JaqalPaw Tutorial : Modulated Mølmer-Sørensen Gates

Matthew N. H. Chow* Daniel Lobser, Ashlyn Burch, Joshua Goldberg, Andrew J. Landahl, Benjamin C. A. Morrison, Kenneth Rudinger, Antonio Russo, Brandon Ruzic, Jay Van Der Wall, Chris Yale, Susan M. Clark

September 2021

1 Introduction

In this example of the Jaqal Pulses and Waveforms (JaqalPaw) code, we provide code that we use to perform modulated Mølmer-Sørensen gates on trapped-ion qubits. The code referenced in this exemplar is located at <**qscout.sandia.gov**>, or can be found at the end of this document. We modulate the frequency, amplitude, and phase of the control light in order to produce pulses found by our optimization script that are designed to have robustness against certain experimental errors [1]. Given that this example contains a framework for essentially arbitrary pulse generation for one or two ion gates, it is our hope that users can simply copy this exemplar code and make small modifications in order to accelerate development of their own pulses.

2 Jaqal Code

In this exemplar, the sole responsibility of the Jaqal code (`Exemplar_ModulatedMS.jaqal`) is to call the JaqalPaw code in a way that allows it to run on the apparatus. To perform this role, it does three basic things.

1. Import JaqalPaw code.
2. Declare parameters that are passed to the gate definition.
3. Call the JaqalPaw gate (`Mod_MS`) within a "prepare_all" - "measure_all" block.

All of these things are covered in the Jaqal specification, and details on syntax can be found at: <**www.sandia.gov/quantum/Projects/QSCOUT_Jaqal.html**>. The main focus of this document is on the JaqalPaw code, although a copy of the Jaqal code is included in the Appendix for completeness.

Quick Note on Import The import statement is given below:

```
from Exemplar_ModulatedMS_PulseDefinitions.ModulatedMSExemplar usepulses *
```

This is a relative import, where `Exemplar_ModulatedMS_PulseDefinitions.py` is the JaqalPaw code, and is located in the same folder as the Jaqal file calling it. `ModulatedMSExemplar` is the name of the class that contains the pulse definitions that are the subject of this exemplar. The `usepulses *` syntax tells Jaqal to import all of the gates that are in that class. Note that, currently, only one `usepulses` statement is allowed and only one JaqalPaw class may be imported in that statement, thus all required gates should be included in that one class by some means. (Typically, we use inheritance in get QSCOUTBuiltins included in this class. More on that later.)

3 JaqalPaw Code

The optimization script that we use for finding these gates outputs a waveform defined by the Raman Rabi rate knots, detuning knots relative to the carrier transition, and phase steps. Knots are used to define cubic splines, whereas steps are taken with discrete jumps. The base functionality of the gate definition that follows is to take these optimizer output parameters and calculate the appropriate RF pulse parameters to apply to each channel.[1] The gate definition returns a list of PulseData objects, built from these RF pulse parameters, that is sent to the RFSoC.

*mnchow@sandia.gov

[1]Since our qubit control is done optically, we are calculating the RF pulse that will be applied to the accousto optic modulators, (AOMs) which in turn convert RF to optical signal and apply light to the ions.

The gate definition is written as a method of the class to be imported in Jaqal (in this case `ModulatedMSExemplar`) that returns a list of `PulseData` objects. The name of the method must begin with `gate_` in order to be accessible within Jaqal after the `usepulses` import. To call the gate from Jaqal, use the rest of the method name after `gate_`. In this case, the method definition statement is:

```
def gate_Mod_MS(self, channel1, channel2, singleion=False, global_duration=-1e6):
    """ General Modulated MS Gate (Produce optimal pulses found by solver).
```

This gate is called in Jaqal by writing `Mod_MS` followed by inputs for the two positional arguments (`channel1` and `channel2`) other than `self` and keyword arguments (`singleion` and `global_duration`). All together, the line of Jaqal that calls this gate is:

```
Mod_MS q[target1] q[target2] singleion global_duration
```

3.1 Referencing Calibration Parameters

In order to calculate the correct AOM drive pulse, there must be some reference to of physical parameters of the system. This is done through reference to the `CalibrationParameters` of `QSCOUTBuiltins`. In order to gain access to these parameters, we inherit the class `QSCOUTBuiltins` in our new class `ModulatedMSExemplar`. Similar to any python class inheritance, this is done in the declaration of the class:

```
class ModulatedMSExemplar(QSCOUTBuiltins):
    # This class inherits QSCOUTBuiltins
```

The calibration parameters are described in the QSCOUTBuiltins specification. For this example, the callibrated parameters that are necessary are the AOM center frequency (`ia_center_frequency`), Raman carrier transition splitting (`adjusted_carrier_splitting`), resonant pi time (`counter_resonant_pi_time`), and the Mølmer-Sørensen amplitudes to give balanced blue-red sideband driving (`MS_blue_amp_list`, `MS_red_amp_list`). After inheriting `QSCOUTBuiltins`, these parameters become fields of the `ModulatedMSExemplar` class and are referenced with `self.parameter_name`.

The first two lines of code in this gate are an example of how to use these `CalibrationParameters`.

```
# Use calibrated, matched pi time for baseline Rabi rate.
rabi_rate_0 = 0.5/self.counter_resonant_pi_time
# Use global beam as the lower leg of the Raman transition.
global_beam_frequency = discretize_frequency(self.ia_center_frequency) -
                        discretize_frequency(self.adjusted_carrier_splitting)
```

The first line uses the given pi time to calculate the nominal Rabi rate. This will be used later for scaling of the amplitude knots. The second line calculates the frequency that will be applied to the global beam AOM. Frequency modulation in this gate is done on the individual addressing (IA) beams, and thus for best performance, the modulation should be centered on the center frequency of the IA AOMs. That means that the global beam frequency should be set at the center frequency of the IA AOMs minus the Raman carrier splitting. Notice that for any math with frequencies is done with the `discretize_frequency` function, which is imported from from `jaqalpaw.utilities.helper_functions`. This allows us to avoid discretization errors and any associated phase errors that could occur as a result. See the "JaqalPaw: A Guide to Defining Pulses and Waveforms for Jaqal" for more information. [2]

3.2 Calculation of Amplitude Knots from Desired Rabi Rate Knots

The input of the Rabi rate knots is an iterable of knots, `rabi_knots`, in Hz (real frequency) and an overall scaling factor, `rabi_fac`. In this example, it is simply a Gaussian pulse shape with a height equal to the nominal Rabi rate for the counterpropagating configuration.

```
# rabi_fac is used as an overall Rabi rate scaling factor.
rabi_fac = 1
# rabi_knots (in Hz) are desired rabi rate knots.
# This example is 13 points along a Gaussian with a height of the nominal Rabi rate.
rabi_knots = self.gauss(npoints=13, A=rabi_rate_0)
```

Note, `self.gauss` is a helper function (described in more detail in section 4.1) that outputs a numpy array of points along a Gaussian. From these values, we convert to an amplitude scale factor by dividing by the nominal Rabi rate. We cast the iterable to a numpy array so that it can be easily multiplied by a scalar.

```
# Convert Rabi rate knots to an amplitude scale.
if len(rabi_knots) > 1:
    amp_scale = [rabi_fac*rk/rabi_rate_0 for rk in rabi_knots]
else:
    #  Default to a square pulse if no input.
    amp_scale = [1, 1]
amp_scale = np.array(amp_scale)
```

Notice in this block we can also use conditionals to set a default of constant amplitude if the input is an empty iterable. This structure could similarly be used to do validity checking on inputs.

After calculation of the amplitude scaling array, we multiply each tone amplitude by scalar for achieving proper blue-red sideband power balace Mølmer-Sørensen (for details, see the `QSCOUTBuiltins` spec), cast to a tuple, and pass the input to `amp0` and `amp1` of the `PulseData` objects for the individual beams.

```
PulseData(channel1, self.MS_pulse_duration,
        ...
        amp0=tuple(self.MS_blue_amp_list[target_idx][amp_index0]*amp_scale),
        amp1=tuple(self.MS_red_amp_list[target_idx][amp_index0]*amp_scale),
        ...),
PulseData(channel2, self.MS_pulse_duration,
        ...
        amp0=tuple(self.MS_blue_amp_list[target_idx][amp_index0]*amp_scale),
        amp1=tuple(self.MS_red_amp_list[target_idx][amp_index0]*amp_scale),
        ...)
```

3.3 Calculation of Frequency Knots from Detuning Knots

The calculation of the frequency knots similarly combines user inputs with calibrated parameters, with an additional step that ensures proper discretization. Since the phase of the red and blue sidebands must sum to twice the carrier, and global phase accumulation in the Octet hardware is tracked with a global counter that is freerunning since the last reset (typically power on time) [2], care must be taken to ensure that the discretized frequency arithmetic is consistent at the least significant bit level. Again, we use the utility function, `discretize_frequency`, to handle it.

The output of the optimizer script is a list of detunings from the carrier Raman transition in Hz, and is entered into the pulse definition as a list of frequency knots. In the example code, we implement a frequency sweep from 100 kHz to 10 kHz and back below the lowest frequency motional mode. [2]

```
# detuning_knots are desired frequency relative to the carrier.
# This simple example is a sweep from 100kHz to 10kHz below the lowest frequency mode.
detuning_knots = [
    self.lower_motional_mode_frequencies[-1]-100e3,
    self.lower_motional_mode_frequencies[-1]-50e3,
    self.lower_motional_mode_frequencies[-1]-10e3,
    self.lower_motional_mode_frequencies[-1]-50e3,
    self.lower_motional_mode_frequencies[-1]-100e3,
    ]
```

[2] This is not an actual optimized output, which would be a longer list of what appear to be magic numbers in the absence of relevant physical context. For example, these were the frequency knots found by the solver on 7/14/2021: [1539666.559627668, 1881455.6803017957, 1877738.7322624624, 1824009.3099547108, 1916441.6482429288, 1899198.92720476, 1848488.216400187, 1818200.1760113079, 1832316.1455718977, 1811761.0404437222, 1854983.75650778, 1882605.5825217813, 1848045.5300807087, 1882605.5825217813, 1854983.75650778, 1811761.0404437222, 1832316.1455718977, 1818200.1760113079, 1848488.216400187, 1899198.92720476, 1916441.6482429288, 1824009.3099547108, 1877738.7322624624, 1881455.6803017957, 1539666.559627668]

From there, the detuning knots are converted to RF drive frequencies by subtracting from the IA AOM center frequency. This centers the modulation on the center frequency of the IA AOM, and as mentioned earlier, the global AOM is set to the proper frequency to make a resonant Raman transition with the center frequency of the IA AOM. In addition, we include a symmetric detuning parameter, `MS_delta` that allows for scanning and calibration of the symmetric detuning. Typically, this value is close to zero for a frequency modulated gate, but may be set to a small offset on the order of a few kHz to account for trap frequency drifts. Subtraction of a list of values from a scaler is done by list comprehension; then the new list is cast to a tuple to so that the RFSoC will generate a cubic spline interpolation when it is passed to PulseData.

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

Finally, `freq_fm0` and `freq_fm1` are passed to `freq0` and `freq1` inputs of the PulseData calls for the IA beams.

```
PulseData(channel1, self.MS_pulse_duration,
        freq0=freq_fm0,
        freq1=freq_fm1,
        ...),
PulseData(channel2, self.MS_pulse_duration,
        freq0=freq_fm0,
        freq1=freq_fm1,
        ...)
```

3.4 Everything Else in the PulseData Objects

The rest of the inputs to the PulseData calls handle phase, frame rotation, synchronization, and feedback.

```
listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
                          ...
                          phase0=phase_steps,
                          phase1=0,
                          sync_mask=0b11,
                          fb_enable_mask=1),
                PulseData(channel1, self.MS_pulse_duration,
                          ...
                          framerot0 = framerot_input,
                          apply_at_eof_mask=framerot_app,
                          phase0=0,
                          phase1=0,
                          sync_mask=0b11,
                          fb_enable_mask=0
                          )]
```

PulseData for `channel2` (the other IA beam) has the same settings as PulseData for `channel1`. The phase data is much like the frequency and amplitude inputs, but more straightforward since there is no real calculation necessary. A list of `phase_steps` in degrees gives the discrete phase steps, and is directly applied to the global beam PulseData object. The IA beams both get zero phase on both tones. As for frame rotation, in this example we just use `frame0` to account for AC Stark shifts. Note that `tone0` and `tone1` are distinct from `frame0` and `frame1`, which are are virtual z rotation accumulators that can be forwarded to either or both tones of a channel. See JaqalPaw guide for more detail [2]. To do the AC Stark compensation, we typically apply a constant rotation at the end of the frame [3], but in the special case of a Gaussian pulse shape, we accumulate phase with an erf function over the course of the gate.

[3] Another exemplar with more detail on AC Stark shifts in progress.

```
        # ERF Stark shift correction for Gaussian pulses. Constant otherwise.
        if is_gaussian:
            framerot_input = tuple(self.erf(len(amp_scale),
                                    self.MS_framerot * amp_scale,
                                    freqwidth=300e3,
                                    total_duration=4e-6))
            framerot_app = 0
        else:
            framerot_input = self.MS_framerot
            framerot_app = 0b11
```

As for synchronization, all tones are synchronized on all channels to give a well defined phase relation to other gates, as is standard practice on QSCOUT. The first frequency knot is used for synchronization on the modulated tones. Finally, the repetition rate feed forward stabilization is done on the lower leg of the Raman transition, and is thus applied to `tone0` of the global beam. QSCOUT does not currently support user control over other parameters of the frequency stabilization, so this convention of stabilizing on the lower leg of the Raman transition should be followed in user code at this time.

4 Useful Techniques

JaqalPaw is written in Python, and thus is quite flexible to user's needs. This example code demonstrates a few helpful features that can be implemented to hopefully inspire user code development.

4.1 Helper Functions

One of the easiest ways to add flexibility to JaqalPaw code is through the use of user-defined helper functions. This can be done in any way that one would typically add access to Python functions in a class, such as:

- Writing a helper function class and inheriting that class in the main gate definitions class.
- Writing `@staticmethod` functions as part of the main class.
- Including static function definitions outside of the class.
- Writing a helper function file and importing it. [4]

In this exemplar, we use a combination of the first and last methods. The `HelperFunctions` method written for this example includes methods for producing evenly spaced points along Gaussian and Erf curves, using `numpy` (aliased as `np`) and `scipy` packages respectively. Code for the Gaussian helper function (using `np.exp` function to take the exponential) is given below:

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

To gain access to the methods from this `HelperFunctions` class, the class definition containing the modulated MS pulse definition inherits this class. Since inheritance of `QSCOUTBuiltins` is also necessary, both classes are passed in to the class declaration of `ModulatedMSExemplar`.

```
class ModulatedMSExemplar(QSCOUTBuiltins, HelperFunctions):
    # This class inherits both QSCOUTBuiltins and HelperFunctions
```

[4] Please talk to us if you plan on using non-standard imports, package dependencies may be complicated and are not guaranteed. Local imports of hand-written files are likely to be successful.

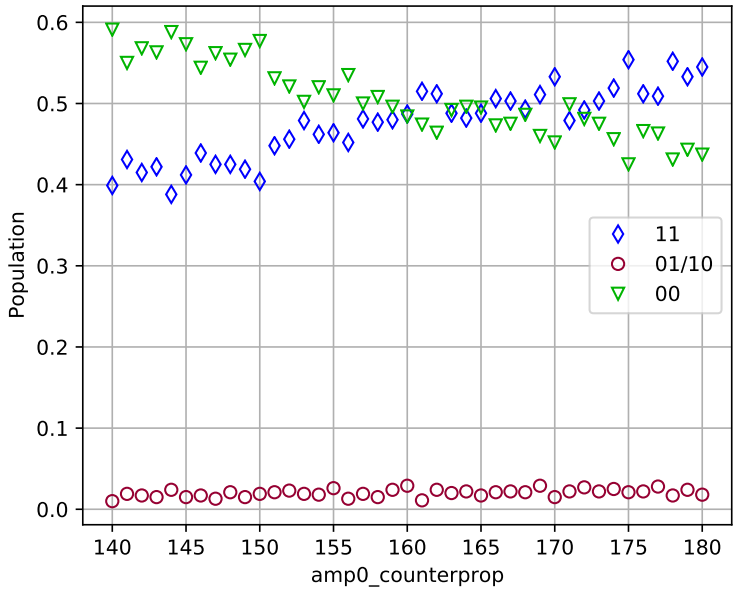

Figure 1: Data for scan of the global beam amplitude for calibration of the Mølmer-Sørensen gate. Ions are prepared in $|00\rangle$ before the gate, so the $\mathrm{XX}(\frac{\pi}{2})$ gate is ideally completed when $|00\rangle$ and $|11\rangle$ are at 0.5, and odd parity states have zero population.

4.2 Using a Parameterized Pulse

Writing pulse definitions based on scannable parameters allows for flexibility and is often useful for calibration and testing. As an example, we show how we parameterize the amplitude of the global beam to allow for calibration of the overall amplitude and spin state tracking over the course of the MS gate. The simplest example of a parameterized pulse is simply allowing a scalar input of a `PulseData` object to be varied. Calibrated parameters given in `QSCOUTBuiltins` are often found in this way. In this example, this simple technique is used for the amplitude of the global beam.

```
        global_amp = self.amp0_counterprop

    listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
                              freq0=global_beam_frequency,
                              freq1=self.global_center_frequency,
                              amp0=global_amp,
```

When we wish to calibrate this amplitude, we scan `self.amp0_counterprop`. Example data from this calibration is given in Fig. 4.2. The ions were prepared in $|00\rangle$, so the gate has completed an $\mathrm{XX}(\frac{\pi}{2})$ rotation when the populations of $|00\rangle$ and $|11\rangle$ both reach 0.5. The calibrated value for this gate is set to 160 to get equal populations of $|00\rangle$ and $|11\rangle$.

Additionally, gate parameters may be passed in as function inputs, or otherwise used to construct more complex `PulseData` inputs. For parameters that are not included in `CalibrationParameters`, an additional let statement must be made in the Jaqal code and that parameter passed in to the input of the gate. In the example code, we implement a simple way to track the spin state over the course of the gate by shuttering the global beam after `global_duration`. Thus we add the parameter `global_duration` in Jaqal and pass it in when we call `Mod_MS`. Then, in the gate definition, if a positive `global_duration` that is less than the total gate duration is given, we calculate a step function that turns off the global beam at that time. Since this turns the Rabi rate of the lower leg of the Raman transition to 0, qubit driving stops. We are then able to scan this input parameter to see how the populations of various spin states evolve over the course of the gate. Note that this example also demonstrates how one might make use of both tones of a Raman transition to create a complex qubit drive. The amplitude modulation on the individual beams is already being modulated with continuous form, so shuttering the individual beams discretely would require a more complex calculation for the `PulseData` input. In a similar fashion

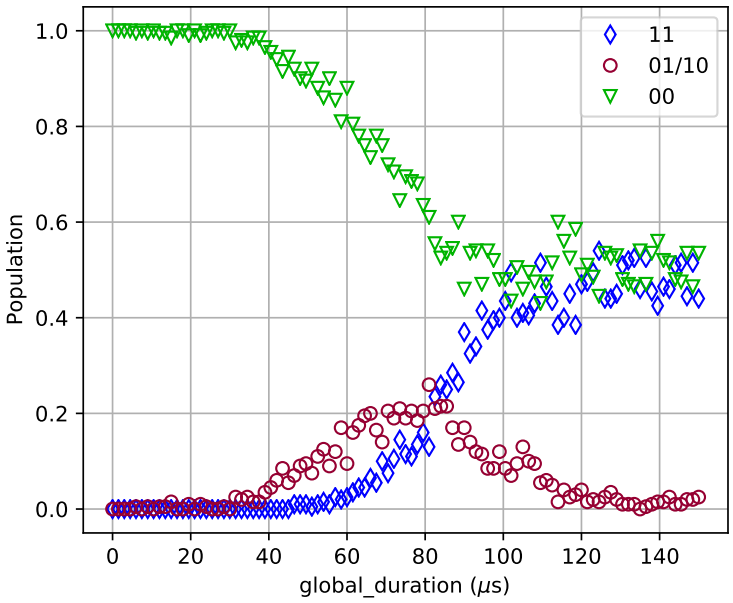

Figure 2: Scan of global duration for a frequency modulated Gaussian MS gate. The population of $|00\rangle$ ($|11\rangle$) steadily decreases (increases) to 0.5 while the population of odd parity states gains non-zero population for intermediate times, but returns to near zero by the end of the gate.

to this pulse timing calculation, pulse parameters may be passed to more complex functions to gain access to more general families of pulses.

4.3 Configure from File

Another useful technique when writing JaqalPaw code is reading in configuration files. This allows a single pulse definition to be used to produce multiple gates. Additionally, it simplifies programmatic production of pulses in that a program designed to produce specific pulses only needs to write the configuration file. For example code, see the `if from_file` block in `gate_Mod_MS` and associated configuration file `Exemplar_ModulatedMS_Config.txt`. In this manner, one can conveniently swap out various pulse designs for pulse shaping studies without changing any JaqalPaw code.

5 Acknowledgements

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research Quantum Testbed Program. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under Contract No. DE-NA0003525. SAND2024-11523O

References

[1] Pak Hong Leung, Kevin A. Landsman, Caroline Figgatt, Norbert M. Linke, Christopher Monroe, and Kenneth R. Brown. Robust 2-qubit gates in a linear ion crystal using a frequency-modulated driving force. *Phys. Rev. Lett.*, 120:020501, Jan 2018.

[2] Daniel Lobser, Joshua Goldberg, Andrew J. Landahl, Peter Maunz, Benjamin C. A. Morrison, Kenneth Rudinger, Antonio Russo, Brandon Ruzic, Daniel Stick, Jay Van Der Wall, and Susan M. Clark. Jaqalpaw: A guide to defining pulses and waveforms for jaqal, June 2021.

Appendix: Copy of the Exemplar Code

5.1 Jaqal Code

```
// Import JaqalPaw Code.
from Exemplar_ModulatedMS_PulseDefinitions.ModulatedMSExemplar usepulses *

// Declare variables.
let target1 2
let target2 3
let singleion 0
let global_duration -1
let ms_loops 1

register q[8]

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

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

measure_all
```

5.2 JaqalPaw Code

```
from jaqalpaw.ir.pulse_data import PulseData
from jaqalpaw.utilities.helper_functions import discretize_frequency
import numpy as np
from scipy.special import erf as _erf
import os
import sys

# Import QSCOUTBuiltins. Note: this is under development, exact syntax subject to change.
from jaqalpaw.utilities.QSCOUTBuiltins import QSCOUTBuiltins, GLOBAL_BEAM

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

    @staticmethod
    def erf(npoints, A, freqwidth=300e3, total_duration=4e-6):
        tdata = np.linspace(-total_duration / 2, total_duration / 2, npoints)
        return (0.5 + _erf(tdata * 2 * np.pi * freqwidth / np.sqrt(2)) / 2) * A

    @staticmethod
    def get_cfg(cfg_file, delimiter=':'):
        """ Method for parsing simple cfguration files.
        Input: cfg_file : string : full path to input file.
        Returns dictionary of key value pairs built from each line.
        File format:
            # Comment lines begin with octothorpe.
            Data lines are <key:value>, or replace : with other delimiter.
                Value must be valid input to eval()."""
        to_return = {}
```

```
        if os.path.exists(cfg_file):
            with open(cfg_file, 'r') as f:
                for line in f:
                    if len(line.strip()) > 0 and line.strip()[0] == "#":
                        # Skip comment lines.
                        continue
                    if line.find("#") < 0:
                        line = line[:line.find("#")] # Allow inline comments.
                    if delimiter in line:
                        line_list = line.split(delimiter)
                        to_return[line_list[0]] = eval(line_list[1].strip())
        return to_return


class ModulatedMSExemplar(QSCOUTBuiltins, HelperFunctions):
    # This class inherits both QSCOUTBuiltins and HelperFunctions

    def gate_Mod_MS(self, channel1, channel2, singleion=False, global_duration=-1e6):
        """ General Modulated MS Gate (Produce optimal pulses found by solver).
        Typically want to read in amplitude, frequency, and phase from cfg file.
        In the manual input version of this example, the global beam does a
        square pulse with constant frequency. The individual beams do a Gaussian
        pulse with symmetric frequency modulated near a red and blue motional
        sidebands. """
        ## Note, tuple input to PulseData indicates a spline. List is instant jumps.

        ## Calculate relevant values from the calibrated parameters.
        # Use calibrated, matched pi time for baseline Rabi rate.
        rabi_rate_0 = 0.5/self.counter_resonant_pi_time
        # Use global beam as the lower leg of the Raman transition.
        global_beam_frequency = discretize_frequency(self.ia_center_frequency)
                                - discretize_frequency(self.adjusted_carrier_splitting)

        # Always have smaller qubit number first.
        amp_index0 = 0
        amp_index1 = 1
        # MS gate pairwise index for parameter lookup.
        target_idx = self.ms_target_idx(channel1,channel2)

        from_file = True
        if from_file:
            ## Use a cfguration file to read in waveform parameters.
            cfg_file = r"Exemplar_ModulatedMS_Config.txt"
            cfg = self.get_cfg(cfg_file)

            # Get Rabi rate knots and scaling from the configuration file.
            rabi_fac = cfg['rabi_fac']
            rabi_knots = cfg['rabi_knots']
            # Get detuning knots relative to the carrier.
            detuning_knots = cfg['detuning_knots']
            # Get phase steps (jumps, not spline knots) and convert to degrees.
            phase_steps = [p*180/np.pi for p in cfg['phase_steps']]
            if len(phase_steps) < 1:
                phase_steps = [0, 0]  # if there are no knots, default to constant phase list.

            # is_gaussian is used in frame rotation decisions
            is_gaussian = cfg['is_gaussian']
```

```
else:
    ## Manually pass in waveform parameters.
    # rabi_fac is used as an overall Rabi rate scaling factor.
    rabi_fac = 1
    # rabi_knots (in Hz) are desired rabi rate knots.
    # This example is 13 points along a Gaussian with a height of the nominal Rabi rate
    rabi_knots = self.gauss(npoints=13, A=rabi_rate_0)
    # detuning_knots are desired frequency relative to the carrier.
    # This example is a sweep from 100kHz to 10kHz below the lowest frequency mode.
    detuning_knots = [
        self.lower_motional_mode_frequencies[-1]-100e3,
        self.lower_motional_mode_frequencies[-1]-50e3,
        self.lower_motional_mode_frequencies[-1]-10e3,
        self.lower_motional_mode_frequencies[-1]-50e3,
        self.lower_motional_mode_frequencies[-1]-100e3,
        ]
    # phase_steps are the phase steps applied to the global beam.
    # This example has constant phase.
    phase_steps = [0, 0]

    # is_gaussian is used in frame rotation decisions
    is_gaussian = True

# Convert Rabi rate knots to an amplitude scale. Default to a square pulse if no input.
if len(rabi_knots) > 1:
    amp_scale = [rabi_fac*rk/rabi_rate_0 for rk in rabi_knots]
else:
    amp_scale = [1, 1]
amp_scale = np.array(amp_scale)

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

# ERF Stark shift correction for Gaussian pulses. Constant otherwise.
if is_gaussian:
    framerot_input = tuple(self.erf(len(amp_scale),
                            self.MS_framerot * amp_scale,
                            freqwidth=300e3,
                            total_duration=4e-6))
    framerot_app = 0
else:
    framerot_input = self.MS_framerot
    framerot_app = 0b11

# If we are scanning global_duration parameter, generate a list of
#   amplitudes to shut off the global beam after global_duration.
if global_duration >= 0 and global_duration < self.MS_pulse_duration:
    global_amp = [self.amp0_counterprop if t <= global_duration
                  else 0 for t in np.linspace(0, self.MS_pulse_duration, 1000)]
else:
    global_amp = self.amp0_counterprop

listtoReturn = [PulseData(GLOBAL_BEAM, self.MS_pulse_duration,
```

```
                        freq0=global_beam_frequency,
                        freq1=self.global_center_frequency,
                        amp0=global_amp,
                        amp1=0,
                        phase0=phase_steps,
                        phase1=0,
                        sync_mask=0b11,
                        fb_enable_mask=1),
                    PulseData(channel1, self.MS_pulse_duration,
                        freq0=tuple(freq_fm0),
                        freq1=tuple(freq_fm1),
                        amp0=tuple(self.MS_blue_amp_list[target_idx][amp_index0]*amp_scale),
                        amp1=tuple(self.MS_red_amp_list[target_idx][amp_index0]*amp_scale),
                        framerot0 = framerot_input,
                        apply_at_eof_mask=framerot_app,
                        phase0=0,
                        phase1=0,
                        sync_mask=0b11,
                        fb_enable_mask=0
                        )]
    if not singleion:
        listtoReturn.append(PulseData(channel2, self.MS_pulse_duration,
                        freq0=tuple(freq_fm0),
                        freq1=tuple(freq_fm1),
                        amp0=tuple(self.MS_blue_amp_list[target_idx][amp_index1]*amp_scale),
                        amp1=tuple(self.MS_red_amp_list[target_idx][amp_index1]*amp_scale),
                        phase0=0,
                        phase1=0,
                        framerot0 = framerot_input,
                        apply_at_eof_mask=framerot_app,
                        sync_mask=0b11,
                        fb_enable_mask=0
                        ))
    return listtoReturn
```