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: March 2, 2023)

Abstract

One of the many challenges of developing an open user testbed such as QSCOUT [1] is providing an interface that maintains simplicity without compromising expressibility or control. This interface comprises two distinct elements: a quantum assembly language designed for specifying quantum circuits at the gate level, and a low-level counterpart used for describing gates in terms of waveforms that realize specific quantum operations. Jaqal, or "Just another quantum assembly language," is the language used in QSCOUT for gate-level descriptions of quantum circuits [2, 3]. JaqalPaw, or "Jaqal pulses and waveforms," is its pulse-level counterpart. This document concerns the latter, and presents a description of the tools needed for precisely defining the underlying waveforms associated with a gate primitive.

* dlobser@sandia.gov; https://qscout.sandia.gov

I. BACKGROUND

QSCOUT uses a linear chain of ^{171}Yb$^+$ trapped ion qubits. Coherent operations on these qubits involve manipulating both their internal spin states, as well as their collective motional states, via optical Raman transitions. The specific experimental details of the hardware involved are described in the QSCOUT manual [1], but a few of these details are repeated here as they relate to certain aspects of the requirements for pulse-level control of these qubits. Chief among these details is understanding the nature of these Raman transitions, and the pulsed laser used to drive them.

For the purposes of this manual, these elements can be distilled down to two basic rules of thumb:

- All operations must involve at least two tones to form a Raman transition.

- The higher frequency tone in each Raman transition must employ a frequency feedback correction.

The specific details of how this frequency feedback system works, and why it is needed, are described in the QSCOUT manual [1, §IV B, §VI A 2] and briefly overviewed in section IV. Beyond these basic rules, there are other subtleties in how these Raman beams can be configured, such as co- or counter-propagating for motionally insensitive or sensitive operations respectively. Counter-propagating gates are achieved via the use of a global beam, which is driven by a specific hardware channel (channel zero by convention) and will be labeled `GLOBAL_BEAM`. All other channels are used for individual qubit addressing.

From the perspective of engineering custom gates in terms of raw waveform data, it is imperative that the user have access to the latest calibration data. Because calibration data is subject to change immediately prior to running a Jaqal program on the experimental apparatus, this information is exposed in the form of variables that are updated with the latest calibration data at run time. Users are thus expected to reference these variables in their pulse definitions in order to ensure that their gates make use of the most recent calibrated data.

However, there are a number of details, independent of the specific knowledge of the calibration parameters, that are necessary for a basic understanding of how to use JaqalPaw to compose specific waveforms. These details will be the primary focus of this manual, and

will build from a kernel of basic elements that require no specific experimental knowledge of the QSCOUT system.

II.   ANATOMY OF A GATE PULSE CLASS

Rather than developing a custom language for defining gates at the pulse level, JaqalPaw is written purely in Python and requires a simple set of conventions. The first line in a Jaqal program should contain the desired set of gate definitions that need to be referenced. This import has the form

```
from GateDefinitionFileName.GateDefinitionClassName usepulses *
```

This references a Python class which contains member functions that can *optionally* be exposed to Jaqal as long as the function name has a "gate_" prefix. These functions are then directly accessible in Jaqal and are identified by the function name where gate_ is implicitly stripped.

```
class GatePulse:                        from GateFile.GatePulse usepulses *
    def gate_G(self, qubit):            register q[8]
        return ...                      G q[2]
```

EX. 1: A side by side comparison of JaqalPaw (left) with Jaqal (right). The gate gate_G is accessed in Jaqal as G.

The function's argument signature, with the exception of *self*, can be used for passing input arguments from Jaqal. In the above example, the gate G has a single input argument that corresponds to the target qubit. The body of the gate definition function is up to the user, as JaqalPaw simply uses Python 3. Likewise, any additional helper functions can be included and should simply leave off the gate_ prefix. Gate pulse definition classes do not use __init__ methods, but instead make use of class-level attributes for passing in calibration data. Calibration parameters always have a type annotation, and they will be overwritten at run time. In order to access the standard calibration parameters, a user-defined class should inherit a standard QSCOUT gate class such as qscout.v1.std. So an extension of the GatePulse class might look like the following

```python
import numpy as np
from qscout.v1.QSCOUTBuiltins import QSCOUTBuiltins

class GatePulse(QSCOUTBuiltins):

    calibrated_amplitude : float = 45.0

    @staticmethod
    def gauss(amp, sigma, num_points):
        x_points = np.linspace(-1.0, 1.0, num_points)
        return amp * np.exp(-x_points**2/2/sigma**2)

    def gate_G(self, qubit, phi):
        scaled_amp = self.calibrated_amplitude * phi / np.pi
        amplitude_envelope = self.gauss(scaled_amp, .3, 7)
        return [PulseData(...)]
```

EX. 2: A basic gate pulse class. Gate definitions start with `gate_` and return a list of `PulseData` objects. Calibration parameters are class-level variables with type annotations.

This covers essentially all of the conventions used for constructing a simple gate pulse class. The final detail is the return signature of a gate definition. A gate's return signature must *always* be a list, and that list must only contain `PulseData` objects, even if there is only a single `PulseData` object in the list.

III. THE PULSE DATA CLASS

All of the low-level hardware control is exposed through the `PulseData` class, which is designed specifically to target the custom "Octet" coherent control hardware developed for QSCOUT. This class is effectively a Python `dataclass` [4], in that it simply carries a set of parameters that are translated directly into a format sent to the coherent control hardware. The full argument signature is shown in Fig. 1, and the purpose and usage of each argument

4

will be described in the remainder of this section.

```
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` at a bare minimum requires a channel and a duration, in which case the output pulse will effectively be a simple NOP for the given duration since the other parameters default to zero. The remaining parameters can be separated into two categories: basic waveform data such as frequency, phase, and amplitude, and metadata used for controlling specific settings for the pulse. Each `PulseData` object contains information for two independent tones that are digitally summed before being converted to an output waveform. As opposed to an arbitrary waveform generator, these waveforms are always sinusoidal, but the frequency, phase, and amplitude of the oscillations are specified by the user.

A.   Controlling Frequency, Phase, and Amplitude

The physical units for these inputs are listed in Table I.

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.

These parameters can be controlled using three basic input types: a constant value, a list of values for discrete modulation, and a tuple of values for spline modulation. For a constant value input, that value is applied for the specified duration of the pulse. A very simple single-tone square pulse can be defined as shown in Ex. 3.

```
def gate_G(self, qubit):
    return [PulseData(qubit,        # output channel
                      1.25e-6,      # duration (s)
                      freq0=200e6,  # frequency (Hz)
                      amp0=50,      # amplitude (arb.)
                      phase0=0)]    # phase (deg.)
```

EX. 3: A simple square pulse is defined in terms of constant parameters.

In Jaqal and JaqalPaw, gates are run back to back without any gaps unless strictly specified by the user. This means that running the following Jaqal code will result in a single square pulse that has 4 times the duration of `G`.

```
loop 4 {
  G q[2]
}
```

If a gap is desired, this could be implemented as either a separate `gap` gate with a short duration (e.g. one that returns `[PulseData(qubit, 0.25e-6)]`), or directly in the gate definition. The back-to-back functionality applies to JaqalPaw in the same way it applies to Jaqal. Namely, returning a list with multiple `PulseData` objects on the same channel will be run in the order specified. Thus, the following three cases are equivalent[1]:

---

[1] To keep the code concise, only `amp0` is defined. `freq0` has been omitted, but would generally be needed to produce output on the hardware.

```python\ndef gate_G(self, qubit):\n return [PulseData(qubit, 1.25e-6, amp0=50)]\n\n\ndef gate_gap(self, qubit):\n return [PulseData(qubit, .25e-6)]\n```	```\nloop 4 {\n G q[2]\n gap q[2]\n}\n```
```python\ndef gate_G_gap(self, qubit):\n return [PulseData(qubit, 1.25e-6, amp0=50),\n PulseData(qubit, 0.25e-6)]\n```	```\nloop 4 {\n G_gap q[2]\n}\n```
```python\ndef gate_G_gap_multi(self, qubit, loops):\n return [PulseData(qubit, 1.25e-6, amp0=50),\n PulseData(qubit, 0.25e-6)] * loops\n```	```\nG_gap_multi q[2] 4\n```

EX. 4:   Three equivalent ways of chaining pulses together. In these examples, gaps are explicitly added after each pulse.

When a series of discrete updates to a parameter is needed, the input to the parameter is a list of values. Discrete modulations are treated as a series of back-to-back square pulses, where each pulse has the same time, t', which evenly subdivides the total duration, t. In other words, a list with N elements comprises N pulses each with duration $t' = t/N$.

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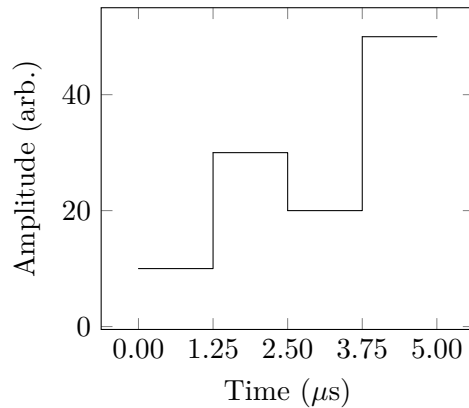

EX. 5:   Discrete updates are represented as a list of inputs and are equally distributed across the duration of the pulse.

For continuous modulation via natural cubic splines, the input format simply changes from a list, `[...]`, to a tuple, `(...)`. The input values are knots of the spline, and are also equally distributed over the duration of the pulse. As opposed to the discrete case, a tuple of k knots comprises $k - 1$ sets of spline coefficients used for interpolation, resulting in a duration of $t' = t/(k - 1)$ between knots.

7

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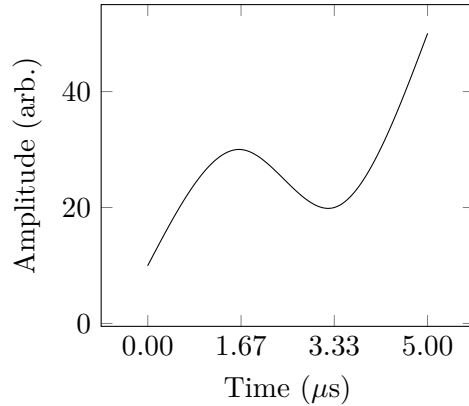

EX. 6: Smooth updates are represented as a tuple.

Multiple parameters can be modulated simultaneously and the length of the list/tuple can differ across all parameters. Asymmetry between the length of different modulation inputs is supported because the rules for equally distributing modulation data over the pulse time are handled separately for each parameter.

```
def gate_G(self, qubit):
    spline_amps = (10,30,20,50,20,60,30,50)
    discrete_amps = [10,30,20,30,50]
    return [PulseData(qubit,
                      5e—6,
                      freq0=200e6,
                      freq1=230e6,
                      amp0=spline_amps,
                      amp1=discrete_amps)]
```
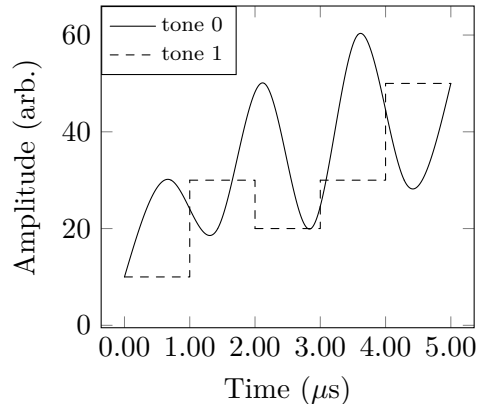

EX. 7: Different modulation types can be used across parameters.

Mixed modulation types on a single parameter are not currently supported in the form of a direct parameter input. However, if one needs to combine modulation types on the same parameter, then multiple `PulseData` objects can be included in the return list to construct piecewise functions. The following example illustrates how one can construct a piecewise function for a pulse, as well as how the natural cubic splines can be used to create linear ramps when only two knots are specified.

8

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
                   amp0=(0,9,41,50)),
          PulseData(qubit, 2e-6,
                   amp0=50),
          PulseData(qubit, 2e-6,
                   amp0=(50,0))]
```
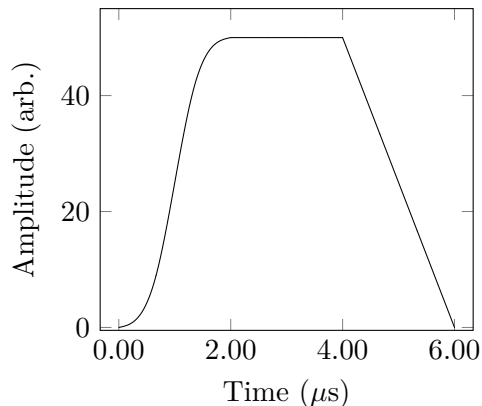
EX. 8: Piecewise functions can be constructed by chaining `PulseData` objects together.

These parameters will be run in the order received on a per-channel basis. If multiple channels are used, the data from each channel will be executed in parallel. In situations where the total duration on each channel differs, channels will be padded with a NOP pulse at the end to ensure the duration is matched across all channels at run time.

```python
def gate_G(self, qubit):
    return [PulseData(qubit, 2e-6,
                   amp0=(0,9,41,50)),
          PulseData(qubit, 2e-6,
                   amp0=50),
          PulseData(qubit, 2e-6,
                   amp0=(50,0)),
          PulseData(GLOBAL_BEAM, 4.5e-6,
                   amp0=(0,30,20,70)]
```
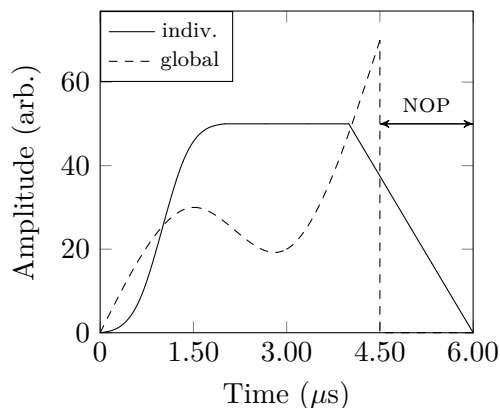
EX. 9: Chaining `PulseData` objects on different channels results in parallel execution. Differences in cumulative duration will be padded with a NOP pulse.

B. Virtual Rotations

In many systems, including QSCOUT, the ability to directly drive gates in all dimensions is often challenging, especially for individually addressed qubits. Typically, two rotation axes are accessible, $e.g.$, $\hat{x}$ and $\hat{y}$, in which case $\hat{z}$ rotations must be handled differently.

9

One approach is to treat $\hat{z}$ rotations virtually. This generally involves redefining gates with adjusted phase offsets to account for the accumulated virtual phase. However, the Octet hardware handles virtual rotations at the hardware level. This allows gate definitions to be decoupled from the context dependency associated with virtual rotations.

The nomenclature used by JaqalPaw is "frame rotation", where the frame accumulates the input rotations until explicitly reset. The input arguments to `PulseData` are `framerot0` and `framerot1`. There is an important distinction between the `0` and `1` labels, which differ from the tone labels.

Frame rotations are intended to represent a virtual equivalent of a physical gate. Because QSCOUT's gates use Raman transitions, the phase of the gate is determined by the *difference* between the phase of the two Raman tones. Thus for co-propagating[2] gates, where both tones are set on the same channel, the frame rotation must only be applied to a single tone. Two-qubit Mølmer-Sørensen gates often implement the motional sideband frequencies on a single channel, and the other leg of the Raman transition on the `GLOBAL_BEAM` channel. The red and blue sideband tones independently form Raman transitions with the global beam. This means that the frame rotation must be applied to both red and blue tones to achieve the proper virtual phase offset.

Other types of gate implementations may have an additional requirement that the virtual phase be inverted. This condition may arise when the absorption and emission paths are swapped. The effective phase of the gate relies on the phase difference between the two tones. This means phase differences `phase0=0, phase1=90` and `phase0=−90, phase1=0` are essentially equivalent[3].

To account for all of these situations, the `framerot` parameters must be *forwarded* to the desired tones, and optionally inverted. This forwarding and inversion can be handled on a per-tone basis, however the frame rotation information is independently tracked on a per-channel basis.

Both the `framerot0` and `framerot1` inputs can be optionally forwarded to one or both tones. Thus their treatment is slightly different than the other parameters such as `freq0` and `freq1`, which control tones 0 and 1 respectively. `framerot1` can be used to track a second

---

[2] Currently, QSCOUT is only using counter-propagating gates because of experimental requirements. However, co-propagating gates are mentioned here in terms of how one might implement them and some of the design choices that were made in terms of co-propagating gates.

[3] The mean phase differs between the two cases, but the beat note at 12.642 GHz formed by the Raman beams will be equivalent between the two cases.

frame, such as one associated with a motional mode. However, the hardware is currently limited to tracking two frames per channel, and doesn't impose any requirements about how they are implemented. Rather, the convention used in QSCOUT is that the frame of the qubit spin is tracked in `frame0`.

Frame rotations obey the same input requirements as the `phase` inputs listed in table I. However, because the `framerot` inputs *accumulate* phase, this means that the inputs need to be treated a little bit differently. Each application of a `PulseData` object with a constant `framerot` will apply that phase with the beginning of the pulse by default. Discrete modulations also obey the same behavior, consider the following definitions of `gate_G` which have the same behavior.

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


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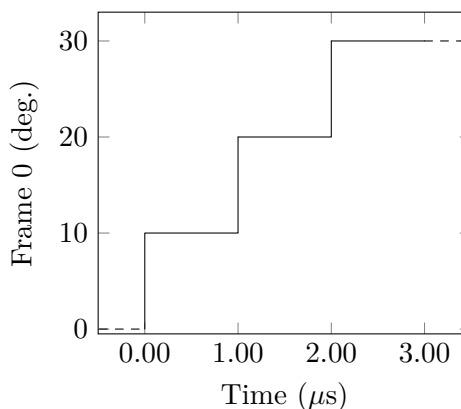

EX. 10: Frame rotation inputs are equivalent to phase, but their values accumulate.

Example 10 only shows the value of the internal frame accumulator. Forwarding the frame accumulator phase to specific tones requires setting metadata inputs in `PulseData`. The structure of the metadata inputs is discussed in III C and outlined in Table III. In Example 11, frame 0 is incremented by 15 degrees in each call to `PulseData`, but the the metadata bits in `fwd_frame0_mask` and `inv_frame0_mask` are used to control how the accumulated phase in frame 0 can be inverted and forwarded to the two tones.

```
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)]
```
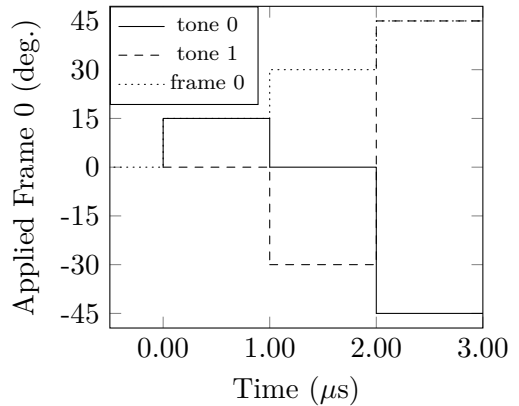

EX. 11: Frames can be optionally applied by forwarding them to one or both tones. Likewise, their sign can be inverted on a per-tone basis.

Note that both frame 0 and frame 1 can be forwarded during the same pulse. However, frame 0 will take precedence if both frames are forwarded to the same tone.

There are a couple more metadata options for frame rotations. Mask settings for these bits are frame specific (frame0 and frame1) and not tone specific (tone0 and tone1). The first option is apply_at_end_mask, which will update the frame for the *next* pulse. This is useful in situations such as accounting for AC Stark effects, where adding a small phase shift in the rotating frame of the qubit after the pulse ends is necessary for global phase synchronization. The second option is rst_frame_mask, which will clear the accumulated phase in the frame. rst_frame_mask always happens at the beginning of a pulse, so a simultaneous application of rst_frame_mask and apply_at_end_mask will clear the frame for the current pulse, and the frame will take on the frame input value with the next pulse.

```
def gate_G(self, qubit):
    return [PulseData(qubit, 1e−6,
                    framerot0=10,
                    apply_at_end_mask=1),
            PulseData(qubit, 1e−6),  # NOP
            PulseData(qubit, 1e−6,
                    framerot0=−5,
                    rst_frame_mask=1)]
```
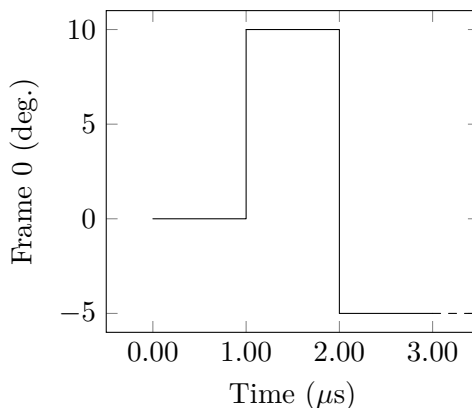
EX. 12: Frame rotations can optionally be applied with the *next* pulse,
and the frames can also be reset.

Frame rotations also accept spline inputs. In this case, the `apply_at_end_mask` is ignored, and the spline will effectively start from the initial value of the accumulator and the final value of the spline will be added to the frame accumulator.

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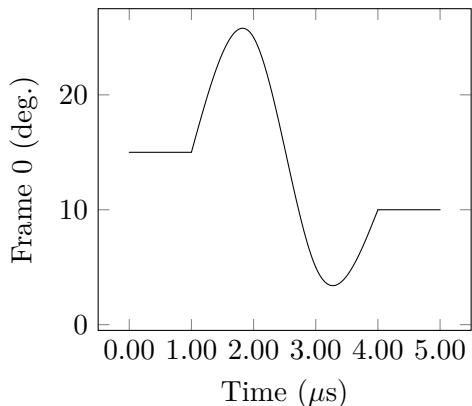
EX. 13: Frame rotations support spline inputs. Only the final value of
the spline is added to the accumulator.

While this might not seem immediately useful, accounting for AC Stark shifts via splines offers certain advantages. For example, the frequency offset associated with an AC Stark shift as a function of pulse amplitude can be represented as the integral of the pulse shape and normalized to the total phase offset from the resulting Stark shift. The benefits in this case are twofold:

- Time-dependent frequency variations are automatically accounted for via the time-dependent phase, allowing one to more closely match resonance conditions during the

13

entire pulse.

- Using frequencies in the natural (unshifted) frame of the qubit greatly simplifies the use of the global synchronization capabilities of the Octet.

C. Metadata

Each `PulseData` object contains a number of entries designed to control certain operations or state associated with the pulse. The list of metadata parameters is shown in Table II. All metadata parameters are listed for the sake of completeness, but users should beware of using certain inputs. Parameters that should generally be avoided are `enable_mask` and `waittrig`. `enable_mask` will change the default output state to off, resulting in a zero output. `waittrig` is used for handshaking with other control hardware to ensure that sequences run after other experimental processes have reached the correct state.

Parameter	Type	Default	Description
sync_mask	int	0b00	Synchronize phase for current frequency
enable_mask	int	0b00	Toggle the output enable state
fb_enable_mask	int	0b00	Enable frequency correction
fwd_frame0_mask	int	0b00	Forward frame 0
fwd_frame1_mask	int	0b00	Forward frame 1
inv_frame0_mask	int	0b00	Invert frame 0 sign
inv_frame1_mask	int	0b00	Invert frame 1 sign
apply_at_end_mask	int	0b00	Apply frame rotation at end of pulse
rst_frame_mask	int	0b00	Reset accumulated frame rotation
waittrig	bool	False	Wait for external trigger

TABLE II: Metadata inputs for `PulseData` objects. All inputs ending in `_mask` range from 0-3 and follow a bitwise convention shown in table III. The format `0b11` is Python notation for a base 2 integer, which in this case corresponds to a decimal value of 3.

Metadata inputs ending with `_mask` take in tone-specific boolean values in the form of an integer. This integer input is meant to be interpreted as a binary bit mask that is `True` if the bit is 1, and `False` if the bit is 0. The least significant bit (LSB) corresponds to tone 0, and the most significant bit (MSB) corresponds to tone 1.

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

TABLE III: Convention used for bitmask enables. The least significant bit (LSB) controls tone 0, the most significant bit (MSB) controls tone 1. The same convention is used for `rst_frame_mask` and `apply_at_end_mask`, however the bits apply to frame 0 and frame 1 for the LSB and MSB respectively.

Most of the metadata is specific to frame rotations, covered in section III B. This leaves two of the most critical metadata entries, `sync_mask` and `fb_enable_mask`. Improper application of these inputs, or lack thereof, can lead to unexpected behavior. As such, these two features have been given dedicated sections.

IV.   FREQUENCY FEEDBACK

The frequency stabilization techniques used are detailed in the QSCOUT manual. A quick summary of this aspect of the system will be provided for completeness. The Raman transitions used for gates in QSCOUT are driven by a pulsed laser. This laser generates a frequency comb in which the comb teeth are separated by $\approx$ 120 MHz. In order to drive the qubit transition, one can make use of a harmonic of the comb that approximately bridges the $\approx$12.642 GHz transition frequency. For 120 MHz, this is roughly the 105th harmonic, where 105×120 MHz $= 12.6$ GHz. This leaves a ≈ 42 MHz offset from resonance, which can easily be accounted for with an acousto-optic modulator (AOM).

The frequencies of the two tones applied for a gate must bridge this leftover 42 MHz frequency difference[4]. However, the pulsed laser used does not have active stabilization of the cavity length. Thermal variation will cause the cavity length to drift, affecting the pulse repetition rate and thus the spacing of the frequency comb. This "breathing" of the frequency comb requires that we actively correct for the resulting frequency offset between

---

[4] In the actual QSCOUT platform, the numbers are slightly different and the resulting offset is closer to 28 MHz, but should be referenced from calibration data.

the two comb teeth needed to achieve resonance. The variation in this repetition rate is constantly tracked, and we translate this variation into the correct frequency offset which is then forwarded to an output tone.

This frequency feedforward scheme is robust against the level of drift we normally encounter, and we've been able to yield coherence times exceeding 10 s [1]. However, it only works if we offset a *single* tone in the Raman transition. In principle, we could add equal and opposite corrections to each tone, with half the total correction being applied. But the sign must differ between the two cases and poses additional challenges based on the configuration of the tones. This correction needs to be properly scaled to account for the total frequency variation at the particular harmonic to which we lock. While we can apply the correction to either tone, this results in a sign change to the scaling factor.

The settings for the harmonic scaling and their sign are currently not reconfigurable between gates. Rather, we've chosen a particular sign convention where the tone which is higher in frequency—in the case of the example numbers given above, this would be 42 MHz higher—should have `fb_enable_mask` set high. The following code demonstrates how co-propagating and counter-propagating gates might implement the feedback enable.

```
def gate_G_coprop(self, qubit):
    return [PulseData(qubit,        1e−6, freq0=200e6,
                                    freq1=242e6, fb_enable_mask=0b10)]

def gate_G_counterprop(self, qubit):
    return [PulseData(qubit,        1e−6, freq0=200e6, fb_enable_mask=0b00),
            PulseData(GLOBAL_BEAM, 1e−6, freq0=242e6, fb_enable_mask=0b01)]
```

V. GLOBAL PHASE SYNCHRONIZATION

One of the most critical aspects of implementing a proper gate is ensuring that the phase of the gate is set correctly with respect to the qubit(s). In many cases, the frequencies associated with a gate are subject to change. For example, a two-qubit gate needs to address motional sidebands of the qubit transition, and certain gates may implement frequency modulation. However, each hardware channel is equipped with only two custom direct digital synthesizer (DDS) modules for the two output tones. Phase is continuous at the boundary where a frequency update occurs, as shown in Fig. 2. The implication is that the
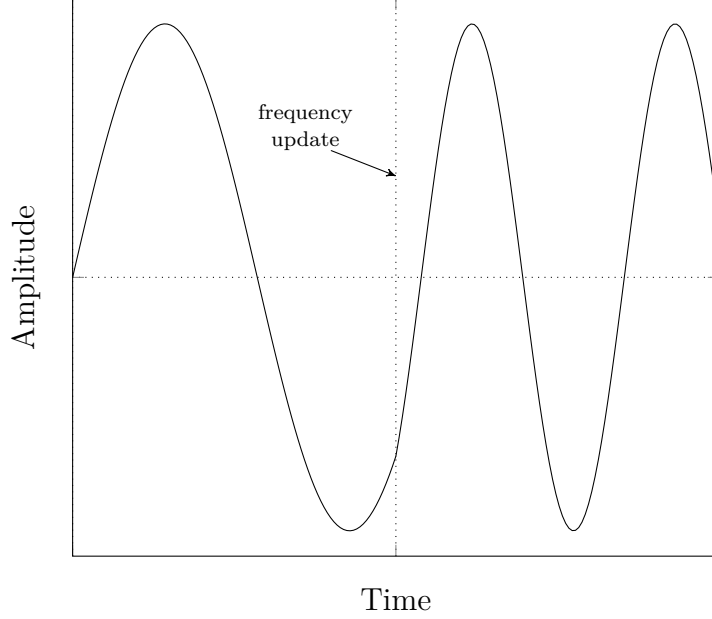
16

FIG. 2: Waveform output after a change in frequency. The phase is
continuous at the boundary where the update occurs.

phase is generally arbitrary for the new frequency.

The absolute phase of the rf output is irrelevant from the perspective of a qubit initialized
to an energy eigenstate. However, the first gate applied to an initialized qubit sets the phase
from which all subsequent gates need to be referenced during a circuit. Thus a free-running
synthesizer at a fixed frequency on resonance with the qubit transition can easily perform
single-qubit gates about different axes by adjusting their phase relative to some initial value.
If the frequency is temporarily changed in order to perform another operation, then returning
to the original frequency will typically result in a phase offset from the original waveform as
shown in Fig. 3.

The resulting phase depends on the value of the other frequency and the duration for
which it was applied. While the phase offset can be easily calculated to determine a cor-
rection for returning to the original phase at the original frequency, this requires a lot of
phase bookkeeping and is strongly context dependent. The Octet hardware handles phase
bookkeeping automatically via a modification to the standard DDS design.

A global counter shared by all output tones is constantly being multiplied by the DDS
modules input frequency word. This produces a global phase, $\Phi = \omega t$, where ω is the input
frequency, and the time, t, is the value of the the global counter. The frequency input to
the DDS is delayed to match the latency of the multiplication stage such that the global
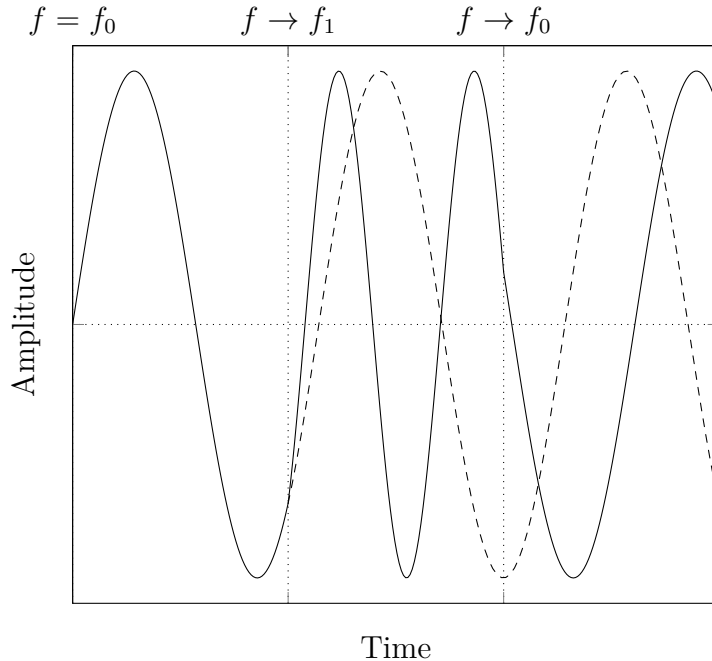
17

FIG. 3: Waveform output after changing frequency and subsequently returning to the original frequency at a later time. The final phase is not aligned to the that of the original waveform.

phase tracks with the input frequency. Setting `sync_mask` high for a particular tone in a `PulseData` object will send a single trigger at the beginning of the pulse. This trigger will overwrite the DDS phase accumulator with the global phase, Φ. The resulting behavior is equivalent to the second frequency update in Fig. 4.

It is worth noting that this mechanism functions the same, regardless of the other settings. In other words, synchronization is not affected by other metadata such as `fb_enable_mask` or settings on the other tone. Synchronization only depends on the frequency at the start of the pulse[5].

A. Synchronization Caveats

Gates that require higher-order frequency and phase relationships might require some extra consideration. For example, the Mølmer-Sørensen gate requires that the red and blue

---

[5] For frequency modulation, the synchronization step is applied with the beginning of the first element in the modulation list or tuple.
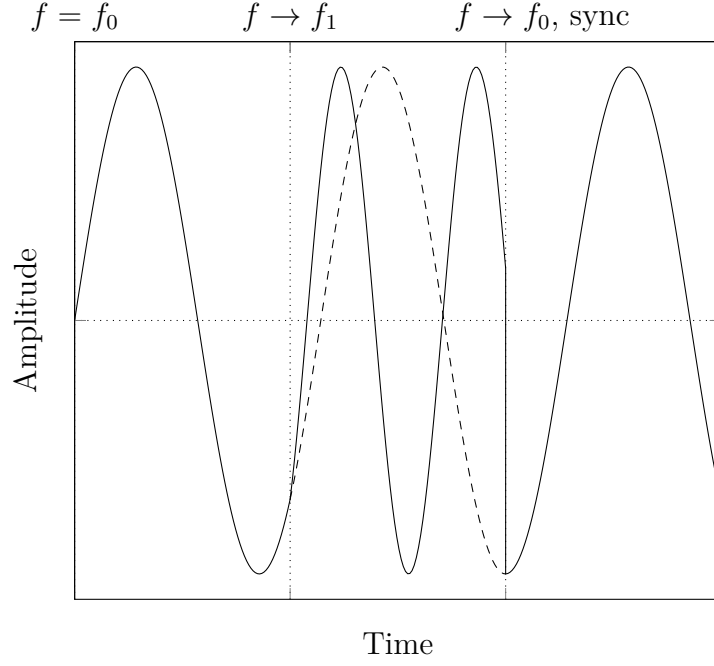
FIG. 4: Waveform output after a change in frequency, followed by a return to the original frequency with a synchronization call applied. In this case, the waveform experiences a discontinuous phase jump when returning to the original frequency *and* phase.

sideband frequencies obey the relationship

$$f_r + f_b = 2f_{qubit} \tag{1}$$

where

$$f_r \equiv f_{qubit} - f_{\text{SB}} \tag{2}$$

$$f_b \equiv f_{qubit} + f_{\text{SB}}. \tag{3}$$

$$\tag{4}$$

This condition maps equivalently onto phase such that

$$\phi_r + \phi_b = 2\phi_{qubit}. \tag{5}$$

19

Because frequencies are digitized before being sent to hardware, rounding effects can result in a small difference in the final frequencies such that

$$F_r + F_b \neq 2F_{qubit}, \tag{6}$$

where F indicates the frequency word in the digital domain. This phase relationship is difficult to predict from the vantage point of the compiler. It is up to the user to ensure that the desired frequency relationships match in the digital domain to ensure proper phase relationships between tones.

These rounding errors typically contribute a 0 or 1 bit difference from the desired result, potentially adding a frequency offset of $\approx 745\mu$Hz. While this may seem negligible, it's important to understand that the global counter is generally set to zero when the hardware is powered on. In some cases, the counter may be accumulating for weeks or months. The global phase error between F_{qubit} and $F_{qubit} + F_{eps}$ is given by $\delta\phi = F_{eps}t_{global}$, which can be quite substantial when $t_{global} >> 0$.

Helper functions are provided to simplify the calculations. For synchronization purposes, `discretize_frequency` can be imported from `jaqalpaw.utilities.helper_functions`. Similar functions `discretize_amplitude` and `discretize_phase` are also provided and can be imported from the same path. The usage for `discretize_frequency` is shown in Ex. 14 and involves converting frequencies to the digital domain before calculating their frequency relationships.

In this case, the resonant qubit frequency is discretized, as well as the desired sideband frequency, prior to the red and blue sideband frequency calculations. The frequency inputs in the `PulseData` call take `rsb_freq` and `bsb_freq`, which have been calculated using discretized frequencies. Note that the discretization effects need only be taken into account for the relevant leg of the Raman transition, which in this case is given by *self*.`aom_center_frequency` + *self*.`effective_qubit_splitting`.

```
from jaqalpaw.utilities.helper_functions import discretize_frequency

class GatePulse:

    aom_center_frequency: float = 200e6
    effective_qubit_splitting: float = 28.123e6
    motional_mode_frequencies: float = [2.5e6,
                                        2.42e6,
                                        2.38e6,
                                        ...]
    amplitude_scaling: float = [.97,
                                .96,
                                .95,
                                ...]

    def gate_G(self, qubit1, qubit2):
        global_aom_frequency = (self.aom_center_frequency +
                                self.effective_qubit_splitting)
        individual_aom_frequency = self.aom_center_frequency
        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

        return [PulseData(GLOBAL_BEAM, 100e-6,
                          freq0=rsb_freq,
                          freq1=bsb_freq,
                          fb_enable_mask=0b11,
                          sync_mask=0b11),
                PulseData(qubit1, 100e-6,
                          freq0=individual_aom_frequency
                          sync_mask=0b01),
                PulseData(qubit1, 100e-6,
                          freq0=individual_aom_frequency
                          sync_mask=0b01),
                ]
```

EX. 14: A basic gate pulse class.

[1] S. M. Clark, D. Lobser, M. Revelle, C. G. Yale, D. Bossert, A. D. Burch, M. N. Chow, C. W. Hogle, M. Ivory, J. Pehr, B. Salzbrenner, D. Stick, W. Sweatt, J. M. Wilson, E. Winrow, and

P. Maunz, Engineering the Quantum Scientific Computing Open User Testbed (QSCOUT): Design details and user guide (2021), in preparation.

[2] A. J. Landahl, D. S. Lobser, B. C. A. Morrison, K. M. Rudinger, A. E. Russo, J. W. Van Der Wall, and P. Maunz, Jaqal, the Quantum Assembly Language for QSCOUT (2020), arXiv:2003.09382 [quant-ph].

[3] B. C. A. Morrison, A. J. Landahl, D. S. Lobser, K. M. Rudinger, A. E. Russo, J. W. Van Der Wall, and P. Maunz, Just another quantum assembly language (Jaqal) (2020), arXiv:2008.08042 [quant-ph].

[4] Python data classes, `https://docs.python.org/3/library/dataclasses.html`, [Online; accessed 6-April-2021].