Untitled-background-results-background-results

November 2, 2020

An Exception was encountered at 'In [6]'.

[17]:
```
#The model is specified in terms of the following parameters:
#Each Duffing oscillator is specified by a frequency  , anharmonicity  , and␣
↪drive strength r, which result in the Hamiltonian terms:

#2 a†a+ a†a(a†a−1)+2 r(a+a†)×D(t),
#where D(t) is the signal on the drive channel for the qubit, and a† and a are,␣
↪respectively, the creation and annihilation operators for the qubit. Note␣
↪that the drive strength r sets the scaling of the control term, with D(t)␣
↪assumed to be a complex and unitless number satisfying |D(t)| 1. - A coupling␣
↪between a pair of oscillators (l,k) is specified by the coupling strength J,␣
↪resulting in an exchange coupling term:

#2 J(ala†k+a†lak),
#where the subscript denotes which qubit the operators act on. - Additionally,␣
↪for numerical simulation, it is necessary to specify a cutoff dimension; the␣
↪Duffing oscillator model is infinite dimensional, and computer simulation␣
↪requires restriction of the operators to a finite dimensional subspace.

%matplotlib inline
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
# The pulse simulator
from qiskit.providers.aer import PulseSimulator
from qiskit.tools.jupyter import *
from qiskit.visualization import *
# Loading your IBM Q account(s)
provider = IBMQ.load_account()
```

```
ibmqfactory.load_account:WARNING:2020-11-02 11:06:19,625: Credentials are
already in use. The existing account in the session will be replaced.
```

[18]:
```
import numpy as np
import matplotlib.pyplot as plt
from qiskit.visualization.bloch import Bloch
```

```
from scipy.optimize import curve_fit
from scipy.signal import find_peaks

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.model_selection import train_test_split

import qiskit.pulse as pulse
import qiskit.pulse.pulse_lib as pulse_lib
from qiskit.compiler import assemble
from qiskit.ignis.characterization.calibrations import rabi_schedules,␣
↪RabiFitter
#from qiskit.pulse.commands import SamplePulse
from qiskit.pulse import *
from qiskit.tools.monitor import job_monitor
# function for constructing duffing models
from qiskit.providers.aer.pulse import duffing_system_model
import os
import sys
import io
import requests
import urllib
import pandas as pd

#readin data
url1 = 'http://homepages.cae.wisc.edu/~ece539/data/eeg/nic23a1.txt'
nic23a1 = urllib.request.urlopen(url1)
#s1=requests.get(url1).content
#nic23a1=pd.read_csv(io.StringIO(s1.decode('utf-8')))
#nic23a1 = pd.read_csv(url1, delimiter='\n')
url2 = 'http://homepages.cae.wisc.edu/~ece539/data/eeg/nic23a3.txt'
nic23a3 = urllib.request.urlopen(url2)
#another obervations
url11 = 'http://homepages.cae.wisc.edu/~ece539/data/eeg/nic8a1.txt'
nic8a1 = urllib.request.urlopen(url11)
url21 = 'http://homepages.cae.wisc.edu/~ece539/data/eeg/nic8a3.txt'
nic8a3 = urllib.request.urlopen(url21)
Dims = 29
Labels = 8

tx1 = []
tx2 = []

#print(np.shape(nic23a1.readlines()))
for line1 in nic23a1.readlines():
#    tx1.append(csv.reader(line, delimiter=' '))
#    tx1.append(line.split('\t'))
    tx1.append(line1.split())
```

```
for line2 in nic23a3.readlines():
#    tx1.append(csv.reader(line, delimiter=' '))
#    tx1.append(line.split('\t'))
    tx2.append(line2.split())
rows,cols = np.shape(tx1) #cols = Dims + Labels
datasets = np.array(tx1)[:][range(Dims)]
label1 = np.array(tx1)[:][range(Dims,Dims+Labels)]
label2 = np.array(tx2)[:][range(Dims,Dims+Labels)]
#print(tx1[0][0])
#preprocess
normdata = np.linalg.norm(datasets,axis = 1)
print(np.shape(normdata))
print(np.shape(datasets))
```

```
(29,)
(29, 37)
```

[19]:
```
#We will experimentally find a  -pulse for each qubit using the following␣
 ↪procedure:
#- A fixed pulse shape is set - in this case it will be a Gaussian pulse.
#- A sequence of experiments is run, each consisting of a Gaussian pulse on the␣
 ↪qubit, followed by a measurement, with each experiment in the sequence␣
 ↪having a subsequently larger amplitude for the Gaussian pulse.
#- The measurement data is fit, and the pulse amplitude that completely flips␣
 ↪the qubit is found (i.e. the  -pulse amplitude).
import warnings
warnings.filterwarnings('ignore')
from qiskit.tools.jupyter import *
%matplotlib inline

from qiskit import IBMQ
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q', group='open', project='main')
backend = provider.get_backend('ibmq_armonk')

backend_config = backend.configuration()
assert backend_config.open_pulse, "Backend doesn't support Pulse"

# cutoff dimension
dim_oscillators = 3
# frequencies for transmon drift terms
# Number of oscillators in the model is determined from len(oscillator_freqs)
oscillator_freqs = [5.0e9, 5.2e9] #harmonic term
anharm_freqs = [-0.33e9, -0.33e9] #anharmonic term

# drive strengths
drive_strengths = [0.02e9, 0.02e9]
```

```
# specify coupling as a dictionary (qubits 0 and 1 are coupled with a␣
↪coefficient 0.002e9)
coupling_dict = {(0,1): 0.002e9}

#sample duration for pulse instructions in accordance
dt = backend_config.dt #1e-9

backend_defaults = backend.defaults()

# unit conversion factors -> all backend properties returned in SI (Hz, sec,␣
↪etc)
GHz = 1.0e9 # Gigahertz
MHz = 1.0e6 # Megahertz
us = 1.0e-6 # Microseconds
ns = 1.0e-9 # Nanoseconds

qubit = 0 # qubit we will analyze
default_qubit_freq = backend_defaults.qubit_freq_est[qubit] # Default qubit␣
↪frequency in Hz.
print(f"Qubit {qubit} has an estimated frequency of {default_qubit_freq/ GHz}␣
↪GHz.")

# scale data (specific to each device)
scale_factor = 1e-14

# number of shots for our experiments
NUM_SHOTS = 1024

### Collect the necessary channels
drive_chan = pulse.DriveChannel(qubit)
meas_chan = pulse.MeasureChannel(qubit)
acq_chan = pulse.AcquireChannel(qubit)
```

```
ibmqfactory.load_account:WARNING:2020-11-02 11:06:33,141: Credentials are
already in use. The existing account in the session will be replaced.
```

```
Qubit 0 has an estimated frequency of 4.974446164901744 GHz.
```

[20]:
```
def get_job_data(job, average):
    """Retrieve data from a job that has already run.
    Args:
        job (Job): The job whose data you want.
        average (bool): If True, gets the data assuming data is an average.
                        If False, gets the data assuming it is for single shots.
    Return:
        list: List containing job result data.
```

```
    """
    job_results = job.result(timeout=120) # timeout parameter set to 120 s
    result_data = []
    for i in range(len(job_results.results)):
        if average: # get avg data
            result_data.append(job_results.get_memory(i)[qubit]*scale_factor)
        else: # get single data
            result_data.append(job_results.get_memory(i)[:, qubit]*scale_factor)
    return result_data

def get_closest_multiple_of_16(num):
    """Compute the nearest multiple of 16. Needed because pulse enabled devices␣
 ↪require
    durations which are multiples of 16 samples.
    """
    return (int(num) - (int(num)%16))

# Drive pulse parameters (us = microseconds)
drive_sigma_us = 0.075                     # This determines the actual width␣
 ↪of the gaussian
drive_samples_us = drive_sigma_us*8        # This is a truncating parameter,␣
 ↪because gaussians don't have
                                           # a natural finite length

drive_sigma = get_closest_multiple_of_16(drive_sigma_us * us /dt)       # The␣
 ↪width of the gaussian in units of dt
drive_samples = get_closest_multiple_of_16(drive_samples_us * us /dt)
# The truncating parameter in units of dt
# Find out which measurement map index is needed for this qubit
meas_map_idx = None
for i, measure_group in enumerate(backend_config.meas_map):
    if qubit in measure_group:
        meas_map_idx = i
        break
assert meas_map_idx is not None, f"Couldn't find qubit {qubit} in the meas_map!"
# Get default measurement pulse from instruction schedule map
inst_sched_map = backend_defaults.instruction_schedule_map
measure = inst_sched_map.get('measure', qubits=backend_config.
 ↪meas_map[meas_map_idx])
```

[21]:
```
# We will sweep 40 MHz around the estimated frequency, with 75 frequencies
#num_freqs = 75
#ground_sweep_freqs = default_qubit_freq + np.linspace(-20*MHz, 20*MHz,␣
 ↪num_freqs)
#ground_freq_sweep_program =␣
 ↪create_ground_freq_sweep_program(ground_sweep_freqs, drive_power=0.3)
```

```
# create the Duffing oscillator system model(returns a PulseSystemModel object,␣
↪which is a general object for storing model information required for␣
↪simulation with the PulseSimulator.)
two_qubit_model = duffing_system_model(dim_oscillators=dim_oscillators,
                                       oscillator_freqs=oscillator_freqs,
                                       anharm_freqs=anharm_freqs,
                                       drive_strengths=drive_strengths,
                                       coupling_dict=coupling_dict,
                                       dt=dt)
```

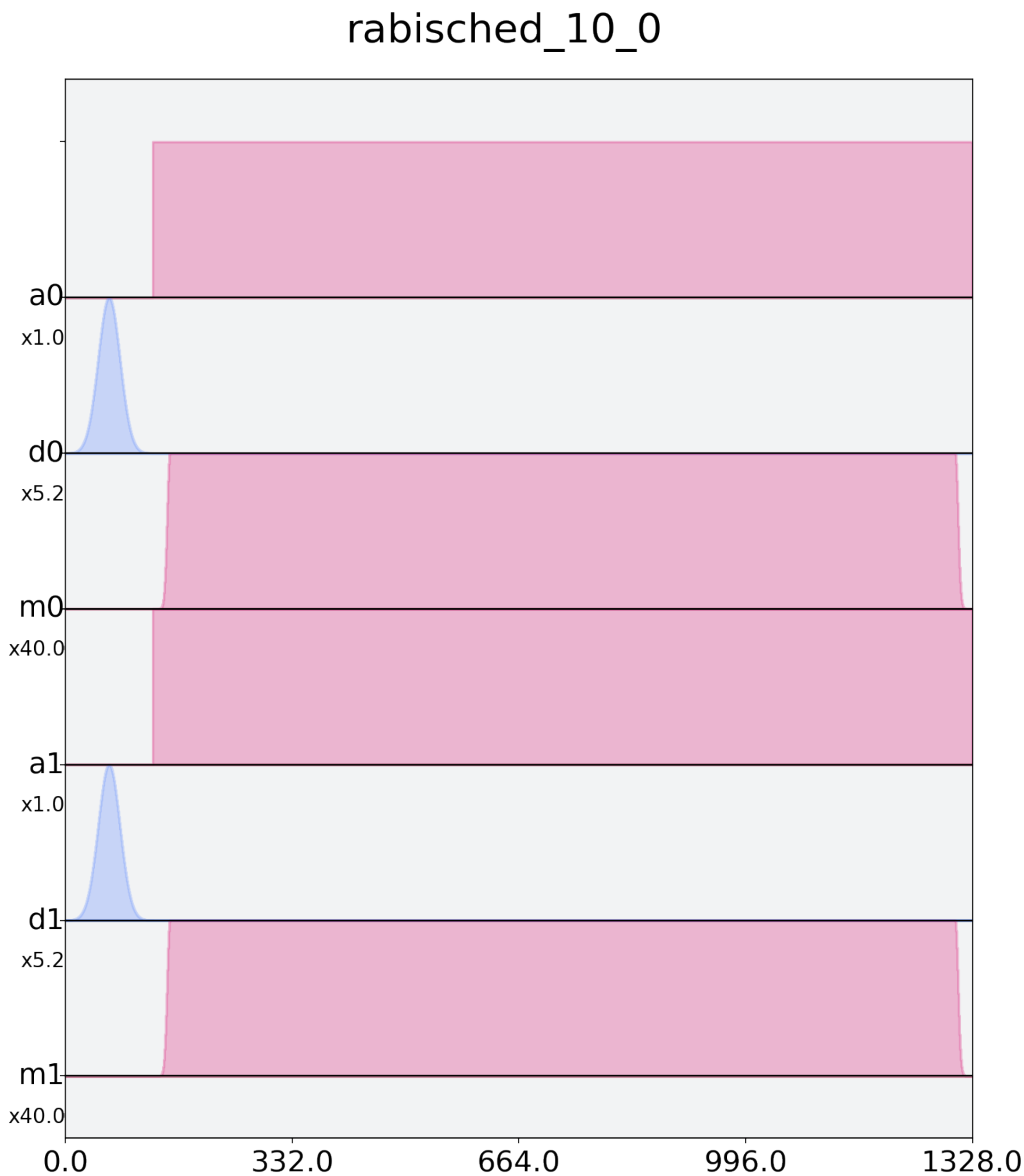
Execution using papermill encountered an exception here and stopped:

[23]:
```
#calibrate pi pulse on each qubit using Ihnis
#4.1 Constructing the schedules
# list of qubits to be used throughout the notebook

qubits = [0, 1]
# Construct a measurement schedule and add it to an InstructionScheduleMap
meas_amp = 0.025
meas_samples = 1200
meas_sigma = 4
meas_width = 1150
meas_pulse = GaussianSquare(duration=meas_samples, amp=meas_amp,
                            sigma=meas_sigma, width=meas_width)

acq_sched = pulse.Acquire(meas_samples, pulse.AcquireChannel(0), pulse.
↪MemorySlot(0))
acq_sched += pulse.Acquire(meas_samples, pulse.AcquireChannel(1), pulse.
↪MemorySlot(1))

measure_sched = pulse.Play(meas_pulse, pulse.MeasureChannel(0)) | pulse.
↪Play(meas_pulse, pulse.MeasureChannel(1)) | acq_sched

inst_map = pulse.InstructionScheduleMap()
inst_map.add('measure', qubits, measure_sched)

#Rabi schedules
#recall: Rabii oscillation
# The magnetic moment is thus {\displaystyle {\boldsymbol {\mu }}={\frac {\hbar␣
↪}{2}}\gamma {\boldsymbol {\sigma }}}{\boldsymbol {\mu }}={\frac {\hbar␣
↪}{2}}\gamma {\boldsymbol {\sigma }}.
```

```
# The Hamiltonian of this system is then given by {H} =-{{\mu }}\cdot{B} =-{\frac {\hbar }{2}}\omega _{0}\sigma _{z}-{\frac {\hbar }{2}}\omega _{1}(\sigma _{x}\cos \omega t-\sigma _{y}\sin \omega t)}\mathbf {H} =-{\boldsymbol {\mu }}\cdot \mathbf {B} =-{\frac {\hbar }{2}}\omega _{0}\sigma _{z}-{\frac {\hbar }{2}}\omega _{1}(\sigma _{x}\cos \omega t-\sigma _{y}\sin \omega t) where {\displaystyle \omega _{0}=\gamma B_{0}}\omega _{0}=\gamma B_{0} and {\displaystyle \omega _{1}=\gamma B_{1}}\omega _{1}=\gamma B_{1}
# Now, let the qubit be in state {\displaystyle |0\rangle }{\displaystyle |0\rangle } at time {\displaystyle t=0}t=0. Then, at time {\displaystyle t}t, the probability of it being found in state {\displaystyle |1\rangle }|1\rangle  is given by {\displaystyle P_{0\to 1}(t)=\left({\frac {\omega _{1}}{\Omega }}\right)^{2}\sin ^{2}\left({\frac {\Omega t}{2}}\right)}{\displaystyle P_{0\to 1}(t)=\left({\frac {\omega _{1}}{\Omega }}\right)^{2}\sin ^{2}\left({\frac {\Omega t}{2}}\right)} where {\displaystyle \Omega ={\sqrt {(\omega -\omega _{0})^{2}+\omega _{1}^{2}}}}\Omega ={\sqrt {(\omega -\omega _{0})^{2}+\omega _{1}^{2}}}
# the qubit oscillates between the {\displaystyle |0\rangle }|0\rang and {\displaystyle |1\rangle }|1\rangle  states.
# The maximum amplitude for oscillation is achieved at {\displaystyle \omega =\omega _{0}}\omega =\omega _{0}, which is the condition for resonance.
# At resonance, the transition probability is given by {\displaystyle P_{0\to 1}(t)=\sin ^{2}\left({\frac {\omega _{1}t}{2}}\right)}{\displaystyle P_{0\to 1}(t)=\sin ^{2}\left({\frac {\omega _{1}t}{2}}\right)}

# pi pulse:
# To go from state {\displaystyle |0\rangle }|0\rang to state {\displaystyle |1\rangle }|1\rangle  it is sufficient to adjust the time {\displaystyle t}t during which the rotating field acts such that {\displaystyle {\frac {\omega _{1}t}{2}}={\frac {\pi }{2}}}{\frac {\omega _{1}t}{2}}={\frac {\pi }{2}} or {\displaystyle t={\frac {\pi }{\omega _{1}}}}t={\frac {\pi }{\omega _{1}}}
#  If a time intermediate between 0 and {\displaystyle {\frac {\pi }{\omega _{1}}}}{\frac {\pi }{\omega _{1}}} is chosen, we obtain a superposition of {\displaystyle |0\rangle }|0\rang and {\displaystyle |1\rangle }|1\rangle .
# Approximation with pi/2 pulse:
# In particular for {\displaystyle t={\frac {\pi }{2\omega _{1}}}}t={\frac {\pi }{2\omega _{1}}}, we have a {\displaystyle {\frac {\pi }{2}}}{\frac {\pi }{2}} pulse, which acts as: {\displaystyle |0\rangle \to {\frac {|0\rangle +i|1\rangle }{\sqrt {2}}}}{\displaystyle |0\rangle \to {\frac {|0\rangle +i|1\rangle }{\sqrt {2}}}}.
```

```
# The equations are essentially identical in the case of a two level atom in␣
 ↪the field of a laser when the generally well satisfied rotating wave␣
 ↪approximation is made. Then {\displaystyle \hbar \omega _{0}}\hbar \omega␣
 ↪_{0} is the energy difference between the two atomic levels, {\displaystyle␣
 ↪\omega }\omega  is the frequency of laser wave and Rabi frequency␣
 ↪{\displaystyle \omega _{1}}\omega _{1} is proportional to the product of the␣
 ↪transition electric dipole moment of atom {\displaystyle {\vec {d}}}{\vec␣
 ↪{d}} and electric field {\displaystyle {\vec {E}}}{\vec {E}} of the laser␣
 ↪wave that is {\displaystyle \omega _{1}\propto \hbar \ {\vec {d}}\cdot {\vec␣
 ↪{E}}}{\displaystyle \omega _{1}\propto \hbar \ {\vec {d}}\cdot {\vec {E}}}.



#experiments

drive_amps = np.linspace(0, 0.9, 48)
drive_sigma = 16
drive_duration = 128

drive_channels = [pulse.DriveChannel(0), pulse.DriveChannel(1)]

rabi_experiments, rabi_amps = rabi_schedules(amp_list=drive_amps,
                                             qubits=qubits,
                                             pulse_width=drive_duration,
                                             pulse_sigma=drive_sigma,
                                             drives=drive_channels,
                                             inst_map=inst_map,
                                             meas_map=[[0, 1]])

rabi_experiments[10].draw()
```

[23]:

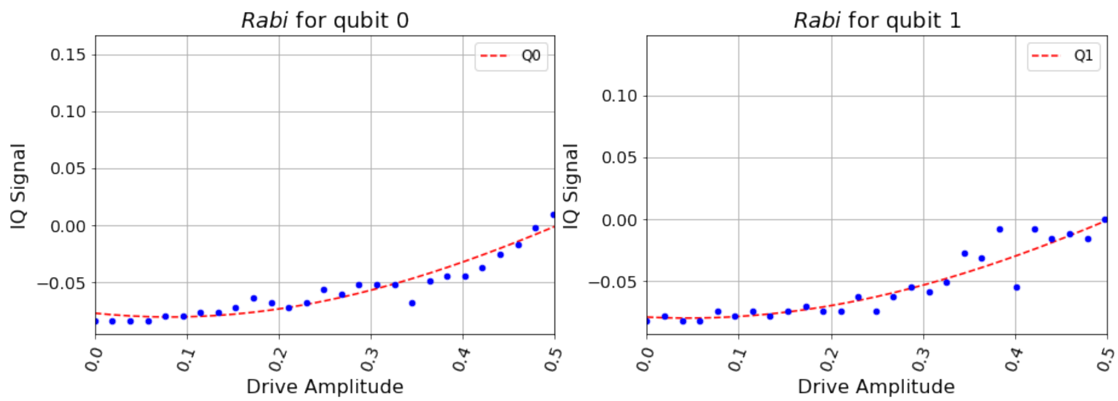


[24]:
```
#ground_freq_sweep_job = backend.run(ground_freq_sweep_program)
#print(ground_freq_sweep_job.job_id())
#job_monitor(ground_freq_sweep_job)
# Get the job data (average)
#ground_freq_sweep_data = get_job_data(ground_freq_sweep_job, average=True)
```

```
#To simulate the Rabi experiments, assemble the Schedule list into a qobj. When␣
↪assembling, pass the PulseSimulator as the backend.#To simulate the Rabi␣
↪experiments, assemble the Schedule list into a qobj. When assembling, pass␣
↪the PulseSimulator as the backend.

# instantiate the pulse simulator
backend_sim = PulseSimulator()

# compute frequencies from the Hamiltonian
qubit_lo_freq = two_qubit_model.hamiltonian.get_qubit_lo_from_drift()

rabi_qobj = assemble(rabi_experiments,
                     backend=backend_sim,
                     qubit_lo_freq=qubit_lo_freq,
                     meas_level=1,
                     meas_return='avg',
                     shots=256)

# run the simulation
rabi_result = backend_sim.run(rabi_qobj, two_qubit_model).result()
```

[25]:
```
rabifit = RabiFitter(rabi_result, rabi_amps, qubits, fit_p0 = [0.5,0.5,0.6,1.5])

plt.figure(figsize=(15, 10))
q_offset = 0
multiplier = 0.5
for qubit in qubits:
    ax = plt.subplot(2, 2, qubit + 1)
    #Xvmin, Xvmax = ax.xaxis.get_data_interval()
    #Yvmin, Yvmax = ax.yaxis.get_data_interval()
    #print(Xvmin, Xvmax,Yvmin, Yvmax)
    Xvmin = multiplier * np.floor(0.1 / multiplier)
    Xvmax = multiplier * np.ceil(0.5 / multiplier)
    ax.set_xlim([Xvmin, Xvmax])
    rabifit.plot(qubit, ax=ax)
    print('Pi Amp: %f'%rabifit.pi_amplitude(qubit))
plt.show()
```

```
Pi Amp: 1.141925
Pi Amp: 1.333972
```

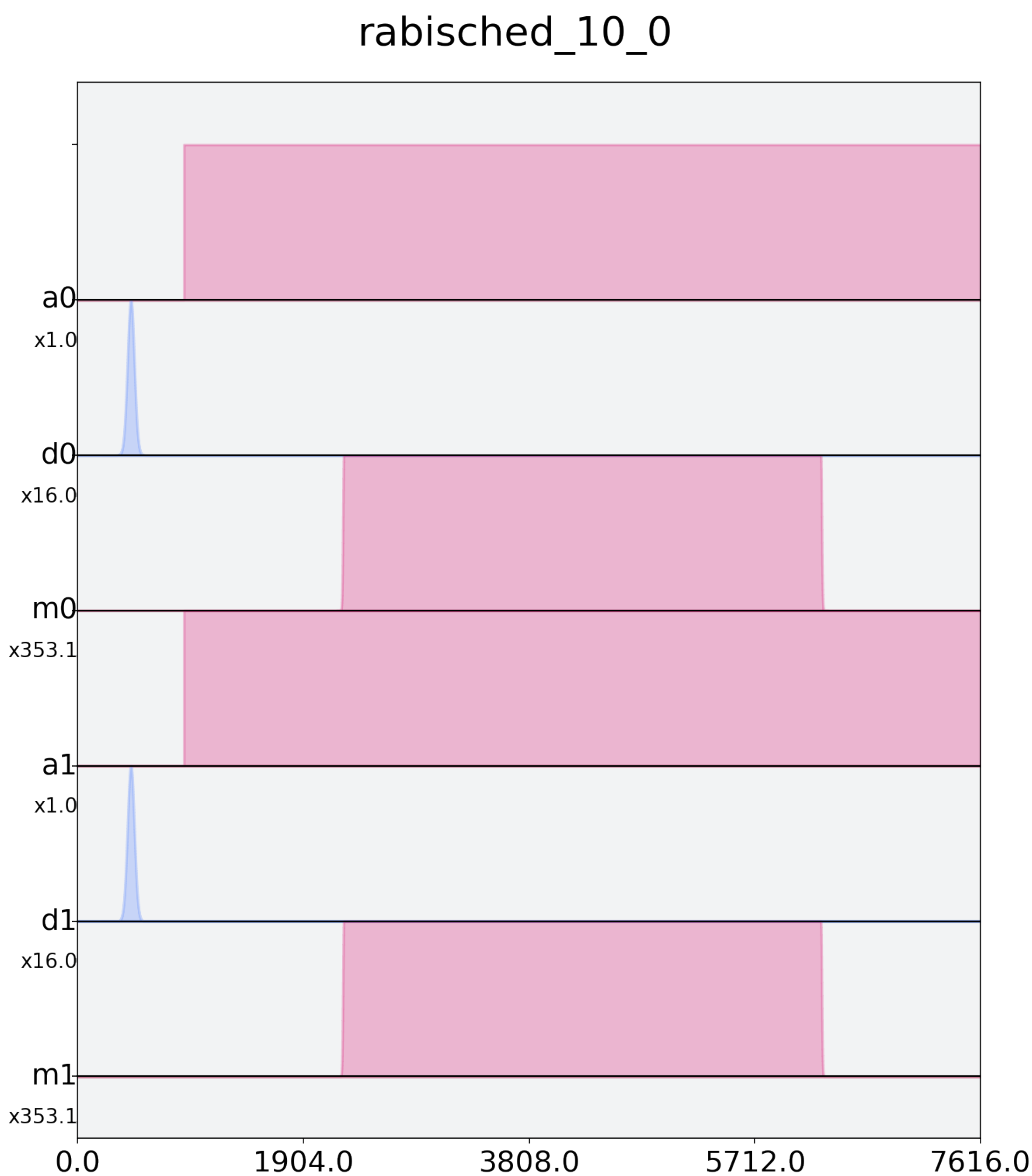


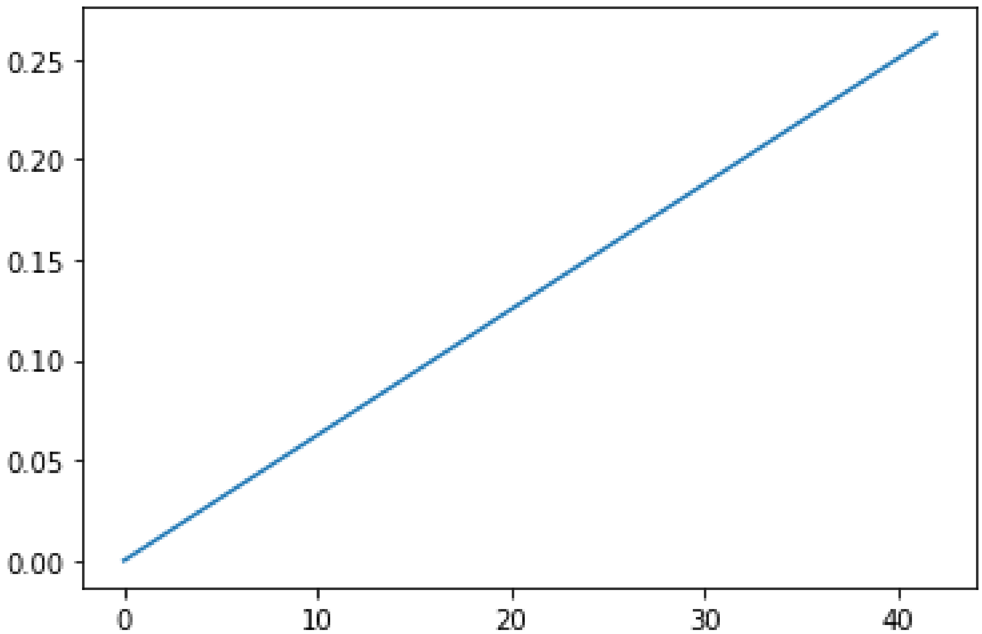
[40]:
```
#experiment data
qubits = [0, 1]

N =  int(Dims*dim_oscillators/2)

meas_amp = Dims/128*0.025/2
meas_samples = int(rows/16*1200/10)
meas_sigma = int(Dims/16*4)
meas_width = int(rows/128*1150/2)
meas_pulse = GaussianSquare(duration=meas_samples, amp=meas_amp,
                            sigma=meas_sigma, width=meas_width)

acq_sched = pulse.Acquire(meas_samples, pulse.AcquireChannel(0), pulse.
 ↪MemorySlot(0))
acq_sched += pulse.Acquire(meas_samples, pulse.AcquireChannel(1), pulse.
 ↪MemorySlot(1))

measure_sched = pulse.Play(meas_pulse, pulse.MeasureChannel(0)) | pulse.
 ↪Play(meas_pulse, pulse.MeasureChannel(1)) | acq_sched

inst_map = pulse.InstructionScheduleMap()
inst_map.add('measure', qubits, measure_sched)

drive_amps = np.linspace(0, 0.9, N)
drive_sigma = Dims
drive_duration = rows

drive_channels = [pulse.DriveChannel(0), pulse.DriveChannel(1)]

rabi_experiments, rabi_amps = rabi_schedules(amp_list=drive_amps/np.linalg.
 ↪norm(drive_amps),
                                             qubits=qubits,
```

```
                                            pulse_width=drive_duration,
                                            pulse_sigma=drive_sigma,
                                            drives=drive_channels,
                                            inst_map=inst_map,
                                            meas_map=[[0, 1]])


print(len(rabi_experiments[10]))
plt.plot(rabi_amps)
rabi_experiments[10].draw()
```

6

[40]:





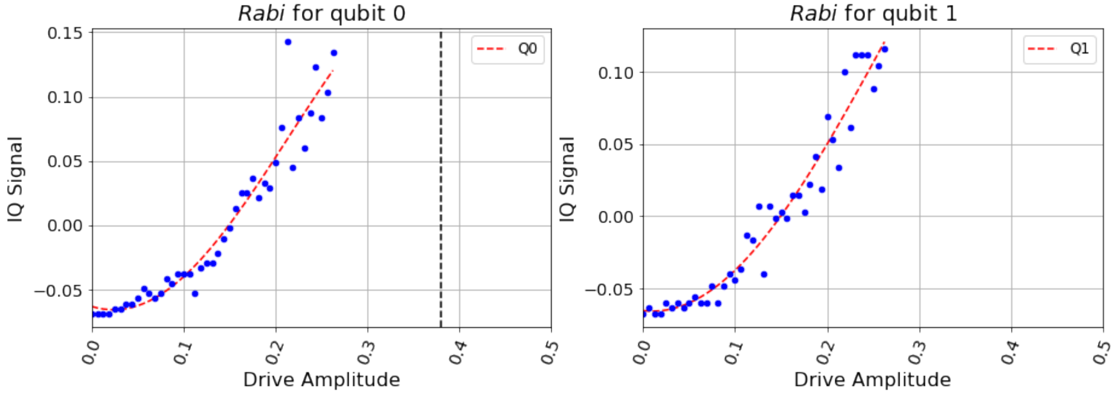

```
def create_ground_freq_sweep_program(freqs, drive_power): ""``Builds a program
that does a freq sweep by exciting the ground state. Depending on drive
power this can reveal the 0->1 frequency or the 0->2 frequency. Args: freqs
(np.ndarray(dtype=float)): Numpy array of frequencies to sweep. drive_power
(float) : Value of drive amplitude. Raises: ValueError: Raised if use more
than 75 frequencies; currently, an error will be thrown on the backend if you
try to do this. Returns: Qobj: Program for ground freq sweep experiment.''"" if
len(freqs) > 75: raise ValueError(``You can only run 75 schedules at a time.'')

# print information on the sweep
print(f"The frequency sweep will go from {freqs[0] / GHz} GHz to {freqs[-1]/ GHz} GHz \

using {len(freqs)} frequencies. The drive power is {drive_power}.")

# Define the drive pulse
ground_sweep_drive_pulse = pulse_lib.gaussian(duration=drive_samples,
                                              sigma=drive_sigma,
                                              amp=drive_power,
                                              name='ground_sweep_drive_pulse')
# Create the base schedule
schedule = pulse.Schedule(name='Frequency sweep starting from ground state.')

schedule |= ground_sweep_drive_pulse(drive_chan)
schedule |= measure << schedule.duration
```

```
    # define frequencies for the sweep
    schedule_freqs = [{drive_chan: freq} for freq in freqs]

    # assemble the program
    # Note: we only require a single schedule since each does the same thing;
    # for each schedule, the LO frequency that mixes down the drive changes
    # this enables our frequency sweep
    ground_freq_sweep_program = assemble(schedule,
                                         backend=backend,
                                         meas_level=1,
                                         meas_return='avg',
                                         shots=NUM_SHOTS,
                                         schedule_los=schedule_freqs)

    return ground_freq_sweep_program
```

[41]:

```
#experiment data
rabifit = RabiFitter(rabi_result, rabi_amps, qubits, fit_p0 = [1.5,1.5,1.6,2.5])

plt.figure(figsize=(15, 10))
q_offset = 0
multiplier = 0.5
for qubit in qubits:
    ax = plt.subplot(2, 2, qubit + 1)
    #Xvmin, Xvmax = ax.xaxis.get_data_interval()
    #Yvmin, Yvmax = ax.yaxis.get_data_interval()
    #print(Xvmin, Xvmax,Yvmin, Yvmax)
    Xvmin = multiplier * np.floor(0.1 / multiplier)
    Xvmax = multiplier * np.ceil(0.5 / multiplier)
    ax.set_xlim([Xvmin, Xvmax])
    rabifit.plot(qubit, ax=ax)
    print('Pi Amp: %f'%rabifit.pi_amplitude(qubit))
plt.show()
```

```
Pi Amp: 0.380605
Pi Amp: 0.508708
```



[42]:
```
# cross-resonance ControlChannel
two_qubit_model.control_channel_index((1,0))
#to perform a cross-resonance drive on qubit 1 with target qubit 0,
#use ControlChannel(1).
```

[42]: 1

[44]:
```
#store pi amplitudes
#given the drive and target indices, and the option to either start with the␣
↪drive qubit in the ground or excited state, returns a list of experiments␣
↪for observing the oscillations.
pi_amps = [rabifit.pi_amplitude(0), rabifit.pi_amplitude(1)]

def cr_drive_experiments(drive_idx,
                         target_idx,
                         flip_drive_qubit = False,

                         #cr_drive_amps=np.linspace(0, 0.9, 16),
                         #cr_drive_samples=800,
                         #cr_drive_sigma=4,
                         #pi_drive_samples=128,
                         #pi_drive_sigma=16
                         #meas_amp = Dims/128*0.025/2
                         #meas_width = int(rows/128*1150/2)
                         cr_drive_amps=np.linspace(0, 0.9, Dims),
                         cr_drive_samples=int(rows/16*1200/10),
                         cr_drive_sigma=int(Dims/16*4),
                         pi_drive_samples=Dims,
                         pi_drive_sigma=rows):
    """Generate schedules corresponding to CR drive experiments.

    Args:
        drive_idx (int): label of driven qubit
        target_idx (int): label of target qubit
        flip_drive_qubit (bool): whether or not to start the driven qubit in␣
↪the ground or excited state
        cr_drive_amps (array): list of drive amplitudes to use
        cr_drive_samples (int): number samples for each CR drive signal
        cr_drive_sigma (float): standard deviation of CR Gaussian pulse
        pi_drive_samples (int): number samples for pi pulse on drive
        pi_drive_sigma (float): standard deviation of Gaussian pi pulse on drive

    Returns:
        list[Schedule]: A list of Schedule objects for each experiment
```

```python
    """

    # Construct measurement commands to be used for all schedules
    #meas_amp = 0.025
    #meas_samples = 1200
    #meas_sigma = 4
    #meas_width = 1150
    meas_amp = Dims/128*0.025/2
    meas_samples = int(rows/16*1200/10)
    meas_sigma = int(Dims/16*4)
    meas_width = int(rows/128*1150/2)
    meas_pulse = GaussianSquare(duration=meas_samples, amp=meas_amp/np.linalg.
↪norm(meas_amp),
                                sigma=meas_sigma, width=meas_width)

    acq_sched = pulse.Acquire(meas_samples, pulse.AcquireChannel(0), pulse.
↪MemorySlot(0))
    acq_sched += pulse.Acquire(meas_samples, pulse.AcquireChannel(1), pulse.
↪MemorySlot(1))

    # create measurement schedule
    measure_sched = (pulse.Play(meas_pulse, pulse.MeasureChannel(0)) |
                     pulse.Play(meas_pulse, pulse.MeasureChannel(1))|
                     acq_sched)

    # Create schedule
    schedules = []
    for ii, cr_drive_amp in enumerate(cr_drive_amps):

        # pulse for flipping drive qubit if desired
        pi_pulse = Gaussian(duration=pi_drive_samples, amp=pi_amps[drive_idx],␣
↪sigma=pi_drive_sigma)

        # cr drive pulse
        cr_width = cr_drive_samples - 2*cr_drive_sigma*4
        cr_rabi_pulse = GaussianSquare(duration=cr_drive_samples,
                                       amp=cr_drive_amp/np.linalg.
↪norm(cr_drive_amp),
                                       sigma=cr_drive_sigma,
                                       width=cr_width)

        # add commands to schedule
        schedule = pulse.Schedule(name='cr_rabi_exp_amp_%s' % cr_drive_amp)
        #schedule = pulse.Schedule(name='cr_rabi_exp_amp_%s' % cr_drive_amp/np.
↪linalg.norm(cr_drive_amp))
```

```
        # flip drive qubit if desired
        if flip_drive_qubit:
            schedule += pulse.Play(pi_pulse, pulse.DriveChannel(drive_idx))

        # do cr drive
        # First, get the ControlChannel index for CR drive from drive to target
        cr_idx = two_qubit_model.control_channel_index((drive_idx, target_idx))
        schedule += pulse.Play(cr_rabi_pulse, pulse.ControlChannel(cr_idx))  <<␣
↪schedule.duration


        schedule += measure_sched << schedule.duration

        schedules.append(schedule)
    return schedules
```

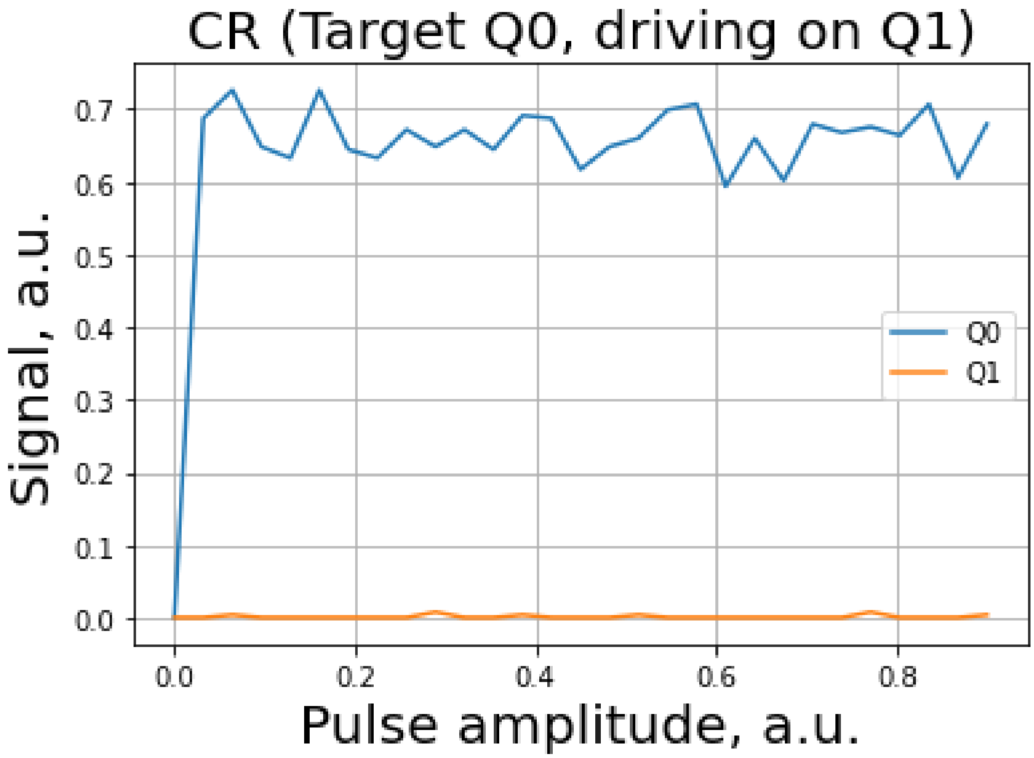
[45]:
```
#create two functions for observing the data: - plot_cr_pop_data - for plotting␣
↪the oscillations between the ground state and the first excited state -␣
↪plot_bloch_sphere - for viewing the trajectory of the target qubit on the␣
↪bloch sphere
def plot_cr_pop_data(drive_idx,
                     target_idx,
                     sim_result,
#                    cr_drive_amps=np.linspace(0, 0.9, 16)):
                     cr_drive_amps=np.linspace(0, 0.9,Dims)):
#                    cr_drive_amps=cr_drive_amp/np.linalg.norm(cr_drive_amp)):
    """Plot the population of each qubit.

    Args:
        drive_idx (int): label of driven qubit
        target_idx (int): label of target qubit
        sim_result (Result): results of simulation
        cr_drive_amps (array): list of drive amplitudes to use for axis labels
    """

    amp_data_Q0 = []
    amp_data_Q1 = []

    for exp_idx in range(len(cr_drive_amps)):
        exp_mem = sim_result.get_memory(exp_idx)
        amp_data_Q0.append(np.abs(exp_mem[0]))
        amp_data_Q1.append(np.abs(exp_mem[1]))

    plt.plot(cr_drive_amps, amp_data_Q0, label='Q0')
    plt.plot(cr_drive_amps, amp_data_Q1, label='Q1')
    plt.legend()
    plt.xlabel('Pulse amplitude, a.u.', fontsize=20)
```

```python
    plt.ylabel('Signal, a.u.', fontsize=20)
    plt.title('CR (Target Q{0}, driving on Q{1})'.format(target_idx,␣
↪drive_idx), fontsize=20)
    plt.grid(True)

def bloch_vectors(drive_idx, drive_energy_level, sim_result):
    """Plot the population of each qubit.

    Args:
        drive_idx (int): label of driven qubit
        drive_energy_level (int): energy level of drive qubit at start of CR␣
↪drive
        sim_result (Result): results of simulation

    Returns:
        list: list of Bloch vectors corresponding to the final state of the␣
↪target qubit
              for each experiment
    """

    # get the dimension used for simulation
    dim = int(np.sqrt(len(sim_result.get_statevector(0))))


    # get the relevant dressed state indices
    idx0 = 0
    idx1 = 0
    if drive_idx == 0:
        if drive_energy_level == 0:
            idx0, idx1 = 0, dim
        elif drive_energy_level == 1:
            idx0, idx1 = 1, dim + 1
    if drive_idx == 1:
        if drive_energy_level == 0:
            idx0, idx1 = 0, 1
        elif drive_energy_level == 1:
            idx0, idx1 = dim, dim + 1

    # construct Pauli operators for correct dressed manifold
    state0 = np.array([two_qubit_model.hamiltonian._estates[idx0]])
    state1 = np.array([two_qubit_model.hamiltonian._estates[idx1]])

    outer01 = np.transpose(state0)@state1
    outer10 = np.transpose(state1)@state0
    outer00 = np.transpose(state0)@state0
    outer11 = np.transpose(state1)@state1
```

```
    X = outer01 + outer10
    Y = -1j*outer01 + 1j*outer10
    Z = outer00 - outer11

    # function for computing a single bloch vector
    bloch_vec = lambda vec: np.real(np.array([np.conj(vec)@X@vec, np.
 ↪conj(vec)@Y@vec, np.conj(vec)@Z@vec]))

    return [bloch_vec(sim_result.get_statevector(idx)) for idx in␣
 ↪range(len(sim_result.results))]

def plot_bloch_sphere(bloch_vectors):
    """Given a list of Bloch vectors, plot them on the Bloch sphere

    Args:
        bloch_vectors (list): list of bloch vectors
    """
    sphere = Bloch()
    sphere.add_points(np.transpose(bloch_vectors))
    sphere.show()
```

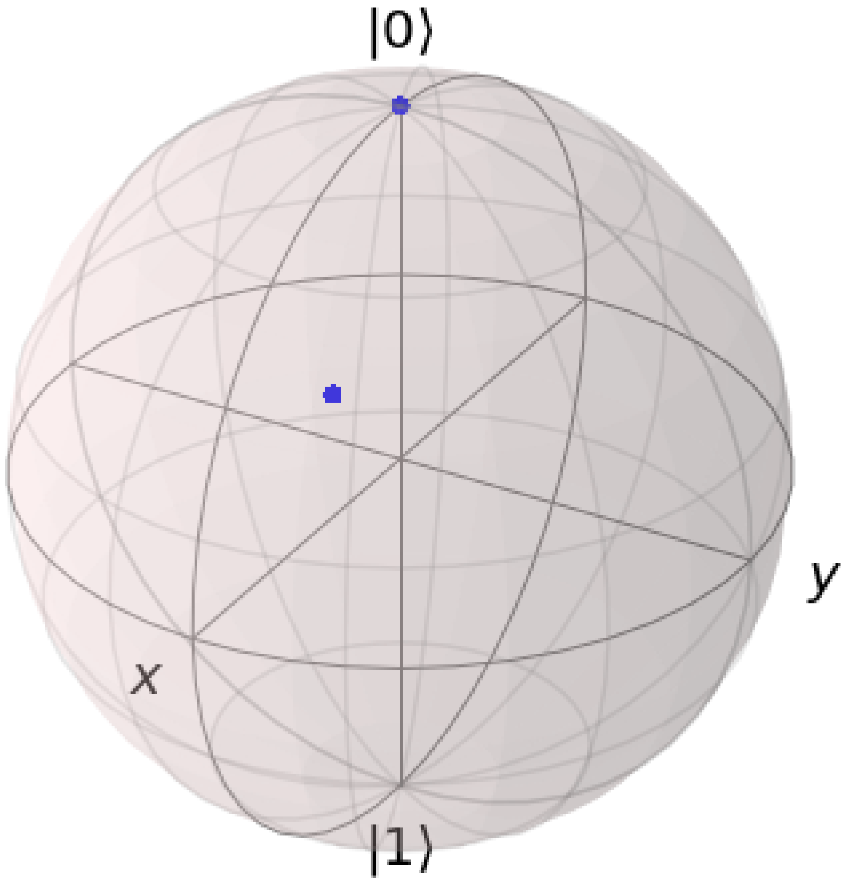
[48]:
```
# construct experiments to observe CR oscillations on qubit 0, driving qubit 1
# Qubit 1 and qubit 0000000000  in the ground state
# construct experiments
drive_idx = 1
target_idx = 0
flip_drive = False
experiments = cr_drive_experiments(drive_idx, target_idx, flip_drive)

# compute frequencies from the Hamiltonian
qubit_lo_freq = two_qubit_model.hamiltonian.get_qubit_lo_from_drift()

# assemble the qobj
cr_rabi_qobj = assemble(experiments,
                        backend=backend_sim,
                        qubit_lo_freq=qubit_lo_freq,
                        meas_level=1,
                        meas_return='avg',
                        shots=256)
```

[55]:
```
#simulation
sim_result = backend_sim.run(cr_rabi_qobj, two_qubit_model).result()

plot_cr_pop_data(drive_idx, target_idx, sim_result)
```



[64]:
```
# observe the trajectory of qubit 0 on the Bloch sphere:
bloch_vecs = bloch_vectors(drive_idx, int(flip_drive), sim_result)
print(np.mean(bloch_vecs*N))
print(N)
if (np.mean(bloch_vecs*N) < 1):
    plot_bloch_sphere(bloch_vecs*N)
else:
    plot_bloch_sphere(bloch_vecs)
```

```
-0.5230914192734953
43
```



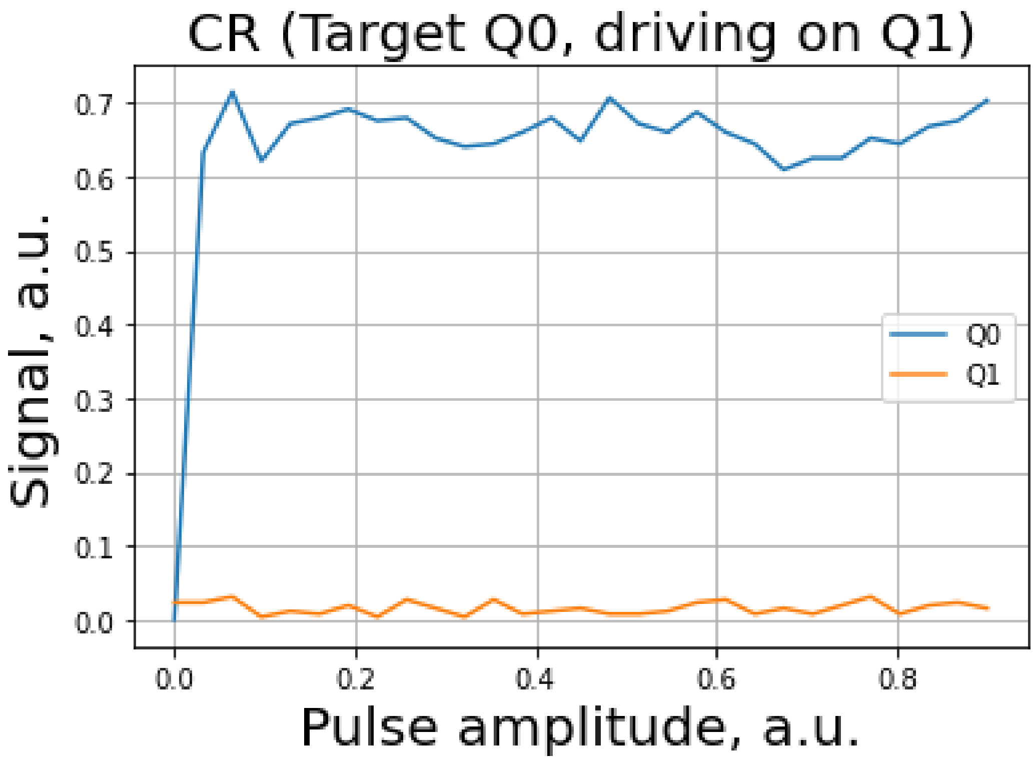
[ ]:
```
# assemble the qobj
cr_rabi_qobj = assemble(experiments,
                        backend=backend_sim,
                        qubit_lo_freq=qubit_lo_freq,
                        meas_level=1,
                        meas_return='avg',
                        shots=256)
```

[68]:
```
# now with flip_drive == True
drive_idx = 1
target_idx = 0
flip_drive = True
experiments = cr_drive_experiments(drive_idx, target_idx, flip_drive)

# compute frequencies from the Hamiltonian
qubit_lo_freq = two_qubit_model.hamiltonian.get_qubit_lo_from_drift()
```

```
# assemble the qobj
cr_rabi_qobj = assemble(experiments,
                        backend=backend_sim,
                        qubit_lo_freq=qubit_lo_freq,
                        meas_level=1,
                        meas_return='avg',
                        shots=256)
```

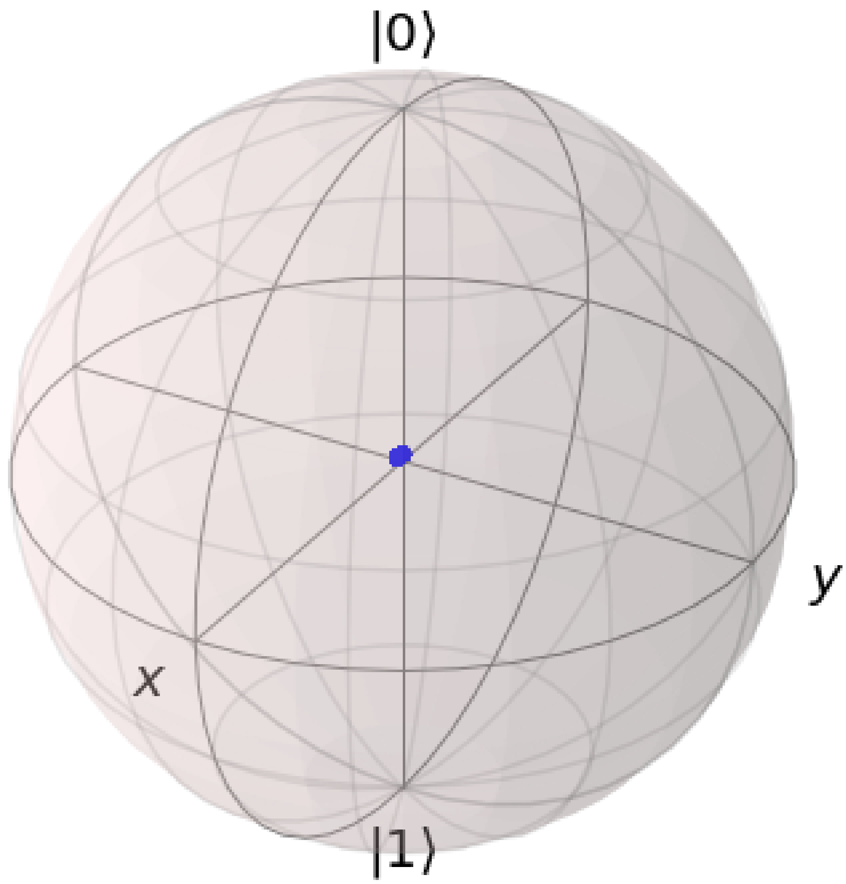
[69]:

```
#Observe that now at a different rate as before, qubit 1 is in the excited␣
↪state, while oscillations are again being driven on qubit 0
sim_result = backend_sim.run(cr_rabi_qobj, two_qubit_model).result()

plot_cr_pop_data(drive_idx, target_idx, sim_result)
```



[71]:

```
#Observe the trajectory of qubit 0 on the Bloch sphere:
bloch_vecs = bloch_vectors(drive_idx, int(flip_drive), sim_result)
plot_bloch_sphere(bloch_vecs*N)
```



def fit_function(x_values, y_values, function, init_params): ""``Fit a function using scipy curve_fit.''"" fitparams, conv = curve_fit(function, x_values, y_values, init_params) y_fit = function(x_values, *fitparams)

return fitparams, y_fit

1 do fit in Hz

(ground_sweep_fit_params, ground_sweep_y_fit) = fit_function(normdata, ground_freq_sweep_data, lambda x, A, q_freq, B, C: (A / np.pi) * (B / ((x - q_freq)2 + B2)) + C, [7, 4.975*GHz, 1*GHz, 3*GHz] # initial parameters for curve_fit)

, *cal_qubit_freq,* , _ = total_loss_fit_params print(f``We've updated our qubit frequency estimate from'' f``{round(default_qubit_freq/GHz, 7)} GHz to {round(cal_qubit_freq/GHz, 7)} GHz.'')