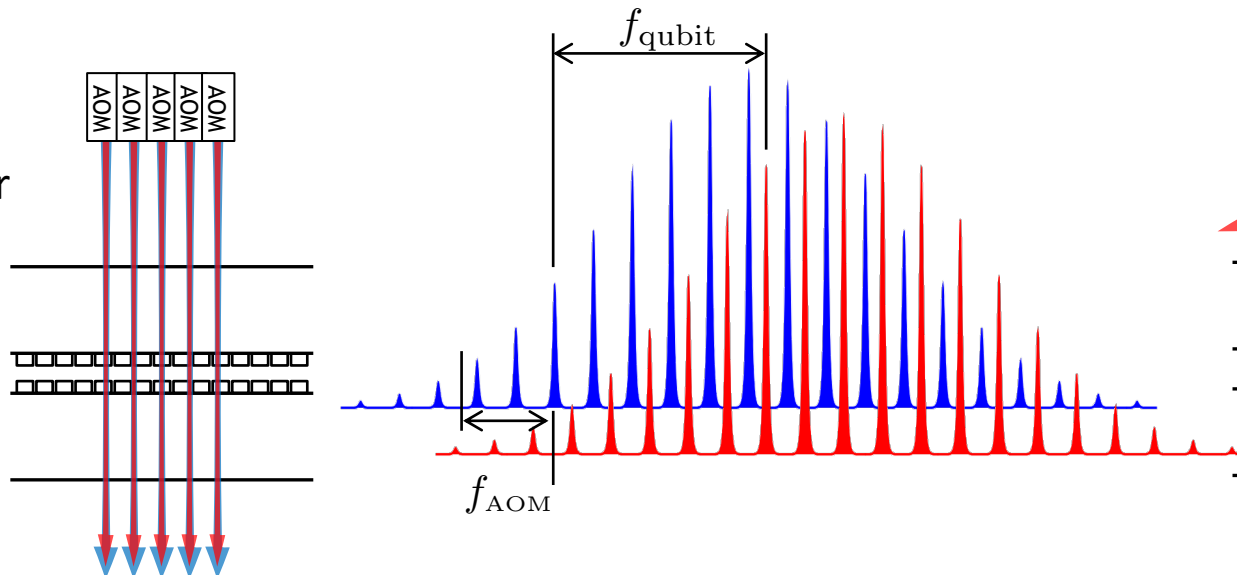
Optical frequency comb to bridges 12.6 GHz via Raman transitions

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

Two configurations: Co- and Counter-propagating

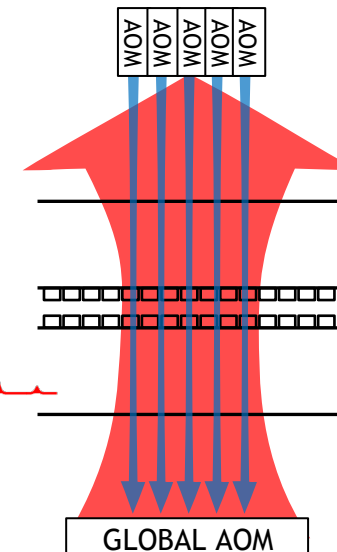
## Co-propagating

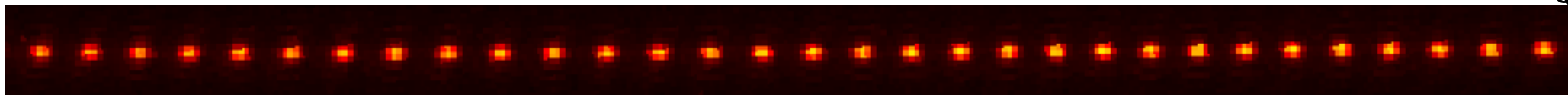
- Immune to Doppler shifts
- Not affected by timing errors and pulse overlap



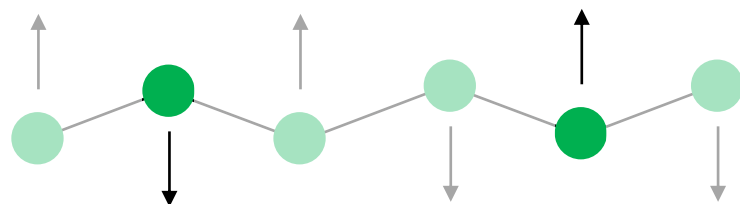
## Counter-propagating

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

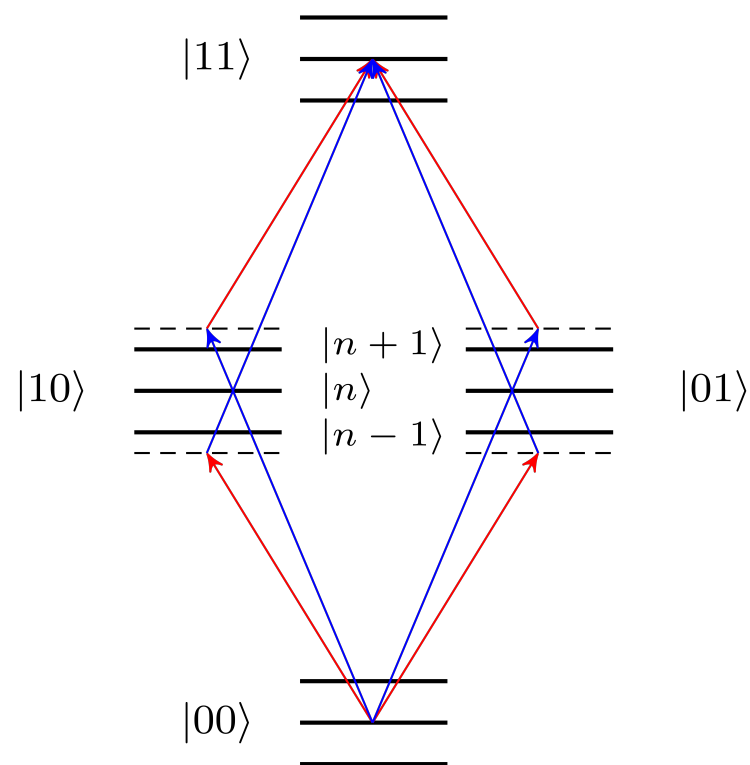
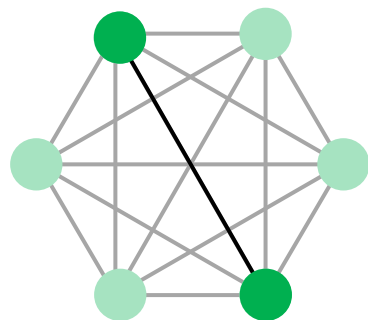


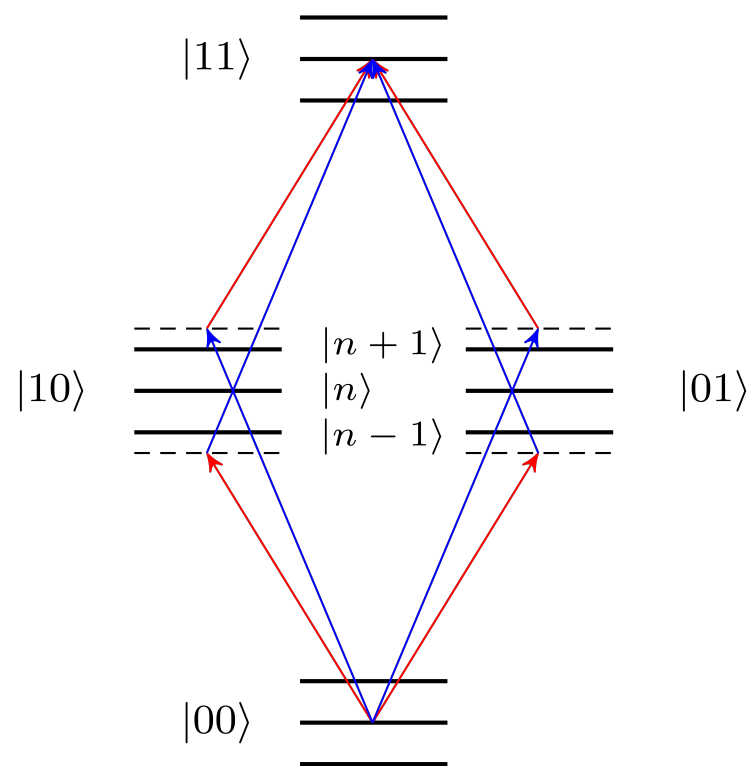
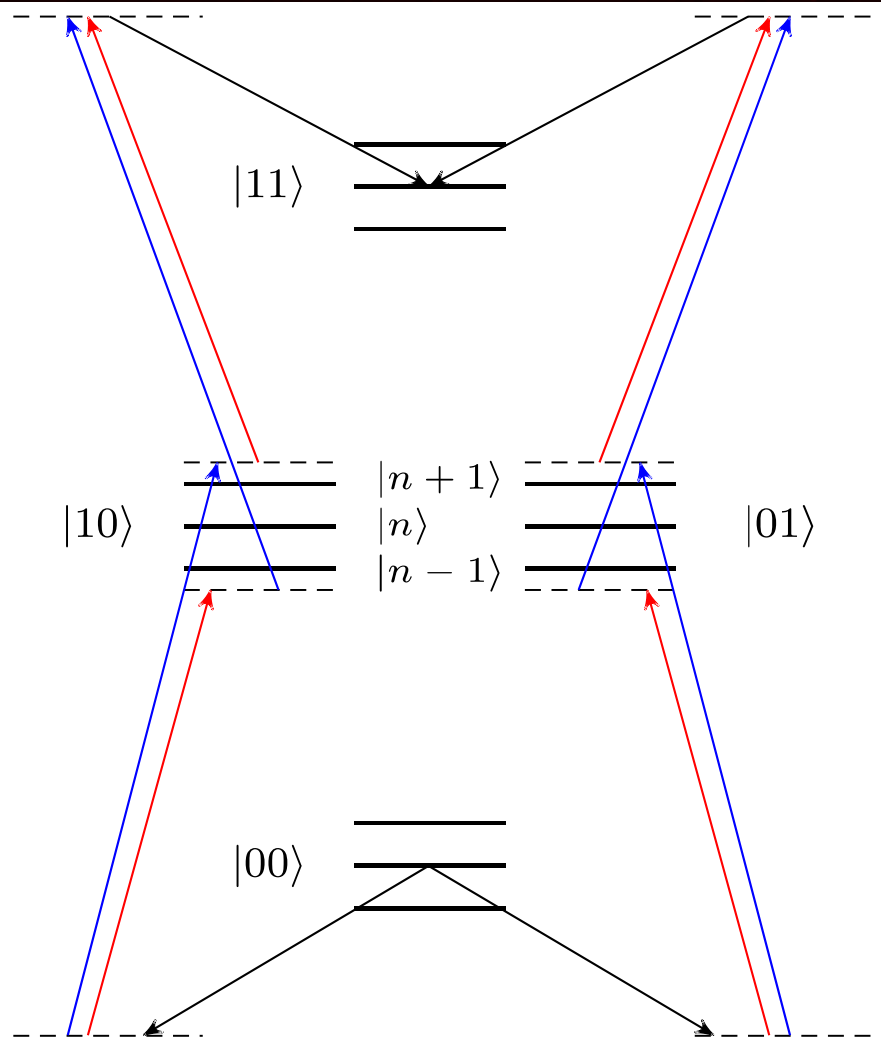
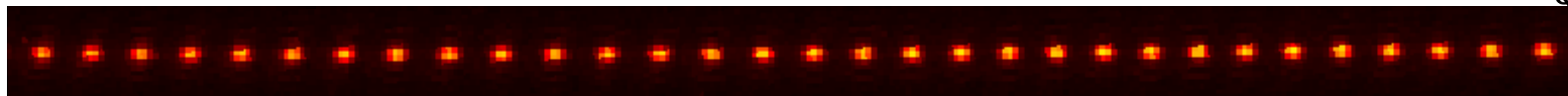


Common motional modes



“Fully connected”





## Experimental Details of Gate Implementation


### **Basic JaqalPaw:** Simple Waveforms

### Advanced JaqalPaw: Experimentally meaningful waveforms

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

<sup>1</sup>Sandia National Laboratories, Albuquerque, NM 87123 USA

<sup>2</sup>Department of Physics and Astronomy, University of New Mexico, Albuquerque, NM 87131 USA

<sup>3</sup>Center for Quantum Information and Control, University of New Mexico, Albuquerque, NM 87131 USA

<sup>4</sup>IonQ, Inc., College Park, MD 20740 USA

## JaqalPaw: A Guide to Defining Pulses and Waveforms for Jaqal

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

<sup>1</sup>Sandia National Laboratories, Albuquerque, New Mexico 87123, USA

<sup>2</sup>Currently at IonQ, College Park, Maryland 20740, USA

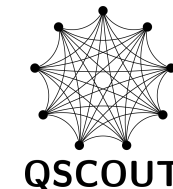
(Dated: April 19, 2021)

<https://qscout.sandia.gov>

QSCOUT Info:

- ▣ [Jaqal Language Specs](#)
- ▣ [Download JaqalPaw](#)
- ▣ [JaqalPaw \(Pulses and Waveforms\)](#)
- ▣ Latest publication: [Engineering the Quantum Scientific Computing Open User Testbed](#)

## 9 JaqalPaw in Broad Strokes



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

Gate definitions are  
defined in a class

They can derive from  
other gate definition  
classes

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

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: Frequency = 200e6 # Hz
```

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: Frequency = 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)))
```

Arbitrary  
helper functions  
allowed

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: Frequency = 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 accessible from Jaqal

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: Frequency = 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(qubit, 10e-6,  
                           amp0=self.gauss(50, sigma, 10),  
                           amp1=self.gauss(50, sigma, 10),  
                           freq0=self.some_calibrated_parameter,  
                           freq1=self.adjusted_carrier_splitting,  
                           fb_enable_mask=0b001)]
```

Arguments after "self" are accessible from Jaqal

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

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 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 .

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 (when nothing else is specified, this is just a NOP)

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

# output channel
# total duration to apply parameters (s)
# tone 0 frequency (Hz)
# tone 0 phase (deg.)
# tone 0 amplitude (arb.)
# tone 1 frequency (Hz)
# tone 1 phase (deg.)
# tone 1 amplitude (arb.)
# frame 0 virtual rotation (deg.)
# frame 1 virtual rotation (deg.)
# synchronize phase for current frequency
# toggle the output enable state
# enable frequency correction
# apply frame rotation at end of pulse
# reset accumulated frame rotation
# forward frame 0
# forward frame 1
# invert frame 0 sign
# invert frame 1 sign
# wait for external trigger
```

FIG. 1: Full argument signature of PulseData .

PulseData objects are the primary building blocks for constructing gates

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

Always requires channel and duration (when nothing else is specified, this is just a NOP)

Frequency, phase, amplitude and frame rotations 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 .

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 and frame rotations can be constant-valued, have multiple discrete updates (lists), or continuous spline modulation (tuples)

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 .

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 and frame rotations can be constant-valued, have multiple discrete updates (lists), or continuous spline modulation (tuples)

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)

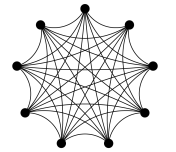
Note that N-1 segments are used in a spline, while N segments are used for discrete updates

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

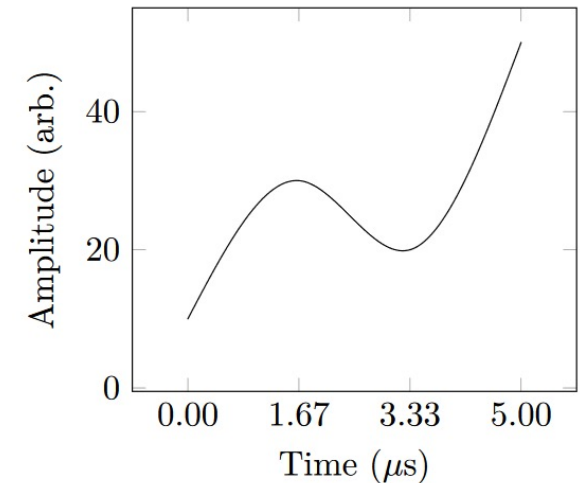
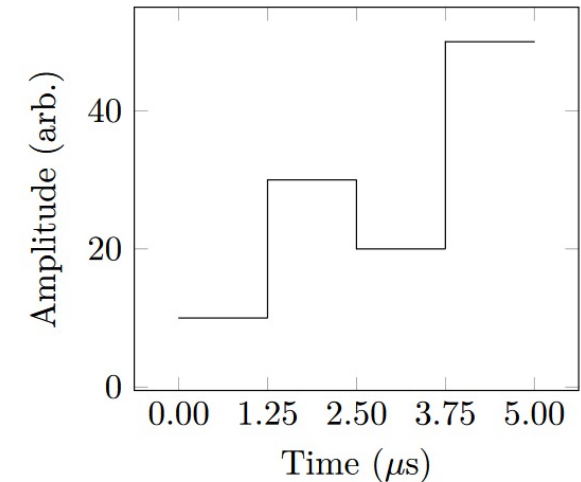
EX. 5: Discrete updates are represented as a list of inputs and are equally distributed across the duration of the pulse.

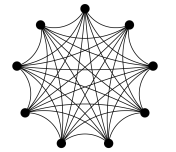
```
def gate_G(self, qubit):
    return [PulseData(qubit,
                      5e-6,
                      freq0=200e6,
                      amp0=(10,30,20,50),
                      phase0=0)]
```

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



QSCOUT

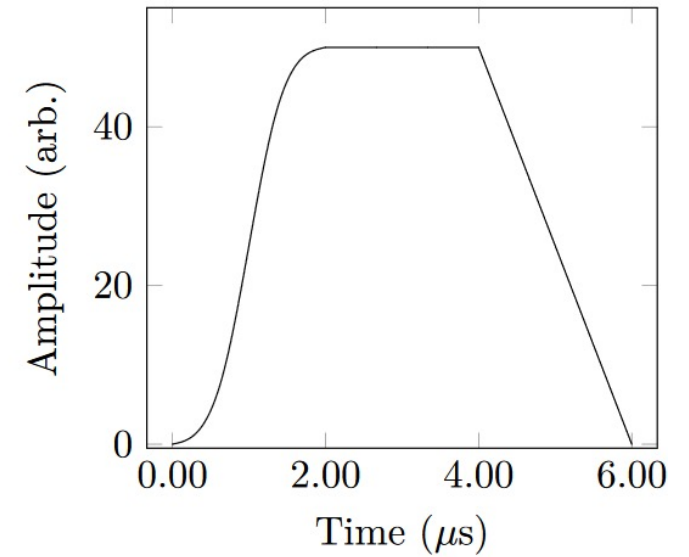




PulseData objects are run back to back when on the same channel

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.

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

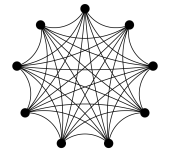
This also applies to gates in general

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

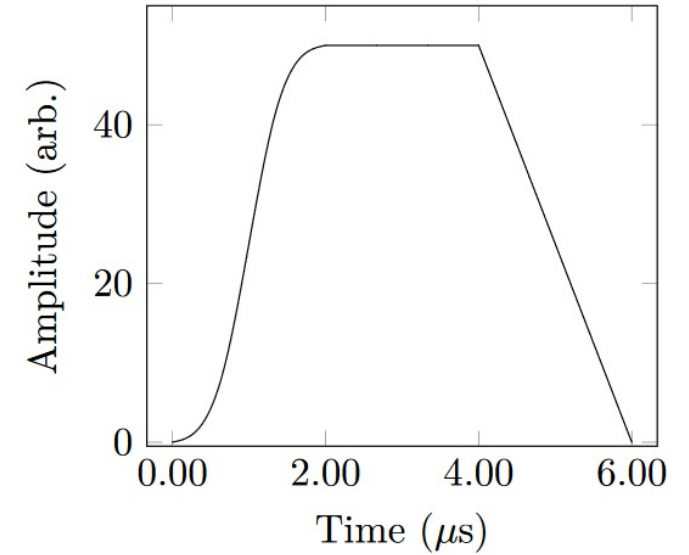
- Different modulation types are nested in a list
- Each list entry is subdivided in time

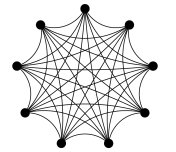
```
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
                      amp0=(0,9,41,50)),
            PulseData(qubit, 2e-6,
                      amp0=50),
            PulseData(qubit, 2e-6,
                      amp0=(50,0))]
```

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



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

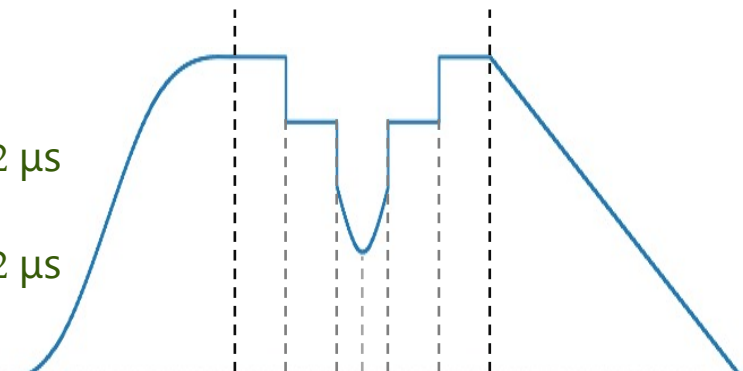
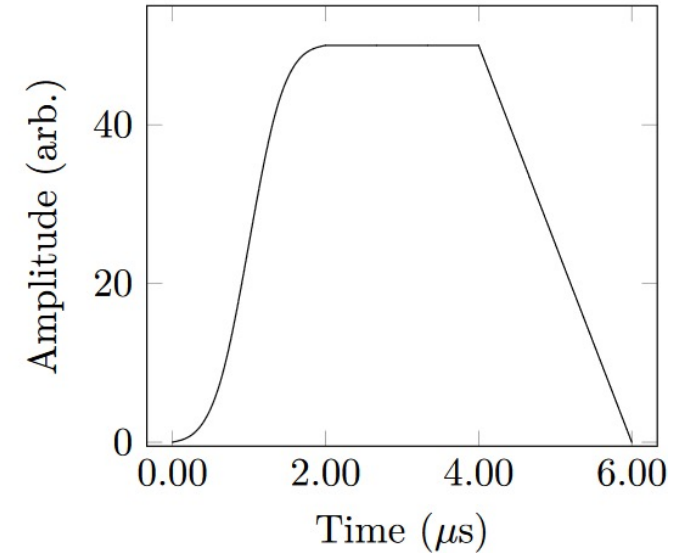
Lists can contain scalar values, lists, or tuples

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
```



PulseData objects on different channels are run in parallel

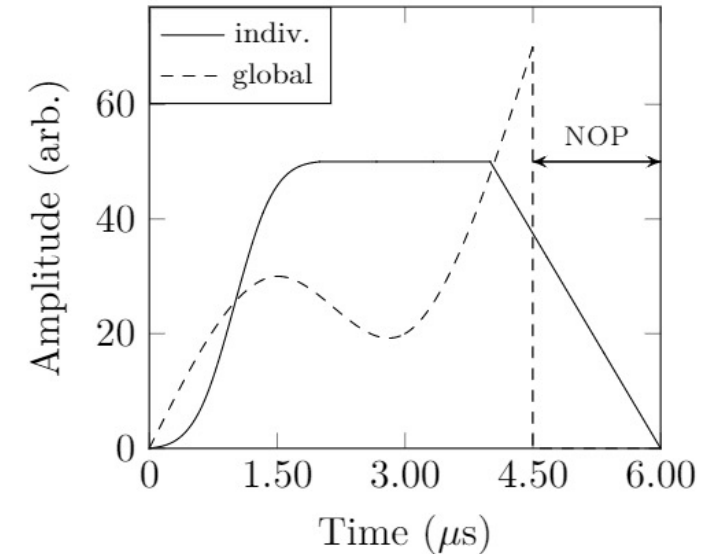
This always applies to PulseData objects in the same gate

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))]
```



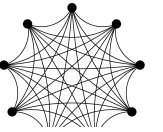
EX. 9: Chaining **PulseData** objects on different channels results in parallel execution. Differences in cumulative duration will be padded with a NOP pulse.

## Experimental Details of Gate Implementation

Basic JaqalPaw: Simple Waveforms

**Advanced JaqalPaw:** Experimentally meaningful waveforms

## Frame Rotations



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

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

$$S_x S_z S_x \rightarrow S_y S_x$$

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

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

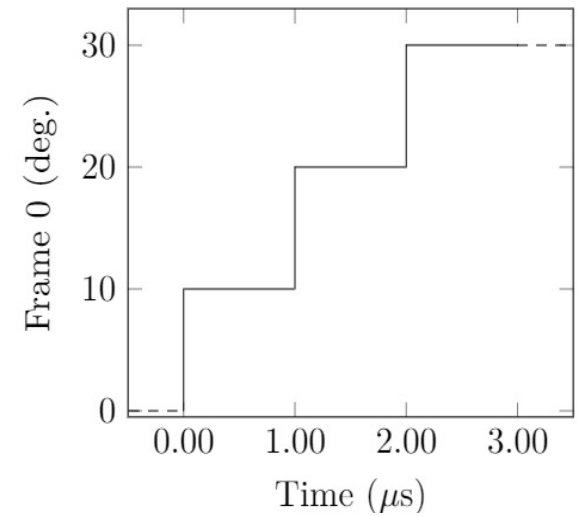
Frame rotations are cumulative and apply to subsequent gates until the frame is explicitly reset

Frame rotations take scalar, discrete, and spline inputs

Spline inputs accumulate only the final value

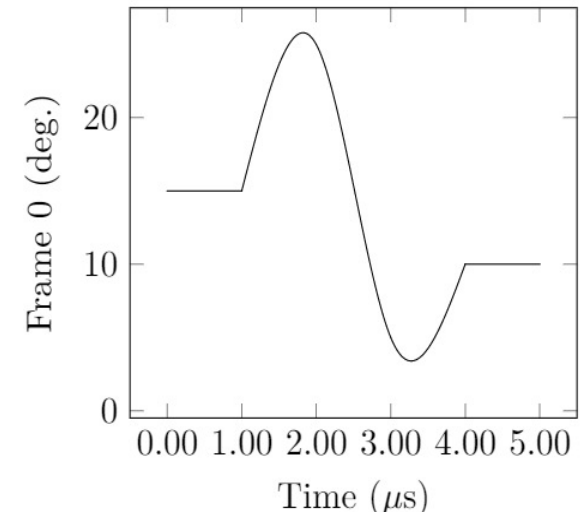
```
def gate_G(self, qubit):
    return [PulseData(qubit, 1e-6,
                      framerot0=10)
            for _ in range(3)]

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



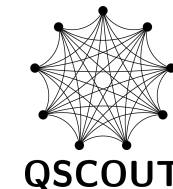
EX. 10: Frame rotation inputs are equivalent to phase, but their values accumulate.

```
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



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

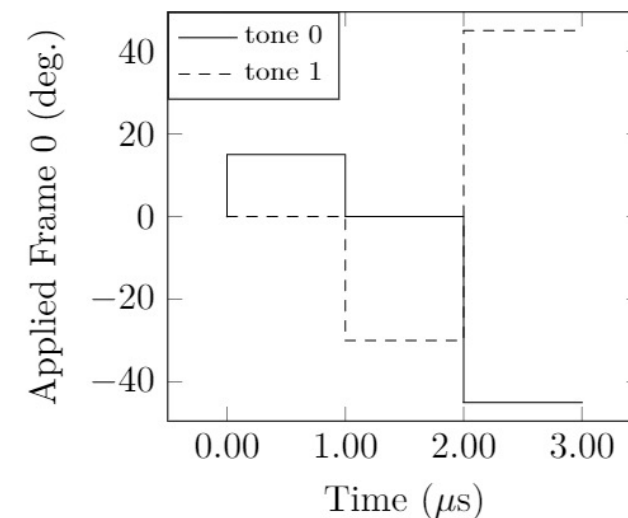
For a two-qubit Mølmer-Sørensen gate, both red and blue sideband inputs must have the frame forwarded

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

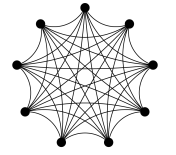
Frame forwarding and inversion is controlled via metadata, which uses a bitmask convention

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

## Comb Instability → Beat Note Lock

Frequency comb is not actively stabilized at the source!

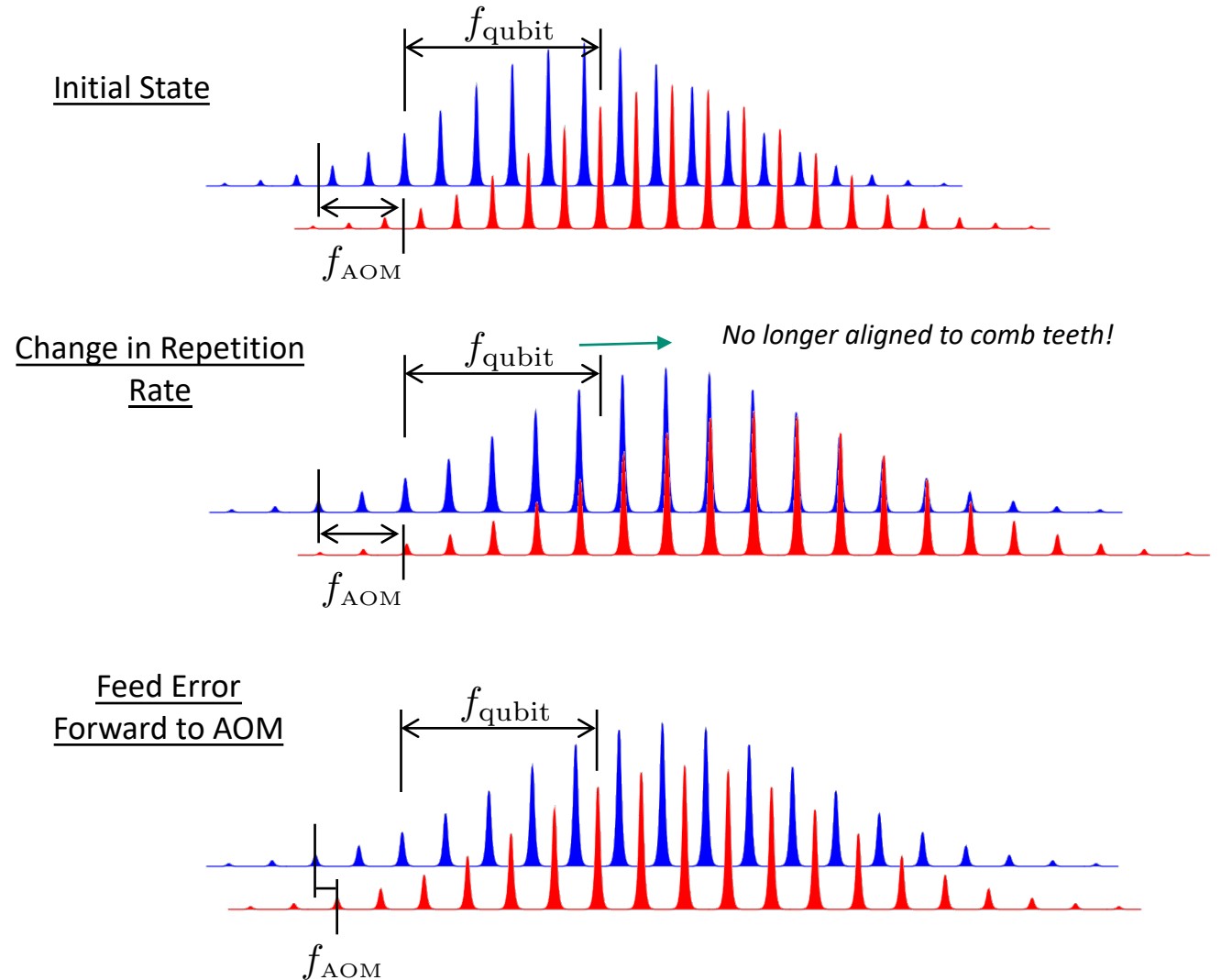
Small variations in the comb spacing require dynamic corrections to stay on resonance

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

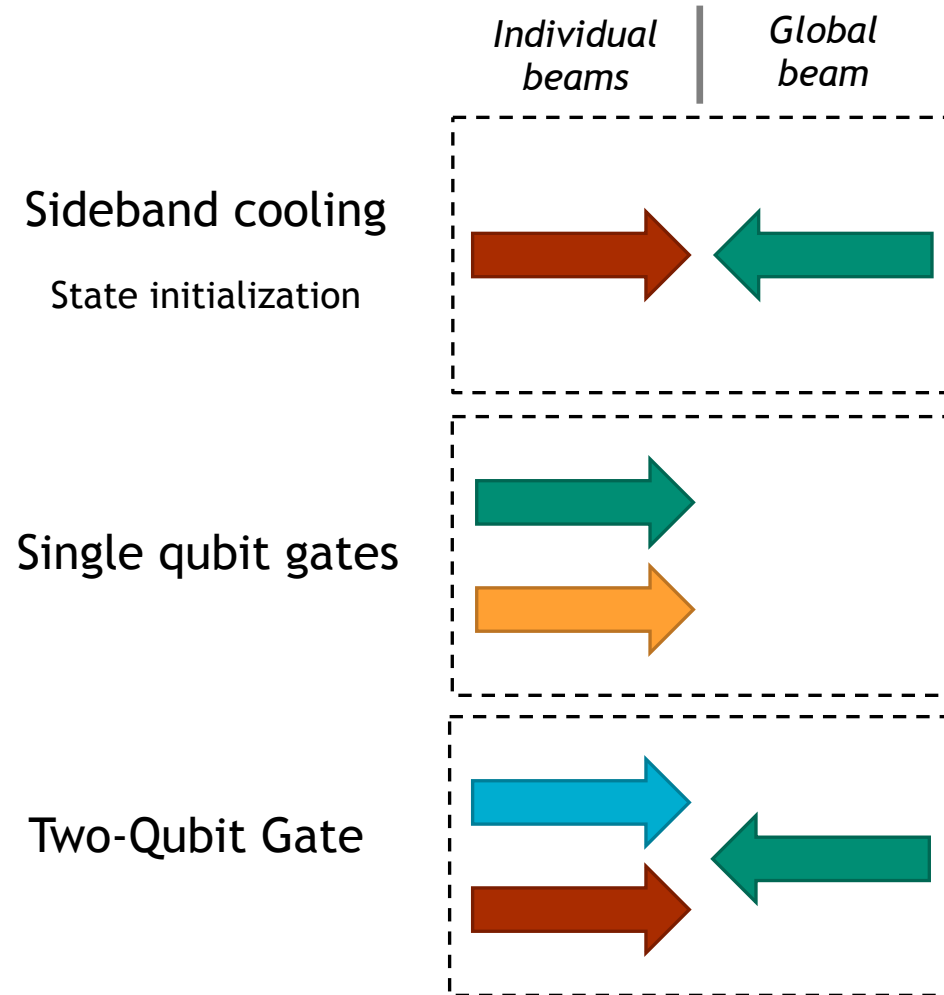
Lock should be applied to the **higher** frequency tone

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



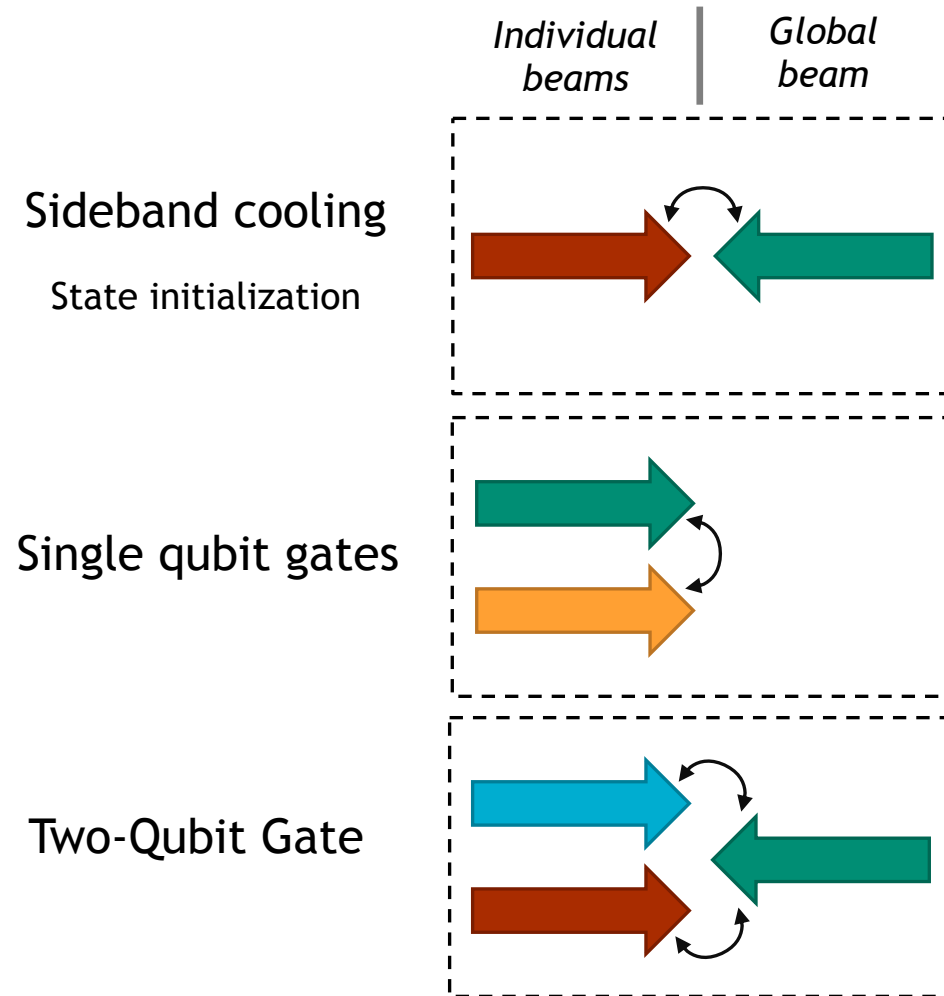
# Challenges: RF Reproducibility and Agility

## Three basic configurations



# Challenges: RF Reproducibility and Agility

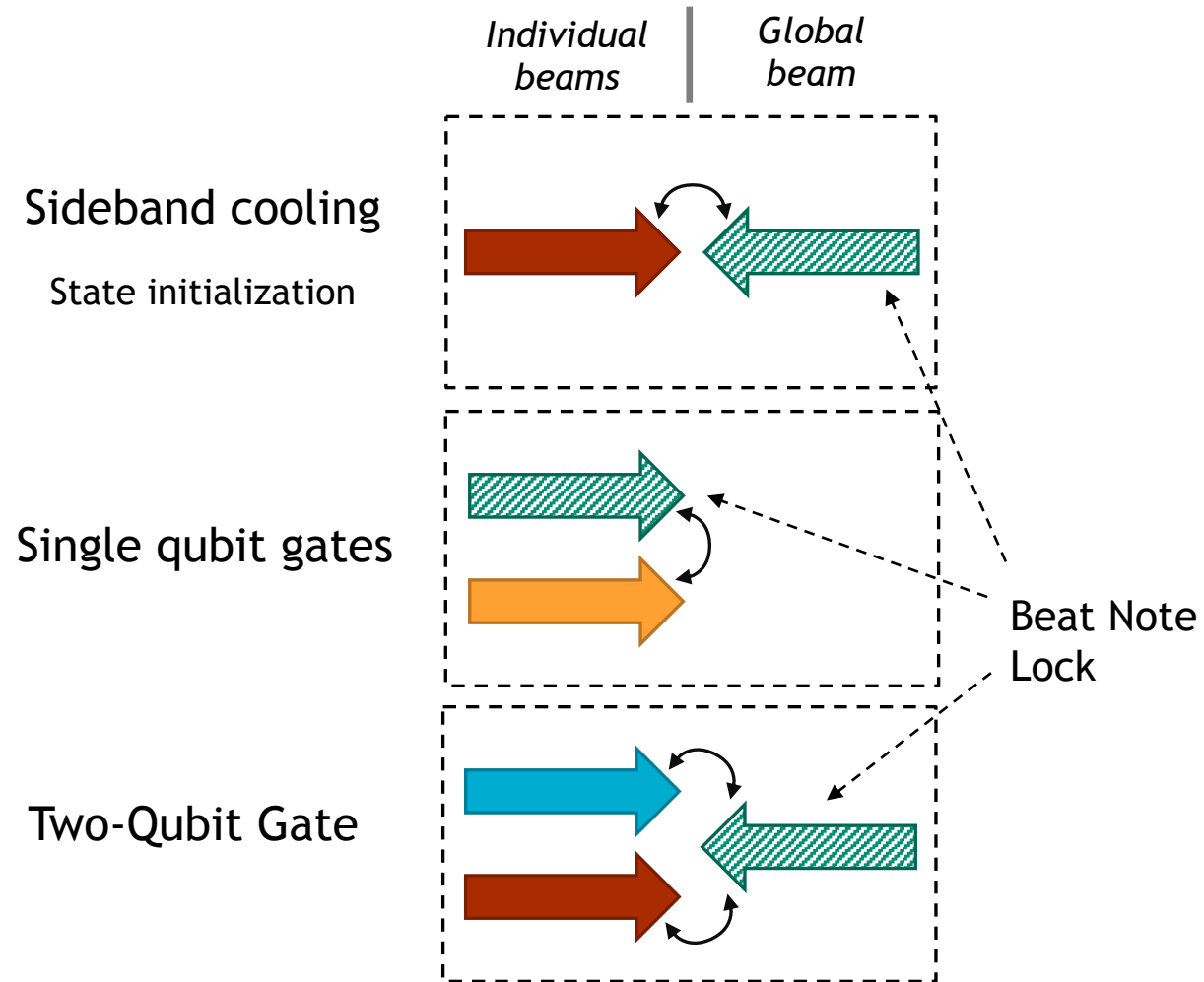
## Three basic configurations



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

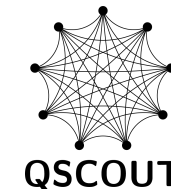
# Challenges: RF Reproducibility and Agility

## Three basic configurations

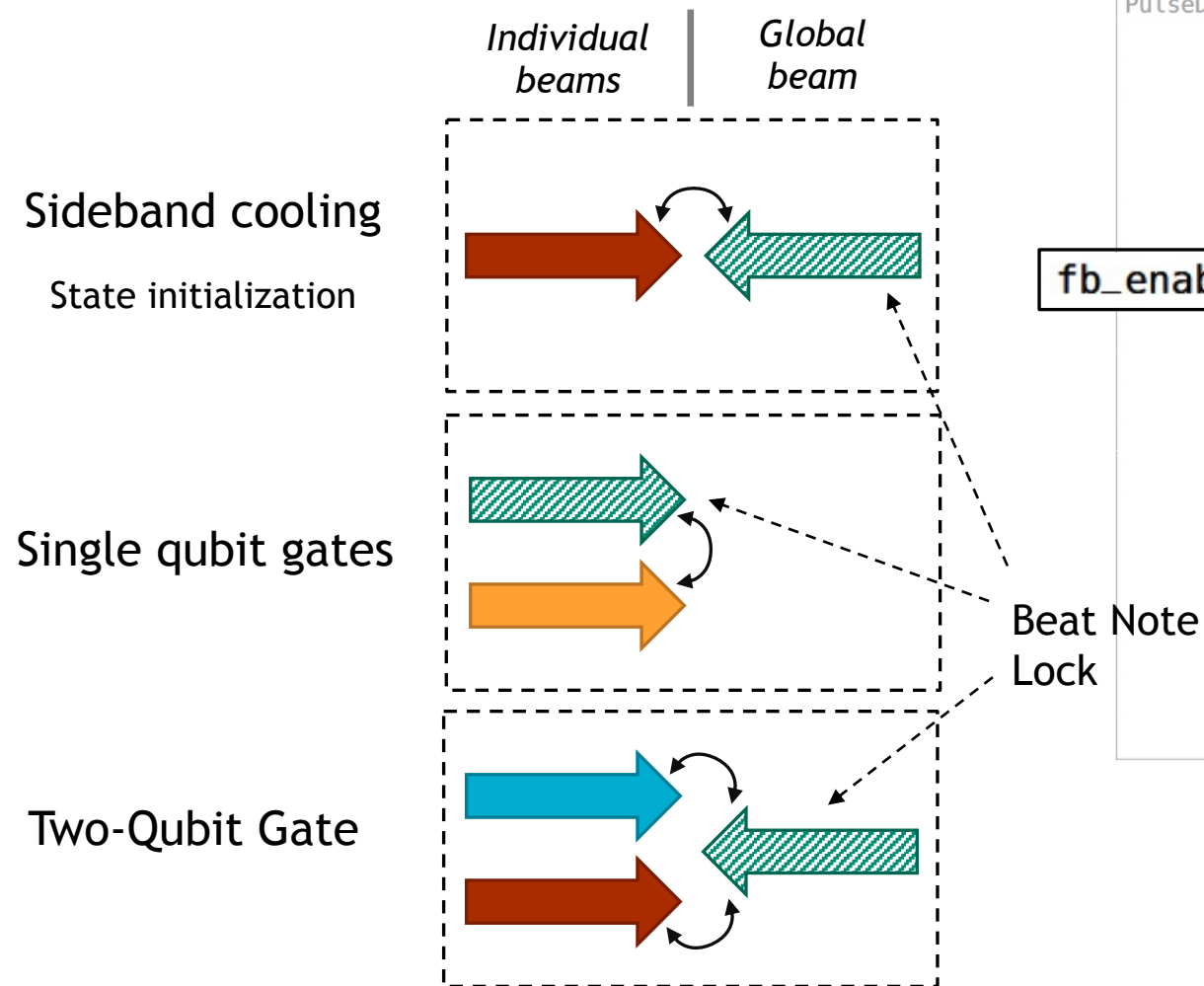


*Lock must be applied to exactly one tone for each Raman pair!*

# Challenges: RF Reproducibility and Agility



## Three basic configurations

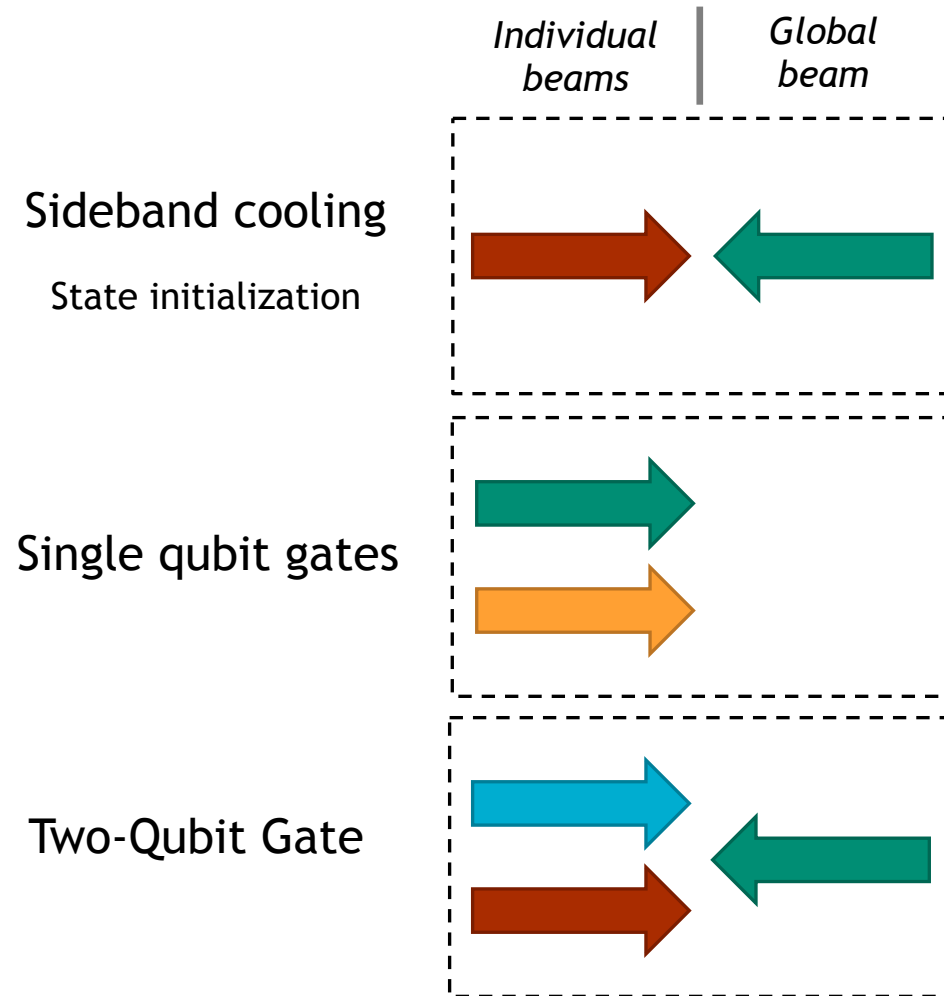


```
PulseData(channel,           # output channel
           dur,              # total duration to apply parameters (s)
           freq0=0,          # tone 0 frequency (Hz)
           phase0=0,         # tone 0 phase (deg.)
           amp0=0,           # tone 0 amplitude (arb.)
           freq1=0,  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

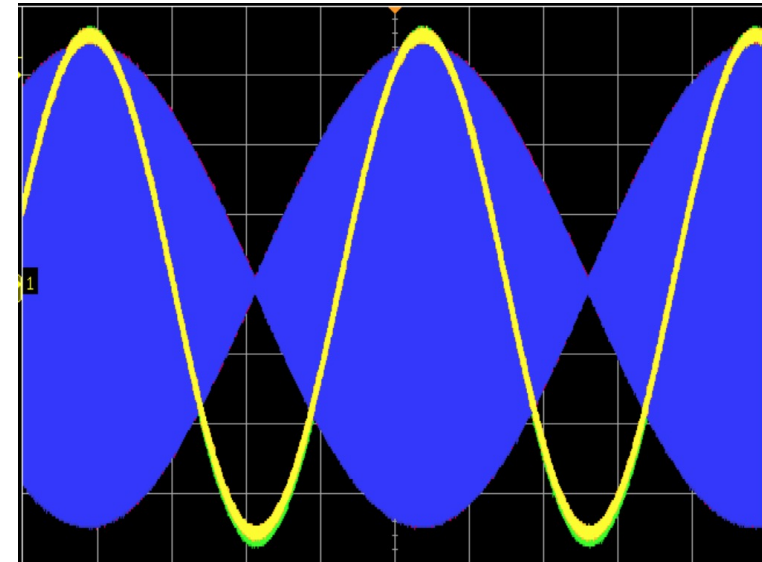
*Lock must be applied to exactly one tone for each Raman pair!*

## 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.

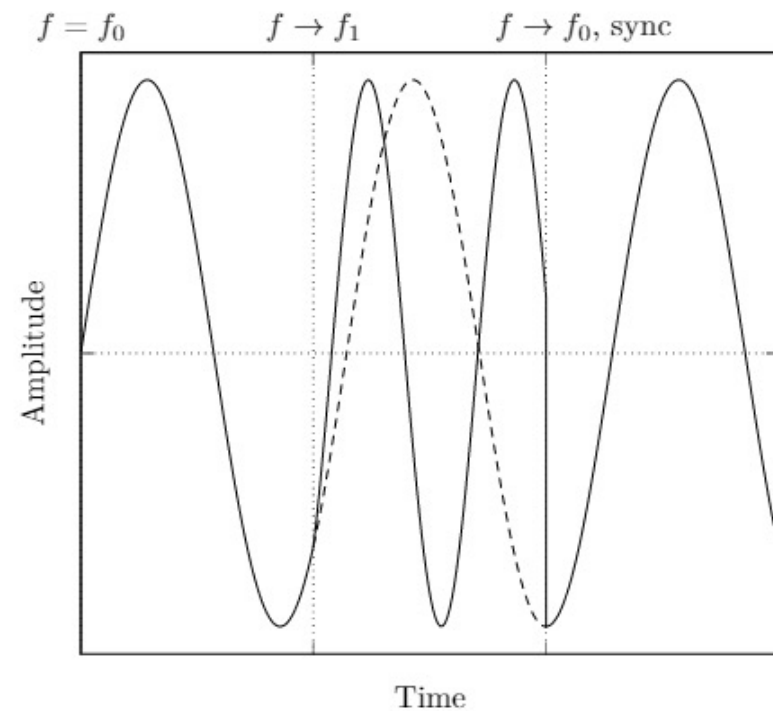
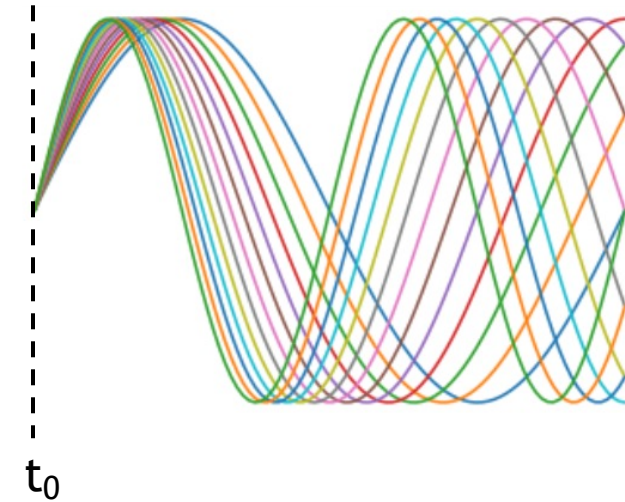
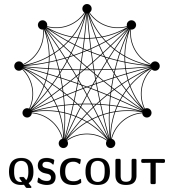
For absolute phase control, one must apply a synchronization trigger to a pulse by setting a non-zero value in the `sync_mask` argument of a `PulseData` object

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

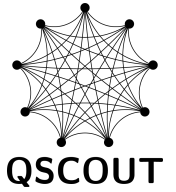
Synchronization pulses must be applied for **all** pulses where phase must be aligned to each other

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

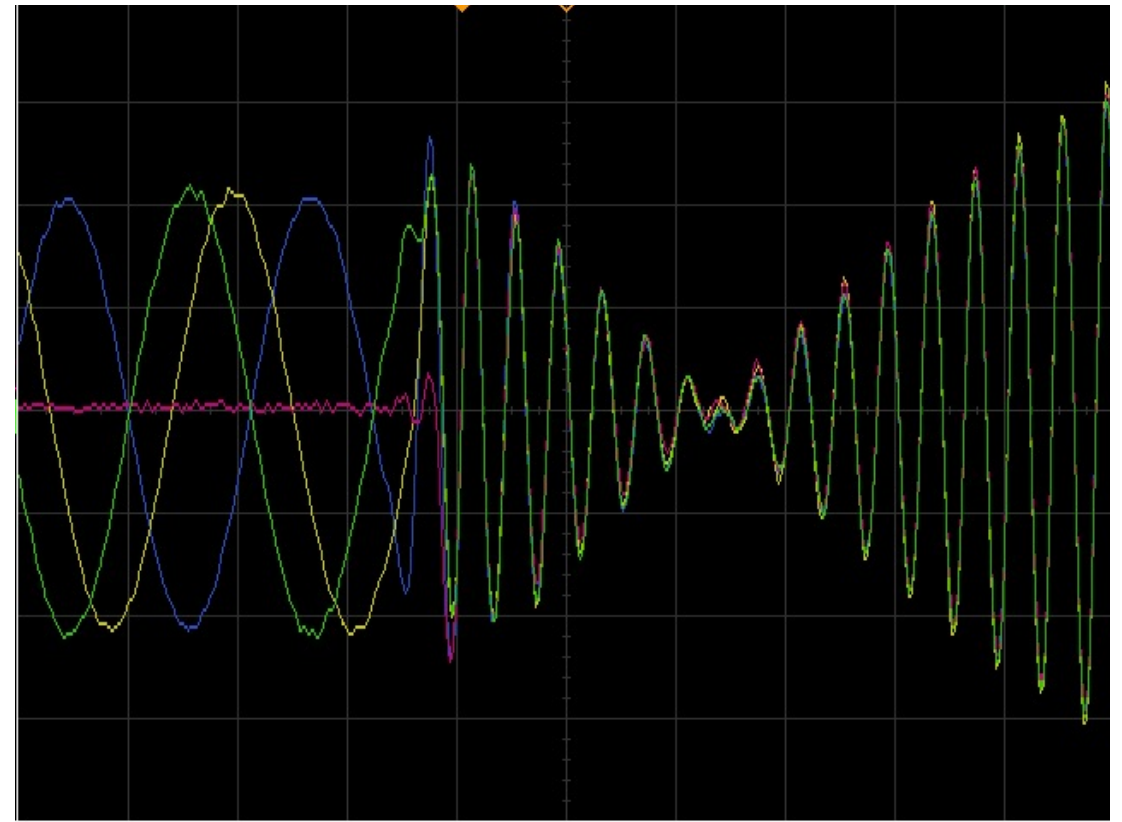
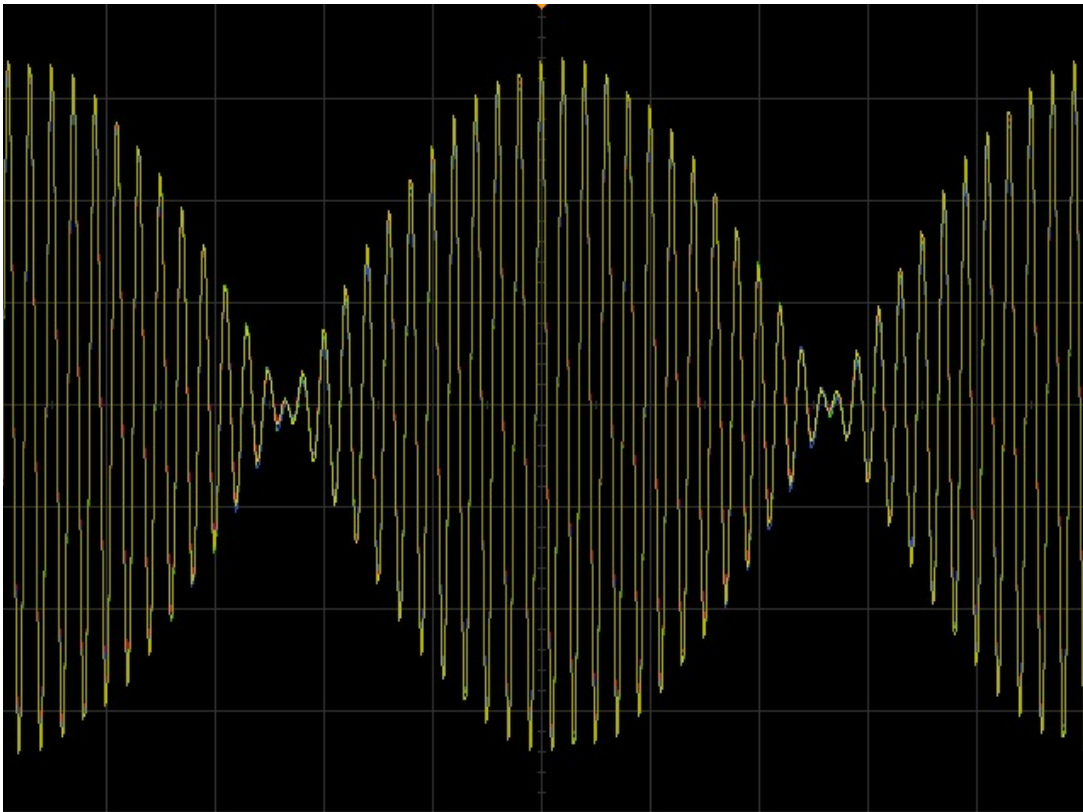


## Hardware Features: Global Phase Synchronization



Tones being manually synchronized to an earlier state

Changing parameters while applying synchronization



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

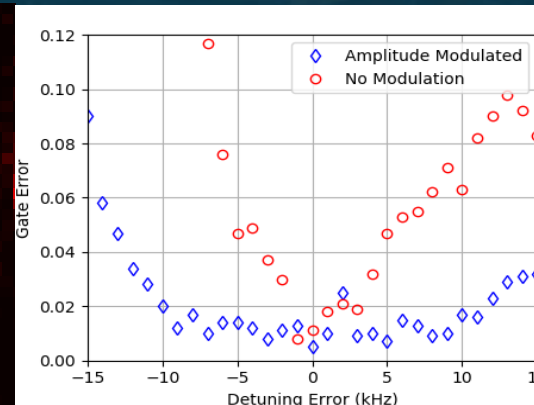
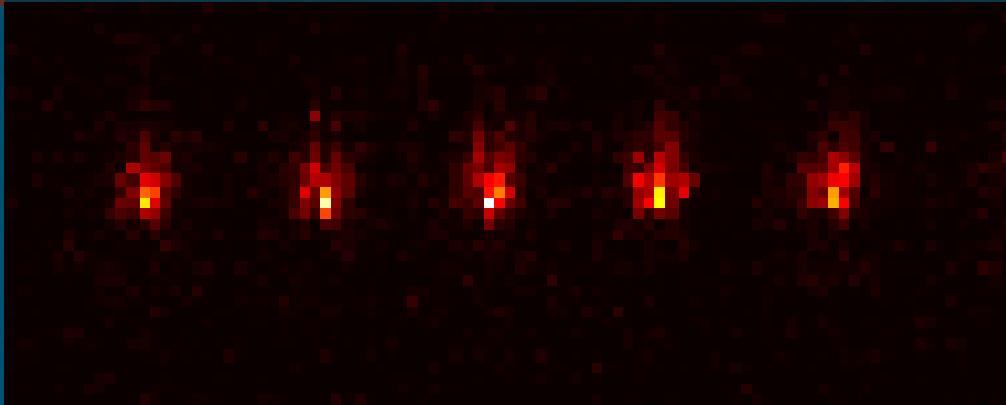
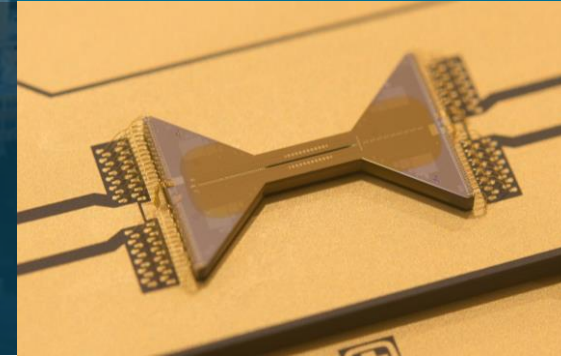
$$f_r + f_b = 2f_{qubit} \quad \text{where} \quad \begin{aligned} f_r &\equiv f_{qubit} - f_{SB} \\ f_b &\equiv f_{qubit} + f_{SB} \end{aligned} \quad \text{but} \quad F_r + F_b \neq 2F_{qubit}$$

Best bet is to use `discretize_frequency` helper functions

```
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
```

# Questions?

# QSCOUT Webinar: JaqlPaw Exemplar Program



*Presented by*

Matthew Chow

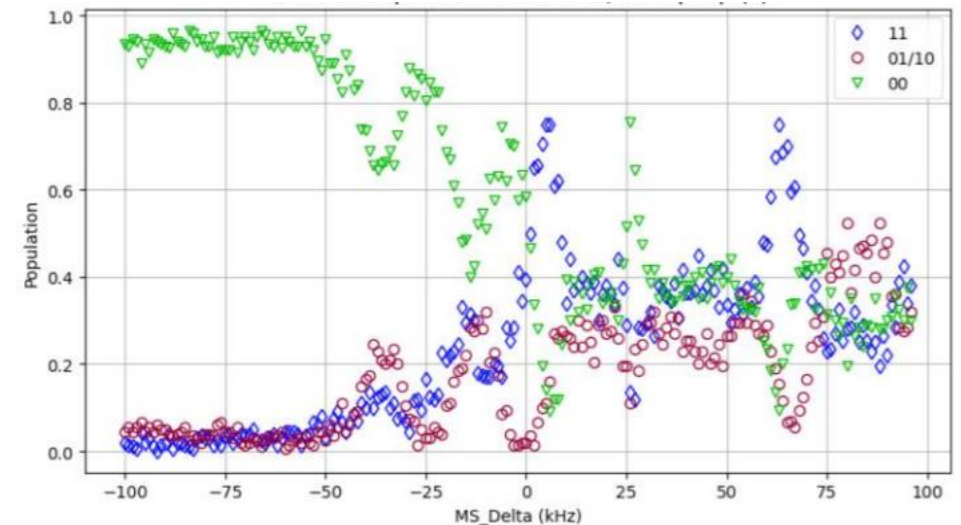
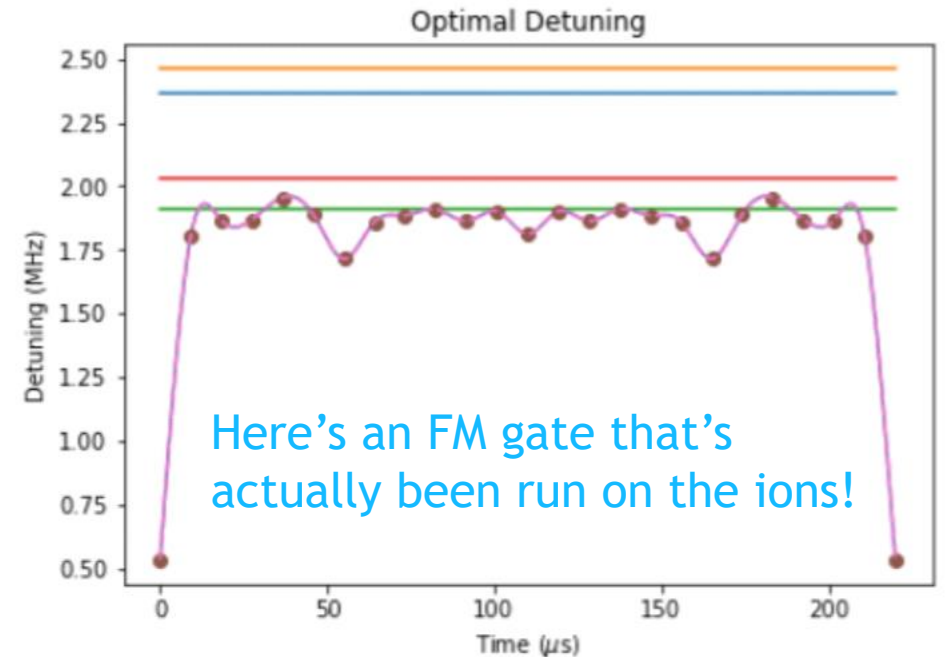
September 15, 2021

# Modulated Gates – JaqalPaw 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, synchronization.
3. Handy features in exemplar
  - a) Parameterized pulses
  - b) Making use of both Raman beams
  - c) Programmatic configuration





Sandia National Laboratories: QS x +

qscout.sandia.gov

Sandia National Laboratories

Quantum Projects Research Highlights Publications

Projects

## Quantum Scientific Computing Open User Testbed (QSCOUT)

A quantum computing testbed based on trapped ions that is available to the research community as an open platform for a range of quantum computing applications.

QSCOUT is an [R&D100 Finalist!](#)

2021 R&D 100 FINALIST

R&D 100 Entry: Quantum Scientific Co... Watch later

QSCOUT Info:

- [Jaqal Language Specs](#)
- [Download JaqalPaw](#)
- [JaqalPaw \(Pulses and Waveforms\)](#)
- [JaqalPaw Example Code](#)
- Latest publication: [Engineering the Quantum Scientific Computing Open User Testbed](#)

Helpful guide for getting started

Complete example

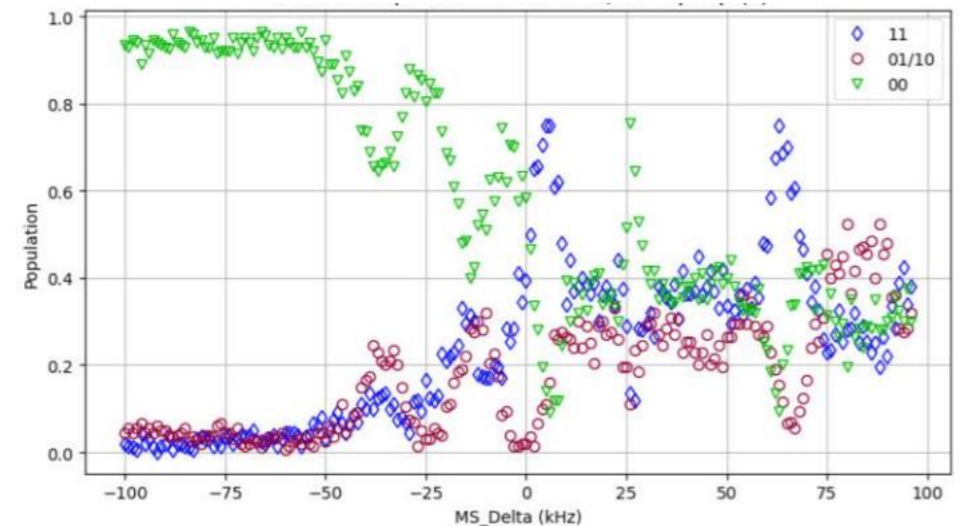
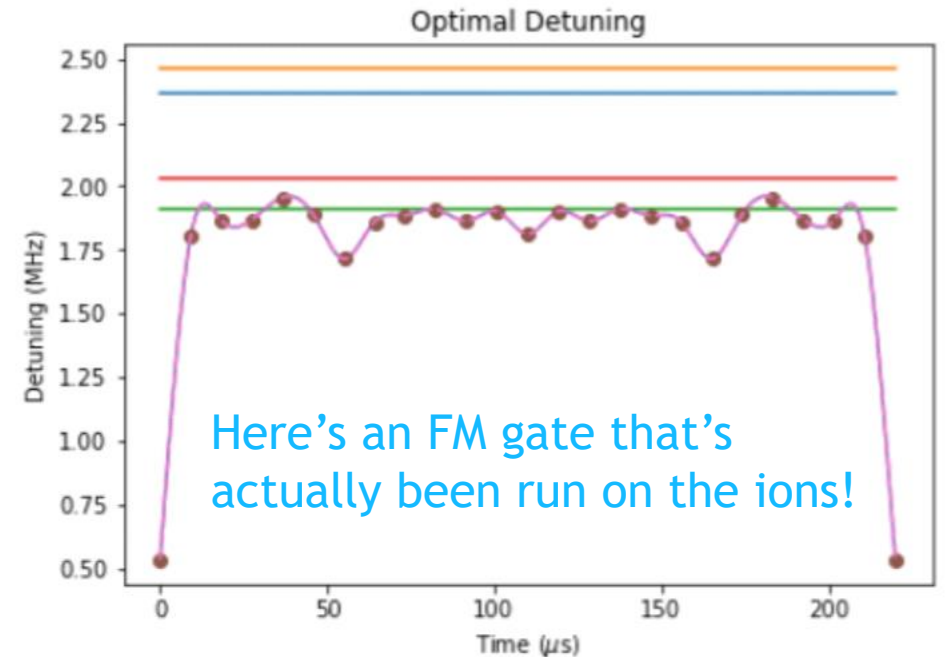
IEEE QSCOUT Manual

# Modulated Gates – JaqalPaw Tutorial



## Topics:

1. Review of basic modulation.
  - a) Calling code from Jaqal
  - b) Referencing calibration parameters
  - c) Frequency discretization, synchronization.
2. Technical details of writing pulse definitions.
  - a) Calling code from Jaqal
  - b) Referencing calibration parameters
  - c) Frequency discretization, synchronization.
3. Handy features in exemplar
  - a) Parameterized pulses
  - b) Making use of both Raman beams
  - c) Programmatic configuration



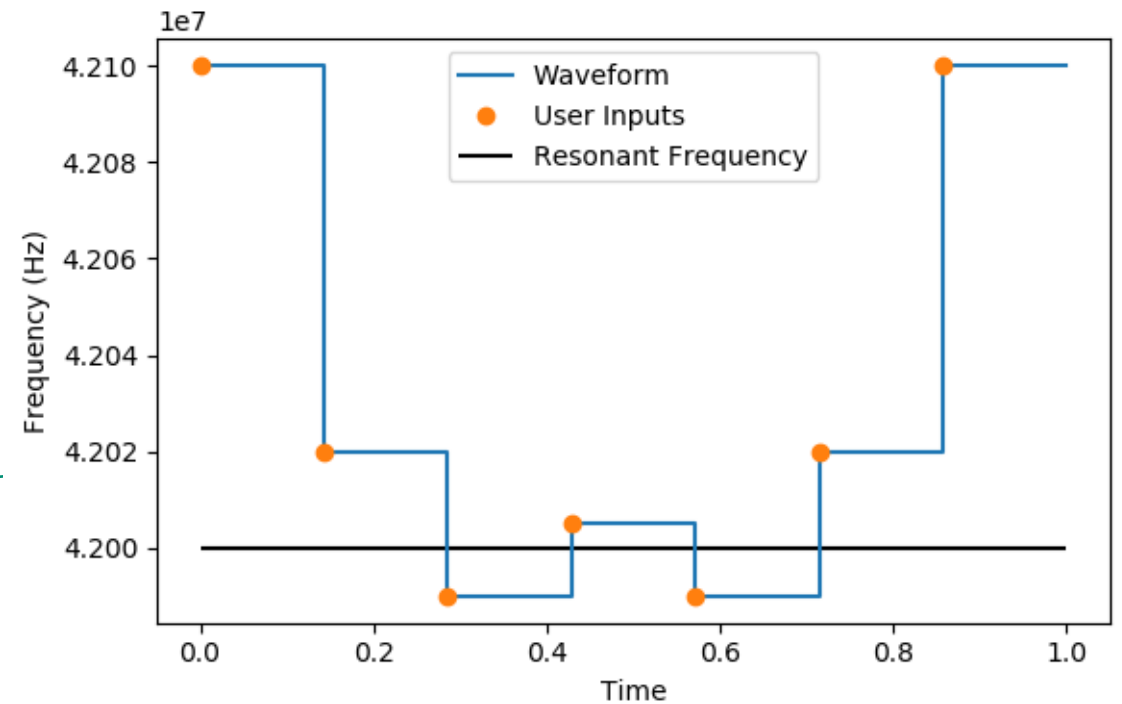
# Example I: Quick Review of Frequency Modulation



```
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                      )]
```

Frequency input is a list.  
Gives frequency jumps.

Frequency jumps applied at  
evenly spaced timing intervals.



# Example 1b: Single Tone Continuous Frequency Modulation

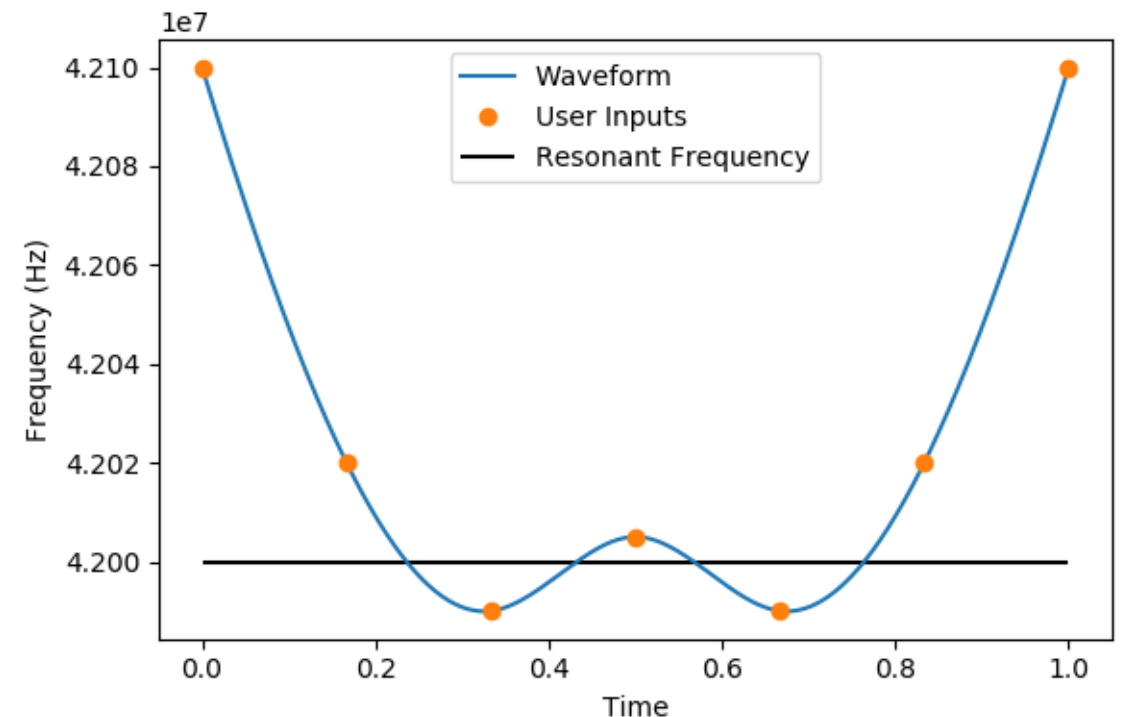


```

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.



## 7 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),
```

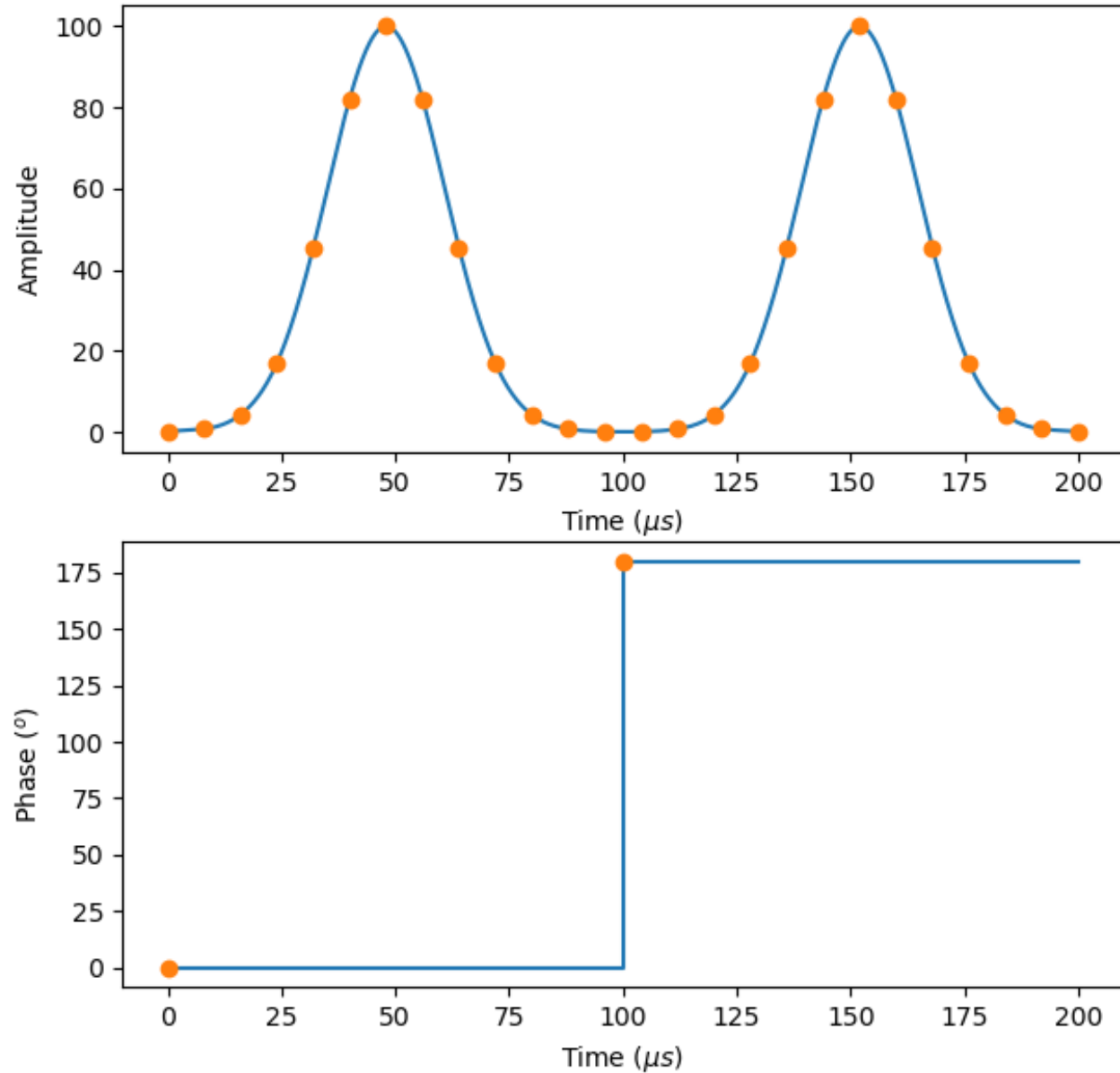
Cast to list for convenient comprehension.

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.

## Example 2 Continued: Gaussian Walsh Gate Waveform

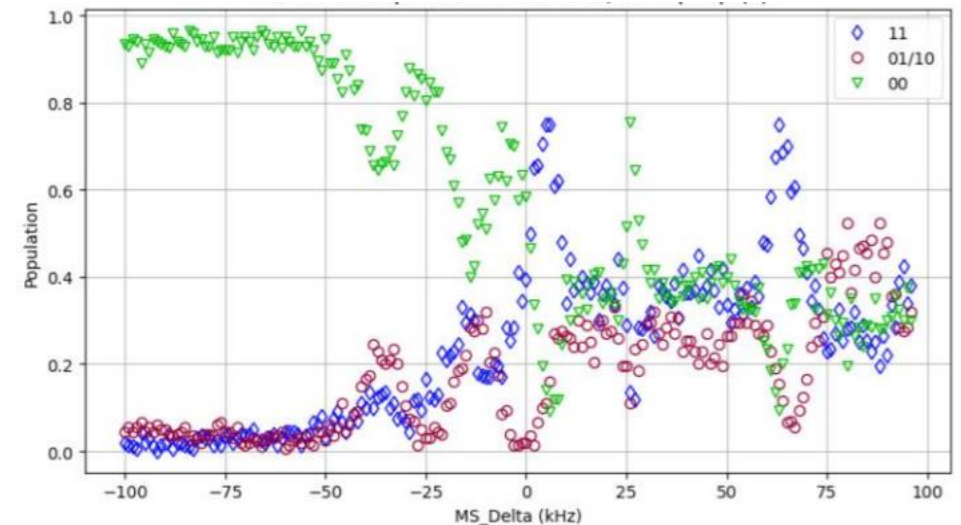
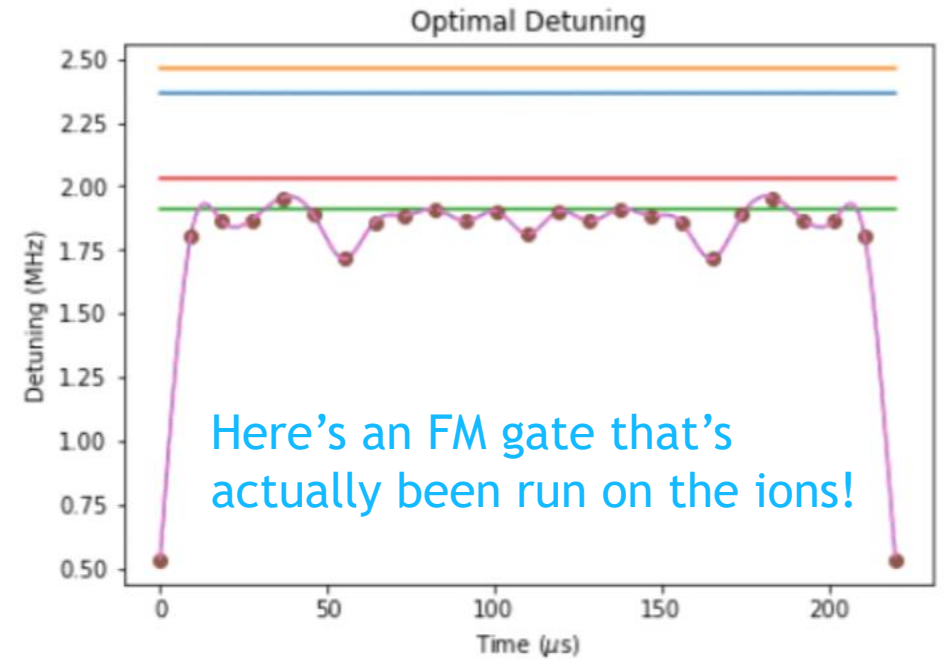


# Modulated Gates – JaqalPaw Tutorial



## 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, synchronization.
3. Handy features in exemplar
  - a) Parameterized pulses
  - b) Making use of both Raman beams
  - c) Programmatic configuration



# 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)

```

1 // Import JaqalPaw Code.
2 from ExemplarPulseDefinitions.ModulatedMSExemplar usepulses *
3
4 // Declare variables.
5 let target1 2
6 let target2 3
7 let ms_loops 1
8
9 register q[8]
10
11 // Prepare = Sideband cool, then Pump to F=0
12 prepare_all
13
14 loop ms_loops {
15     Mod_MS q[target1] q[target2]
16 }
17
18 measure_all

```

## Responsible for:

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

## JaqalPaw has all the substance

```

def run_q[0] (self, channel, channel, dummy_exp_time, global_duration) {
    // ... (substance code) ...
}

```

This code available at [qscout.sandia.gov](https://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     ## Principal 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 is the current structure the calibration parameters, but it is subject to change.
- If you need access to other parameters, talk to us and we'll work with you.

# Technical Details: Math with Frequencies; Synchronization

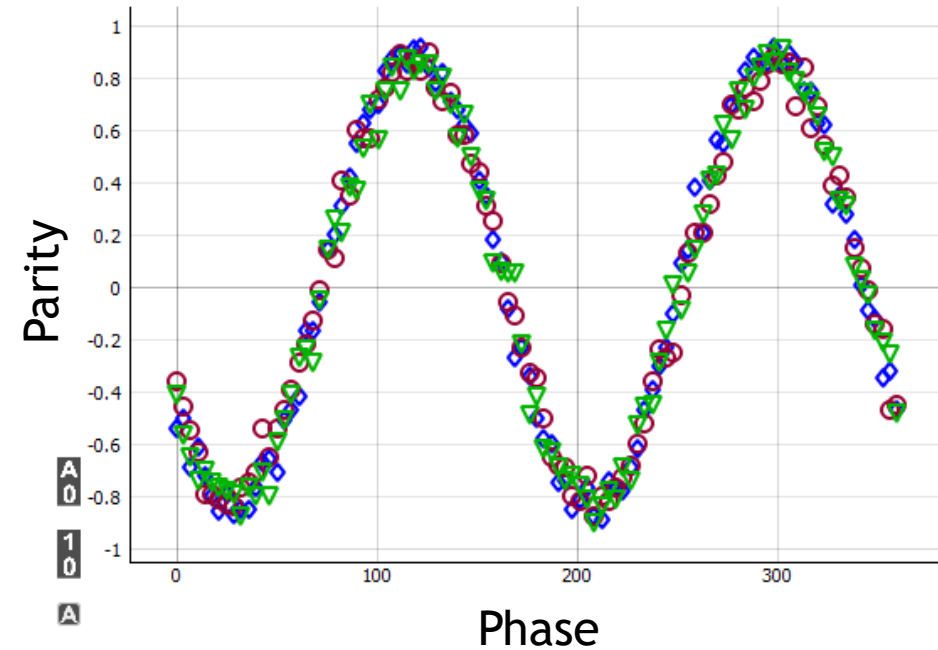
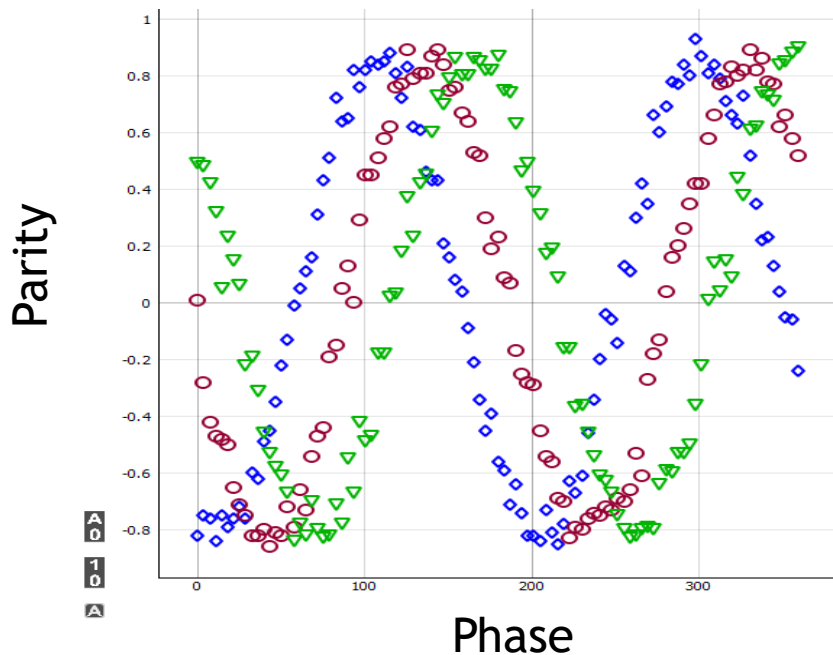


Math with frequencies: To avoid detrimental rounding errors from discretization, use the jaqalpaw utility function, “discretize\_frequency.”

```
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])
139
```

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

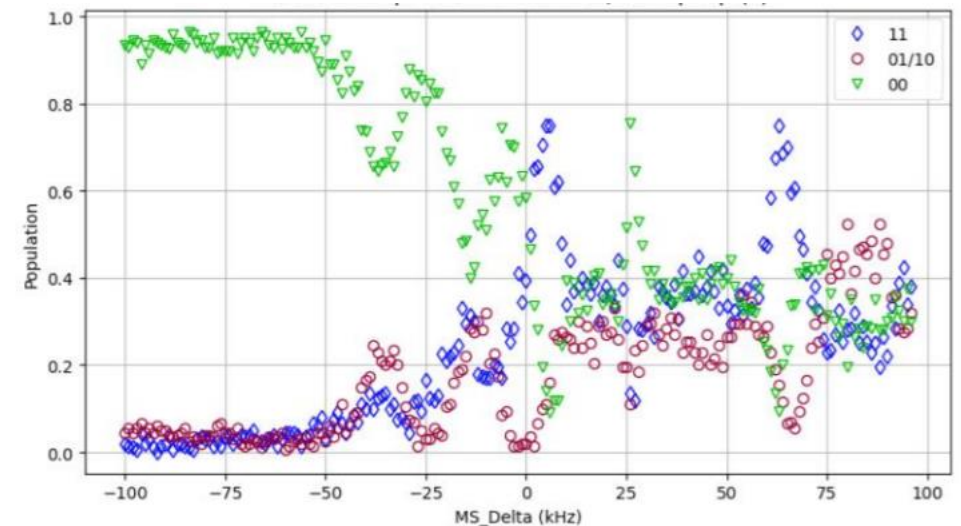
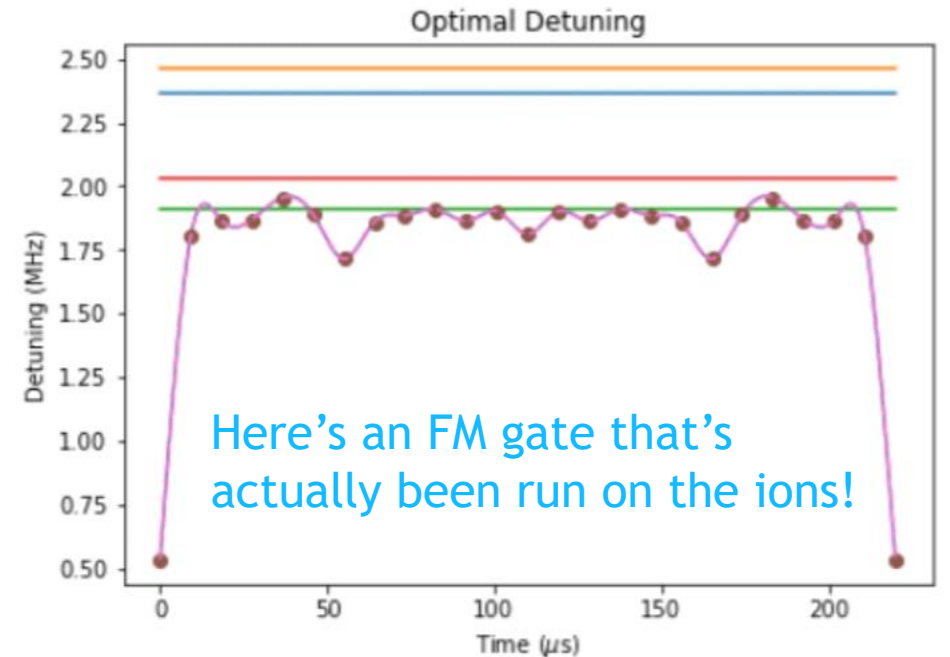


# Modulated Gates – JaqalPaw Tutorial



## 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, synchronization.
3. Handy features in exemplar
  - a) Parameterized pulses
  - b) Making use of both Raman beams
  - c) Programmatic configuration



# Build in Options by Parameterizing Pulses

## Ex 3a: MS Amplitude Calibration – Simple



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

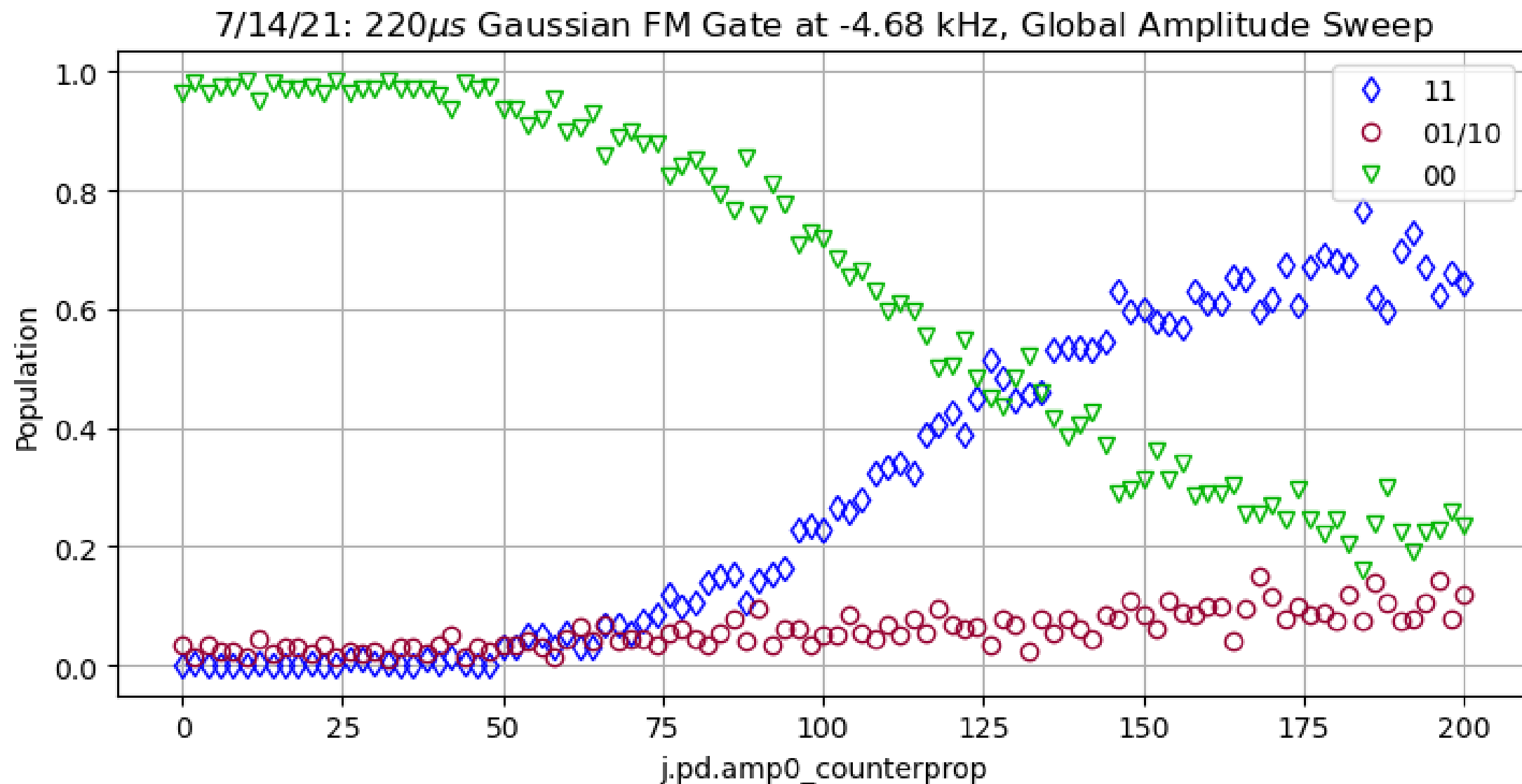
```
70 class ModulatedMSExemplar(QSCOUTBuiltins):  
71  
72     def gate_Mod_MS(self, channel1, channel2):  
73  
74         global_amp = self.amp0_counterprop  
75  
76         listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,  
77                                 freq0=global_beam_frequency,  
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.

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

```

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
161

```

Jaqal code modification:

```

...
10 let s 0

```

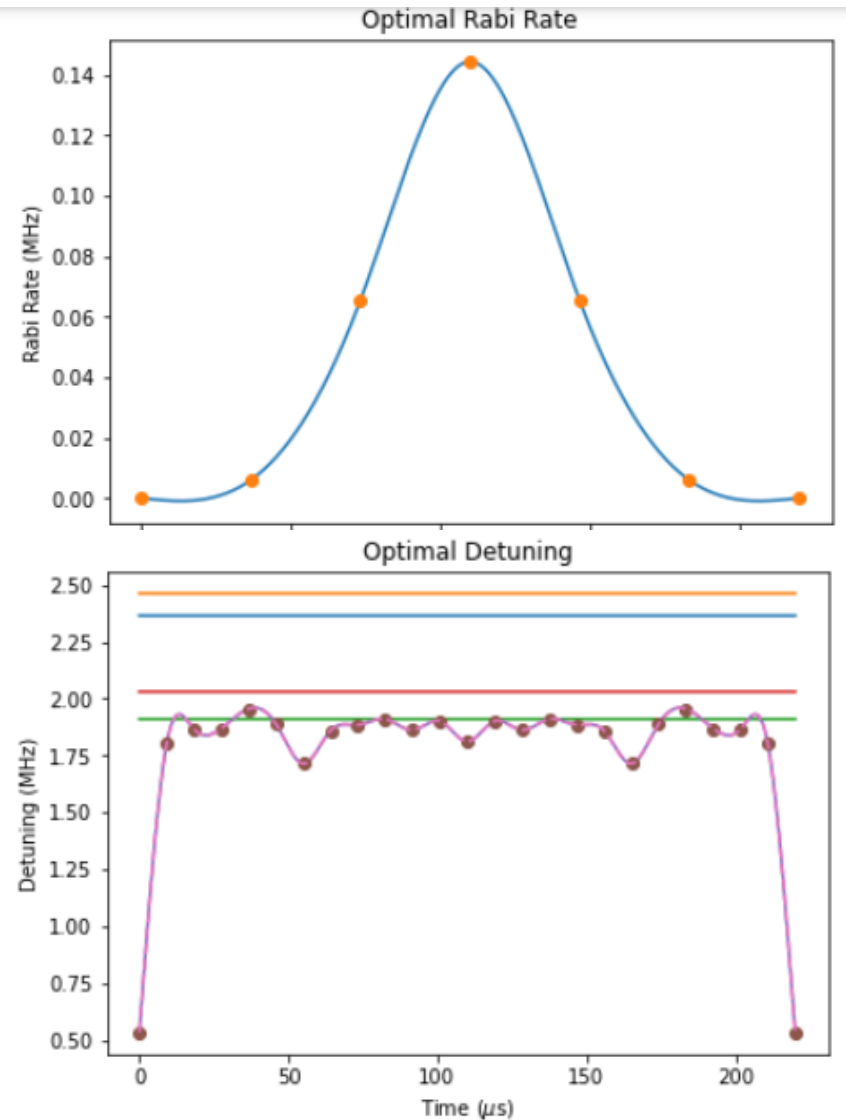
```

...
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.

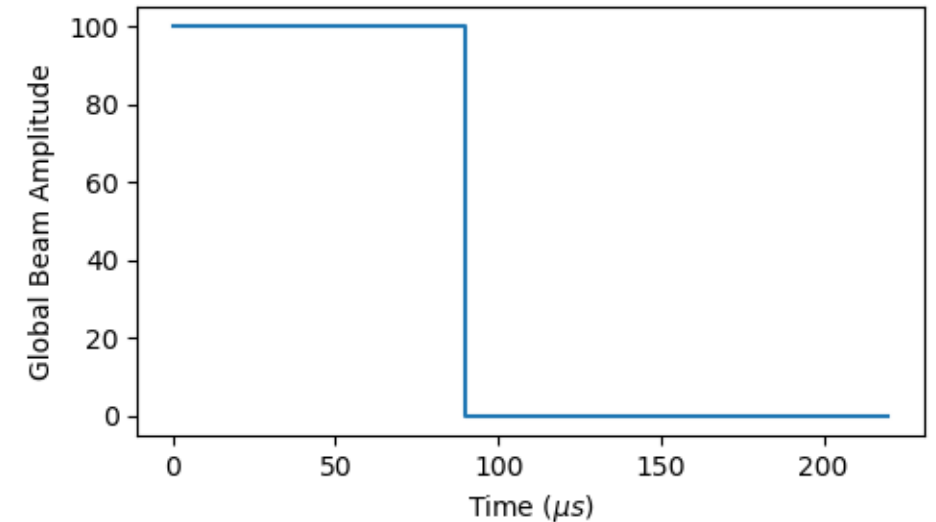
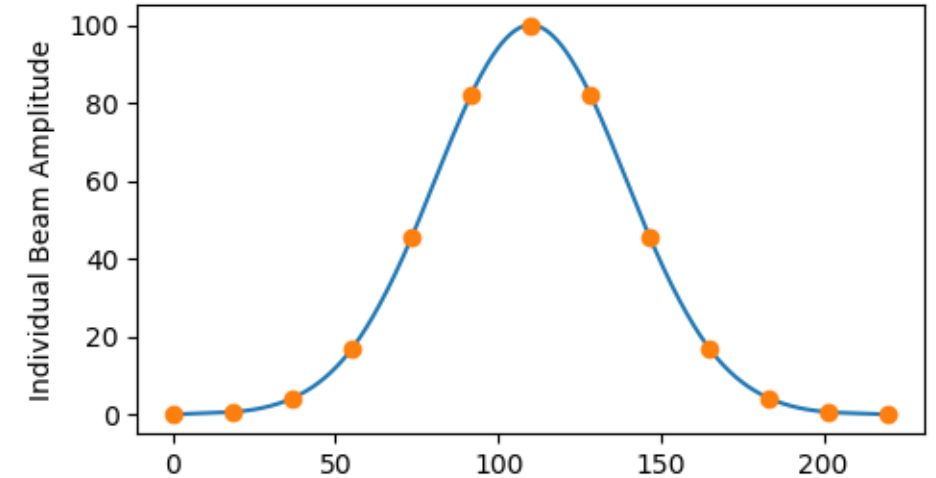
# Code for Spin State Tracking



```

240 def gate_Mod_MS(self, channel1, channel2, global_duration=-1e6):
241     """ General Modulated MS Gate (Produce optimal pulses found by solver).
242
243     ## Calculate gaussian knots to apply to IA beams.
244     rabi_rate_0 = 0.5/self.counter_resonant_pi_time
245     rabi_fac = 1
246     rabi_knots = self.gauss(npoints=13, A=rabi_rate_0,
247                             freqwidth=300e3, total_duration=4e-6)
248     amp_scale = [rabi_fac*rk/rabi_rate_0 for rk in rabi_knots]
249     amp_scale = np.array(amp_scale)
250
251     ## If global duration is within valid range, shut off GLOBAL_BEAM after that time.
252     if global_duration >= 0 and global_duration < self.MS_pulse_duration:
253         global_amp = [self.amp0_counterprop if t <= global_duration
254                       else 0 for t in np.linspace(0, self.MS_pulse_duration, 1000)]
255     else:
256         global_amp = self.amp0_counterprop
257
258     ## Frequency knots and other waveform info here.
259
260     listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
261                               freq0=global_beam_frequency,
262                               freq1=self.global_center_frequency,
263                               amp0=global_amp,
264                               amp1=0,
265                               phase0=phase_steps,
266                               phase1=0,
267                               sync_mask=0 if dummy_sync else 3,
268                               fb_enable_mask=0),
269                     PulseData(channel1, self.MS_pulse_duration,
270                               freq0=tuple(freq_fm0),
271                               freq1=tuple(freq_fm1),
272                               amp0=tuple(self.MS_blue_amp_list[channel1]*amp_scale),
273                               amp1=tuple(self.MS_red_amp_list[channel1]*amp_scale),
274                               framerot0 = framerot_input,
275                               apply_at_eof_mask=framerot_app,
276                               phase0=0,
277                               phase1=0,
278                               sync_mask=0 if dummy_sync else 3,
279                               fb_enable_mask=1
280                               ),
281

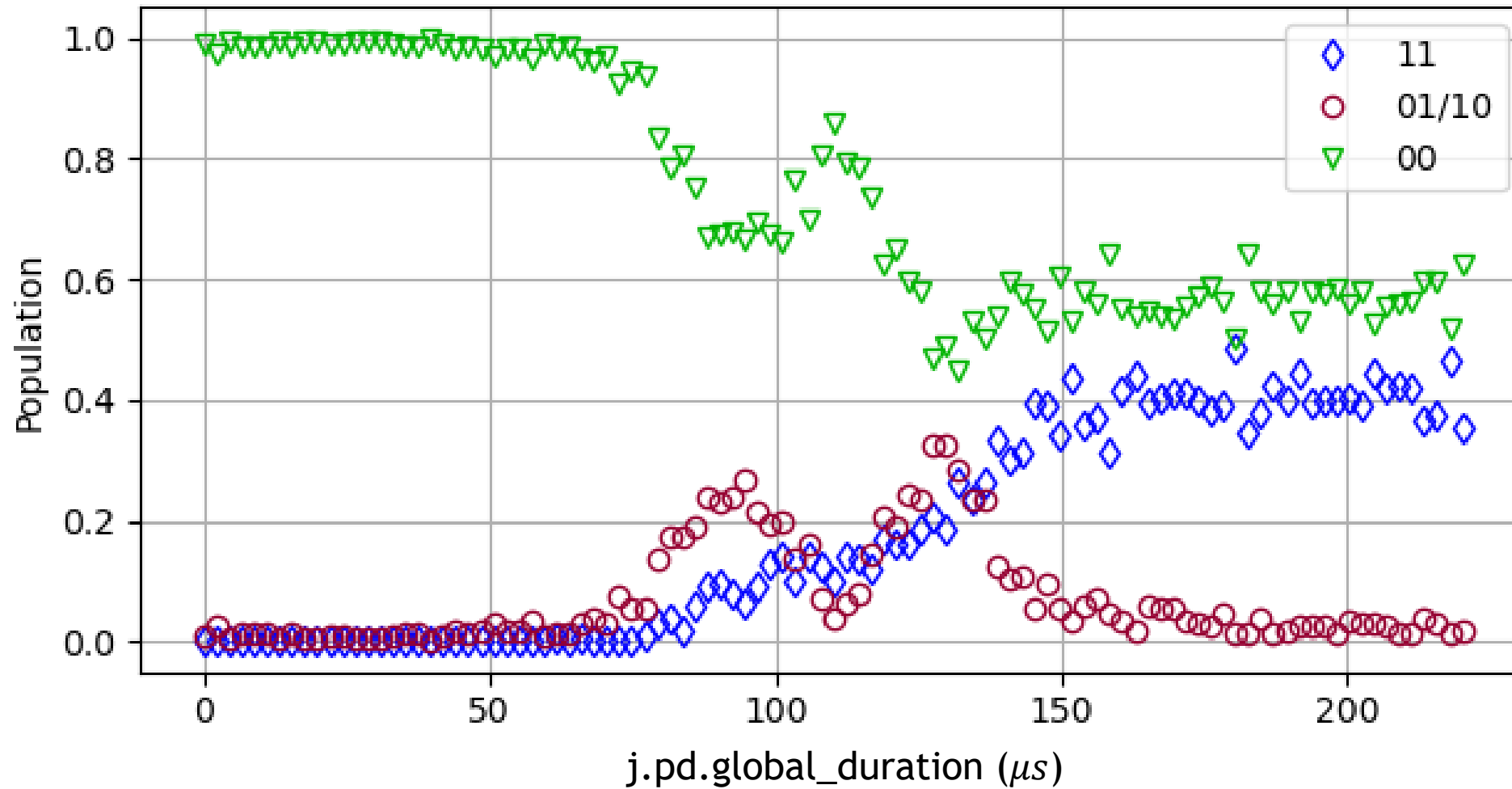
```



# Data For Spin State Tracking



220 $\mu$ s Gaussian FM Gate Dynamics



# Programmatic Configurations



```

25 @staticmethod
26 def get_cfg(cfg_file, delimiter=':'):
27     """ Method for parsing simple cfiguration 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 cfiguration 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 cfiguration 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]
105

```

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.

# Questions?





Sandia  
National  
Laboratories

# Transpiling Your Code in Other Languages to Run on QSCOUT

*Presented By*


B. C. A. Morrison



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.



- JaqalPaq-Extras contains tools for interoperability with other quantum languages/packages
  - Google's Cirq: `jaqalpaq.transpilers.cirq`
  - Rigetti's pyquil and Quil: `jaqalpaq.transpilers.quil`
  - ETH Zurich's ProjectQ: `jaqalpaq.transpilers.projectq`
  - IBM's Qiskit and OpenQASM: `jaqalpaq.transpilers.qiskit`
  - CQC's pytket and  $t|ket\rangle$ : `jaqalpaq.transpilers.tket`
  - QSCOUT hardware compatible scheduler: `jaqalpaq.scheduler`
- These allow you to quickly get up and running with Jaqal if you have existing code in other languages
- We'll go over each of these with examples:
  - Bell state preparation: Cirq, pyquil, and ProjectQ
  - VQE: Qiskit,  $t|ket\rangle$ , and the automatic scheduler



# Simple Example: Bell State Preparation

---

`jaqalpaq.transpilers.cirq`  
`jaqalpaq.transpilers.quil`  
`jaqalpaq.transpilers.projectq`

# Bell State Preparation in Cirq



```
In [1]: import cirq
from jaqalpaq.transpilers.cirq import jaqal_circuit_from_cirq_circuit
from jaqalpaq.generator import generate_jaqal_program
```

```
In [2]: cirq_bell = cirq.Circuit()
qubits = [cirq.LineQubit(0), cirq.LineQubit(1)]
cirq_bell.append(cirq.H.on(qubits[0]))
cirq_bell.append(cirq.CNOT(*qubits))
print(cirq_bell)
```

```
0: —H—@—
      |
1: ———X—
```

```
In [3]: cirq_ion_bell = cirq.ConvertToIonGates().convert_circuit(cirq_bell)
print(cirq_ion_bell)
```

```
0: —PhX(1.0)———MS(0.25π)—PhX(-0.5)^0.5—S^-1—
      |
1: —————MS(0.25π)—PhX(1.0)^0.5———
```

```
In [4]: jaqal_bell = jaqal_circuit_from_cirq_circuit(cirq_ion_bell)
print(generate_jaqal_program(jaqal_bell))
```

```
register allqubits[2]

prepare_all
R allqubits[0] 3.141592653589793 3.141592653589793
MS allqubits[0] allqubits[1] 0 1.5707963267948966
<
    R allqubits[0] -1.5707963267948972 1.5707963267948966
    R allqubits[1] 3.141592653589793 1.5707963267948966
>
Rz allqubits[0] -1.5707963267948966
measure_all
```

# Bell State Preparation in Quil



```
In [5]: import pyquil
        from pyquil.gates import *
        from numpy import pi
        from jaqalpaq.transpilers.quil import get_ion_qc, quil_gates
```

```
In [6]: qg = quil_gates()
        MS = qg["MS"]
        R = qg["R"]
        quil_bell = pyquil.Program()
        quil_bell += X(0)
        quil_bell += MS(0, pi/2, 0, 1)
        quil_bell += R(-pi/2, pi/2, 0)
        quil_bell += R(pi, pi/2, 1)
        quil_bell += RZ(-pi/2, 0)
        print(quil_bell)
```

```
In [7]: quil_qc = get_ion_qc(2)
        jaqal_bell = quil_qc.compile(quil_bell)
        print(generate_jaqal_program(jaqal_bell))
```

```
X 0
MS(0,pi/2) 0 1
R(-pi/2,pi/2) 0
R(pi,pi/2) 1
RZ(-pi/2) 0
```

```
register qreg[2]
```

```
{
    prepare_all
    Px qreg[0]
    MS qreg[0] qreg[1] 0.0 1.5707963267948966
    R qreg[0] -1.5707963267948966 1.5707963267948966
    R qreg[1] 3.141592653589793 1.5707963267948966
    Rz qreg[0] -1.5707963267948966
    measure_all
}
```

# Bell State Preparation in ProjectQ

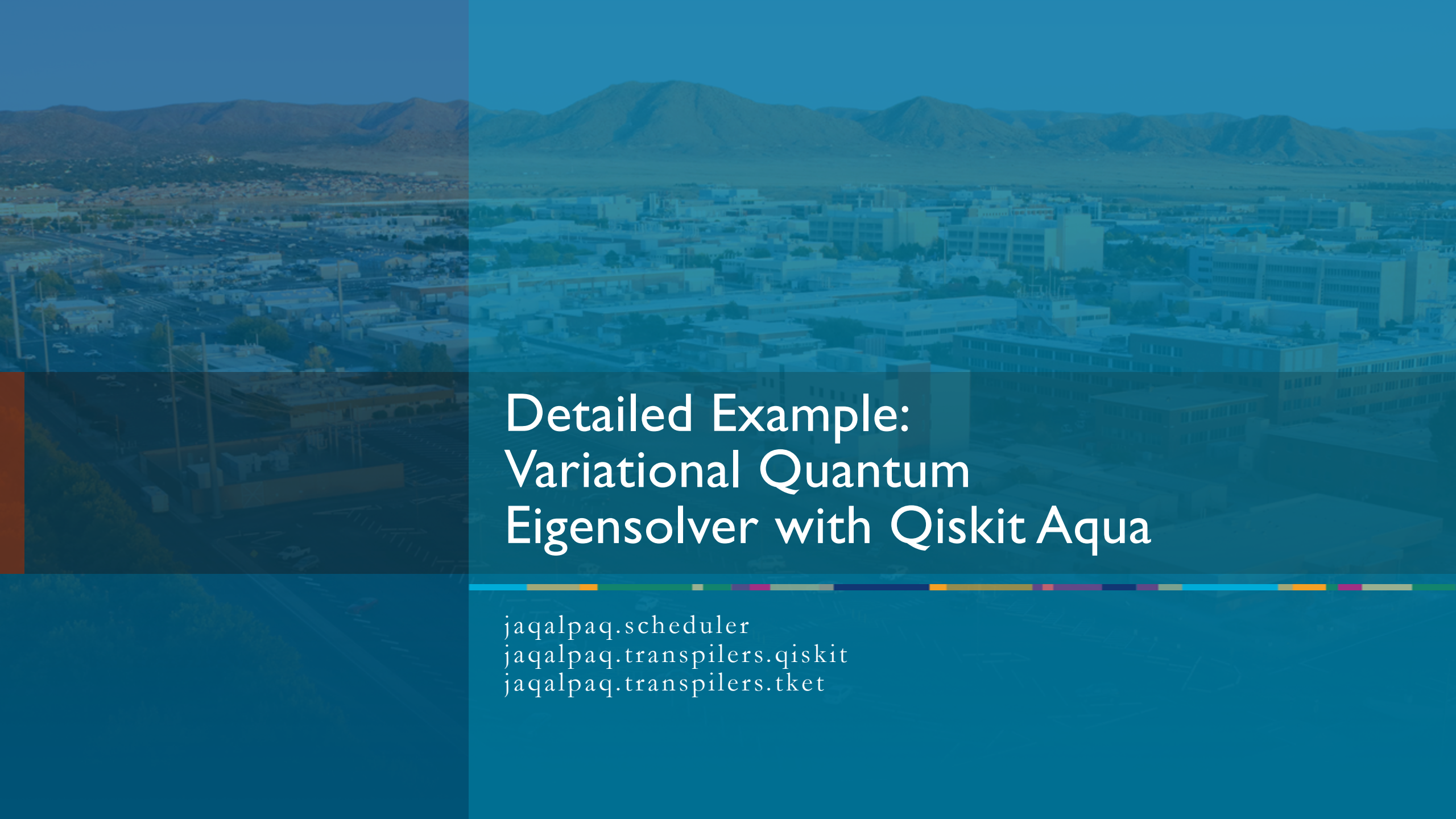


```
In [8]: import projectq
        from projectq.cengines import MainEngine, DummyEngine
        from projectq.ops import H, CNOT, Measure, All
        from jaqalpaq.transpilers.projectq import get_engine_list, JaqalBackend
```

```
In [9]: backend = JaqalBackend()
        engine_list = get_engine_list()
        engine = MainEngine(backend, engine_list, verbose=True)
        q1 = engine.allocate_qubit()
        q2 = engine.allocate_qubit()
        H | q1
        CNOT | (q1, q2)
        All(Measure) | [q1, q2]
        engine.flush()
        print(generate_jaqal_program(backend.circuit))
```

```
register q[2]
```

```
{
    prepare_all
    R q[0] 0 3.14159265359
    R q[0] 1.5707963267948966 3.141592653589413
    R q[0] 0 4.712388980384414
    R q[1] 0 1.570796326795
    MS q[0] q[1] 0 1.570796326795
    R q[0] 1.5707963267948966 1.570796326795
    measure_all
}
```



# Detailed Example: Variational Quantum Eigensolver with Qiskit Aqua

---

```
jaqalpaq.scheduler  
jaqalpaq.transpilers.qiskit  
jaqalpaq.transpilers.tket
```



```

register q[2]

{
    prepare_all
    R q[0] 0 3.14159265359
    R q[0] 1.5707963267948966 3.141592653589413
    R q[0] 0 4.712388980384414
    R q[1] 0 1.570796326795
    MS q[0] q[1] 0 1.570796326795
    R q[0] 1.5707963267948966 1.570796326795
    measure_all
}

```

```
In [10]: from jaqalpaq.scheduler import schedule_circuit
```

```
In [11]: scheduled_circuit = schedule_circuit(backend.circuit)
print(generate_jaqal_program(scheduled_circuit))
```

```

register q[2]

{
    prepare_all
    <
        R q[0] 0 3.14159265359
        R q[1] 0 1.570796326795
    >
    R q[0] 1.5707963267948966 3.141592653589413
    R q[0] 0 4.712388980384414
    MS q[0] q[1] 0 1.570796326795
    R q[0] 1.5707963267948966 1.570796326795
    measure_all
}

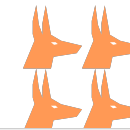
```



```
In [34]: 1 import qiskit
          2 from qiskit.chemistry.components.initial_states import HartreeFock
          3 from qiskit.chemistry.components.variational_forms import UCCSD
          4 from qiskit.transpiler import PassManager
          5 from qiskit.chemistry.drivers import PySCFDriver, UnitsType
          6 from qiskit.chemistry import FermionicOperator
          7
          8 from jaqalpaq.transpilers.qiskit import jaqal_circuit_from_qiskit_circuit, ion_pass_manager, get_ion_instance
          9 from jaqalpaq.emulator import run_jaqal_circuit
         10 from qscout.v1.std import NATIVE_GATES
```

- Specify atom positions
- Construct molecule object and Hamiltonian as fermionic operator
- Map Hamiltonian to qubits
- Construct Hartree-Fock state
- Construct unitary coupled cluster singles and doubles ansatz
- Build circuit

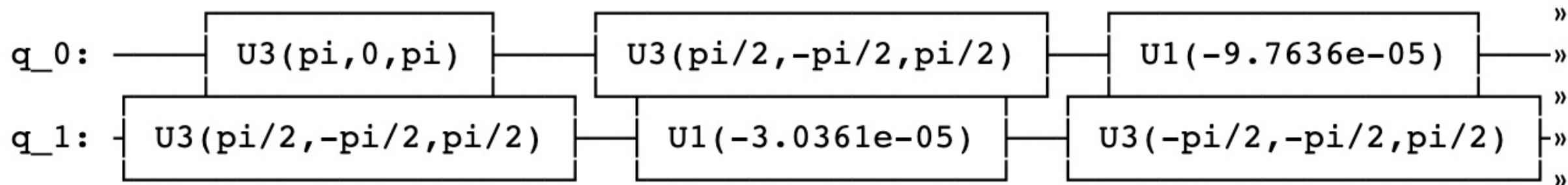
# Constructing the VQE circuit in Qiskit



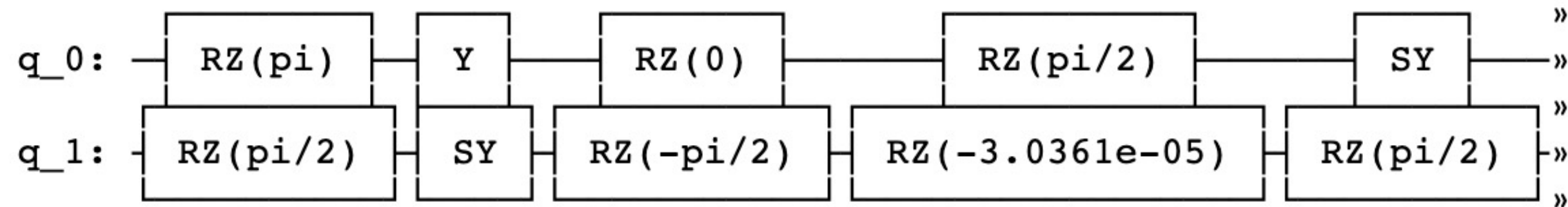
In [35]:

```
1 atom_positions = [[0.0, 0.0, 0.0], [1.0, 0.0, 0.0]]
2
3 molecule = PySCFDriver(
4     atom=''.join(["H %f %f %f; " % tuple(a) for a in atom_positions]),
5     unit=UnitsType.BOHR,
6     charge=0,
7     spin=0,
8     basis='sto6g'
9 ).run()
10
11 fermionic_op = FermionicOperator(molecule.one_body_integrals, molecule.two_body_integrals)
12 qubit_op = fermionic_op.mapping(map_type='parity', threshold=0.00000001)
13 hfs = HartreeFock(num_orbitals=4, num_particles=2, qubit_mapping='parity', two_qubit_reduction=True)
14
15 uccsd = UCCSD(
16     num_orbitals=4,
17     num_particles=2,
18     initial_state=hfs,
19     qubit_mapping="parity",
20     two_qubit_reduction=True,
21     num_time_slices=1
22 )
23
24 params = (4.881817576986406e-05, -1.5180312771430236e-05, -0.07605047960428524)
25 superconducting_circuit = uccsd.construct_circuit(params).decompose()
26 print(superconducting_circuit)
```

# Unrolling the VQE Circuit to Ion Gates



```
In [36]: 1 ion_circuit = ion_pass_manager().run(superconducting_circuit)
          2 print(ion_circuit)
```



# Transpiling the VQE Circuit to Jaqal



```
In [62]: jaqal_circuit = jaqal_circuit_from_qiskit_circuit(ion_circuit)
print(generate_jaqal_program(jaqal_circuit))
```

```
register baseregister[2]
map q baseregister[0:2:1]
{
    prepare_all
    Rz q[0] 3.141592653589793
    Py q[0]
    Rz q[0] 0.0
    Rz q[0] 1.5707963267948966
    Sy q[0]
    Rz q[0] -1.5707963267948966
    Rz q[0] -9.76363515397281e-05
    Rz q[0] 1.5707963267948966
    Syd q[0]
    Rz q[0] -1.5707963267948966
    Rz q[0] 1.5707963267948966
    Sy q[0]
    Rz q[0] -1.5707963267948966
    Sy q[0]
    Rz q[1] 1.5707963267948966
    Sy q[1]
    Rz q[1] -1.5707963267948966
    Rz q[1] -3.0360625542860462e-05
    Rz q[1] 1.5707963267948966
    Syd q[1]
    Rz q[1] -1.5707963267948966
    Rz q[1] 3.141592653589793

    Sy q[1]
    Rz q[1] 0.0
    MS q[0] q[1] 0.0 1.5707963267948966
    Sxd q[0]
    Syd q[0]
    Sy q[0]
    Sxd q[1]
    Rz q[1] 0.07605047960428521
    MS q[0] q[1] 0.0 1.5707963267948966
    Sxd q[0]
    Syd q[0]
    Rz q[0] 1.5707963267948966
    Syd q[0]
    Rz q[0] -1.5707963267948966
    Rz q[0] 3.141592653589793
    Sy q[0]
    Rz q[0] 0.0
    Sy q[0]
    Sxd q[1]
    Rz q[1] 3.141592653589793
    Sy q[1]
    Rz q[1] 0.0
    Rz q[1] 1.5707963267948966
    Sy q[1]
    Rz q[1] -1.5707963267948966

    MS q[0] q[1] 0.0 1.5707963267948966
    Sxd q[0]
    Syd q[0]
    Sy q[0]
    Rz q[0] 0.0
    Sxd q[1]
    Rz q[1] 1.5707963267948966
    Syd q[1]
    Rz q[1] -1.5707963267948966
    measure_all
}
```

# Scheduling and Emulating the VQE Circuit



```
In [36]: scheduled_jaqal_circuit = schedule_circuit(jaqal_circuit)
print(generate_jaqal_program(scheduled_jaqal_circuit))
```

```
register baseregister[2]

map q baseregister[0:2:1]

{
    prepare_all
    <
        Rz q[0] 3.141592653589793
        Rz q[1] 1.5707963267948966
    >
    <
        Py q[0]
        Sy q[1]
    >
    <
        Rz q[0] 0.0
        Rz q[1] -1.5707963267948966
    >
}
```

```
run_jaqal_circuit(scheduled_jaqal_circuit).subcircuits[0].probability_by_str
```

```
OrderedDict([('00', 2.3832142370672216e-09),
             ('10', 0.9942274635465904),
             ('01', 0.005772533839753652),
             ('11', 2.3044191554966176e-10)])
```



```
In [40]: 1 from jaqalpaq.transpilers.tket import jaqal_circuit_from_tket_circuit
          2 from pytket.predicates import CompilationUnit
          3 from pytket.extensions.qiskit import qiskit_to_tk
          4 from pytket.passes import SynthesiseUMD, DecomposeBoxes
```

```
In [41]: 1 tket_circuit = qiskit_to_tk(superconducting_circuit)
          2 tket_circuit
```

```
[U3(1*PI, 0*PI, 1*PI) q[0]; U3(0.5*PI, 1.5*PI, 0.5*PI) q[1]; U3(0.5*PI, 1.5*PI, 0.5*PI) q[0]; U1(1.99999*PI) q[1]; U1
(1.99997*PI) q[0]; U3(1.5*PI, 1.5*PI, 0.5*PI) q[1]; U3(1.5*PI, 1.5*PI, 0.5*PI) q[0]; U2(0*PI, 1*PI) q[1]; U3(0.5*PI,
1.5*PI, 0.5*PI) q[0]; CX q[0], q[1]; U1(0.0242076*PI) q[1]; CX q[0], q[1]; U3(1.5*PI, 1.5*PI, 0.5*PI) q[0]; U2(0*PI,
1*PI) q[1]; U2(0*PI, 1*PI) q[0]; U3(0.5*PI, 1.5*PI, 0.5*PI) q[1]; CX q[0], q[1]; U1(1.97579*PI) q[1]; CX q[0], q[1];
U2(0*PI, 1*PI) q[0]; U3(1.5*PI, 1.5*PI, 0.5*PI) q[1]; ]
```

# Optimizing Circuits for Ion Hardware with t|ket>



```
unit = CompilationUnit(tket_circuit)
DecomposeBoxes().apply(unit)
SynthesiseUMD().apply(unit)
tket_jaqal_circuit = jaqal_circuit_from_tket_circuit(unit.circuit)
print(generate_jaqal_program(tket_jaqal_circuit))

register baseregister[2]
```

```
map q baseregister[0:2:1]
```

```
{
    prepare_all
    Rz q[0] 1.5707963267948966
    Rz q[1] 1.5707659661693572
    R q[0] 3.141592653589793 9.76363494493584e-05
    R q[1] 1.570765966169354 1.5707963267948966
    MS q[0] q[1] 0 1.5707963267948966
    R q[0] 3.141592653589793 1.5707963267948966
    Rz q[1] 0.0760504796042852
    R q[1] 0.0 4.71238898038469
    MS q[0] q[1] 0 1.5707963267948966
    R q[0] 1.5707963267948966 1.5707963267948966
    Rz q[1] 3.141592653589793
    R q[1] 1.5707963267948966 1.5707963267948966
    MS q[0] q[1] 0 1.5707963267948966
    R q[0] 3.141592653589793 1.5707963267948966
    Rz q[1] 6.2071348275753015
    R q[1] 0.0 4.71238898038469
    MS q[0] q[1] 0 1.5707963267948966
    R q[1] 0.0 4.71238898038469
    measure_all
}
```

```
run_jaqal_circuit(scheduled_jaqal_circuit).subcircuits[0].probability_by_str
```

```
OrderedDict([('00', 2.3832142370672216e-09),
             ('10', 0.9942274635465904),
             ('01', 0.005772533839753652),
             ('11', 2.3044191554966176e-10)])
```

```
optimized_jaqal_circuit = schedule_circuit(tket_jaqal_circuit)
print(generate_jaqal_program(optimized_jaqal_circuit))
```

```
map q baseregister[0:2:1]
```

```
{
    prepare_all
    <
        Rz q[0] 1.5707963267948966
        Rz q[1] 1.5707659661693572
    >
    <
        R q[0] 3.141592653589793 9.76363494493584e-05
        R q[1] 1.570765966169354 1.5707963267948966
    >
    MS q[0] q[1] 0 1.5707963267948966
    <
        R q[0] 3.141592653589793 1.5707963267948966
        Rz q[1] 0.0760504796042852
    >
    R q[1] 0.0 4.71238898038469
    MS q[0] q[1] 0 1.5707963267948966
    <
        R q[0] 1.5707963267948966 1.5707963267948966
        Rz q[1] 3.141592653589793
    >
    R q[1] 1.5707963267948966 1.5707963267948966
    MS q[0] q[1] 0 1.5707963267948966
    <
        R q[0] 3.141592653589793 1.5707963267948966
        Rz q[1] 6.2071348275753015
    >
    R q[1] 0.0 4.71238898038469
    MS q[0] q[1] 0 1.5707963267948966
    R q[1] 0.0 4.71238898038469
    measure_all
}
```

```
run_jaqal_circuit(optimized_jaqal_circuit).subcircuits[0].probability_by_str
```

```
OrderedDict([('00', 2.383214070533768e-09),
             ('10', 0.9942274635465904),
             ('01', 0.005772533839753652),
             ('11', 2.304418045273593e-10)])
```