phys4410finalprojectproof

May 10, 2021

[1]:
```
#just analyze ibmq_armonk. no other pulse providers.
```

[2]:
```
from qiskit.tools.jupyter import *
```

[3]:
```
from qiskit import IBMQ
IBMQ.load_account()
provider = IBMQ.get_provider(hub='ibm-q', group='open', project='main')
backend = provider.get_backend('ibmq_armonk')
```

```
C:\Users\johnn\.julia\conda\3\lib\site-
packages\qiskit\providers\ibmq\ibmqfactory.py:192: UserWarning: Timestamps in
IBMQ backend properties, jobs, and job results are all now in local time instead
of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '
```

[4]:
```
backend_config = backend.configuration()
assert backend_config.open_pulse, "Backend doesn't support Pulse"
```

[5]:
```
dt = backend_config.dt
print(f"Sampling time: {dt*1e9} ns")    # The configuration returns dt in␣
 ↪seconds, so multiply by
                                        # 1e9 to get nanoseconds
```

```
Sampling time: 0.2222222222222222 ns
```

[6]:
```
backend_defaults = backend.defaults()
```

[7]:
```
import numpy as np

# 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

# We will find the qubit frequency for the following qubit.
```

```
qubit = 0

# The sweep will be centered around the estimated qubit frequency.
center_frequency_Hz = backend_defaults.qubit_freq_est[qubit]        # The␣
 ↪default frequency is given in Hz
                                                                    # warning:␣
 ↪this will change in a future release
print(f"Qubit {qubit} has an estimated frequency of {center_frequency_Hz / GHz}␣
 ↪GHz.")

# scale factor to remove factors of 10 from the data
scale_factor = 1e-14

# We will sweep 40 MHz around the estimated frequency
frequency_span_Hz = 40 * MHz
# in steps of 1 MHz.
frequency_step_Hz = 1 * MHz

# We will sweep 20 MHz above and 20 MHz below the estimated frequency
frequency_min = center_frequency_Hz - frequency_span_Hz / 2
frequency_max = center_frequency_Hz + frequency_span_Hz / 2
# Construct an np array of the frequencies for our experiment
frequencies_GHz = np.arange(frequency_min / GHz,
                            frequency_max / GHz,
                            frequency_step_Hz / GHz)

print(f"The sweep will go from {frequency_min / GHz} GHz to {frequency_max /␣
 ↪GHz} GHz \
in steps of {frequency_step_Hz / MHz} MHz.")
```

```
Qubit 0 has an estimated frequency of 4.97186208283383 GHz.
The sweep will go from 4.95186208283383 GHz to 4.99186208283383 GHz in steps of
1.0 MHz.
```

[8]:
```
# samples need to be multiples of 16
def get_closest_multiple_of_16(num):
    return int(num + 8 ) - (int(num + 8 ) % 16)
```

[9]:
```
from qiskit import pulse            # This is where we access all of our Pulse␣
 ↪features!
from qiskit.pulse import Play
# This Pulse module helps us build sampled pulses for common pulse shapes
from qiskit.pulse import library as pulse_lib


# 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
drive_amp = 0.05
# Drive pulse samples
drive_pulse = pulse_lib.gaussian(duration=drive_samples,
                                 sigma=drive_sigma,
                                 amp=drive_amp,
                                 name='freq_sweep_excitation_pulse')
```

[10]:
```
# Find out which group of qubits need to be acquired with 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!"
```

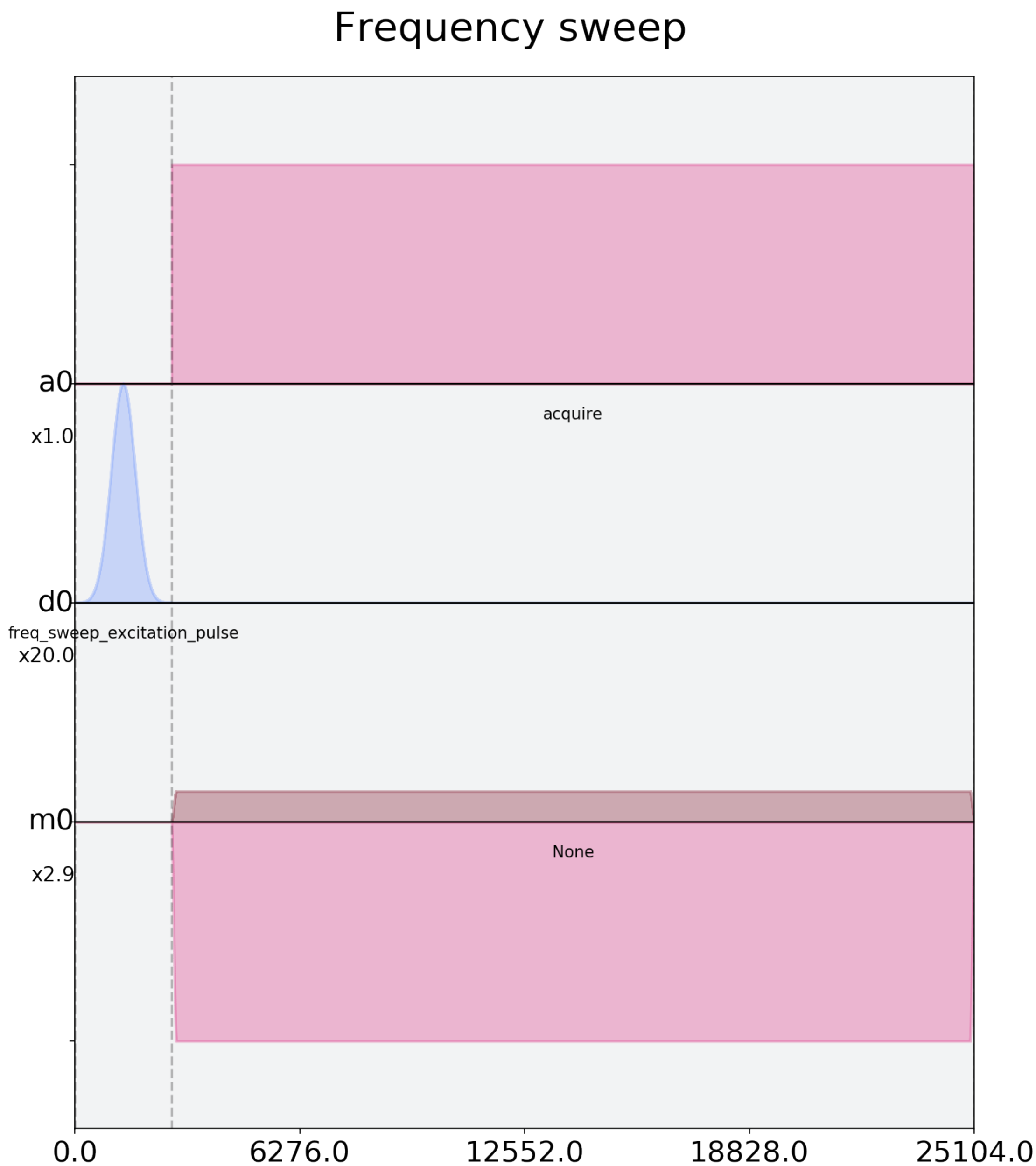
[11]:
```
inst_sched_map = backend_defaults.instruction_schedule_map
measure = inst_sched_map.get('measure', qubits=backend_config.
↪meas_map[meas_map_idx])
```

[12]:
```
### Collect the necessary channels
drive_chan = pulse.DriveChannel(qubit)
meas_chan = pulse.MeasureChannel(qubit)
acq_chan = pulse.AcquireChannel(qubit)
```

[13]:
```
# Create the base schedule
# Start with drive pulse acting on the drive channel
schedule = pulse.Schedule(name='Frequency sweep')
schedule += Play(drive_pulse, drive_chan)
# The left shift `<<` is special syntax meaning to shift the start time of the␣
↪schedule by some duration
schedule += measure << schedule.duration

# Create the frequency settings for the sweep (MUST BE IN HZ)
frequencies_Hz = frequencies_GHz*GHz
schedule_frequencies = [{drive_chan: freq} for freq in frequencies_Hz]
```

[14]:
```
schedule.draw(label=True)
```

[14]:



[15]:
```
from qiskit import assemble

num_shots_per_frequency = 1024
frequency_sweep_program = assemble(schedule,
                                   backend=backend,
                                   meas_level=1,
                                   meas_return='avg',
                                   shots=num_shots_per_frequency,
                                   schedule_los=schedule_frequencies)
```

[16]:
```
job = backend.run(frequency_sweep_program)
```

[17]:
```
# print(job.job_id())
from qiskit.tools.monitor import job_monitor
job_monitor(job)
```

```
Job Status: job has successfully run
```

[18]:
```
job2 = backend.run(frequency_sweep_program)
```

[19]:
```
# print(job.job_id())
from qiskit.tools.monitor import job_monitor
job_monitor(job2)
```

```
Job Status: job has successfully run
```

[20]:
```
job3 = backend.run(frequency_sweep_program)
```

[21]:
```
# print(job.job_id())
from qiskit.tools.monitor import job_monitor
job_monitor(job3)
```

```
Job Status: job has successfully run
```

[22]:
```
job4 = backend.run(frequency_sweep_program)
```

[23]:
```
# print(job.job_id())
from qiskit.tools.monitor import job_monitor
job_monitor(job4)
```

```
Job Status: job has successfully run
```

[24]:
```
job5 = backend.run(frequency_sweep_program)
```

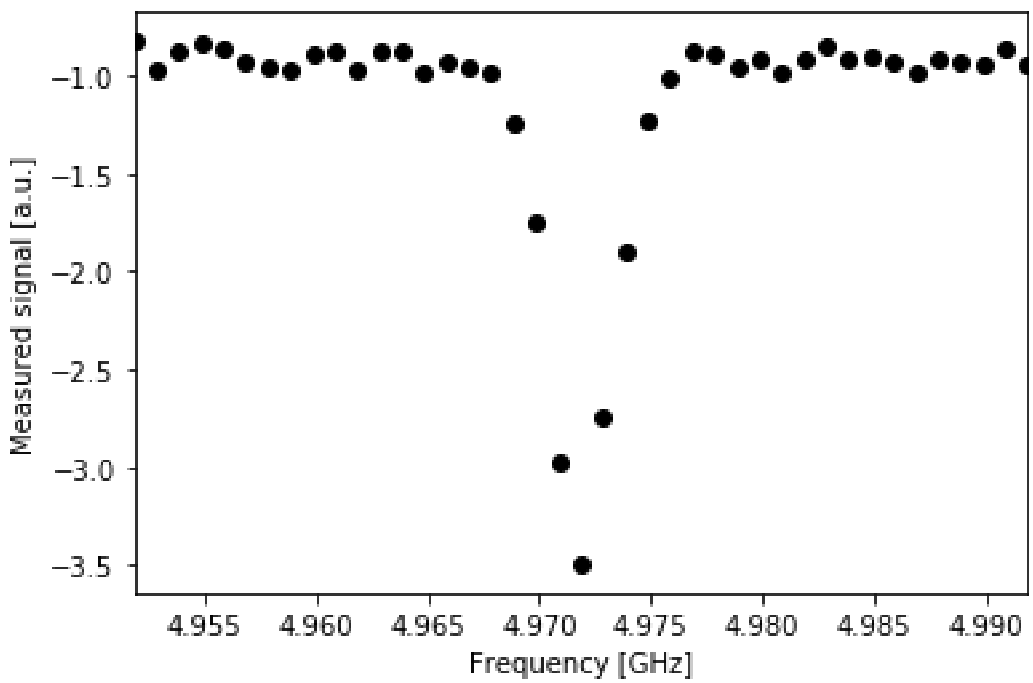
[25]:
```
# print(job.job_id())
from qiskit.tools.monitor import job_monitor
job_monitor(job5)
```

```
Job Status: job has successfully run
```

[68]:
```
frequency_sweep_results = job.result(timeout=120)
```

[69]:
```
import matplotlib.pyplot as plt

sweep_values = []
sweep_values1 = []
for i in range(len(frequency_sweep_results.results)):
    # Get the results from the ith experiment
```

```
    res = frequency_sweep_results.get_memory(i)*scale_factor
    # Get the results for `qubit` from this experiment
    sweep_values.append(res[qubit])
    sweep_values1.append(res[qubit])
sweep_values1 = np.real(sweep_values1)

plt.scatter(frequencies_GHz, np.real(sweep_values), color='black') # plot real␣
 ↪part of sweep values
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```

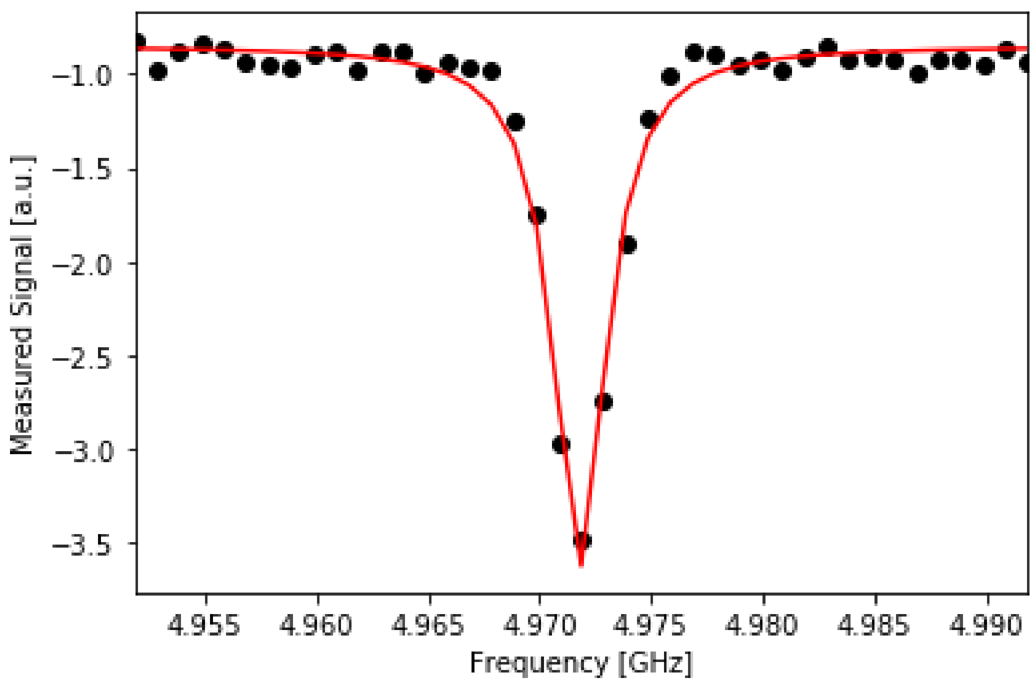


[70]:
```
from scipy.optimize import curve_fit

def fit_function(x_values, y_values, function, init_params):
    fitparams, conv = curve_fit(function, x_values, y_values, init_params)
    y_fit = function(x_values, *fitparams)

    return fitparams, y_fit
```

[71]:
```
fit_params, y_fit = fit_function(frequencies_GHz,
                                 np.real(sweep_values),
                                 lambda x, A, q_freq, B, C: (A / np.pi) * (B /␣
 ↪((x - q_freq)**2 + B**2)) + C,
```

```
                                        [-5, 4.975, 1, 5] # initial parameters for␣
↪curve_fit
                                        )
```

[72]:
```
plt.scatter(frequencies_GHz, np.real(sweep_values), color='black')
plt.plot(frequencies_GHz, y_fit, color='red')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])

plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")
plt.show()
```



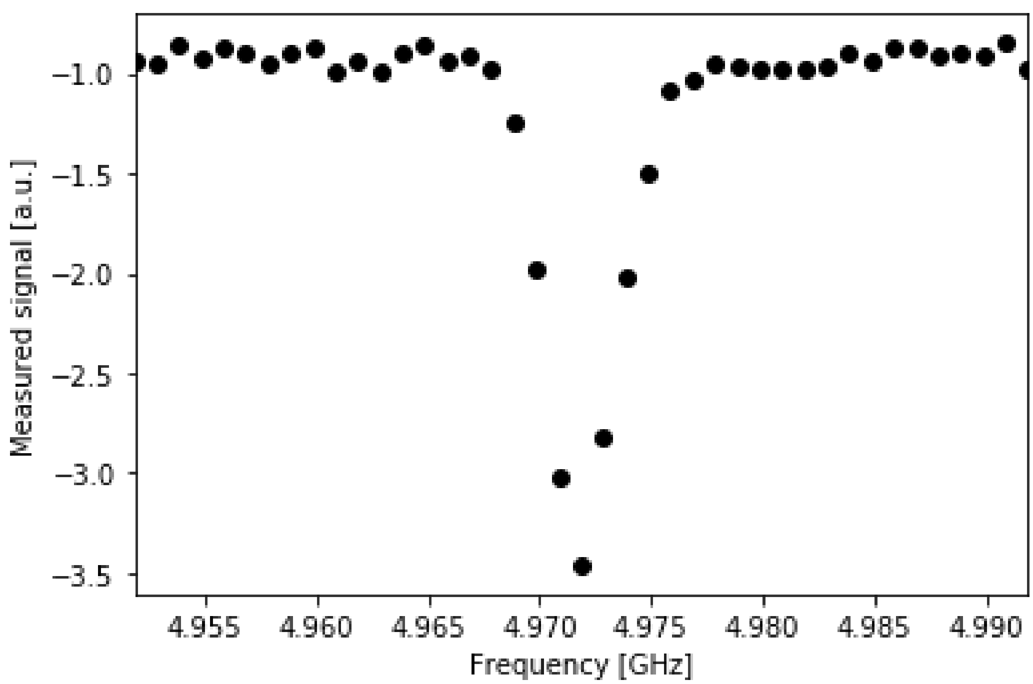
[73]:
```
A, rough_qubit_frequency1, B, C = fit_params
rough_qubit_frequency1 = rough_qubit_frequency1*GHz # make sure qubit freq is␣
↪in Hz
print(f"We've updated our qubit frequency estimate from "
      f"{round(backend_defaults.qubit_freq_est[qubit] / GHz, 5)} GHz to␣
↪{round(rough_qubit_frequency1/GHz, 5)} GHz.")
```

```
We've updated our qubit frequency estimate from 4.97186 GHz to 4.97181 GHz.
```

[74]:
```
fit_params1 = fit_params
```

[75]:
```
frequency_sweep_results = job2.result(timeout=120)
```

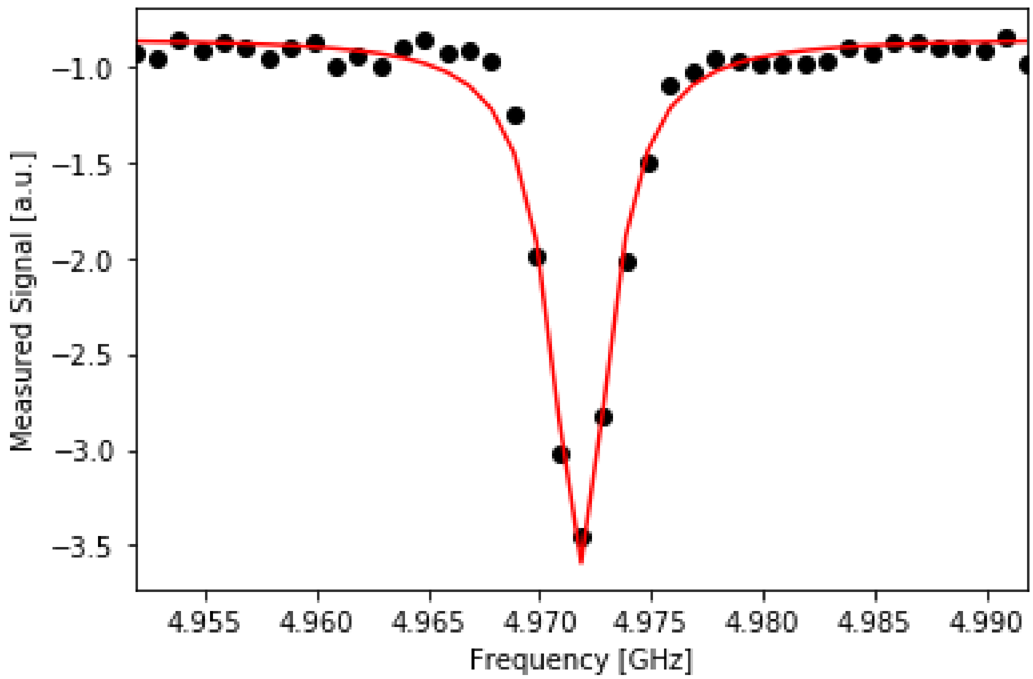
[76]:
```
import matplotlib.pyplot as plt

sweep_values = []
sweep_values2 = []
for i in range(len(frequency_sweep_results.results)):
    # Get the results from the ith experiment
    res = frequency_sweep_results.get_memory(i)*scale_factor
    # Get the results for `qubit` from this experiment
    sweep_values.append(res[qubit])
    sweep_values2.append(res[qubit])
sweep_values2 = np.real(sweep_values2)

plt.scatter(frequencies_GHz, np.real(sweep_values), color='black') # plot real␣
↪part of sweep values
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[77]:
```
fit_params, y_fit = fit_function(frequencies_GHz,
                                 np.real(sweep_values),
                                 lambda x, A, q_freq, B, C: (A / np.pi) * (B /␣
↪((x - q_freq)**2 + B**2)) + C,
                                 [-5, 4.972, 1, 5] # initial parameters for␣
↪curve_fit
```

```
                                    )
```

[78]:
```
plt.scatter(frequencies_GHz, np.real(sweep_values), color='black')
plt.plot(frequencies_GHz, y_fit, color='red')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])

plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")
plt.show()
```



[79]:
```
A, rough_qubit_frequency2, B, C = fit_params
rough_qubit_frequency2 = rough_qubit_frequency2*GHz # make sure qubit freq is␣
 ↪in Hz
print(f"We've updated our qubit frequency estimate from "
      f"{round(backend_defaults.qubit_freq_est[qubit] / GHz, 5)} GHz to␣
 ↪{round(rough_qubit_frequency2 / GHz, 5)} GHz.")
```

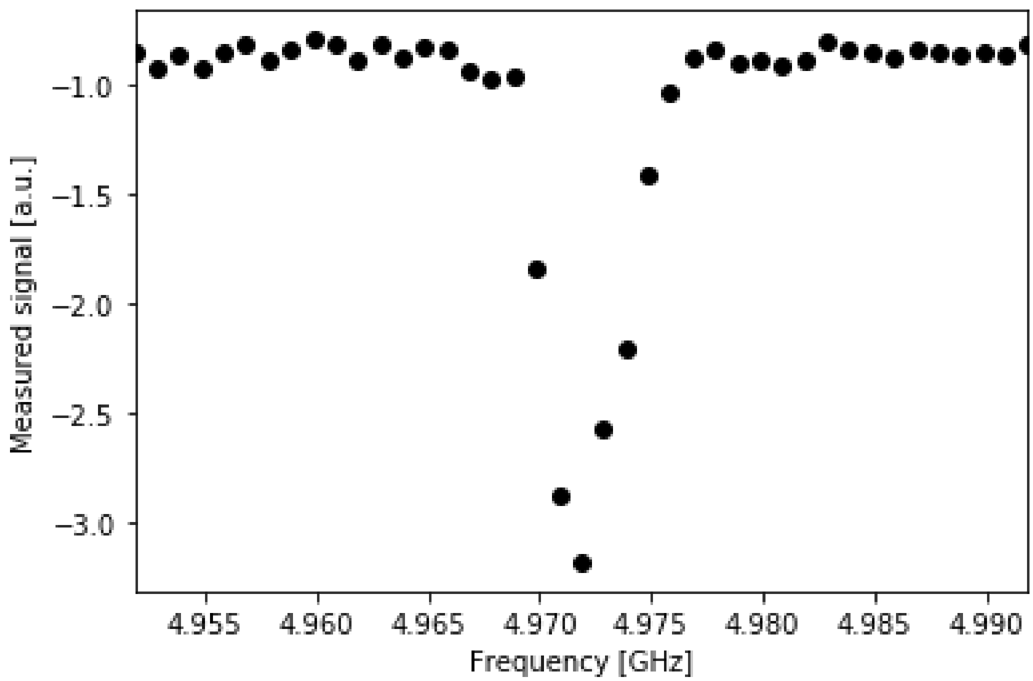
```
We've updated our qubit frequency estimate from 4.97186 GHz to 4.97184 GHz.
```

[80]:
```
fit_params2 = fit_params
```

[81]:
```
frequency_sweep_results = job3.result(timeout=120)
```

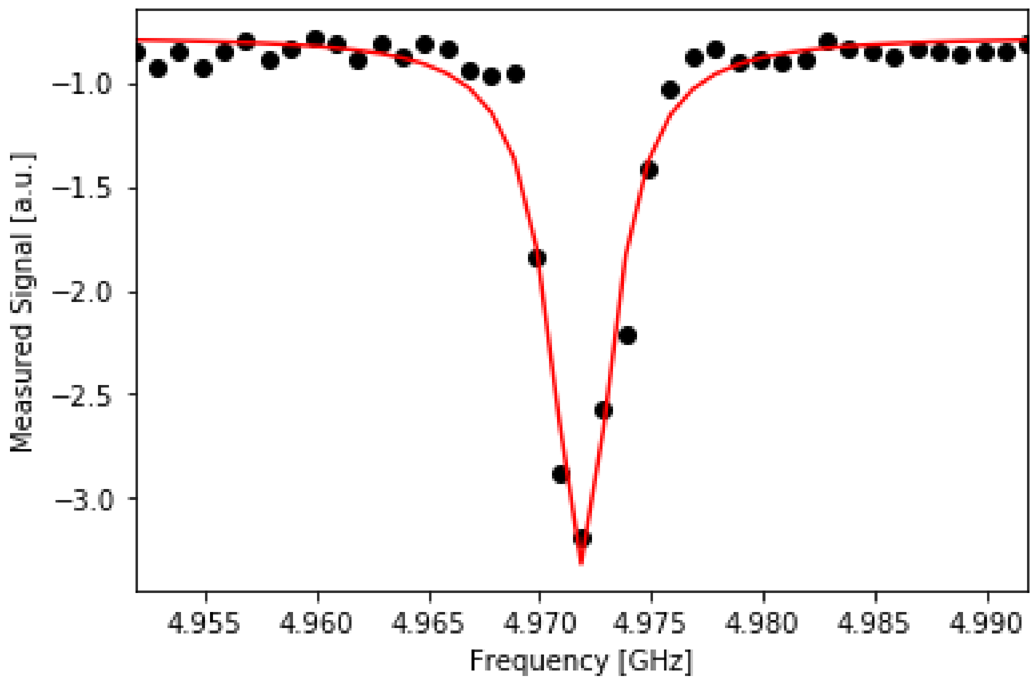
[82]:
```
import matplotlib.pyplot as plt

sweep_values = []
```

```
sweep_values3 = []
for i in range(len(frequency_sweep_results.results)):
    # Get the results from the ith experiment
    res = frequency_sweep_results.get_memory(i)*scale_factor
    # Get the results for `qubit` from this experiment
    sweep_values.append(res[qubit])
    sweep_values3.append(res[qubit])
sweep_values3 = np.real(sweep_values3)

plt.scatter(frequencies_GHz, np.real(sweep_values), color='black') # plot real␣
 ↪part of sweep values
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[83]:
```
fit_params, y_fit = fit_function(frequencies_GHz,
                                 np.real(sweep_values),
                                 lambda x, A, q_freq, B, C: (A / np.pi) * (B /␣
 ↪((x - q_freq)**2 + B**2)) + C,
                                 [-5, 4.972, 1, 5] # initial parameters for␣
 ↪curve_fit
                                )
```

[84]:
```
plt.scatter(frequencies_GHz, np.real(sweep_values), color='black')
plt.plot(frequencies_GHz, y_fit, color='red')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])

plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")
plt.show()
```



[85]:
```
A, rough_qubit_frequency2, B, C = fit_params
rough_qubit_frequency1 = rough_qubit_frequency1*GHz # make sure qubit freq is␣
↪in Hz
print(f"We've updated our qubit frequency estimate from "
      f"{round(backend_defaults.qubit_freq_est[qubit] / GHz, 5)} GHz to␣
↪{round(rough_qubit_frequency2, 5)} GHz.")
```

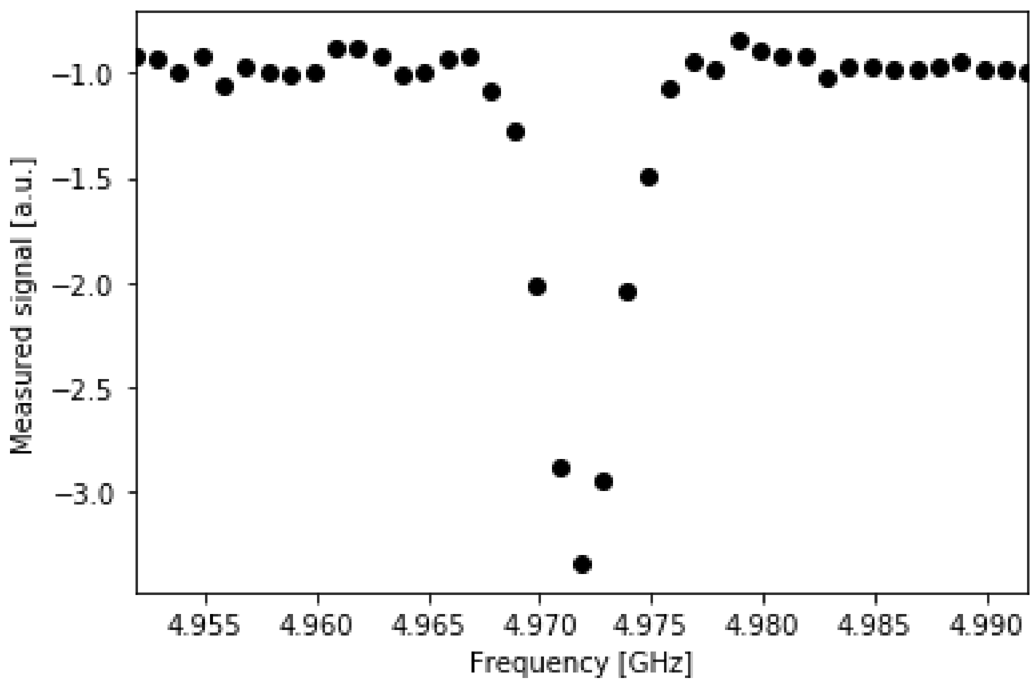
```
We've updated our qubit frequency estimate from 4.97186 GHz to 4.97189 GHz.
```

[86]:
```
fit_params3 = fit_params
```

[87]:
```
frequency_sweep_results = job4.result(timeout=120)
```

[88]:
```
import matplotlib.pyplot as plt

sweep_values = []
sweep_values4 = []
```

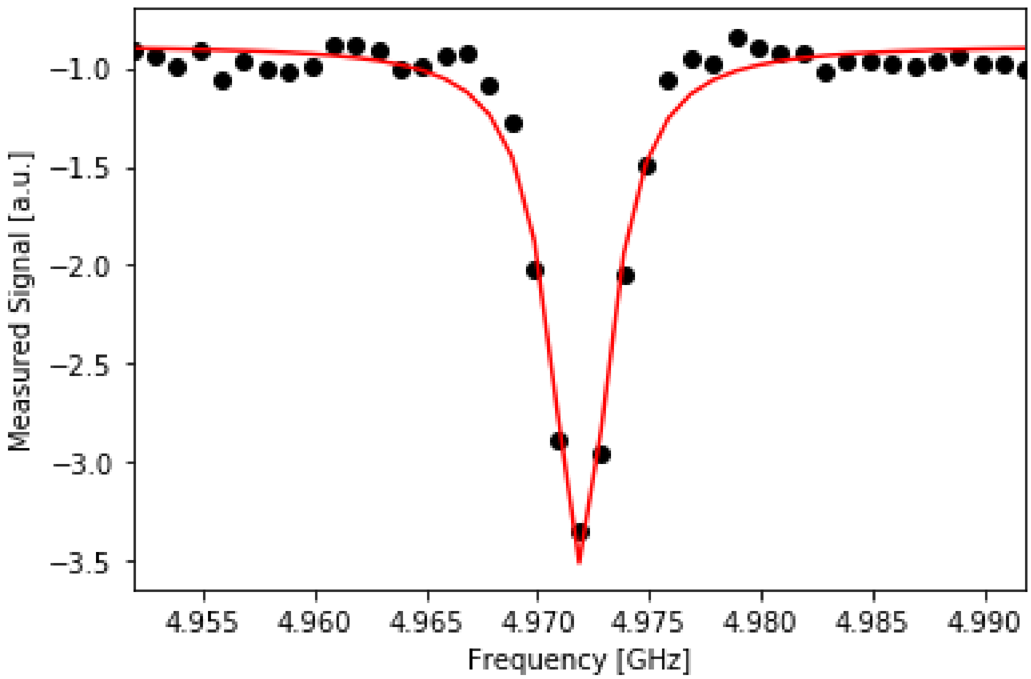
```
for i in range(len(frequency_sweep_results.results)):
    # Get the results from the ith experiment
    res = frequency_sweep_results.get_memory(i)*scale_factor
    # Get the results for `qubit` from this experiment
    sweep_values.append(res[qubit])
    sweep_values4.append(res[qubit])
sweep_values4 = np.real(sweep_values4)

plt.scatter(frequencies_GHz, np.real(sweep_values), color='black') # plot real␣
 ↪part of sweep values
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[89]:
```
fit_params, y_fit = fit_function(frequencies_GHz,
                                 np.real(sweep_values),
                                 lambda x, A, q_freq, B, C: (A / np.pi) * (B /␣
 ↪((x - q_freq)**2 + B**2)) + C,
                                 [-5, 4.972, 1, 5] # initial parameters for␣
 ↪curve_fit
                                )
```

[90]:
```
plt.scatter(frequencies_GHz, np.real(sweep_values), color='black')
plt.plot(frequencies_GHz, y_fit, color='red')
```

```
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])

plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")
plt.show()
```



[91]:
```
A, rough_qubit_frequency2, B, C = fit_params
rough_qubit_frequency2 = rough_qubit_frequency2*GHz # make sure qubit freq is␣
↪in Hz
print(f"We've updated our qubit frequency estimate from "
      f"{round(backend_defaults.qubit_freq_est[qubit] / GHz, 5)} GHz to␣
↪{round(rough_qubit_frequency2 / GHz, 5)} GHz.")
```

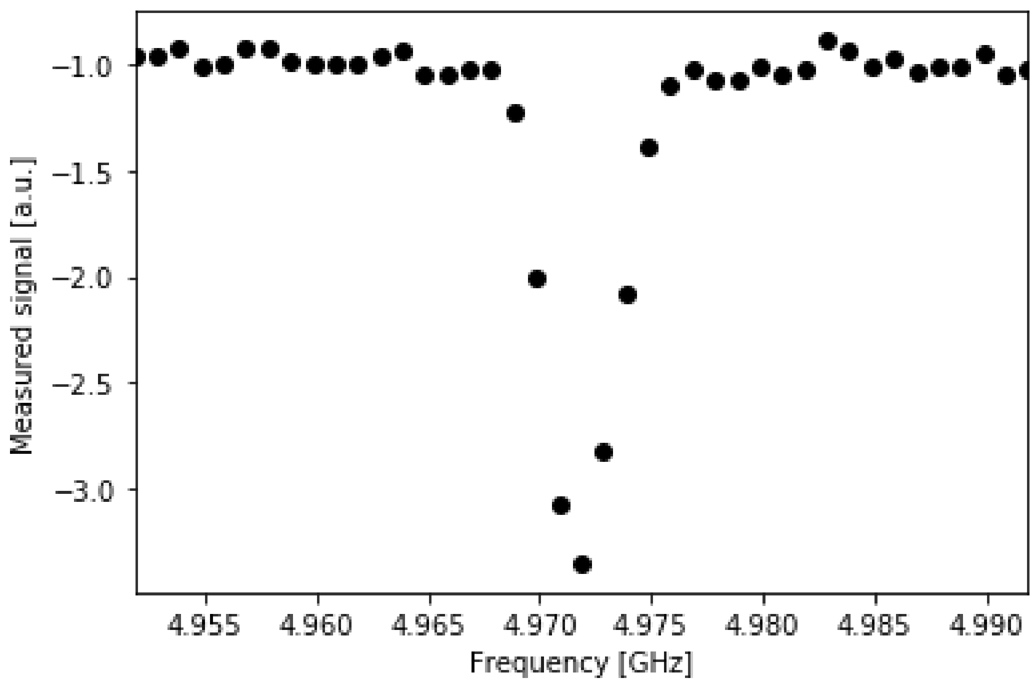
```
We've updated our qubit frequency estimate from 4.97186 GHz to 4.97191 GHz.
```

[92]:
```
fit_params4 = fit_params
```

[93]:
```
frequency_sweep_results = job5.result(timeout=120)
```

[94]:
```
import matplotlib.pyplot as plt

sweep_values = []
sweep_values5 = []
for i in range(len(frequency_sweep_results.results)):
    # Get the results from the ith experiment
```

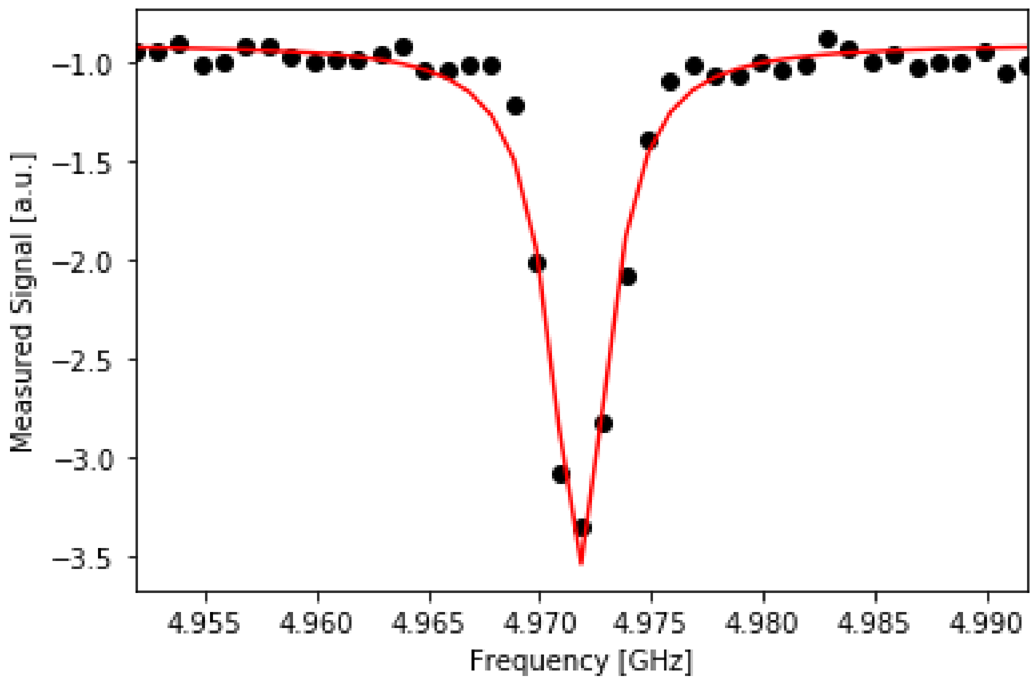
```
    res = frequency_sweep_results.get_memory(i)*scale_factor
    # Get the results for `qubit` from this experiment
    sweep_values.append(res[qubit])
    sweep_values5.append(res[qubit])
sweep_values5 = np.real(sweep_values5)

plt.scatter(frequencies_GHz, np.real(sweep_values), color='black') # plot real␣
 ↪part of sweep values
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[95]:
```
fit_params, y_fit = fit_function(frequencies_GHz,
                                 np.real(sweep_values),
                                 lambda x, A, q_freq, B, C: (A / np.pi) * (B /␣
 ↪((x - q_freq)**2 + B**2)) + C,
                                 [-5, 4.972, 1, 5] # initial parameters for␣
 ↪curve_fit
                                )
```

[96]:
```
plt.scatter(frequencies_GHz, np.real(sweep_values), color='black')
plt.plot(frequencies_GHz, y_fit, color='red')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
```

```
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")
plt.show()
```



[97]:
```
A, rough_qubit_frequency2, B, C = fit_params
rough_qubit_frequency1 = rough_qubit_frequency1*GHz # make sure qubit freq is␣
↪in Hz
print(f"We've updated our qubit frequency estimate from "
      f"{round(backend_defaults.qubit_freq_est[qubit] / GHz, 5)} GHz to␣
↪{round(rough_qubit_frequency2, 5)} GHz.")
```

```
We've updated our qubit frequency estimate from 4.97186 GHz to 4.97181 GHz.
```

[98]:
```
fit_params5 = fit_params
```

[99]:
```
fit_params
```

[99]:
```
array([-1.29720976e-02,  4.97180773e+00,  1.57209146e-03, -9.14074696e-01])
```

[100]:
```
fit_params1[1]
```

[100]:
```
4.97180721713975
```

[101]:
```
fit_params2[1]
```

[101]:
```
4.971839052436073
```

[102]:
```
fit_params3[1]
```

[102]: 4.971891650918128

[103]:
```
fit_params4[1]
```

[103]: 4.971912583987321

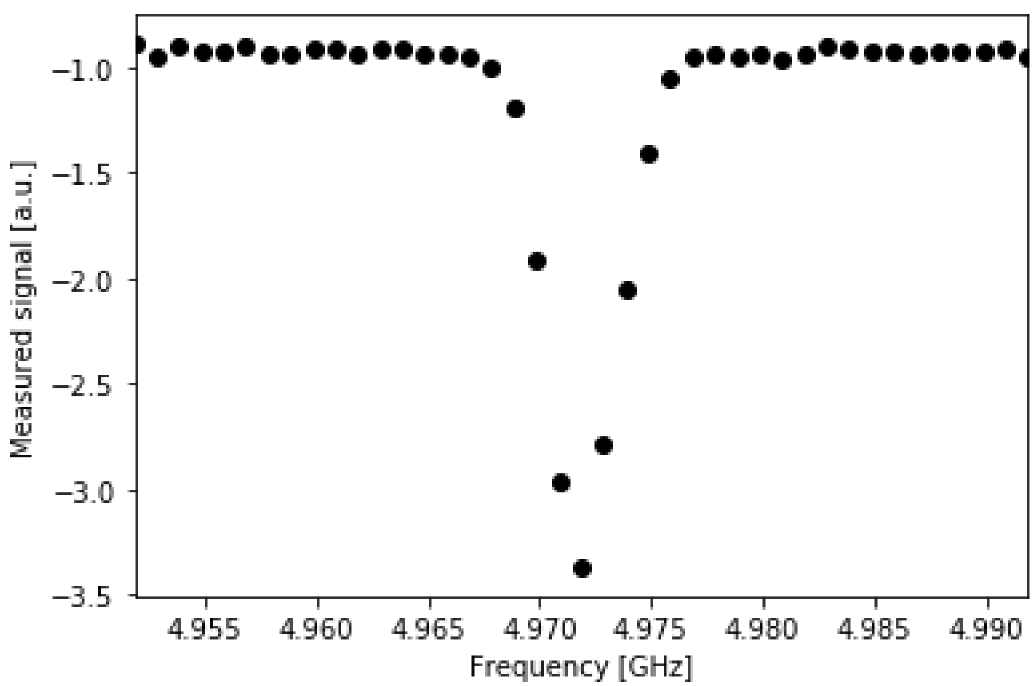
[104]:
```
fit_params5[1]
```

[104]: 4.971807727137528

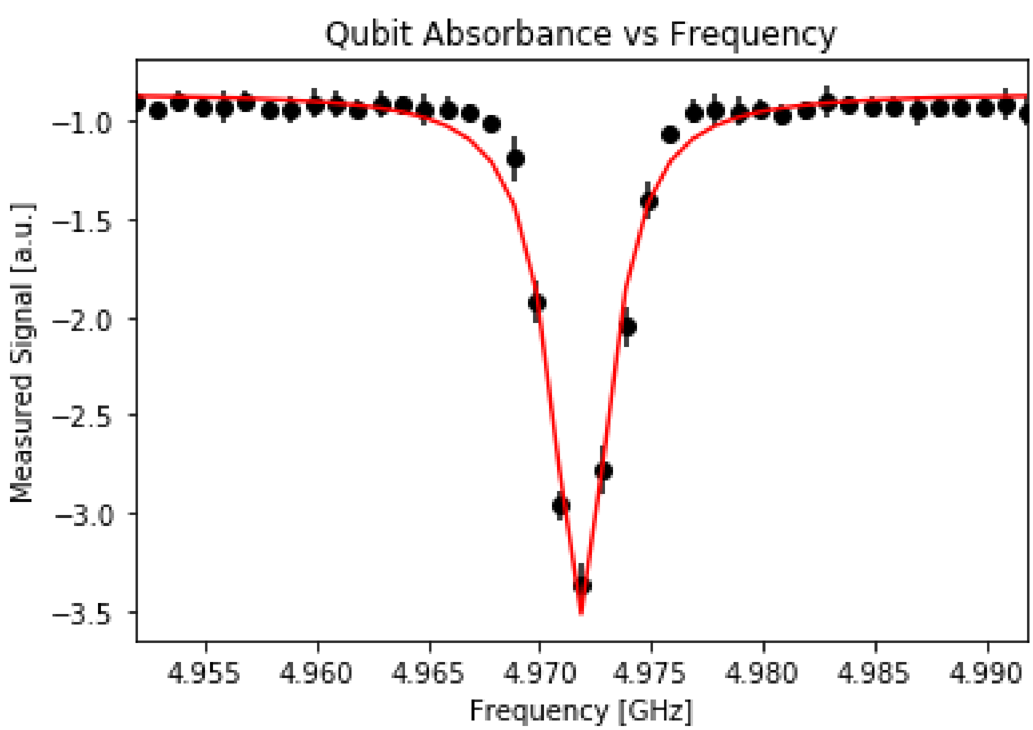
[105]:
```
rough_mean_vals = np.mean([sweep_values1, sweep_values2, sweep_values3,␣
↪sweep_values4, sweep_values5], axis=0)
```

[107]:
```
fit_params, y_fit = fit_function(frequencies_GHz,
                                  rough_mean_vals,
                                  lambda x, A, q_freq, B, C: (A / np.pi) * (B /␣
↪((x - q_freq)**2 + B**2)) + C,
                                  [-5, 4.972, 1, 5] # initial parameters for␣
↪curve_fit
                                 )
```

[106]:
```
plt.scatter(frequencies_GHz, rough_mean_vals, color='black') # plot real part␣
↪of sweep values
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])
plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[134]:
```
plt.scatter(frequencies_GHz, rough_mean_vals, color='black')

plt.xlabel("Frequency [GHz]")
plt.ylabel("Measured Signal [a.u.]")

arr = np.vstack((sweep_values1, sweep_values2, sweep_values3, sweep_values4,␣
 ↪sweep_values5))

yerr = np.std(arr,axis=0)

plt.errorbar(frequencies_GHz, rough_mean_vals, yerr=yerr, color='black', ls =␣
 ↪'none')

plt.plot(frequencies_GHz, y_fit, color='red')
plt.xlim([min(frequencies_GHz), max(frequencies_GHz)])

plt.title("Qubit Absorbance vs Frequency")

plt.show()
```



[126]:
```
avg_freq = fit_params[1]
```

[127]:
```
#avg_freq = (fit_params1[1] + fit_params2[1] + fit_params3[1] + fit_params4[1]␣
 ↪+ fit_params5[1])/5
```

[128]:
```
avg_freq
```

[128]: 4.971848666925875

[129]:
```
freq_std = np.sqrt(((fit_params1[1] - avg_freq)**2 +␣
↪(fit_params2[1]-avg_freq)**2 + (fit_params3[1] - avg_freq)**2 +␣
↪(fit_params4[1] - avg_freq)**2 + (fit_params5[1] - avg_freq)**2)/5)
```

[130]:
```
freq_std
```

[130]: 4.340415599402016e-05

[ ]:
```
#rough qubit frequency is over. Take avg_freq and
```

[ ]:
```
#4.97184 +/- 0.00006 GHz
```

[142]:
```
rough_qubit_frequency = avg_freq * GHz
```

[143]:
```
rough_qubit_frequency
```

[143]: 4971848666.925875

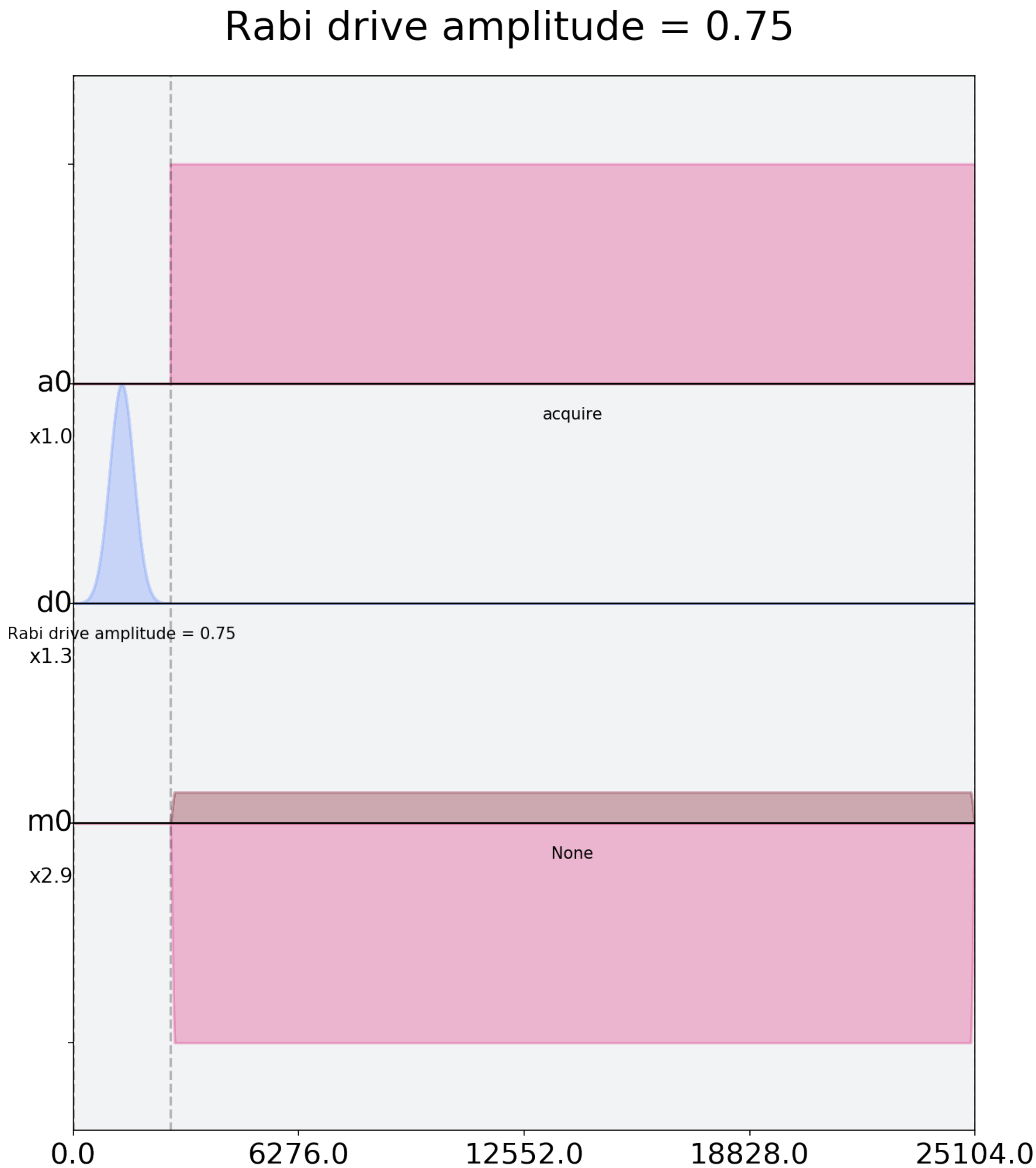
[137]:
```
#90 and pi pulses!!!!!
```

[138]:
```
# This experiment uses these values from the previous experiment:
    # `qubit`,
    # `measure`, and
    # `rough_qubit_frequency`.

# Rabi experiment parameters
num_rabi_points = 50

# Drive amplitude values to iterate over: 50 amplitudes evenly spaced from 0 to␣
↪0.75
drive_amp_min = 0
drive_amp_max = 0.75
drive_amps = np.linspace(drive_amp_min, drive_amp_max, num_rabi_points)
```

[139]:
```
# Build the Rabi experiments:
#    A drive pulse at the qubit frequency, followed by a measurement,
#    where we vary the drive amplitude each time.
rabi_schedules = []
for drive_amp in drive_amps:
    rabi_pulse = pulse_lib.gaussian(duration=drive_samples, amp=drive_amp,
                                    sigma=drive_sigma, name=f"Rabi drive␣
↪amplitude = {drive_amp}")
    this_schedule = pulse.Schedule(name=f"Rabi drive amplitude = {drive_amp}")
    this_schedule += Play(rabi_pulse, drive_chan)
    # Reuse the measure instruction from the frequency sweep experiment
    this_schedule += measure << this_schedule.duration
    rabi_schedules.append(this_schedule)
```

[140]:
```
rabi_schedules[-1].draw(label=True)
```

[140]:



[144]:
```
# Assemble the schedules into a Qobj
num_shots_per_point = 1024

rabi_experiment_program = assemble(rabi_schedules,
                                   backend=backend,
                                   meas_level=1,
                                   meas_return='avg',
                                   shots=num_shots_per_point,
```

```
                                    schedule_los=[{drive_chan:␣
↪rough_qubit_frequency}]
                                                 * num_rabi_points)
```

[145]:
```
print(job.job_id())
job = backend.run(rabi_experiment_program)
job_monitor(job)
```

```
6098f7e2eef9e1aeb418cf21
Job Status: job has successfully run
```

[146]:
```
rabi_results1 = job.result(timeout=120)
```

[147]:
```
print(job.job_id())
job = backend.run(rabi_experiment_program)
job_monitor(job)
```

```
60990302c3a1cd8b9527287d
Job Status: job has successfully run
```

[148]:
```
rabi_results2 = job.result(timeout=120)
```

[149]:
```
print(job.job_id())
job = backend.run(rabi_experiment_program)
job_monitor(job)
```

```
609905ff7ccc8bbb62281cf7
Job Status: job has successfully run
```

[150]:
```
rabi_results3 = job.result(timeout=120)
```

[151]:
```
print(job.job_id())
job = backend.run(rabi_experiment_program)
job_monitor(job)
```

```
6099064a5411236b3a3f5375
Job Status: job has successfully run
```

[152]:
```
rabi_results4 = job.result(timeout=120)
```

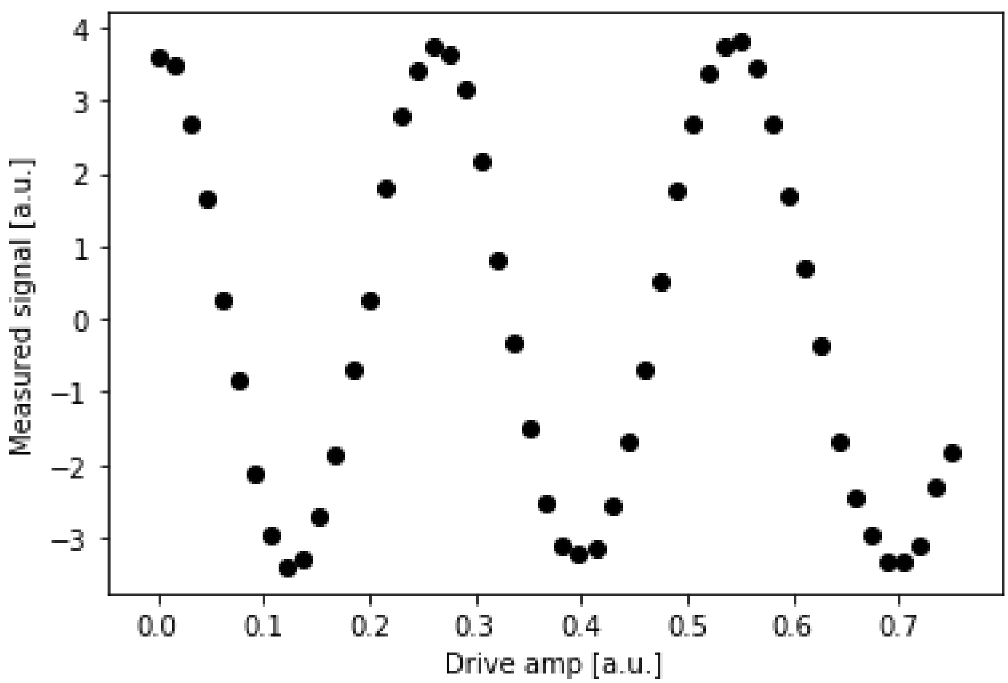
[153]:
```
print(job.job_id())
job = backend.run(rabi_experiment_program)
job_monitor(job)
```

```
60990695541123 83f53f5377
Job Status: job has successfully run
```

[154]:
```
rabi_results5 = job.result(timeout=120)
```

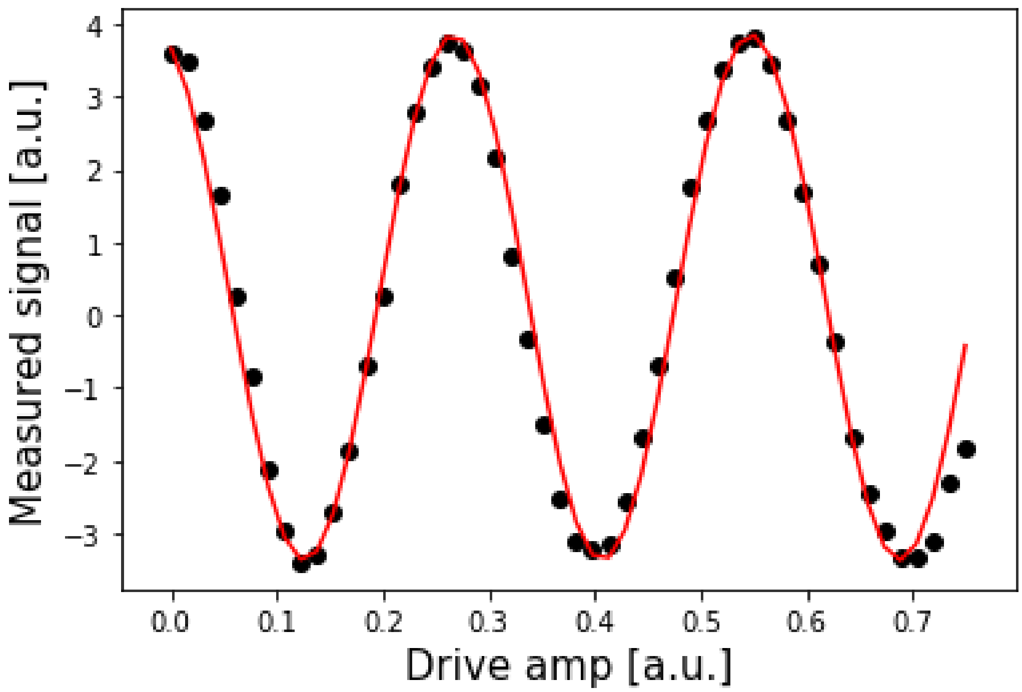
[155]:
```
# center data around 0
def baseline_remove(values):
    return np.array(values) - np.mean(values)
```

[156]:
```
rabi_values1 = []
for i in range(num_rabi_points):
    # Get the results for `qubit` from the ith experiment
    rabi_values1.append(rabi_results1.get_memory(i)[qubit]*scale_factor)

rabi_values1 = np.real(baseline_remove(rabi_values1))

plt.xlabel("Drive amp [a.u.]")
plt.ylabel("Measured signal [a.u.]")
plt.scatter(drive_amps, rabi_values1, color='black') # plot real part of Rabi␣
 ↪values
plt.show()
```



[157]:
```
fit_params, y_fit = fit_function(drive_amps,
                                 rabi_values1,
                                 lambda x, A, B, drive_period, phi: (A*np.
 ↪cos(2*np.pi*x/drive_period - phi) + B),
                                 [3, 0.1, 0.4, 0])

plt.scatter(drive_amps, rabi_values1, color='black')
```

```
plt.plot(drive_amps, y_fit, color='red')

drive_period = fit_params[2] # get period of rabi oscillation


plt.xlabel("Drive amp [a.u.]", fontsize=15)
plt.ylabel("Measured signal [a.u.]", fontsize=15)
plt.show()
```

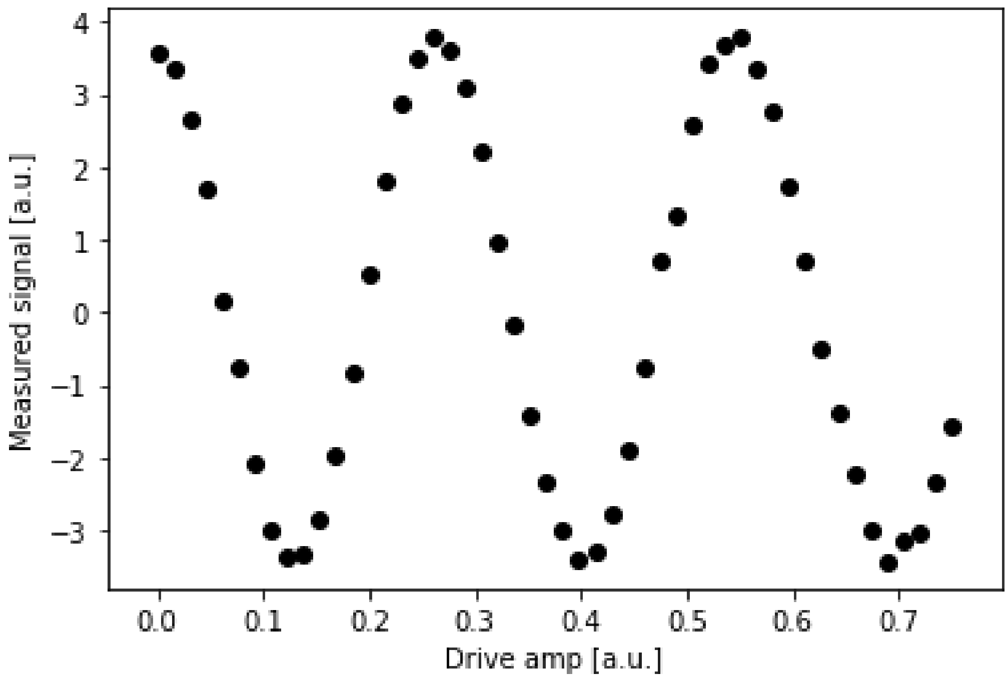


[158]:
```
amp1 = abs(drive_period / 2)
print(f"Pi Amplitude = {amp1}")
```

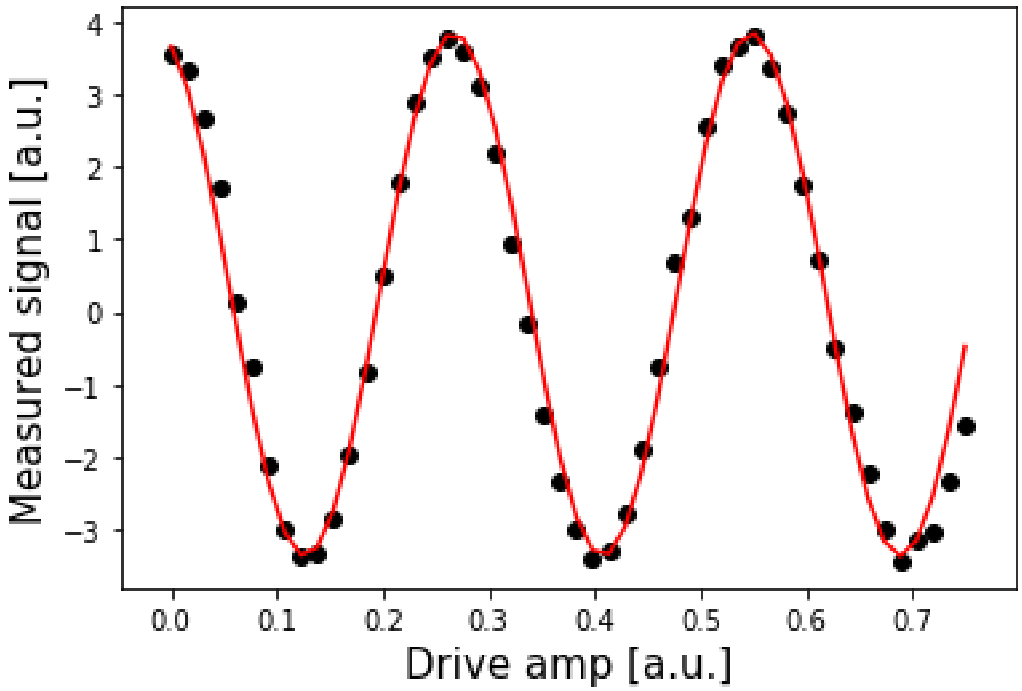
```
Pi Amplitude = 0.14051706708510328
```

[159]:
```
rabi_values2 = []
for i in range(num_rabi_points):
    # Get the results for `qubit` from the ith experiment
    rabi_values2.append(rabi_results2.get_memory(i)[qubit]*scale_factor)

rabi_values2 = np.real(baseline_remove(rabi_values2))

plt.xlabel("Drive amp [a.u.]")
plt.ylabel("Measured signal [a.u.]")
```

```
plt.scatter(drive_amps, rabi_values2, color='black') # plot real part of Rabi␣
↪values
plt.show()
```



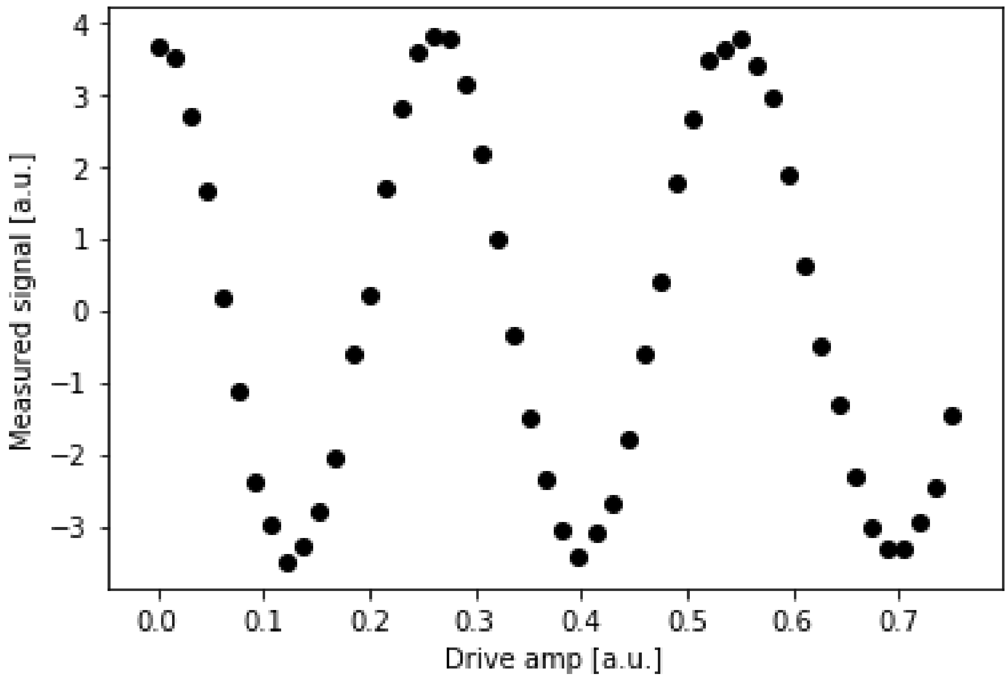
[160]:
```
fit_params, y_fit = fit_function(drive_amps,
                                 rabi_values2,
                                 lambda x, A, B, drive_period, phi: (A*np.
↪cos(2*np.pi*x/drive_period - phi) + B),
                                 [3, 0.1, 0.4, 0])

plt.scatter(drive_amps, rabi_values2, color='black')
plt.plot(drive_amps, y_fit, color='red')

drive_period = fit_params[2] # get period of rabi oscillation


plt.xlabel("Drive amp [a.u.]", fontsize=15)
plt.ylabel("Measured signal [a.u.]", fontsize=15)
plt.show()
```



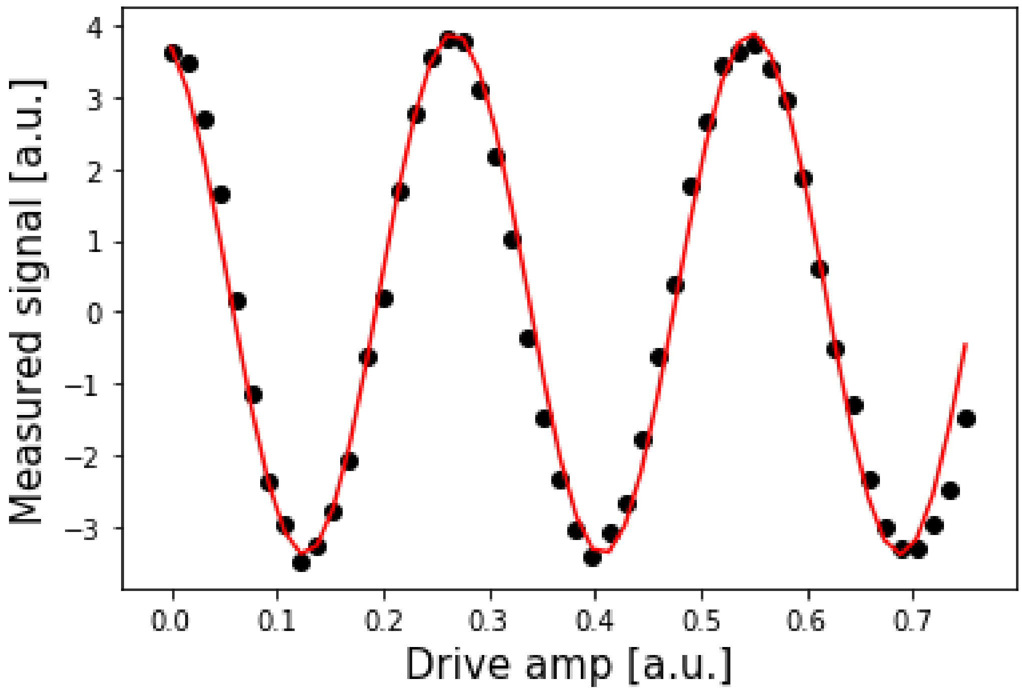
[161]:
```
amp2 = abs(drive_period / 2)
print(f"Pi Amplitude = {amp2}")
```

```
Pi Amplitude = 0.14060554394695005
```

[162]:
```
rabi_values3 = []
for i in range(num_rabi_points):
    # Get the results for `qubit` from the ith experiment
    rabi_values3.append(rabi_results3.get_memory(i)[qubit]*scale_factor)

rabi_values3 = np.real(baseline_remove(rabi_values3))

plt.xlabel("Drive amp [a.u.]")
plt.ylabel("Measured signal [a.u.]")
plt.scatter(drive_amps, rabi_values3, color='black') # plot real part of Rabi␣
↪values
plt.show()
```



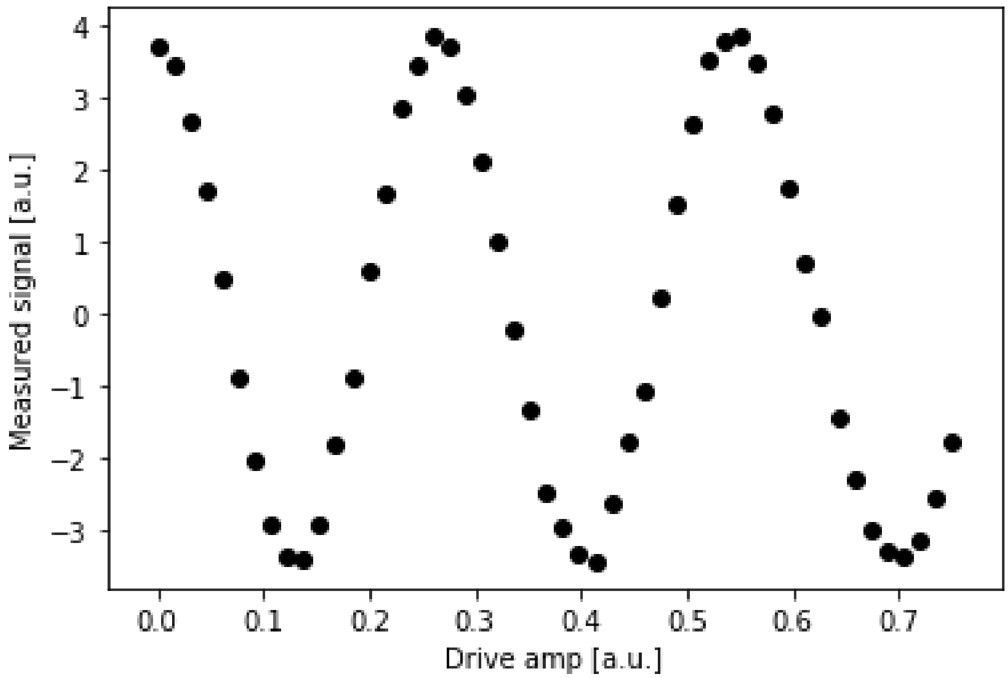
[163]:
```
fit_params, y_fit = fit_function(drive_amps,
                                 rabi_values3,
                                 lambda x, A, B, drive_period, phi: (A*np.
 ↪cos(2*np.pi*x/drive_period - phi) + B),
                                 [3, 0.1, 0.4, 0])

plt.scatter(drive_amps, rabi_values3, color='black')
plt.plot(drive_amps, y_fit, color='red')

drive_period = fit_params[2] # get period of rabi oscillation


plt.xlabel("Drive amp [a.u.]", fontsize=15)
plt.ylabel("Measured signal [a.u.]", fontsize=15)
plt.show()
```



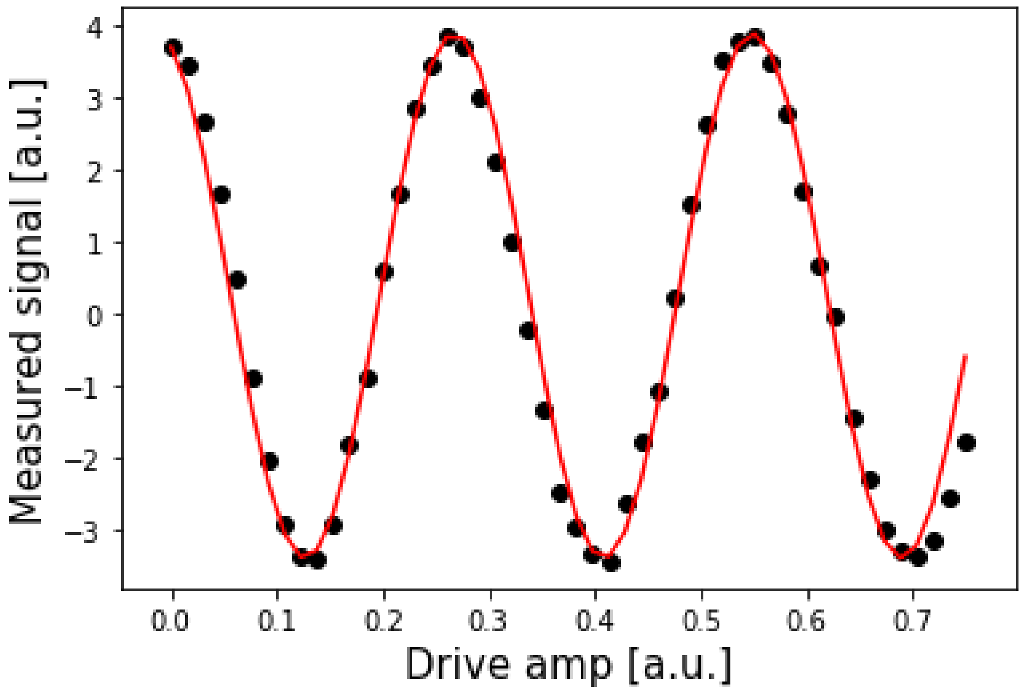
[164]:
```
amp3 = abs(drive_period / 2)
print(f"Pi Amplitude = {amp3}")
```

```
Pi Amplitude = 0.14068012121747528
```

[165]:
```
rabi_values4 = []
for i in range(num_rabi_points):
    # Get the results for `qubit` from the ith experiment
    rabi_values4.append(rabi_results4.get_memory(i)[qubit]*scale_factor)

rabi_values4 = np.real(baseline_remove(rabi_values4))

plt.xlabel("Drive amp [a.u.]")
plt.ylabel("Measured signal [a.u.]")
plt.scatter(drive_amps, rabi_values4, color='black') # plot real part of Rabi␣
 ↪values
plt.show()
```



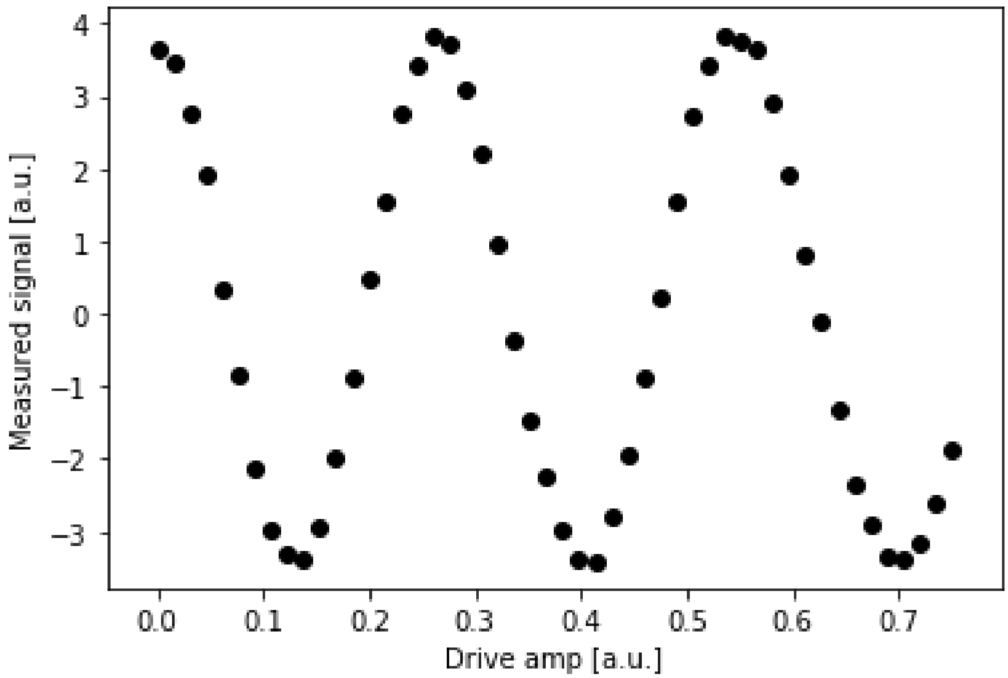
[166]:
```
fit_params, y_fit = fit_function(drive_amps,
                                 rabi_values4,
                                 lambda x, A, B, drive_period, phi: (A*np.
 ↪cos(2*np.pi*x/drive_period - phi) + B),
                                 [3, 0.1, 0.4, 0])

plt.scatter(drive_amps, rabi_values4, color='black')
plt.plot(drive_amps, y_fit, color='red')

drive_period = fit_params[2] # get period of rabi oscillation


plt.xlabel("Drive amp [a.u.]", fontsize=15)
plt.ylabel("Measured signal [a.u.]", fontsize=15)
plt.show()
```



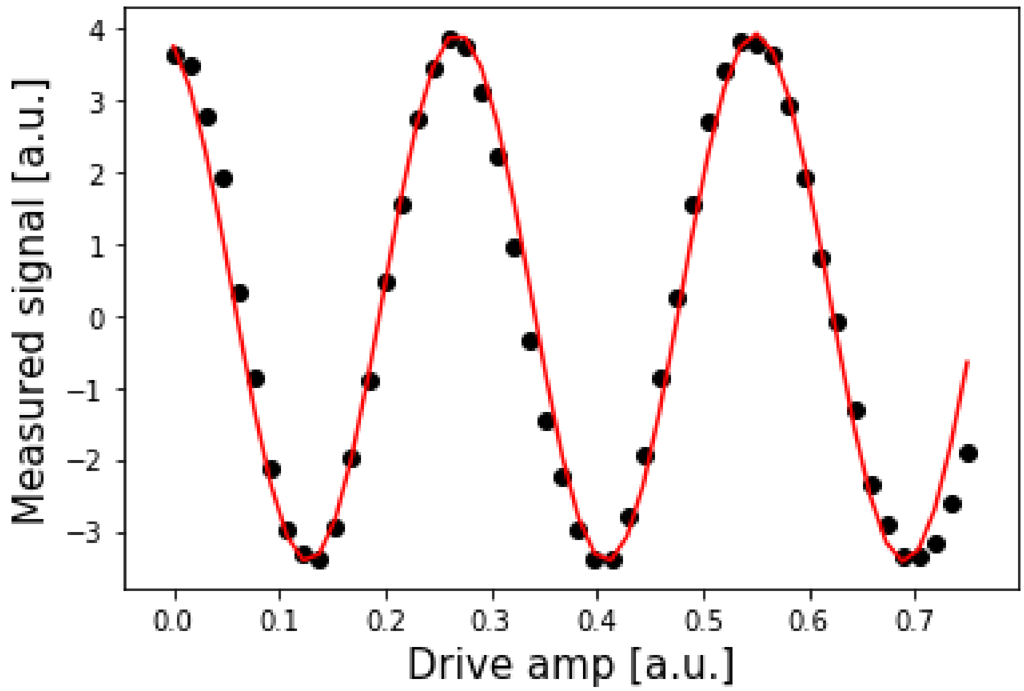
[167]:
```
amp4 = abs(drive_period / 2)
print(f"Pi Amplitude = {amp4}")
```

```
Pi Amplitude = 0.14086619307222828
```

[168]:
```
rabi_values5 = []
for i in range(num_rabi_points):
    # Get the results for `qubit` from the ith experiment
    rabi_values5.append(rabi_results5.get_memory(i)[qubit]*scale_factor)

rabi_values5 = np.real(baseline_remove(rabi_values5))

plt.xlabel("Drive amp [a.u.]")
plt.ylabel("Measured signal [a.u.]")
plt.scatter(drive_amps, rabi_values5, color='black') # plot real part of Rabi␣
 ↪values
plt.show()
```



[177]:
```
fit_params, y_fit = fit_function(drive_amps,
                                 rabi_values5,
                                 lambda x, A, B, drive_period, phi: (A*np.
 ↪cos(2*np.pi*x/drive_period - phi) + B),
                                 [3, 0.1, 0.3, 0])

plt.scatter(drive_amps, rabi_values5, color='black')
plt.plot(drive_amps, y_fit, color='red')

drive_period = fit_params[2] # get period of rabi oscillation


plt.xlabel("Drive amp [a.u.]", fontsize=15)
plt.ylabel("Measured signal [a.u.]", fontsize=15)
plt.show()
```



[178]:
```
amp5 = abs(drive_period / 2)
print(f"Pi Amplitude = {amp5}")
```

```
Pi Amplitude = 0.14090025466819067
```

[179]:
```
avg_amp = (amp1 + amp2 + amp3 + amp4 + amp5) / 5
```

[180]:
```
avg_amp
```

[180]: 0.1407138359979895

[181]:
```
amp_std = np.sqrt(((amp1 - avg_amp)**2 + (amp2 - avg_amp)**2 + (amp3 -␣
 ↪avg_amp)**2 + (amp4 - avg_amp)**2 + (amp5 - avg_amp)**2) / 5)
```

[182]:
```
amp_std
```

[182]: 0.00014801787627989575

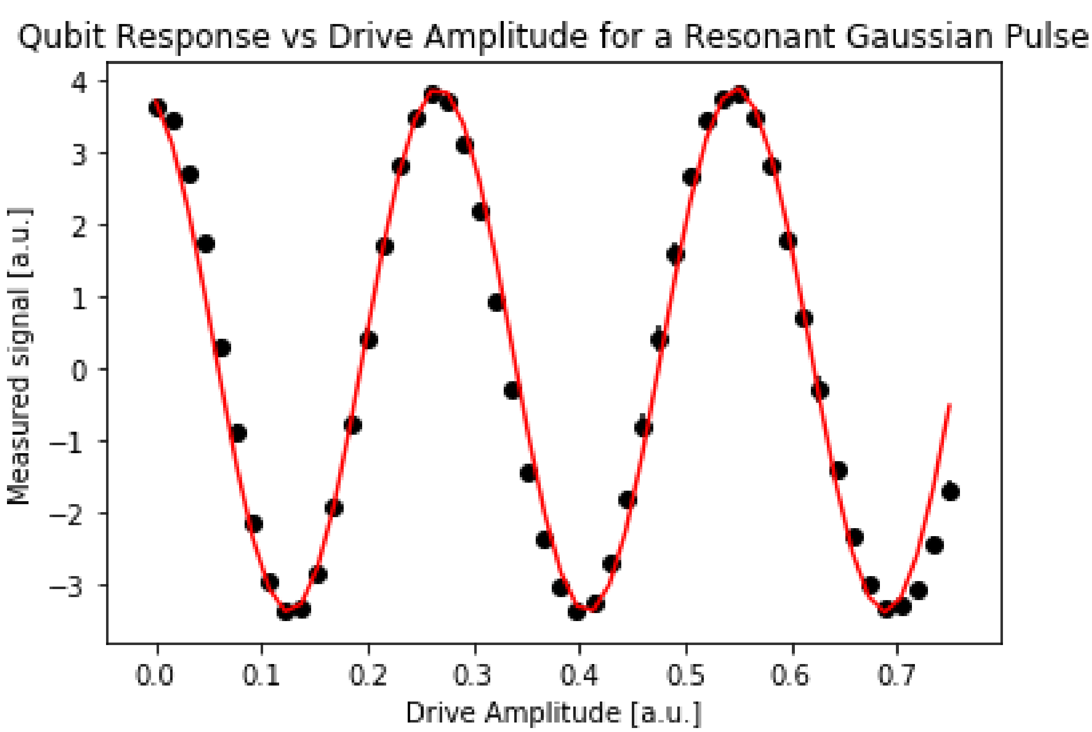
[183]:
```
rabi_values_mean = np.mean([rabi_values1, rabi_values2, rabi_values3,␣
 ↪rabi_values4, rabi_values5], axis = 0)
```

[184]:
```
rabi_stds = np.std([rabi_values1, rabi_values2, rabi_values3, rabi_values4,␣
 ↪rabi_values5], axis = 0)
```

[607]:
```
fit_params, y_fit = fit_function(drive_amps,
                                 rabi_values_mean,
                                 lambda x, A, B, drive_period, phi: (A*np.
 ↪cos(2*np.pi*x/drive_period - phi) + B),
```

```
                                [3, 0.1, 0.3, 0])

plt.xlabel("Drive Amplitude [a.u.]")
plt.ylabel("Measured signal [a.u.]")
plt.scatter(drive_amps, rabi_values_mean, color='black') # plot real part of␣
 ↪Rabi values

print(len(frequencies_GHz))

print(len(rabi_values_mean))

yerr = [3] * len(drive_amps)

plt.errorbar(drive_amps, rabi_values_mean, yerr=rabi_stds, color = 'black', ls␣
 ↪= 'none')

plt.plot(drive_amps, y_fit, color = 'red')

plt.title("Qubit Response vs Drive Amplitude for a Resonant Gaussian Pulse")

plt.show()
```

```
41
50
```



[608]:
```
avg_amp = abs(drive_period / 2)
print(f"Pi Amplitude = {amp5}")
```

```
Pi Amplitude = 0.14090025466819067
```

[609]:
```
amp_std = np.sqrt(((amp1 - avg_amp)**2 + (amp2 - avg_amp)**2 + (amp3 -␣
↪avg_amp)**2 + (amp4 - avg_amp)**2 + (amp5 - avg_amp)**2) / 5)
```

[610]:
```
amp_std
```

[610]:
```
0.0002380361575433018
```

[201]:
```
#avg_amp = 0.14072 +/- (9 * 10^-5)
```

[202]:
```
#avg_amp = 0.140720
```

[203]:
```
#define pi pulse
```

[204]:
```
pi_pulse = pulse_lib.gaussian(duration=drive_samples,
                              amp=avg_amp,
                              sigma=drive_sigma,
                              name='pi_pulse')
```

[205]:
```
x90_pulse = pulse_lib.gaussian(duration=drive_samples,
                              amp=avg_amp / 2,
                              sigma=drive_sigma,
                              name='x90_pulse')
```

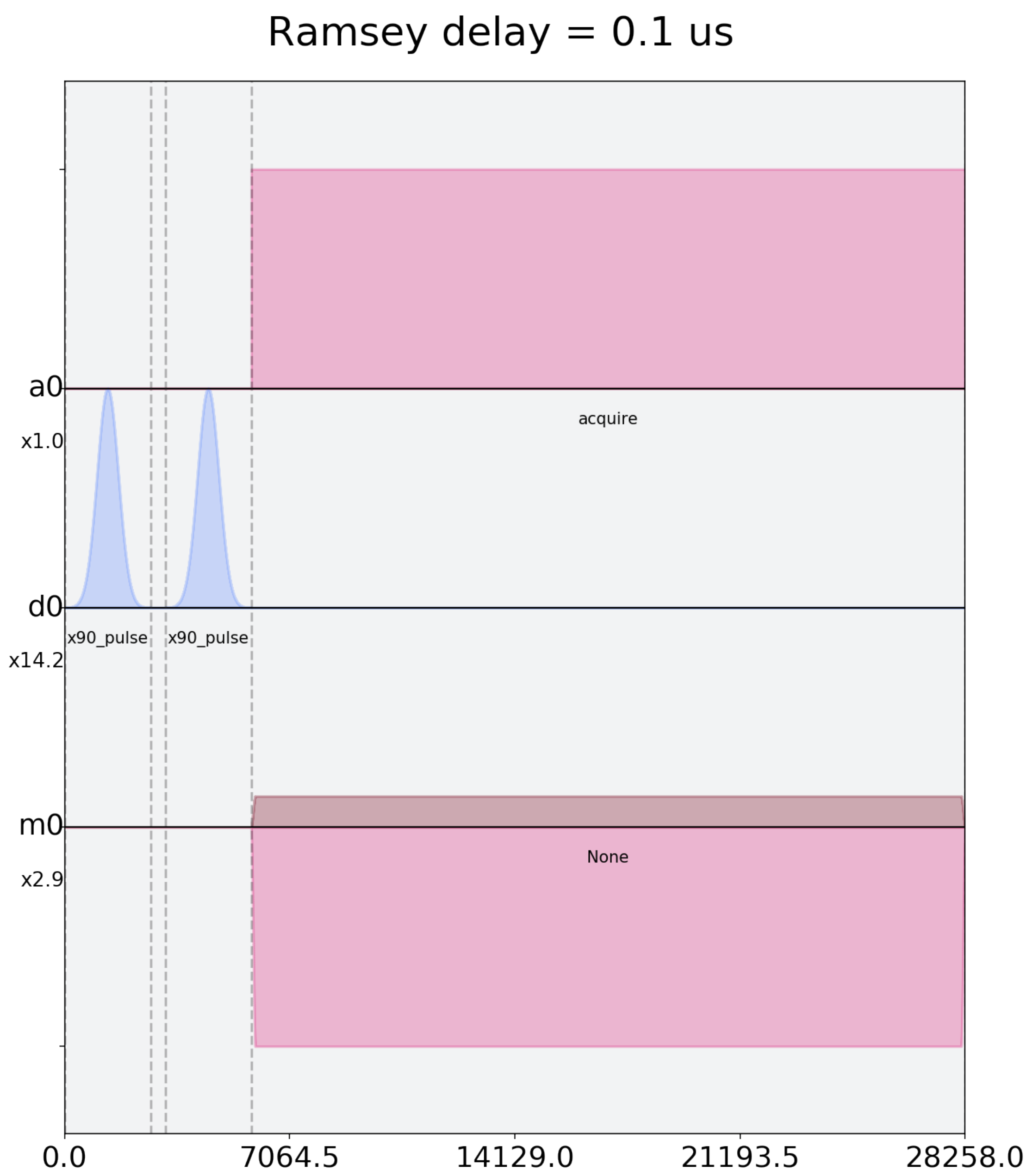
[206]:
```
#now we do a ramsey experiment to get a more precise qubit measurement
```

[618]:
```
# Ramsey experiment parameters
time_max_us = 1.8
time_step_us = 0.025
times_us = np.arange(0.1, time_max_us, time_step_us)
# Convert to units of dt
delay_times_dt = times_us * us / dt
```

[208]:
```
ramsey_schedules = []

for delay in delay_times_dt:
    this_schedule = pulse.Schedule(name=f"Ramsey delay = {delay * dt / us} us")
    this_schedule |= Play(x90_pulse, drive_chan)
    this_schedule |= Play(x90_pulse, drive_chan) << int(this_schedule.duration␣
↪+ delay)
    this_schedule |= measure << int(this_schedule.duration)

    ramsey_schedules.append(this_schedule)
```

[209]:
```
ramsey_schedules[0].draw(label=True)
```

[209]:



[210]:
```
# Execution settings
num_shots = 256

detuning_MHz = 2
ramsey_frequency = round(rough_qubit_frequency + detuning_MHz * MHz, 6) # need␣
↪ramsey freq in Hz
ramsey_program = assemble(ramsey_schedules,
                          backend=backend,
                          meas_level=1,
```

```
                                meas_return='avg',
                                shots=num_shots,
                                schedule_los=[{drive_chan:␣
↪ramsey_frequency}]*len(ramsey_schedules)
                               )
```

[211]:
```
job1 = backend.run(ramsey_program)
# print(job.job_id())
job_monitor(job1)
```

```
Job Status: job has successfully run
```

[212]:
```
job2 = backend.run(ramsey_program)
# print(job.job_id())
job_monitor(job2)
```

```
Job Status: job has successfully run
```

[213]:
```
job3 = backend.run(ramsey_program)
# print(job.job_id())
job_monitor(job3)
```

```
Job Status: job has successfully run
```

[214]:
```
job4 = backend.run(ramsey_program)
# print(job.job_id())
job_monitor(job4)
```

```
Job Status: job has successfully run
```

[215]:
```
job5 = backend.run(ramsey_program)
# print(job.job_id())
job_monitor(job5)
```

```
Job Status: job has successfully run
```

[216]:
```
ramsey_results1 = job1.result(timeout=120)
```

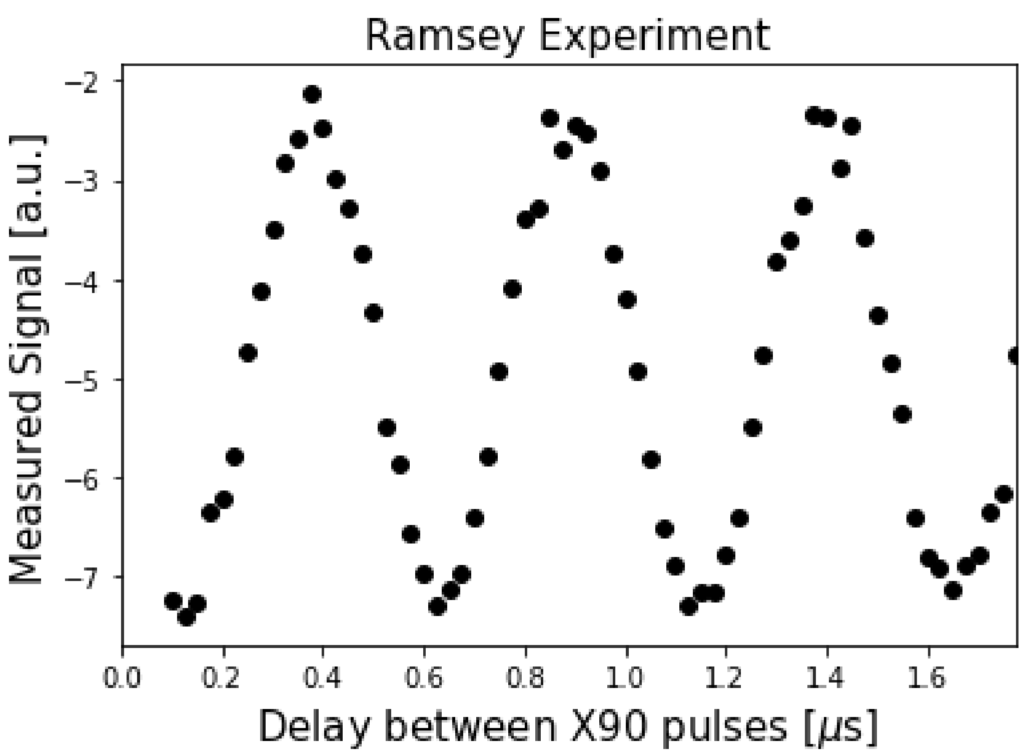
[217]:
```
ramsey_results2 = job2.result(timeout=120)
```

[218]:
```
ramsey_results3 = job3.result(timeout=120)
```

[219]:
```
ramsey_results4 = job4.result(timeout=120)
```

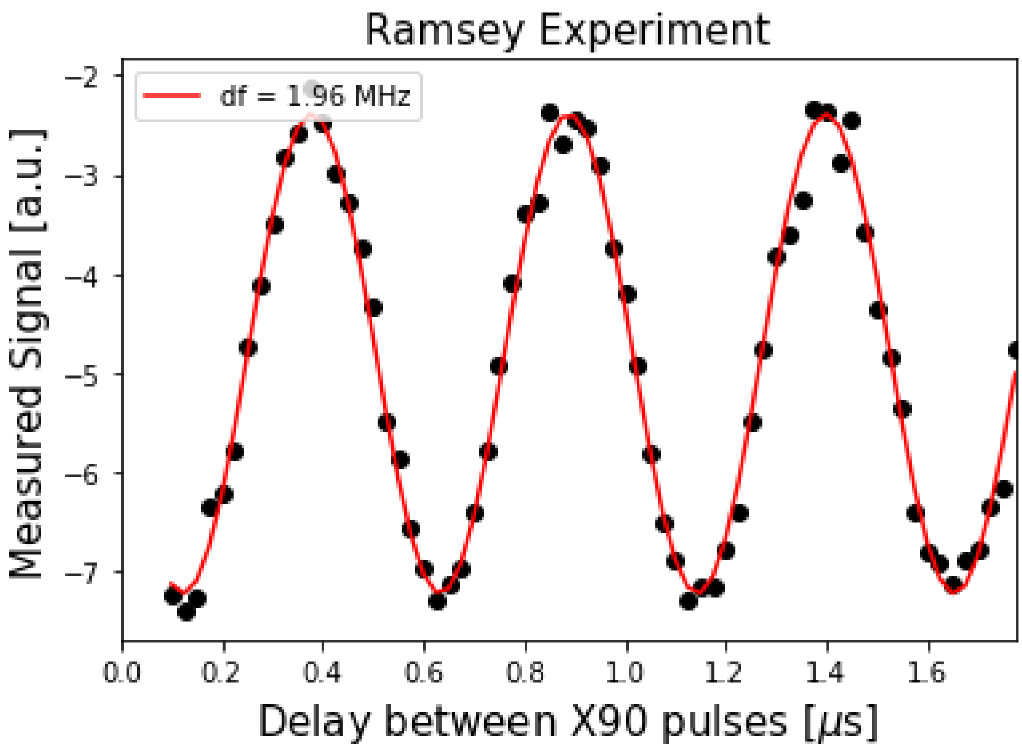
[220]:
```
ramsey_results5 = job5.result(timeout=120)
```

[221]:
```
ramsey_values = []
for i in range(len(times_us)):
    ramsey_values.append(ramsey_results1.get_memory(i)[qubit]*scale_factor)

ramsey_values1 = ramsey_values

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.xlim(0, np.max(times_us))
plt.title("Ramsey Experiment", fontsize=15)
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.show()
```



[222]:
```
fit_params, y_fit = fit_function(times_us, np.real(ramsey_values),
                                 lambda x, A, del_f_MHz, C, B: (
                                          A * np.cos(2*np.pi*del_f_MHz*x - C) +␣
 ↪B
                                         ),
                                 [5, 1./0.4, 0, 0.25]
                                )

# Off-resonance component
_, del_f_MHz, _, _, = fit_params # freq is MHz since times in us
```

```python
plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.plot(times_us, y_fit, color='red', label=f"df = {del_f_MHz:.2f} MHz")
plt.xlim(0, np.max(times_us))
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Ramsey Experiment', fontsize=15)
plt.legend()
plt.show()
```

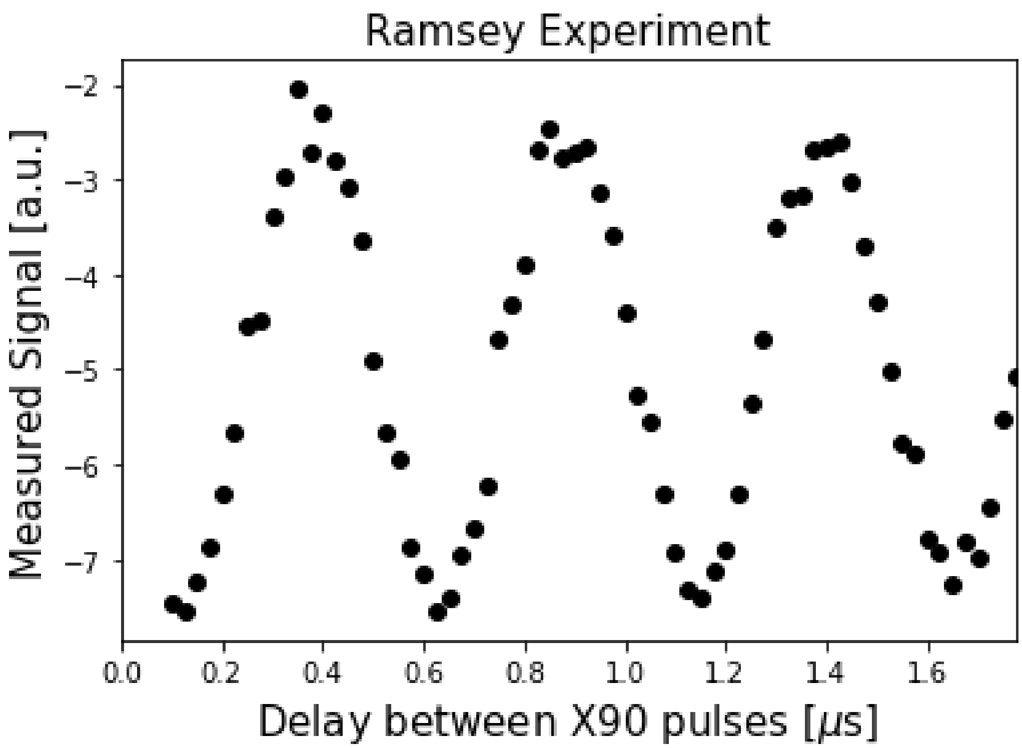


[223]:
```python
ramsey_shift1 = del_f_MHz
ramsey_shift1
```

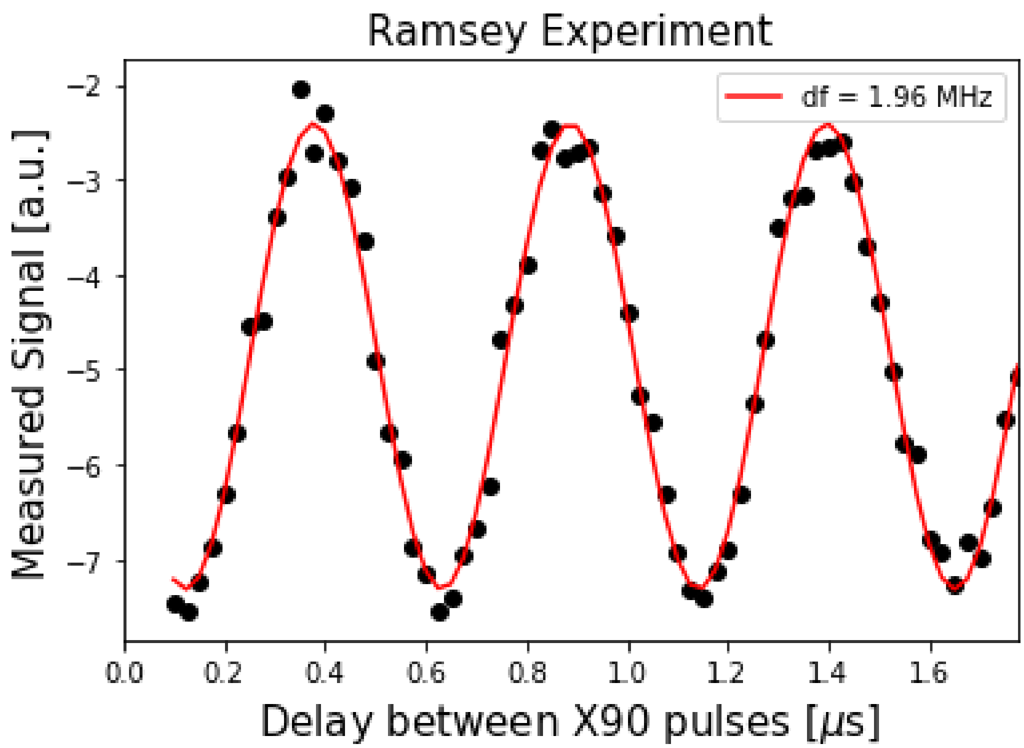
[223]: 1.959572567610101

[224]:
```python
ramsey_values = []
for i in range(len(times_us)):
    ramsey_values.append(ramsey_results2.get_memory(i)[qubit]*scale_factor)

ramsey_values2 = ramsey_values

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.xlim(0, np.max(times_us))
plt.title("Ramsey Experiment", fontsize=15)
```

```
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.show()
```



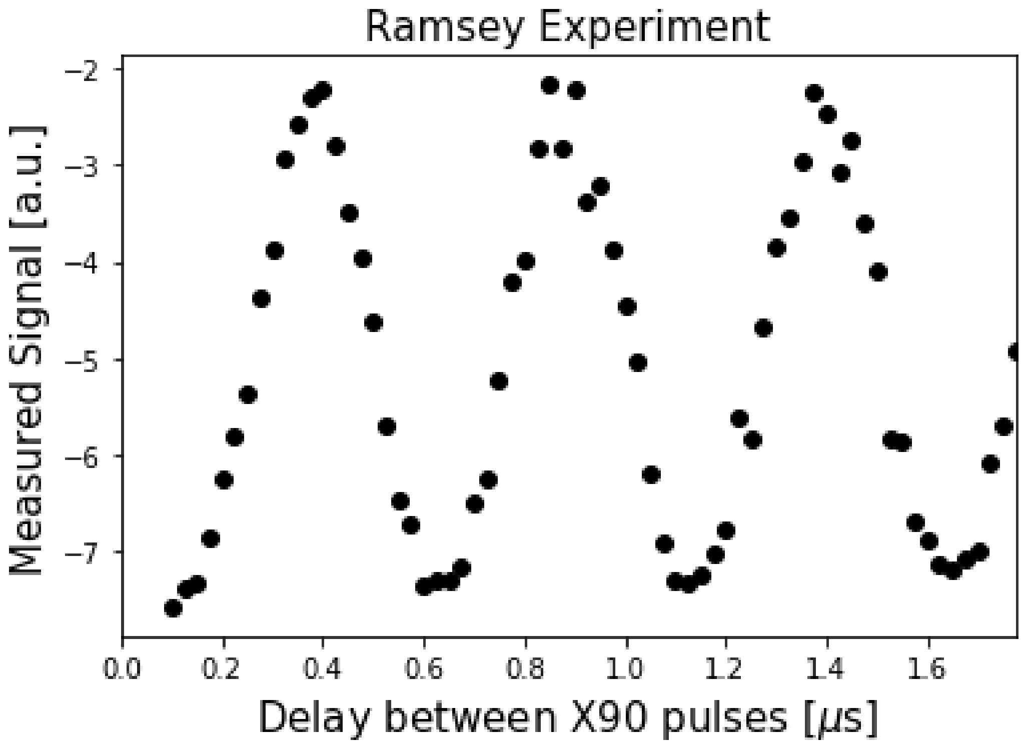
[225]:
```
fit_params, y_fit = fit_function(times_us, np.real(ramsey_values),
                                 lambda x, A, del_f_MHz, C, B: (
                                          A * np.cos(2*np.pi*del_f_MHz*x - C) +
 ↪B
                                         ),
                                 [5, 1./0.4, 0, 0.25]
                                )

# Off-resonance component
_, del_f_MHz, _, _, = fit_params # freq is MHz since times in us

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.plot(times_us, y_fit, color='red', label=f"df = {del_f_MHz:.2f} MHz")
plt.xlim(0, np.max(times_us))
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Ramsey Experiment', fontsize=15)
plt.legend()
plt.show()
```



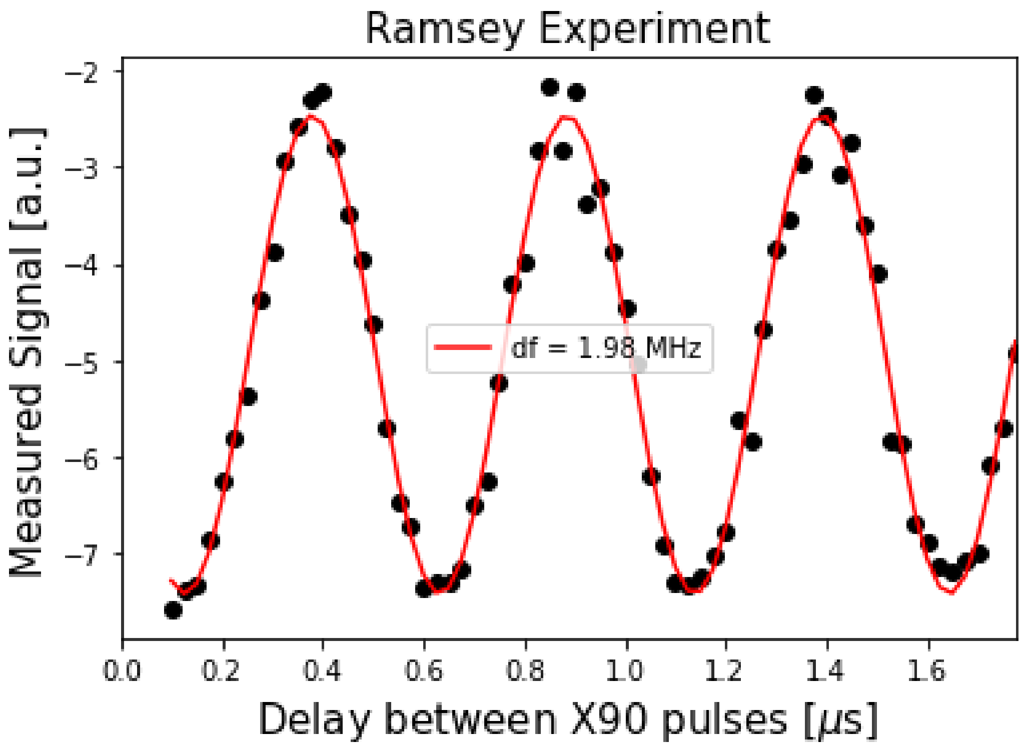
[226]:
```
ramsey_shift2 = del_f_MHz
ramsey_shift2
```

[226]: 1.9638423216125036

[227]:
```
ramsey_values = []
for i in range(len(times_us)):
    ramsey_values.append(ramsey_results3.get_memory(i)[qubit]*scale_factor)

ramsey_values3 = ramsey_values

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.xlim(0, np.max(times_us))
plt.title("Ramsey Experiment", fontsize=15)
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.show()
```



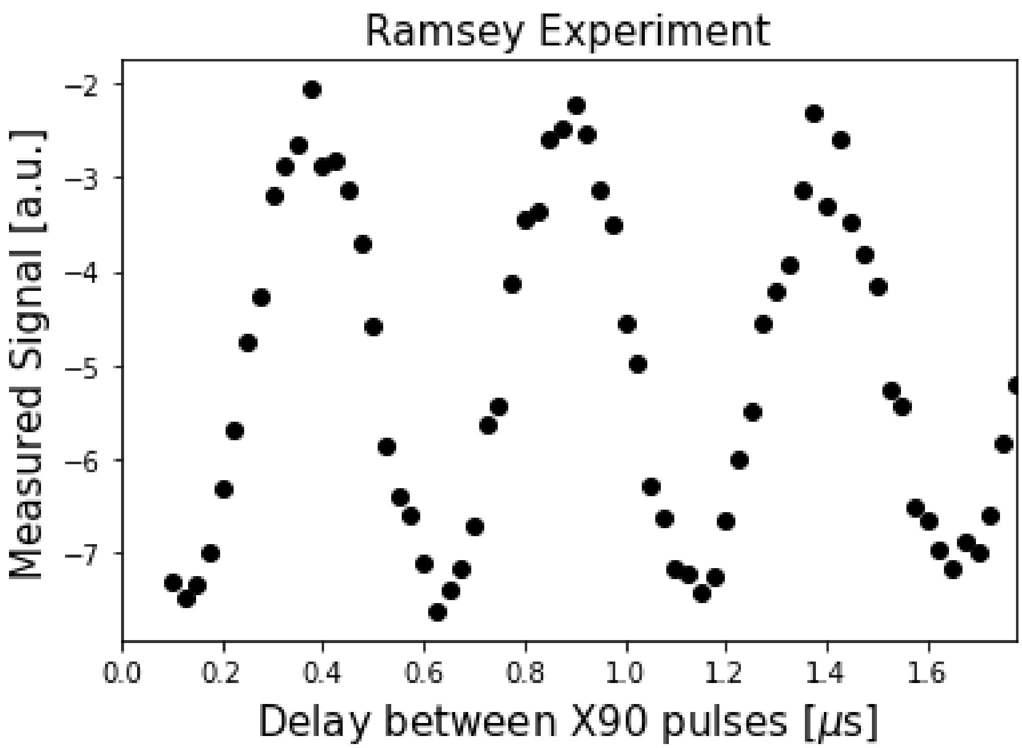
[228]:
```
fit_params, y_fit = fit_function(times_us, np.real(ramsey_values),
                                 lambda x, A, del_f_MHz, C, B: (
                                          A * np.cos(2*np.pi*del_f_MHz*x - C) +
 ↪B
                                         ),
                                 [5, 1./0.4, 0, 0.25]
                                )

# Off-resonance component
_, del_f_MHz, _, _, = fit_params # freq is MHz since times in us

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.plot(times_us, y_fit, color='red', label=f"df = {del_f_MHz:.2f} MHz")
plt.xlim(0, np.max(times_us))
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Ramsey Experiment', fontsize=15)
plt.legend()
plt.show()
```



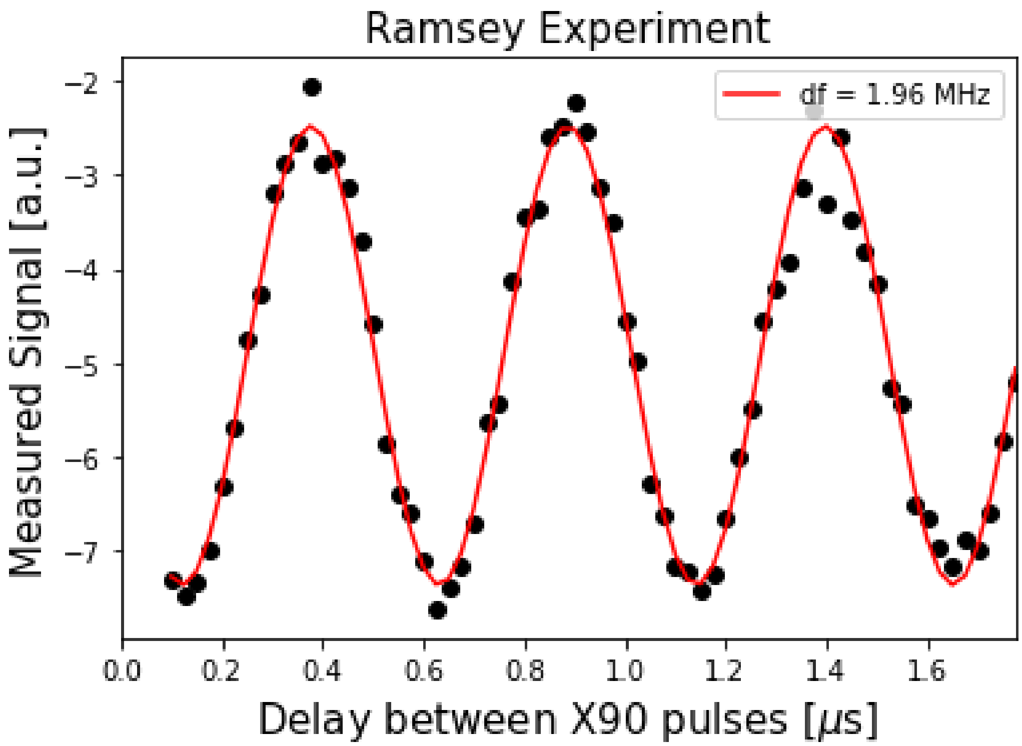
[229]:
```
ramsey_shift3 = del_f_MHz
ramsey_shift3
```

[229]: 1.9770036017140786

[230]:
```
ramsey_values = []
for i in range(len(times_us)):
    ramsey_values.append(ramsey_results4.get_memory(i)[qubit]*scale_factor)

ramsey_values4 = ramsey_values

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.xlim(0, np.max(times_us))
plt.title("Ramsey Experiment", fontsize=15)
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.show()
```



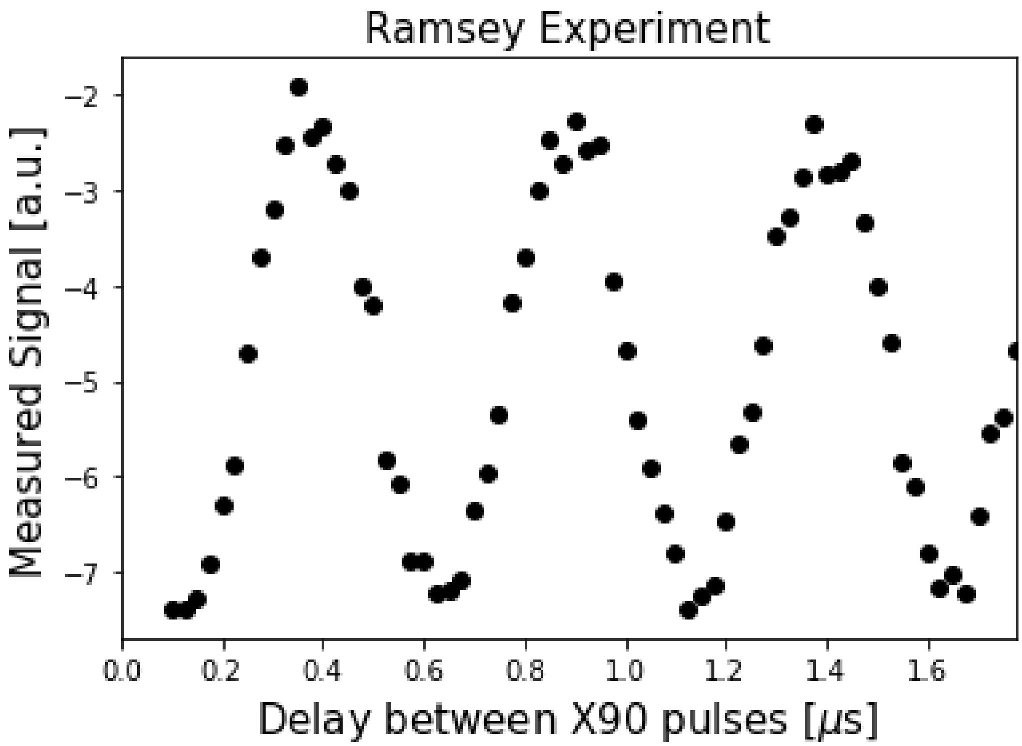
[231]:
```
fit_params, y_fit = fit_function(times_us, np.real(ramsey_values),
                                 lambda x, A, del_f_MHz, C, B: (
                                          A * np.cos(2*np.pi*del_f_MHz*x - C) +
 ↪B
                                         ),
                                 [5, 1./0.4, 0, 0.25]
                                )

# Off-resonance component
_, del_f_MHz, _, _, = fit_params # freq is MHz since times in us

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.plot(times_us, y_fit, color='red', label=f"df = {del_f_MHz:.2f} MHz")
plt.xlim(0, np.max(times_us))
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Ramsey Experiment', fontsize=15)
plt.legend()
plt.show()
```



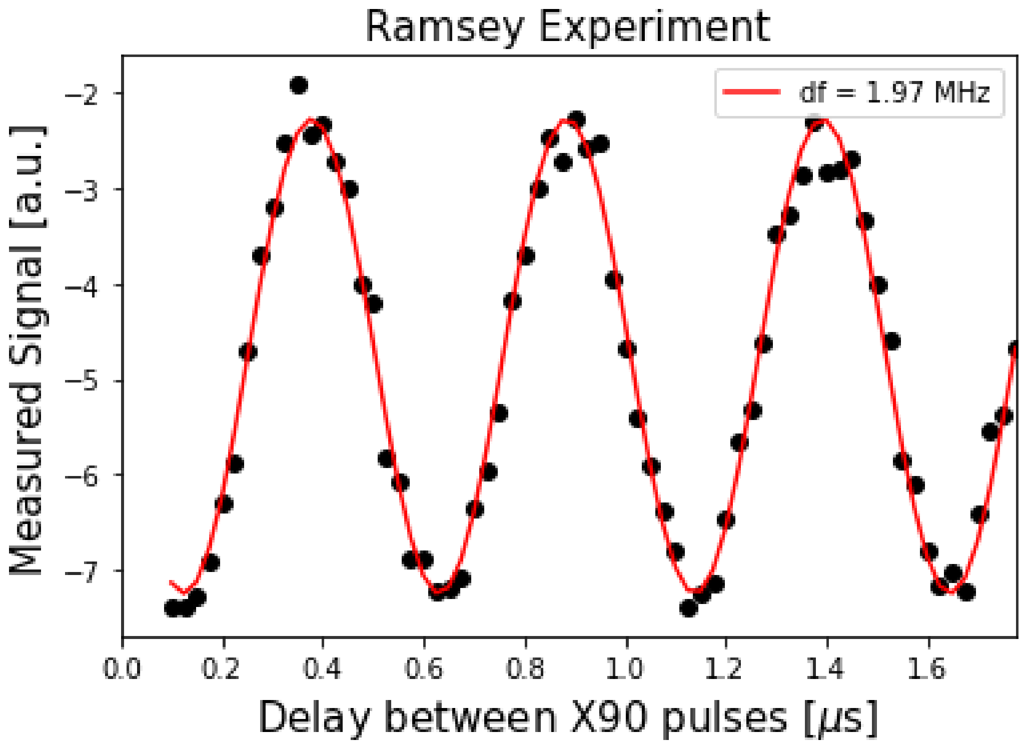
[232]:
```
ramsey_shift4 = del_f_MHz
ramsey_shift4
```

[232]: 1.9603092498664338

[233]:
```
ramsey_values = []
for i in range(len(times_us)):
    ramsey_values.append(ramsey_results5.get_memory(i)[qubit]*scale_factor)

ramsey_values5 = ramsey_values

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.xlim(0, np.max(times_us))
plt.title("Ramsey Experiment", fontsize=15)
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.show()
```



[234]:
```
fit_params, y_fit = fit_function(times_us, np.real(ramsey_values),
                                 lambda x, A, del_f_MHz, C, B: (
                                          A * np.cos(2*np.pi*del_f_MHz*x - C) +␣
 ↪B
                                         ),
                                 [5, 1./0.4, 0, 0.25]
                                )

# Off-resonance component
_, del_f_MHz, _, _, = fit_params # freq is MHz since times in us

plt.scatter(times_us, np.real(ramsey_values), color='black')
plt.plot(times_us, y_fit, color='red', label=f"df = {del_f_MHz:.2f} MHz")
plt.xlim(0, np.max(times_us))
plt.xlabel('Delay between X90 pulses [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Ramsey Experiment', fontsize=15)
plt.legend()
plt.show()
```



[235]:
```
ramsey_shift5 = del_f_MHz
ramsey_shift5
```

[235]: 1.9722919222459898

[288]:
```
#ramsey_shift_avg = (ramsey_shift1 + ramsey_shift2 + ramsey_shift3 +␣
↪ramsey_shift4 + ramsey_shift5) / 5
```

[464]:
```
#ramsey_shift_avg
```

[464]: 1.97

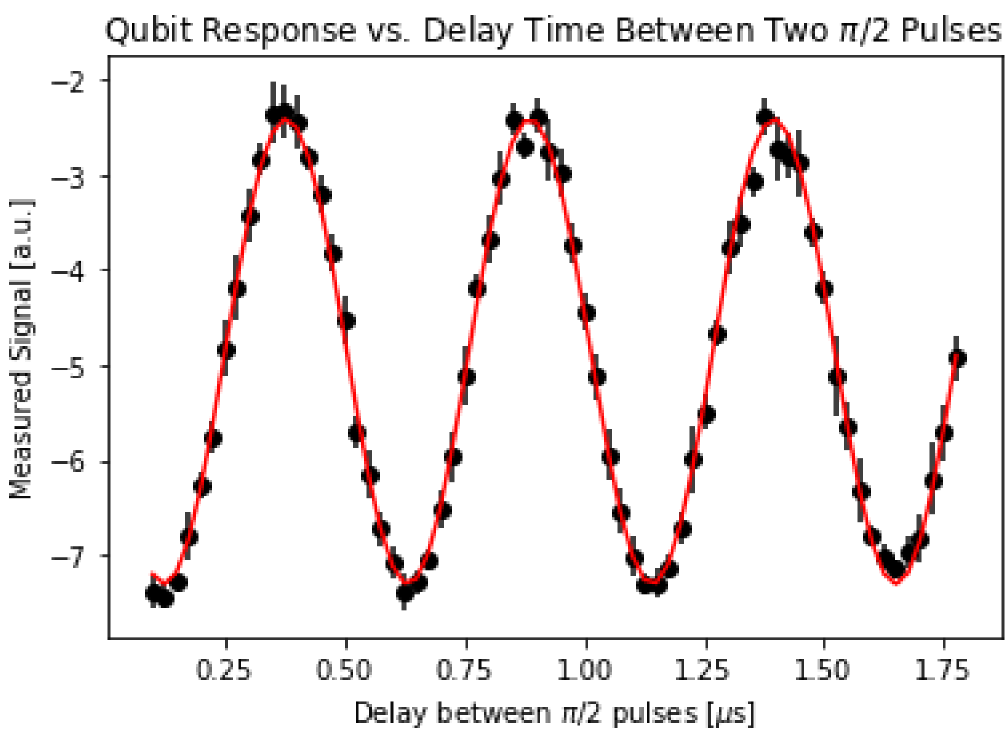
[612]:
```
ramsey_shift_means = np.mean([ramsey_values1, ramsey_values2, ramsey_values3,␣
↪ramsey_values4, ramsey_values5], axis=0)
```

[613]:
```
ramsey_shift_stds = np.std([ramsey_values1, ramsey_values2, ramsey_values3,␣
↪ramsey_values4, ramsey_values5], axis=0)
```

[619]:
```
print(len(ramsey_shift_means))

fit_params, y_fit = fit_function(times_us, np.real(ramsey_shift_means),
                                 lambda x, A, del_f_MHz, C, B: (
                                          A * np.cos(2*np.pi*del_f_MHz*x - C) +␣
↪B
                                         ),
                                 [5, 1./0.4, 0, 0.25]
```

```
                                )

plt.xlabel('Delay between $\pi / 2$ pulses [$\mu$s]')
plt.ylabel('Measured Signal [a.u.]')
plt.scatter(times_us, np.real(ramsey_shift_means), color='black') # plot real␣
 ↪part of Rabi values


plt.errorbar(times_us, np.real(ramsey_shift_means), yerr=np.
 ↪real(ramsey_shift_stds), color = 'black', ls = 'none')

plt.plot(times_us, y_fit, color = 'red')

plt.title("Qubit Response vs. Delay Time Between Two $\pi/2$ Pulses")

plt.show()
```

68



[241]:
```
ramsey_shift_avg = del_f_MHz
ramsey_shift_avg
```

[241]: 1.9722919222459898

[242]:
```
ramsey_shift_std = np.sqrt(((ramsey_shift1 - ramsey_shift_avg)**2 +␣
↪(ramsey_shift2 - ramsey_shift_avg)**2 + (ramsey_shift3 -␣
↪ramsey_shift_avg)**2 + (ramsey_shift4 - ramsey_shift_avg)**2 +␣
↪(ramsey_shift5 - ramsey_shift_avg)**2) / 5)
```

[243]:
```
ramsey_shift_std
```

[243]:
```
0.008932660230689397
```

[ ]:
```
#avg_ramsey_shift = 1.970 +/- 0.003 (all MHz)
```

[ ]:
```
#4.97184 +/- 0.00006 GHz is rough qubit
```

[ ]:
```
#qubit_freq = 4.97381 +/- 0.000063 GHz
```

[247]:
```
qubit_freq = rough_qubit_frequency + (ramsey_shift_avg - detuning_MHz) * MHz
```

[248]:
```
qubit_freq
```

[248]:
```
4971820958.848121
```

[249]:
```
detuning_MHz
```

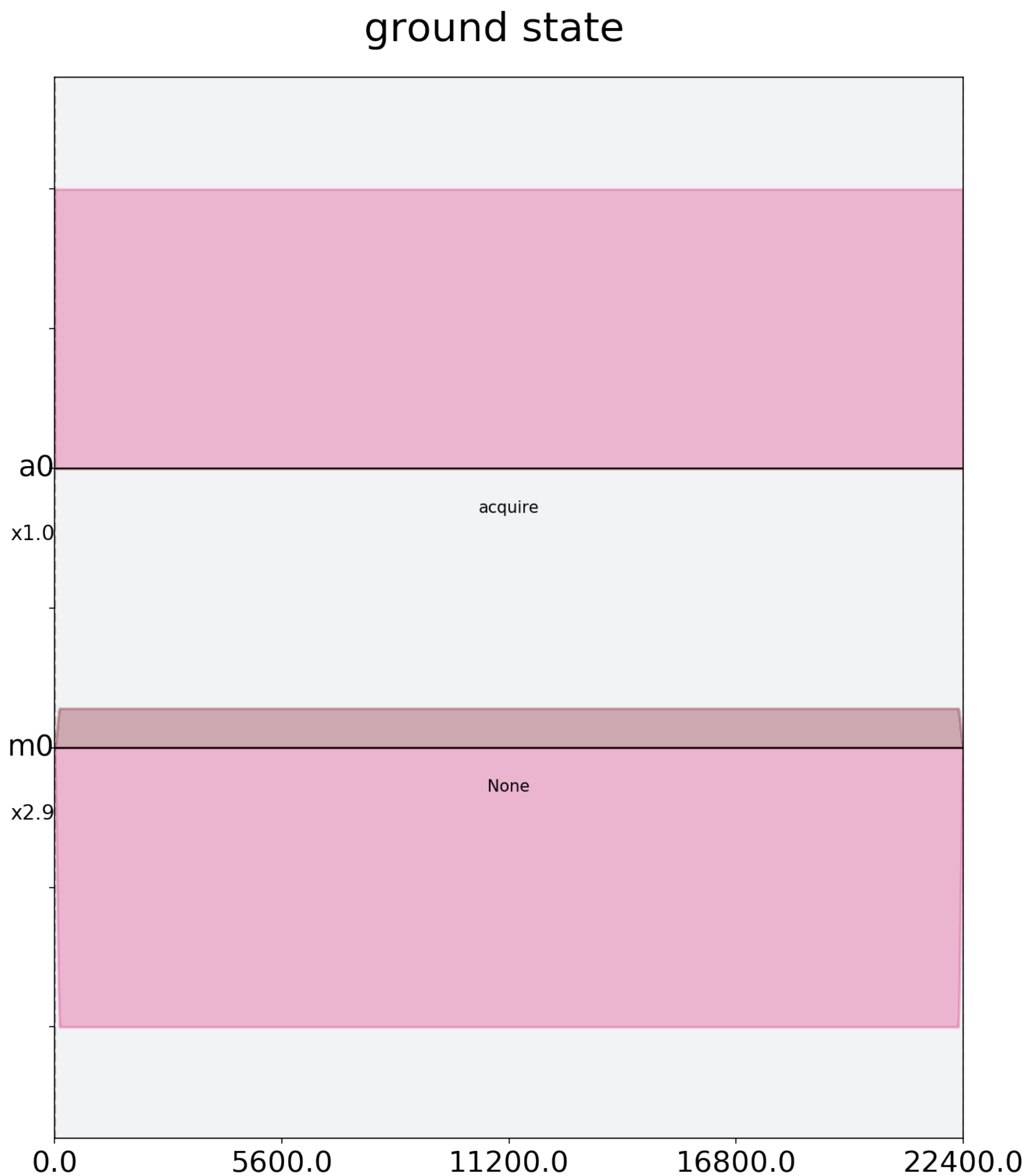
[249]:
```
2
```

[250]:
```
#Now we evaluate a 0-1 discriminator.
```

[251]:
```
# Create two schedules

# Ground state schedule
gnd_schedule = pulse.Schedule(name="ground state")
gnd_schedule += measure

# Excited state schedule
exc_schedule = pulse.Schedule(name="excited state")
exc_schedule += Play(pi_pulse, drive_chan)  # We found this in Part 2A above
exc_schedule += measure << exc_schedule.duration
```

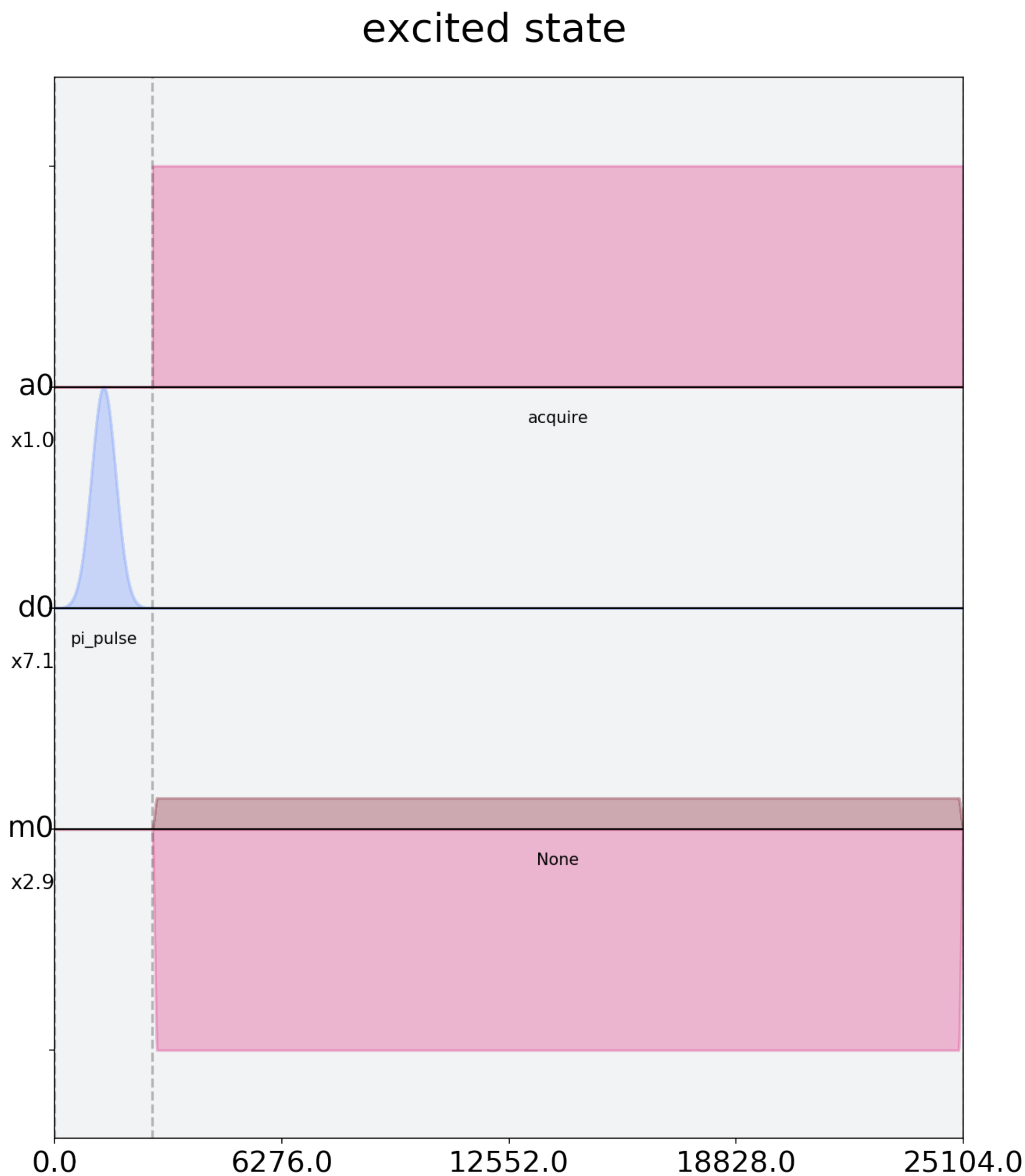
[252]:
```
gnd_schedule.draw(label=True)
```

[252]:



[253]:
```
exc_schedule.draw(label=True)
```

[253]:



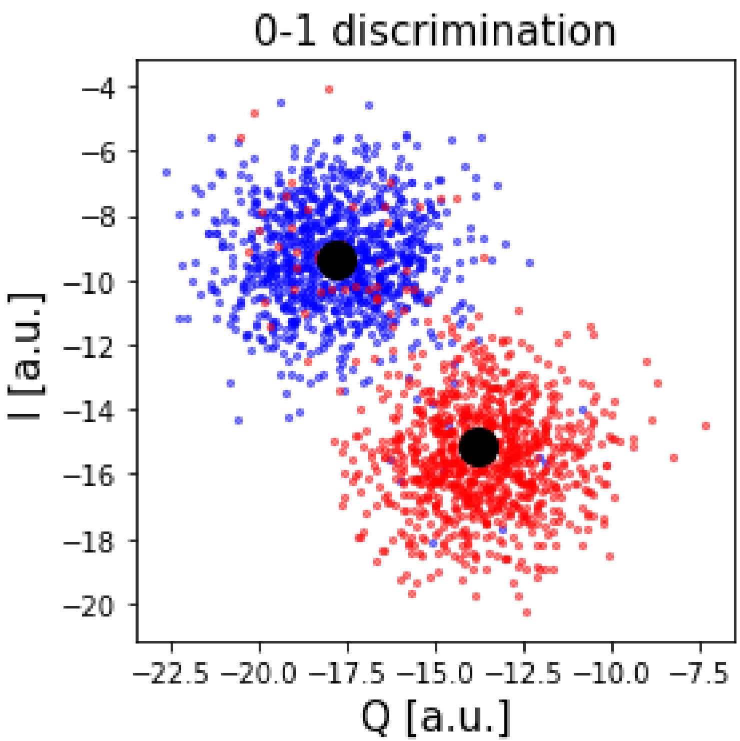
[254]:
```
# Execution settings
num_shots = 1024

gnd_exc_program = assemble([gnd_schedule, exc_schedule],
                           backend=backend,
                           meas_level=1,
                           meas_return='single',
                           shots=num_shots,
                           schedule_los=[{drive_chan: qubit_freq}] * 2)
```

[255]:
```
# print(job.job_id())
job = backend.run(gnd_exc_program)
job_monitor(job)
```

```
Job Status: job has successfully run
```

[256]:
```
gnd_exc_results = job.result(timeout=120)
```

[257]:
```
gnd_results = gnd_exc_results.get_memory(0)[:, qubit]*scale_factor
exc_results = gnd_exc_results.get_memory(1)[:, qubit]*scale_factor

plt.figure(figsize=[4,4])
# Plot all the results
# All results from the gnd_schedule are plotted in blue
plt.scatter(np.real(gnd_results), np.imag(gnd_results),
                s=5, cmap='viridis', c='blue', alpha=0.5, label='state_0')
# All results from the exc_schedule are plotted in red
plt.scatter(np.real(exc_results), np.imag(exc_results),
                s=5, cmap='viridis', c='red', alpha=0.5, label='state_1')

# Plot a large dot for the average result of the 0 and 1 states.
mean_gnd = np.mean(gnd_results) # takes mean of both real and imaginary parts
mean_exc = np.mean(exc_results)
plt.scatter(np.real(mean_gnd), np.imag(mean_gnd),
            s=200, cmap='viridis', c='black',alpha=1.0, label='state_0_mean')
plt.scatter(np.real(mean_exc), np.imag(mean_exc),
            s=200, cmap='viridis', c='black',alpha=1.0, label='state_1_mean')

plt.ylabel('I [a.u.]', fontsize=15)
plt.xlabel('Q [a.u.]', fontsize=15)
plt.title("0-1 discrimination", fontsize=15)

plt.show()
```



[258]:
```
rough_qubit_frequency
```

[258]: 4971848666.925875

[259]:
```
#Now we evaluate the accuracy of our classifier.
```

[260]:
```
import math

def classify(point: complex):
    """Classify the given state as |0> or |1>."""
    def distance(a, b):
        return math.sqrt((np.real(a) - np.real(b))**2 + (np.imag(a) - np.
 ↪imag(b))**2)
    return int(distance(point, mean_exc) < distance(point, mean_gnd))
```

[261]:
```
total_ground = 0
total_exc = 0
total = 0
for point in gnd_results:
    total += 1
    if classify(point) == 0:
        total_ground += 1

for point in exc_results:
    total += 1
```

```
    if classify(point) == 1:
        total_exc += 1
```

[267]:
```
total_ground
```

[267]: 1001

[268]:
```
total_exc
```

[268]: 971

[269]:
```
total = total / 2
```

[270]:
```
(total - (total_exc)) / total
```

[270]: 0.0517578125

[ ]:
```
#This corresponds to 5.17578125% infidelity.
```

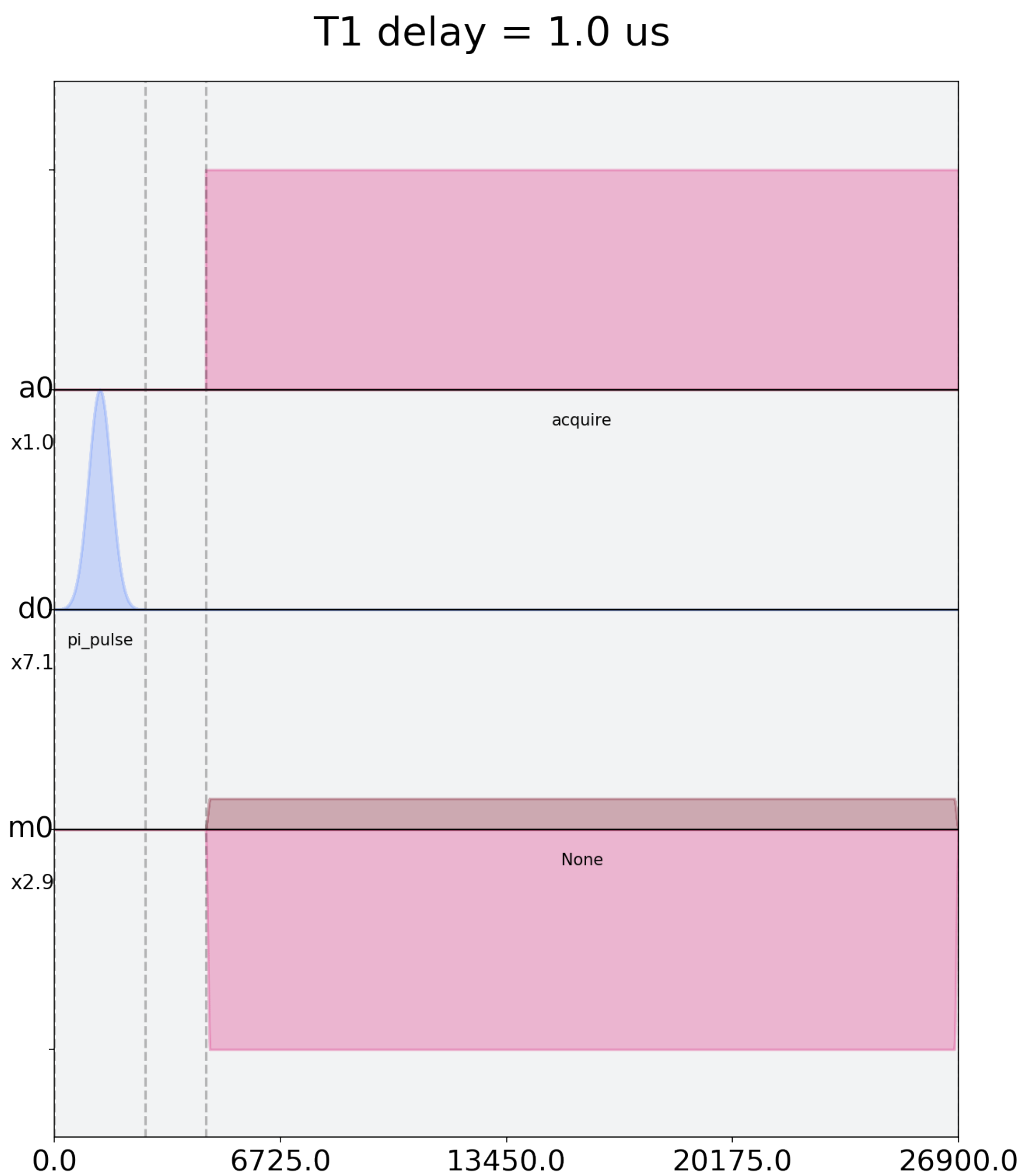
[271]:
```
#repeat 4 times, get T1, T2 times.
```

[272]:
```
# T1 experiment parameters
time_max_us = 450
time_step_us = 6
times_us = np.arange(1, time_max_us, time_step_us)
# Convert to units of dt
delay_times_dt = times_us * us / dt
# We will use the same `pi_pulse` and qubit frequency that we calibrated and␣
 ↪used before
```

[273]:
```
# Create schedules for the experiment
t1_schedules = []
for delay in delay_times_dt:
    this_schedule = pulse.Schedule(name=f"T1 delay = {delay * dt/us} us")
    this_schedule += Play(pi_pulse, drive_chan)
    this_schedule |= measure << int(delay)
    t1_schedules.append(this_schedule)
```

[274]:
```
sched_idx = 0
t1_schedules[sched_idx].draw(label=True)
```

[274]:



[275]:
```
# Execution settings
num_shots = 256

t1_experiment = assemble(t1_schedules,
                         backend=backend,
                         meas_level=1,
                         meas_return='avg',
                         shots=num_shots,
```

```
                              schedule_los=[{drive_chan: qubit_freq}] *␣
↪len(t1_schedules))
```

[276]:
```
job1 = backend.run(t1_experiment)
# print(job.job_id())
job_monitor(job)
```

```
Job Status: job has successfully run
```

[277]:
```
t1_results1 = job1.result(timeout=120)
```

[278]:
```
job2 = backend.run(t1_experiment)
# print(job.job_id())
job_monitor(job2)
```

```
Job Status: job has successfully run
```

[279]:
```
t1_results2 = job2.result(timeout=120)
```

[280]:
```
job3 = backend.run(t1_experiment)
# print(job.job_id())
job_monitor(job3)
```

```
Job Status: job has successfully run
```

[281]:
```
t1_results3 = job3.result(timeout=120)
```

[282]:
```
job4 = backend.run(t1_experiment)
# print(job.job_id())
job_monitor(job4)
```

```
Job Status: job has successfully run
```

[283]:
```
t1_results4 = job4.result(timeout=120)
```

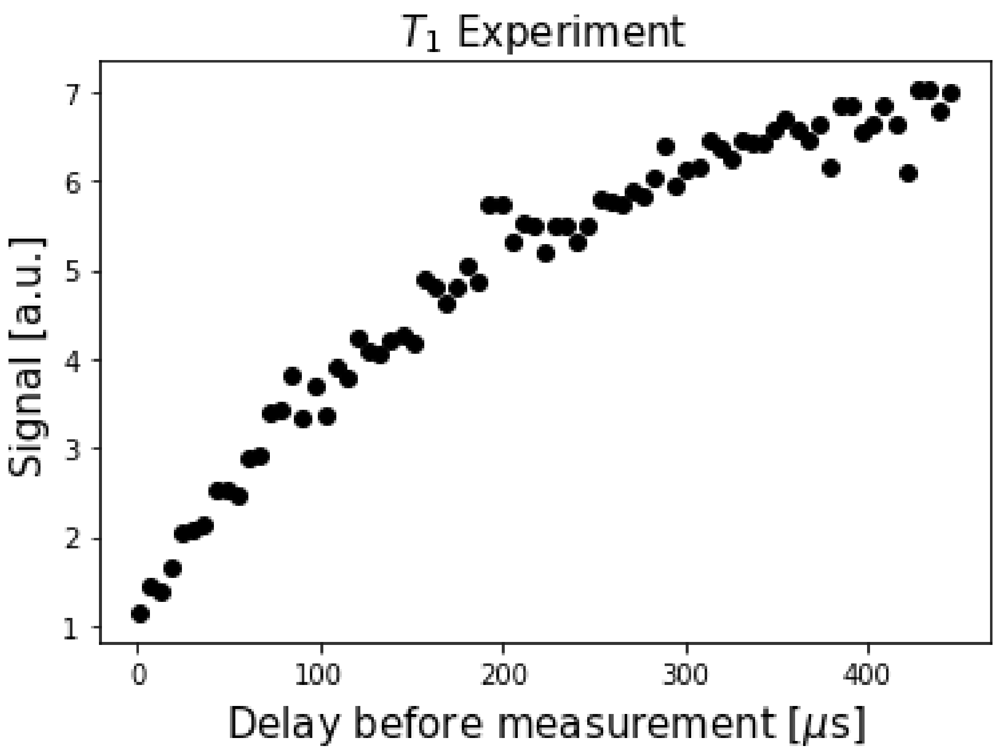
[284]:
```
job5 = backend.run(t1_experiment)
# print(job.job_id())
job_monitor(job5)
```

```
Job Status: job has successfully run
```

[285]:
```
t1_results5 = job5.result(timeout=120)
```

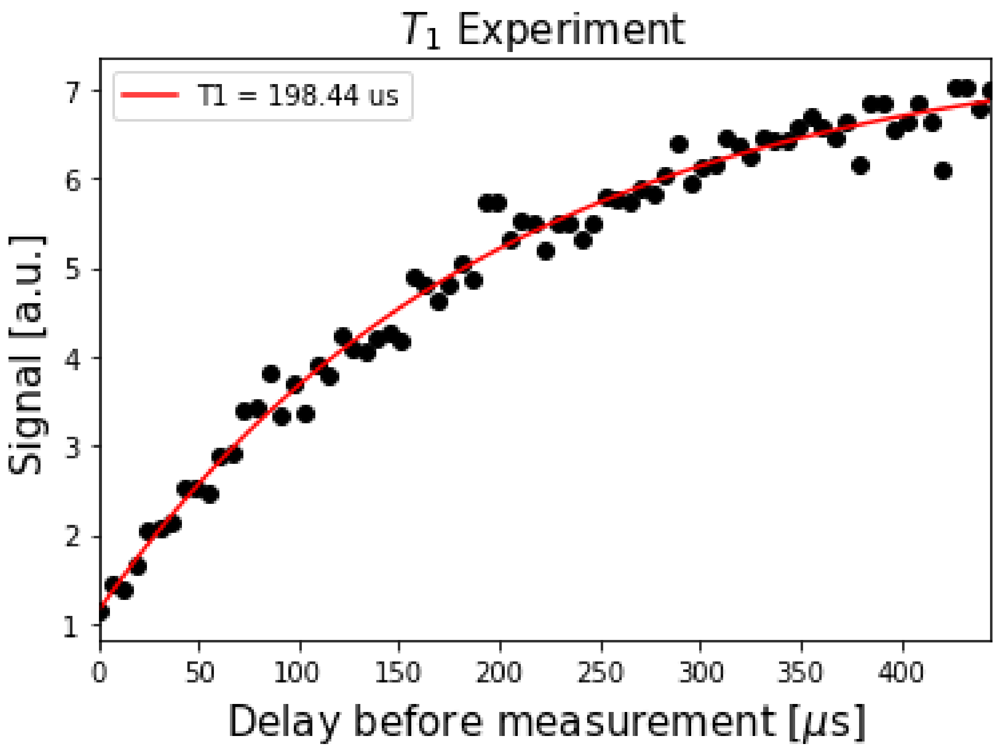
[375]:
```
t1_values = []
for i in range(len(times_us)):
    t1_values.append(t1_results1.get_memory(i)[qubit]*scale_factor)
t1_values = np.real(t1_values)
```

```
t1_values1 = t1_values

plt.scatter(times_us, t1_values, color='black')
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.show()
```



[376]:
```
# Fit the data
fit_params, y_fit = fit_function(times_us, t1_values,
            lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
            [-3, 3, 100]
            )

_, _, T1 = fit_params

plt.scatter(times_us, t1_values, color='black')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.xlim(0, np.max(times_us))
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.legend()
```

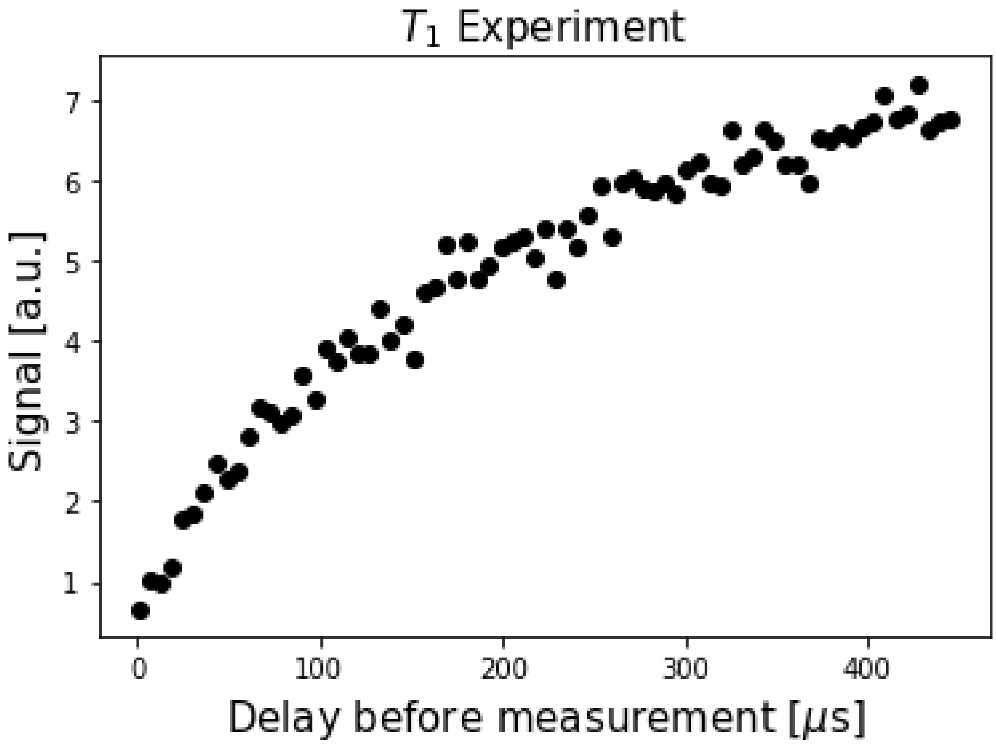
```
plt.show()
```



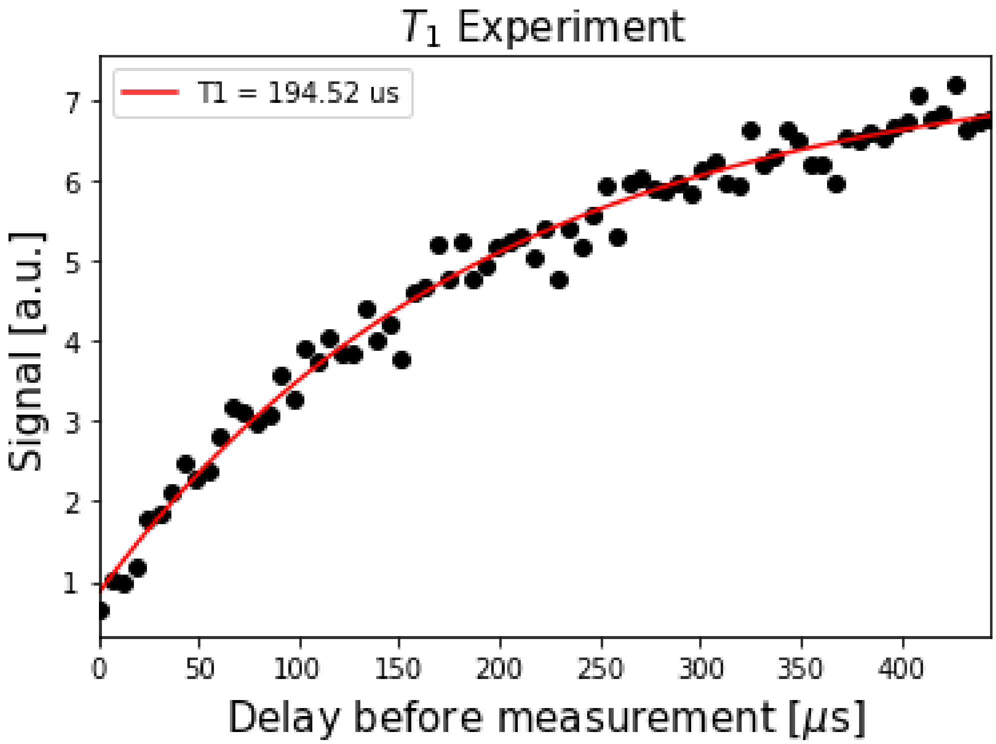
[377]:
```
T1_1 = T1
T1_1
```

[377]: 198.43842743713702

[378]:
```
t1_values = []
for i in range(len(times_us)):
    t1_values.append(t1_results2.get_memory(i)[qubit]*scale_factor)
t1_values = np.real(t1_values)

t1_values2 = t1_values

plt.scatter(times_us, t1_values, color='black')
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.show()
```



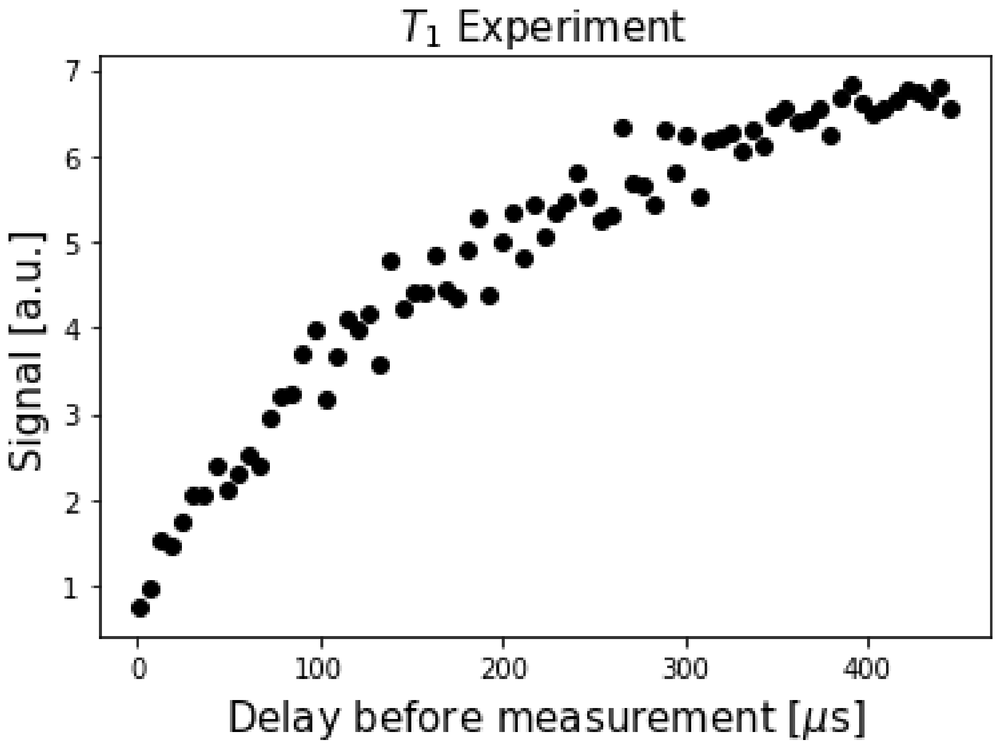
[379]:
```
# Fit the data
fit_params, y_fit = fit_function(times_us, t1_values,
            lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
            [-3, 3, 100]
            )

_, _, T1 = fit_params

plt.scatter(times_us, t1_values, color='black')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.xlim(0, np.max(times_us))
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.legend()
plt.show()
```



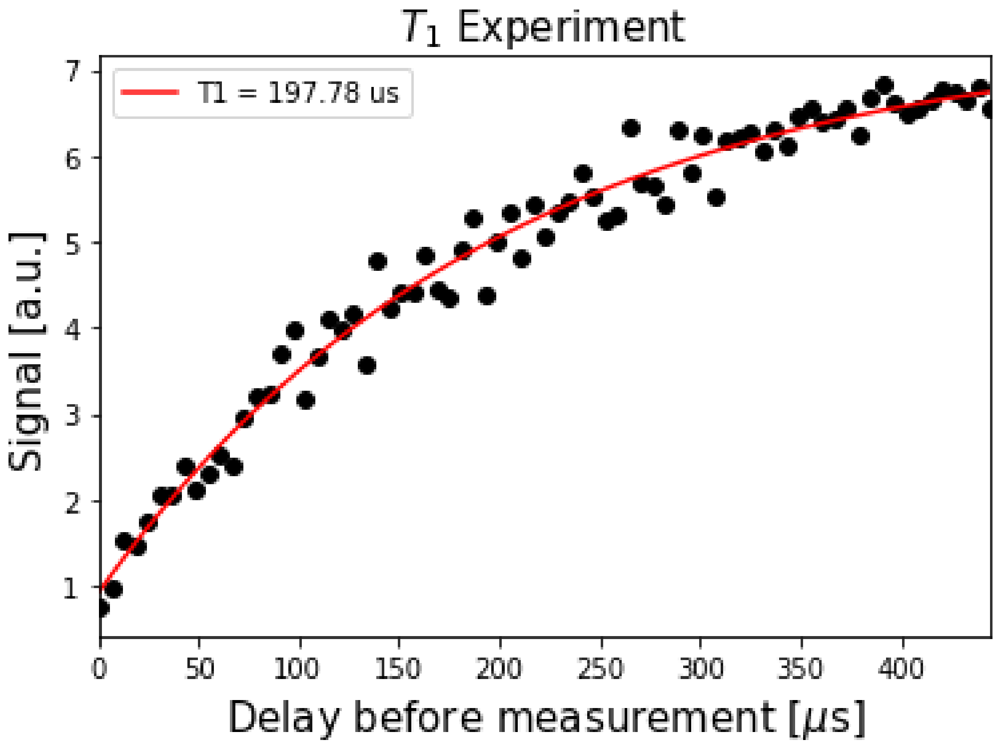
[380]:
```
T1_2 = T1
T1_2
```

[380]: 194.51856912108892

[381]:
```
t1_values = []
for i in range(len(times_us)):
    t1_values.append(t1_results3.get_memory(i)[qubit]*scale_factor)
t1_values = np.real(t1_values)

t1_values3 = t1_values

plt.scatter(times_us, t1_values, color='black')
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.show()
```



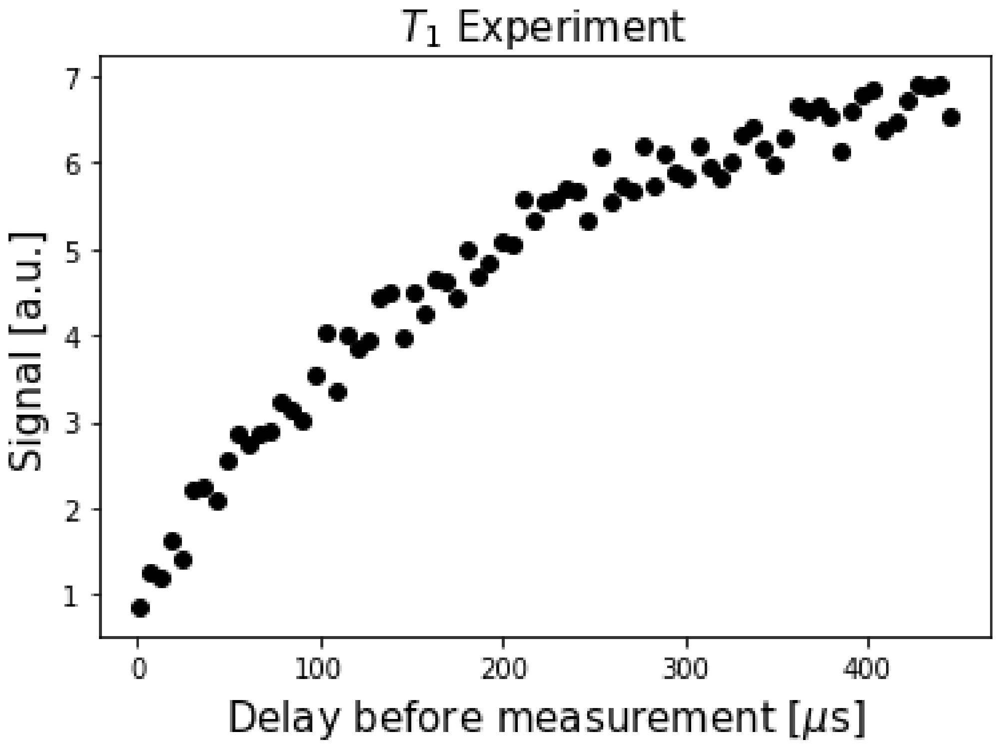
[382]:
```
# Fit the data
fit_params, y_fit = fit_function(times_us, t1_values,
            lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
            [-3, 3, 100]
            )

_, _, T1 = fit_params

plt.scatter(times_us, t1_values, color='black')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.xlim(0, np.max(times_us))
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.legend()
plt.show()
```



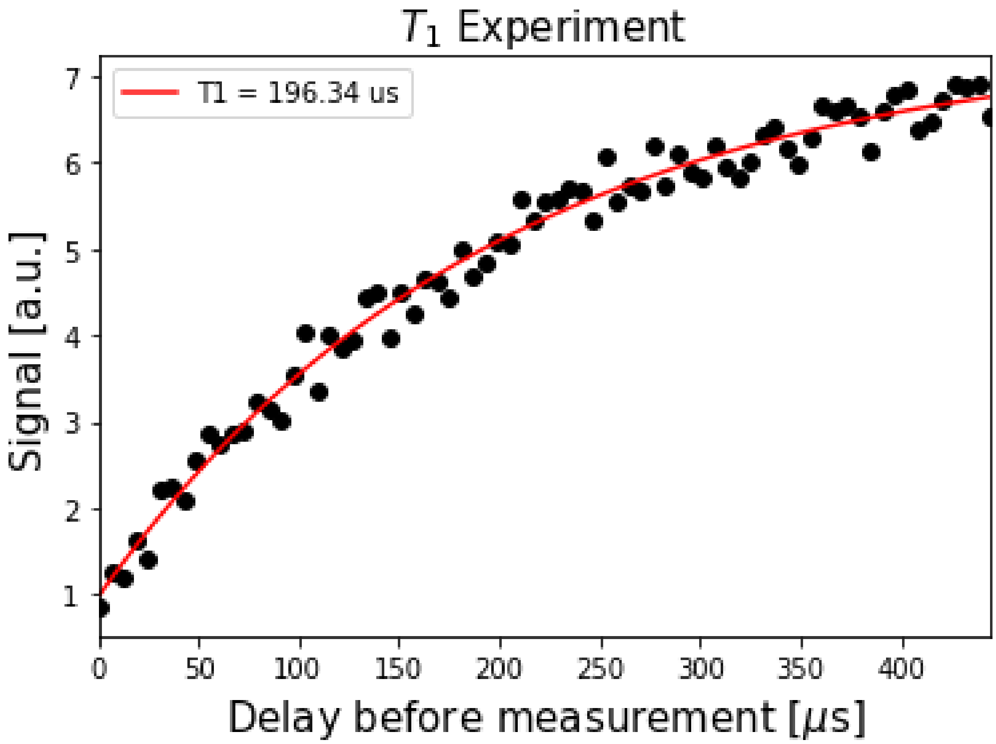
[383]:
```
T1_3 = T1
T1_3
```

[383]: 197.77725439208365

[384]:
```
t1_values = []
for i in range(len(times_us)):
    t1_values.append(t1_results4.get_memory(i)[qubit]*scale_factor)
t1_values = np.real(t1_values)

t1_values4 = t1_values

plt.scatter(times_us, t1_values, color='black')
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.show()
```



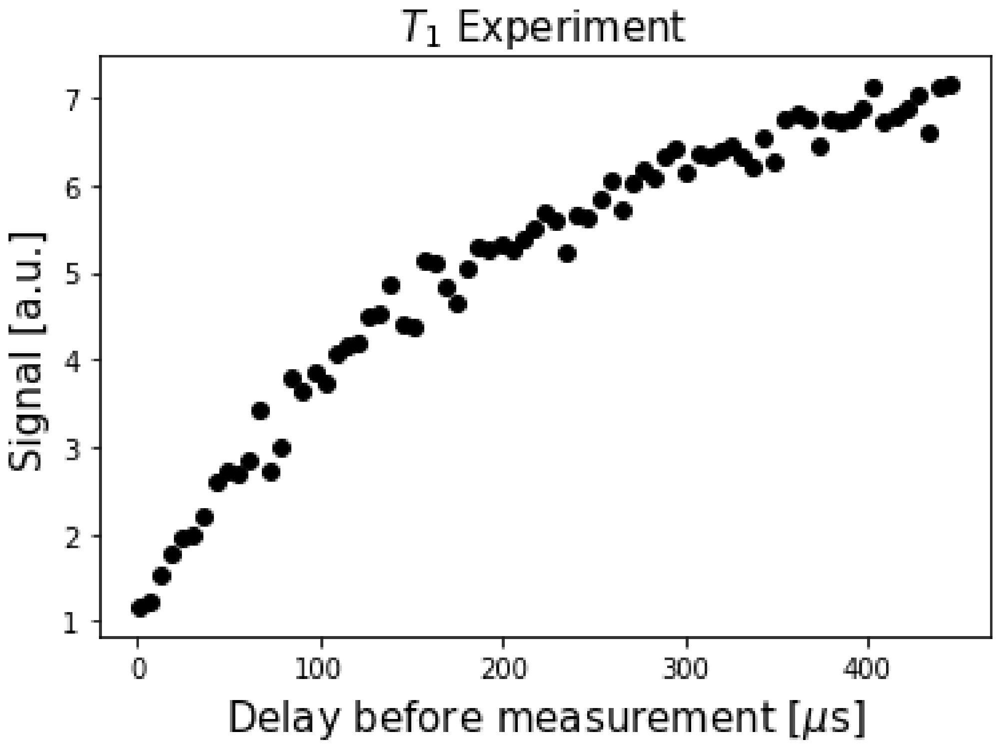
[385]:
```
# Fit the data
fit_params, y_fit = fit_function(times_us, t1_values,
            lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
            [-3, 3, 100]
            )

_, _, T1 = fit_params

plt.scatter(times_us, t1_values, color='black')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.xlim(0, np.max(times_us))
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.legend()
plt.show()
```



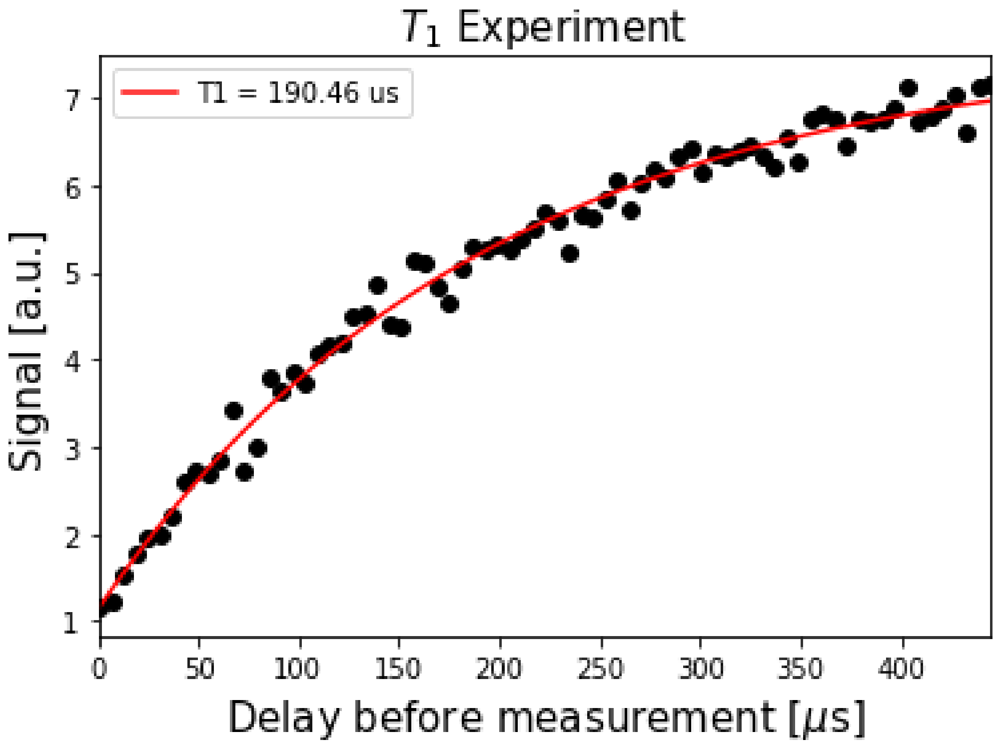
[386]:
```
T1_4 = T1
T1_4
```

[386]: 196.3370814736288

[387]:
```
t1_values = []
for i in range(len(times_us)):
    t1_values.append(t1_results5.get_memory(i)[qubit]*scale_factor)
t1_values = np.real(t1_values)

t1_values5 = t1_values

plt.scatter(times_us, t1_values, color='black')
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.show()
```



[388]:
```
# Fit the data
fit_params, y_fit = fit_function(times_us, t1_values,
            lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
            [-3, 3, 100]
            )

_, _, T1 = fit_params

plt.scatter(times_us, t1_values, color='black')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.xlim(0, np.max(times_us))
plt.title("$T_1$ Experiment", fontsize=15)
plt.xlabel('Delay before measurement [$\mu$s]', fontsize=15)
plt.ylabel('Signal [a.u.]', fontsize=15)
plt.legend()
plt.show()
```



[389]:
```
T1_5 = T1
T1_5
```

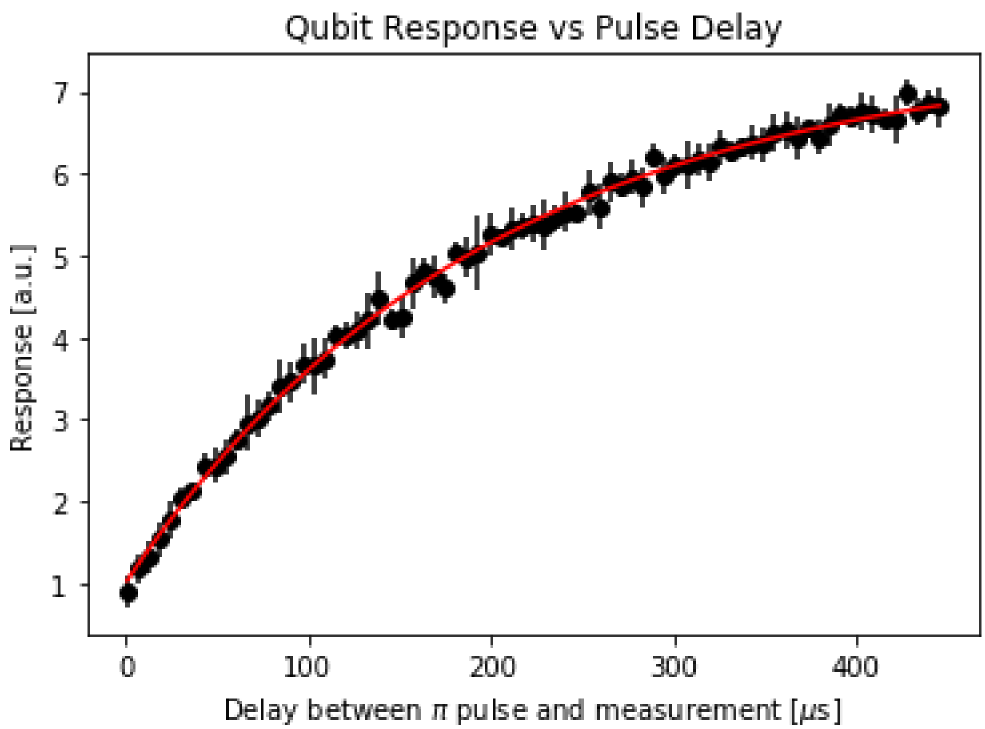
[389]: 190.4615040121705

[390]:
```
#avg_T1 = (T1_1 + T1_2 + T1_3 + T1_4 + T1_5) / 5
#avg_T1
```

[390]: 195.5065672872218

[402]:
```
t1_vals_avg = np.mean([t1_values1, t1_values2, t1_values3, t1_values4,␣
 ↪t1_values5], axis = 0)
t1_vals_std = np.std([t1_values1, t1_values2, t1_values3, t1_values4,␣
 ↪t1_values5], axis = 0)

fit_params, y_fit = fit_function(times_us, t1_vals_avg,
            lambda x, A, C, T1: (A * np.exp(-x / T1) + C),
            [-3, 3, 100]
            )

_, _, T1 = fit_params

plt.scatter(times_us, t1_vals_avg, color = 'black')
plt.errorbar(times_us, t1_vals_avg, yerr = t1_vals_std, color = 'black', ls =␣
 ↪'none')
```

```
plt.title("Qubit Response vs Pulse Delay")
plt.xlabel('Delay between $\pi$ pulse and measurement [$\mu$s]')
plt.ylabel('Response [a.u.]')
plt.plot(times_us, y_fit, color='red', label=f"T1 = {T1:.2f} us")
plt.show()
```



[395]:
```
avg_T1 = T1
```

[399]:
```
avg_T1
```

[399]: 195.4857703500917

[396]:
```
T1_std = np.sqrt(((T1_1 - avg_T1)**2 + (T1_2 - avg_T1) + (T1_3 - avg_T1)**2 +
 ↪(T1_4 - avg_T1)**2 + (T1_5 - avg_T1)**2) / 5)
```

[397]:
```
T1_std
```

[397]: 2.7917687806409024

[304]:
```
#avg T1 is 180.37 +/- 13.058 microseconds
```

[305]:
```
#T2 experiment time!
```

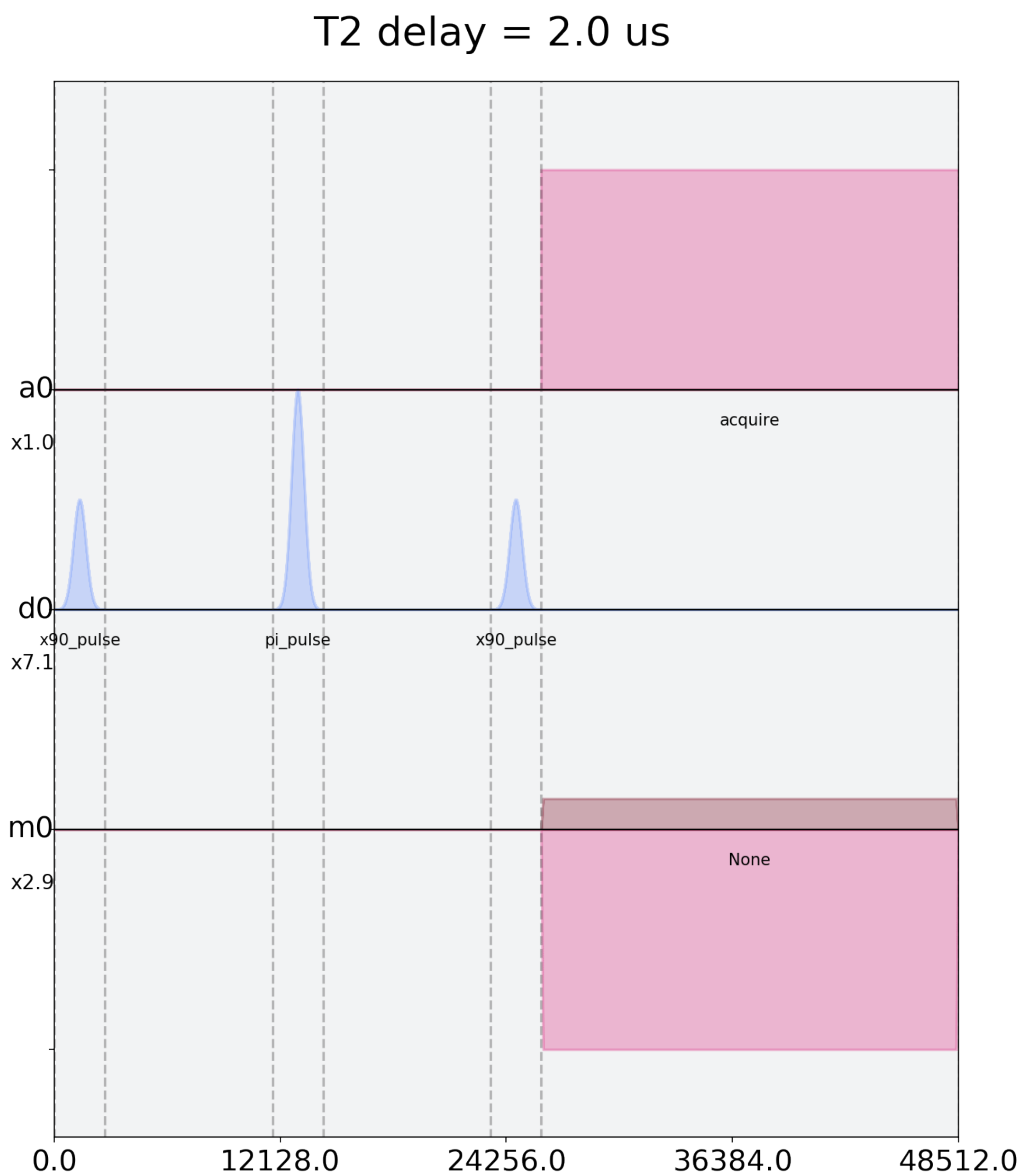
[306]:
```
# T2 experiment parameters
tau_max_us = 200
tau_step_us = 4
taus_us = np.arange(2, tau_max_us, tau_step_us)
```

```
# Convert to units of dt
delay_times_dt = taus_us * us / dt

# We will use the pi_pulse and x90_pulse from previous experiments
```

[307]:
```
t2_schedules = []
for tau in delay_times_dt:
    this_schedule = pulse.Schedule(name=f"T2 delay = {tau *dt/us} us")
    this_schedule |= Play(x90_pulse, drive_chan)
    this_schedule |= Play(pi_pulse, drive_chan) << int(this_schedule.duration +␣
↪tau)
    this_schedule |= Play(x90_pulse, drive_chan) << int(this_schedule.duration␣
↪+ tau)
    this_schedule |= measure << int(this_schedule.duration)

    t2_schedules.append(this_schedule)
```

[308]:
```
t2_schedules[0].draw(label=True)
```

[308]:



[309]:
```
# Execution settings
num_shots_per_point = 512

t2_experiment = assemble(t2_schedules,
                         backend=backend,
                         meas_level=1,
                         meas_return='avg',
                         shots=num_shots_per_point,
                         schedule_los=[{drive_chan: qubit_freq}]
```

```
                                              * len(t2_schedules))
```

[310]:
```
job1 = backend.run(t2_experiment)
# print(job.job_id())
job_monitor(job1)
```

```
Job Status: job has successfully run
```

[311]:
```
t2_results1 = job1.result(timeout=120)
```

[312]:
```
job2 = backend.run(t2_experiment)
# print(job.job_id())
job_monitor(job2)
```

```
Job Status: job has successfully run
```

[313]:
```
t2_results2 = job2.result(timeout=120)
```

[314]:
```
job3 = backend.run(t2_experiment)
# print(job.job_id())
job_monitor(job3)
```

```
Job Status: job has successfully run
```

[315]:
```
t2_results3 = job3.result(timeout=120)
```

[316]:
```
job4 = backend.run(t2_experiment)
# print(job.job_id())
job_monitor(job4)
```

```
Job Status: job has successfully run
```

[317]:
```
t2_results4 = job4.result(timeout=120)
```

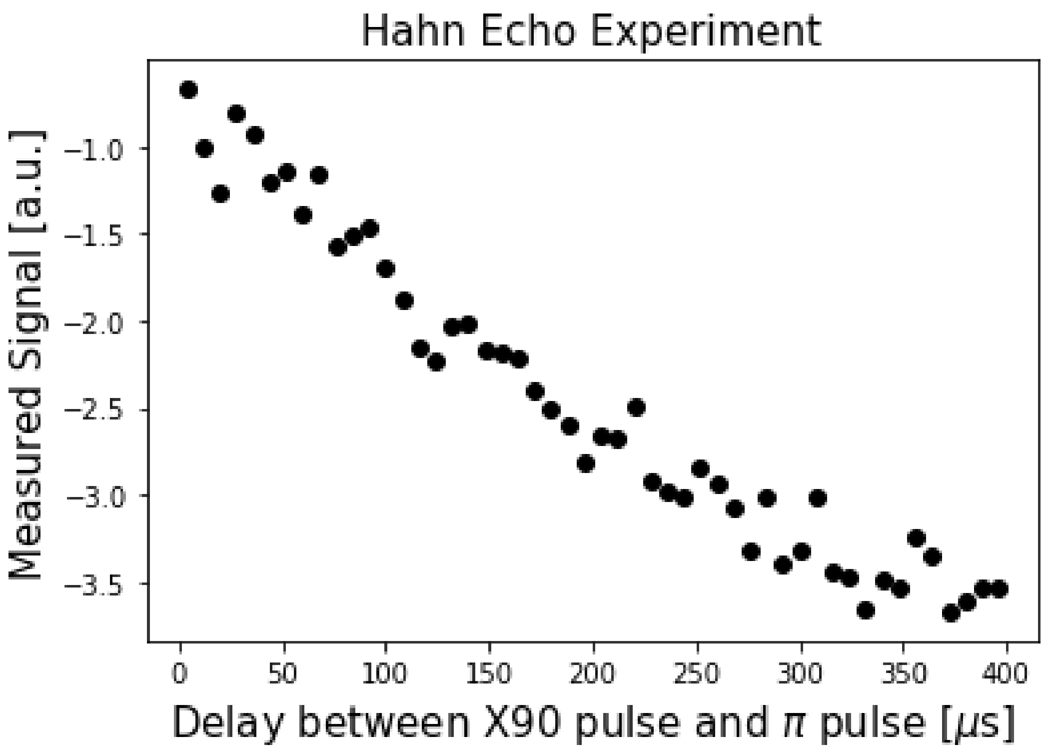
[318]:
```
job5 = backend.run(t2_experiment)
# print(job.job_id())
job_monitor(job5)
```

```
Job Status: job has successfully run
```

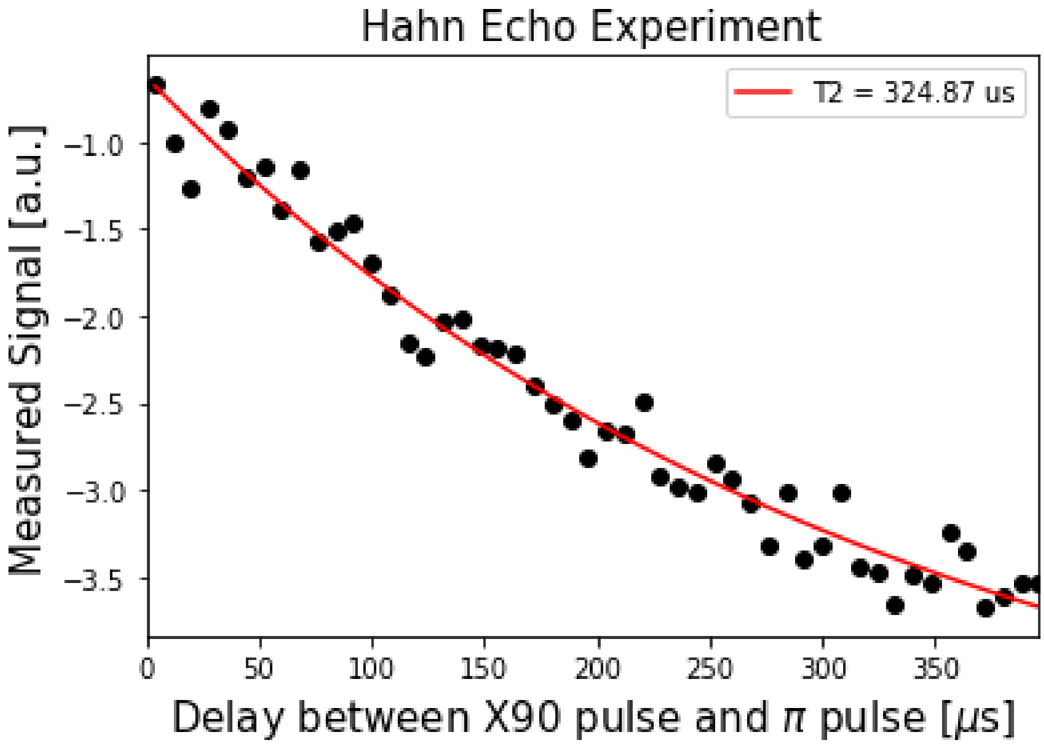
[344]:
```
t2_results5 = job5.result(timeout=120)
```

[404]:
```
t2_values = []
for i in range(len(taus_us)):
    t2_values.append(t2_results1.get_memory(i)[qubit]*scale_factor)

t2_values1 = t2_values
```

```
plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.show()
```



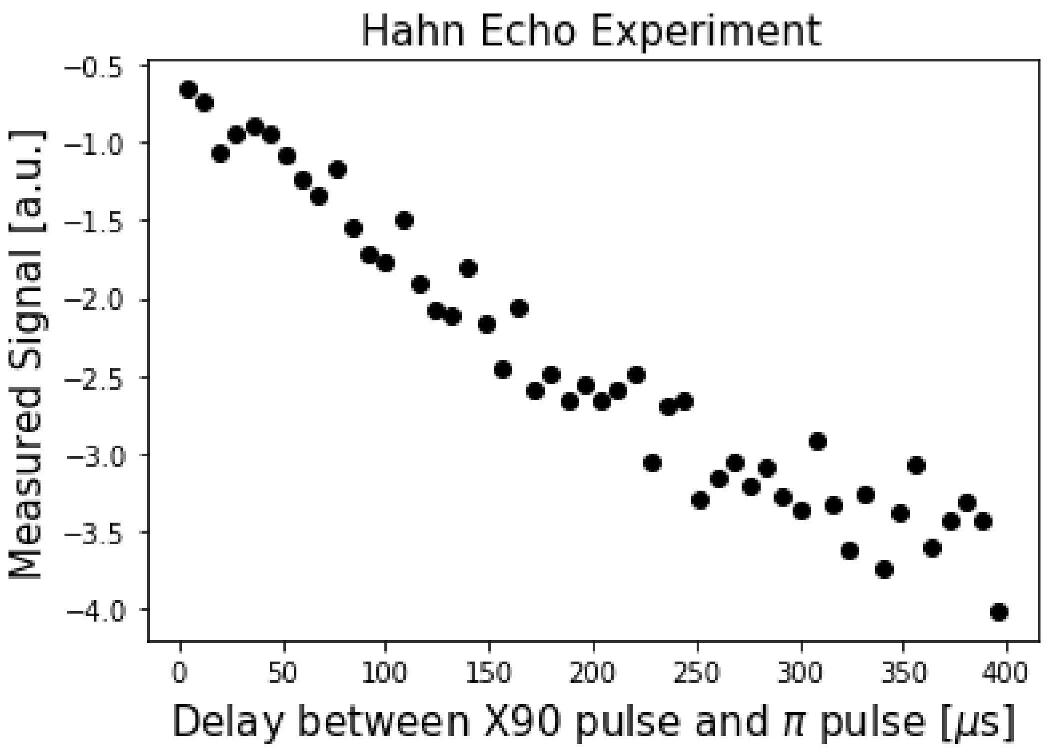
[405]:
```
fit_params, y_fit = fit_function(2*taus_us, np.real(t2_values),
             lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
             [-3, 0, 300])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.plot(2*taus_us, y_fit, color='red', label=f"T2 = {T2:.2f} us")
plt.xlim(0, np.max(2*taus_us))
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.legend()
plt.show()
```



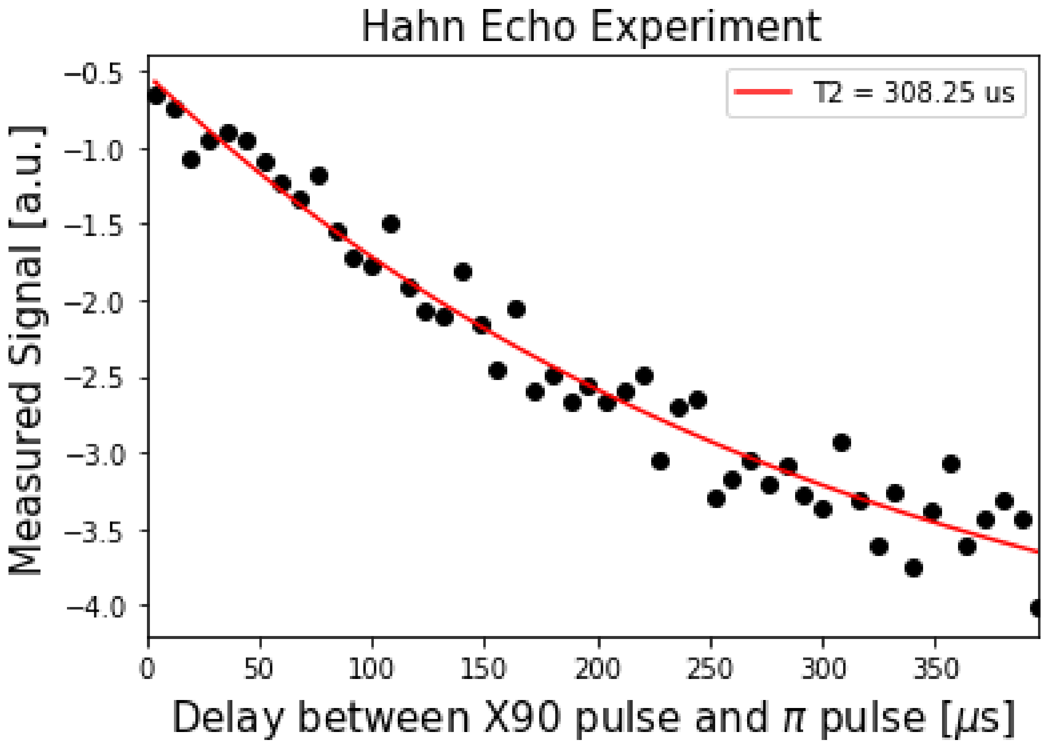
[406]:
```
T2_1 = T2
T2_1
```

[406]: 324.86919937044684

[407]:
```
t2_values = []
for i in range(len(taus_us)):
    t2_values.append(t2_results2.get_memory(i)[qubit]*scale_factor)

t2_values2 = t2_values

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.show()
```



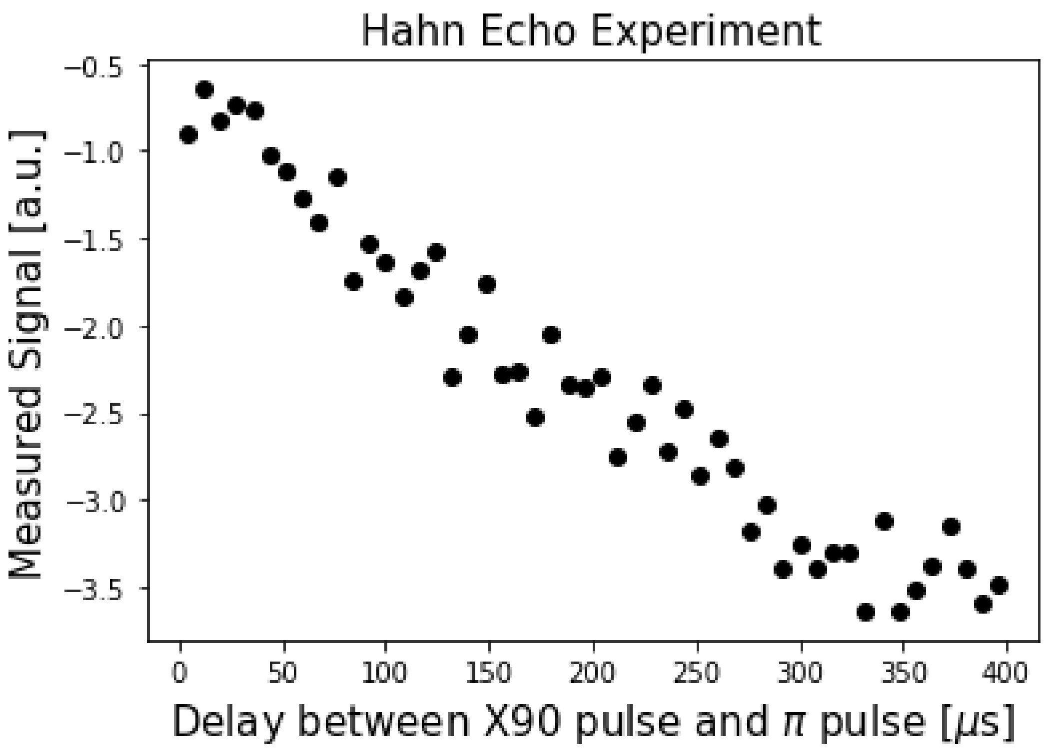
[408]:
```
fit_params, y_fit = fit_function(2*taus_us, np.real(t2_values),
                    lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
                    [-3, 0, 300])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.plot(2*taus_us, y_fit, color='red', label=f"T2 = {T2:.2f} us")
plt.xlim(0, np.max(2*taus_us))
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.legend()
plt.show()
```



[409]:
```
T2_2 = T2
T2_2
```

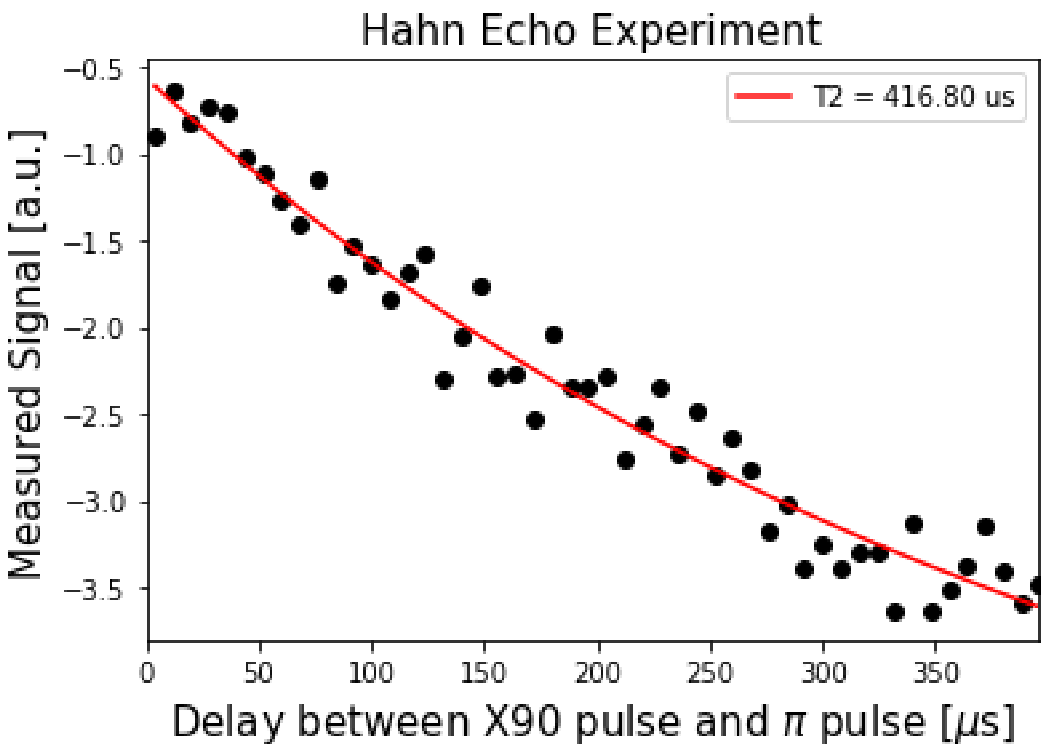
[409]: 308.2546393836576

[410]:
```
t2_values = []
for i in range(len(taus_us)):
    t2_values.append(t2_results3.get_memory(i)[qubit]*scale_factor)

t2_values3 = t2_values

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.show()
```



[411]:
```
fit_params, y_fit = fit_function(2*taus_us, np.real(t2_values),
             lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
             [-3, 0, 100])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.plot(2*taus_us, y_fit, color='red', label=f"T2 = {T2:.2f} us")
plt.xlim(0, np.max(2*taus_us))
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.legend()
plt.show()
```

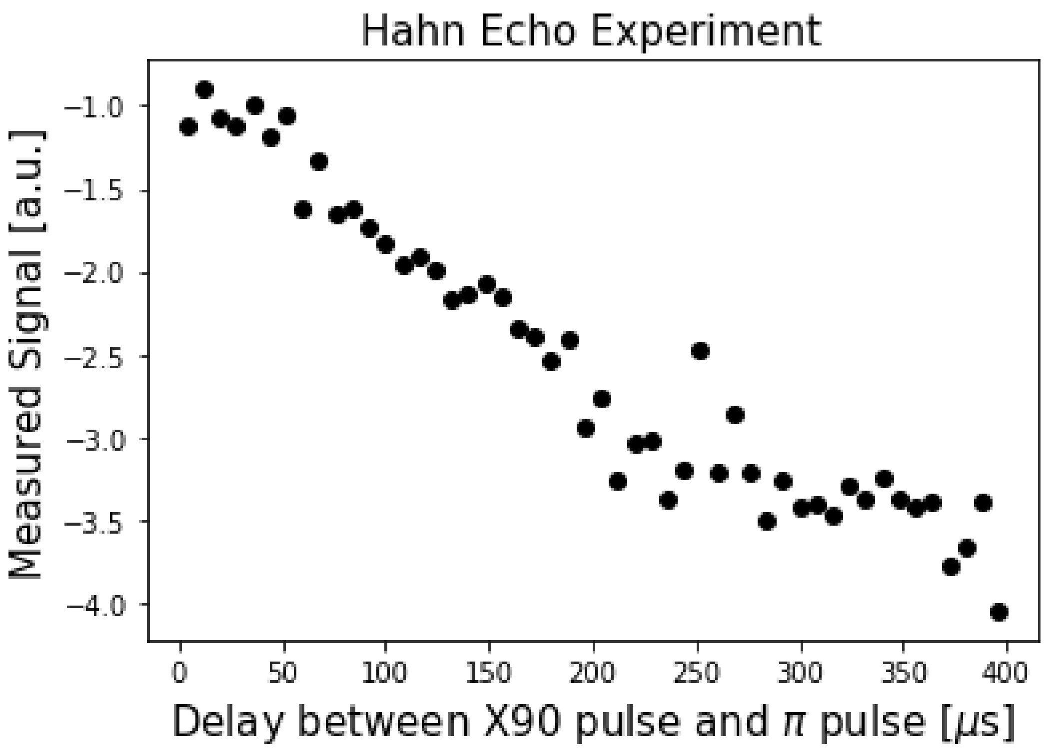
```
C:\Users\johnn\.julia\conda\3\lib\site-packages\ipykernel_launcher.py:2:
RuntimeWarning: overflow encountered in exp
```



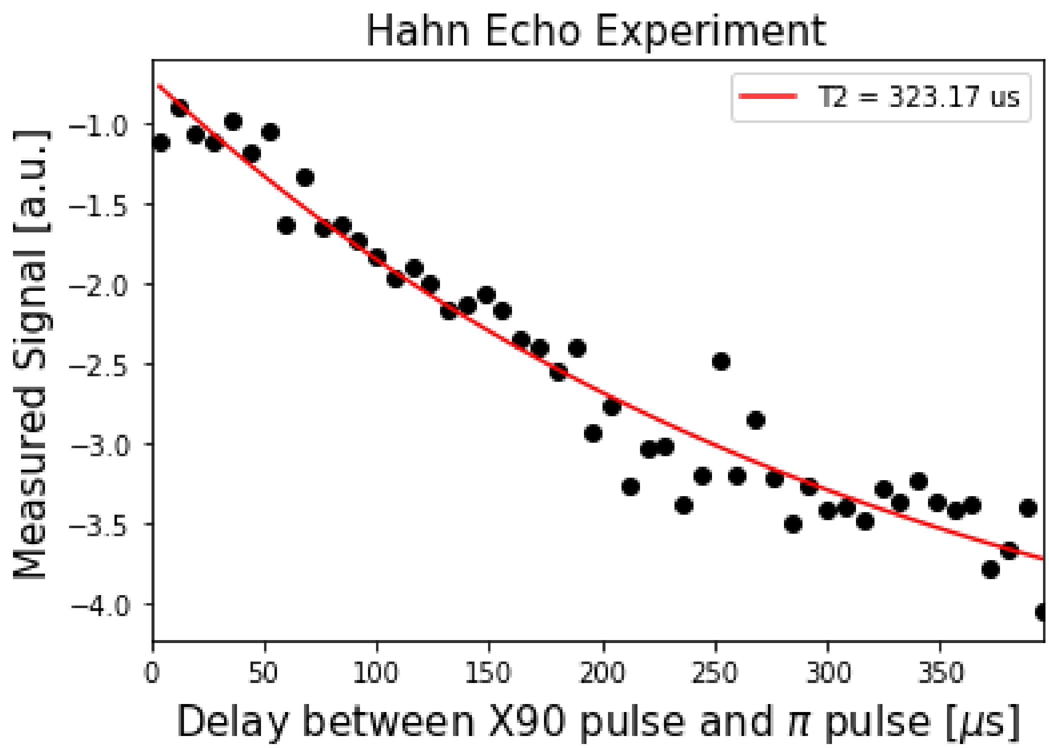
[412]:
```
T2_3 = T2
T2_3
```

[412]: 416.80474822479465

[413]:
```
t2_values = []
for i in range(len(taus_us)):
    t2_values.append(t2_results4.get_memory(i)[qubit]*scale_factor)

t2_values4 = t2_values

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.show()
```



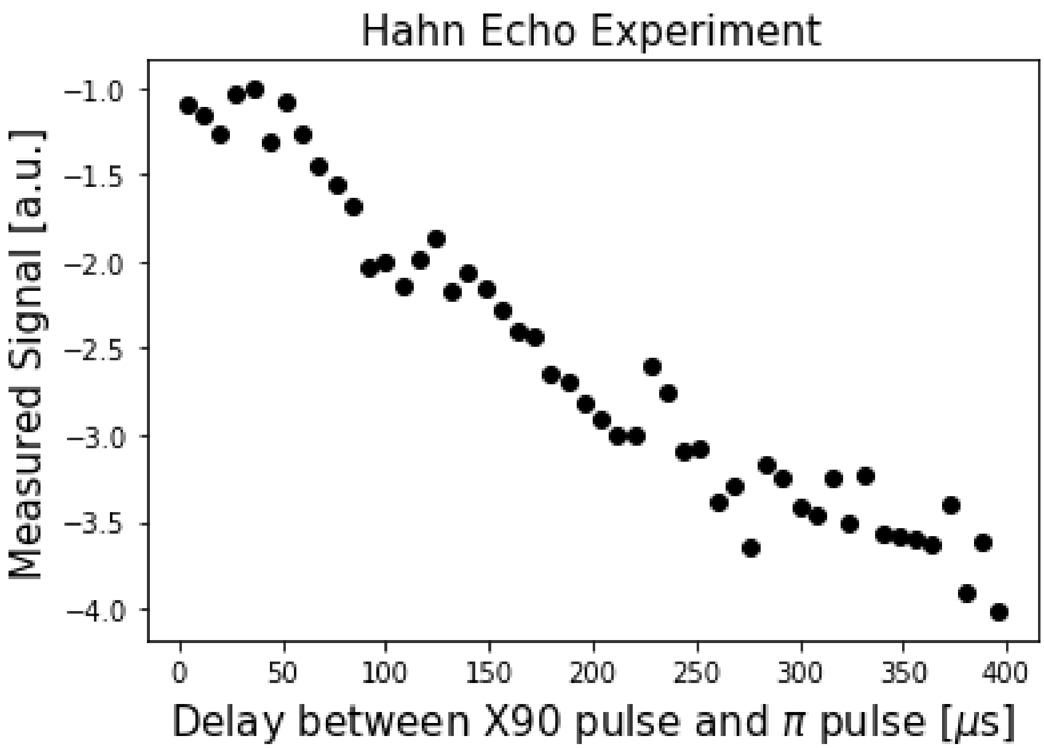
[414]:
```
fit_params, y_fit = fit_function(2*taus_us, np.real(t2_values),
             lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
             [-3, 0, 100])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.plot(2*taus_us, y_fit, color='red', label=f"T2 = {T2:.2f} us")
plt.xlim(0, np.max(2*taus_us))
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.legend()
plt.show()
```



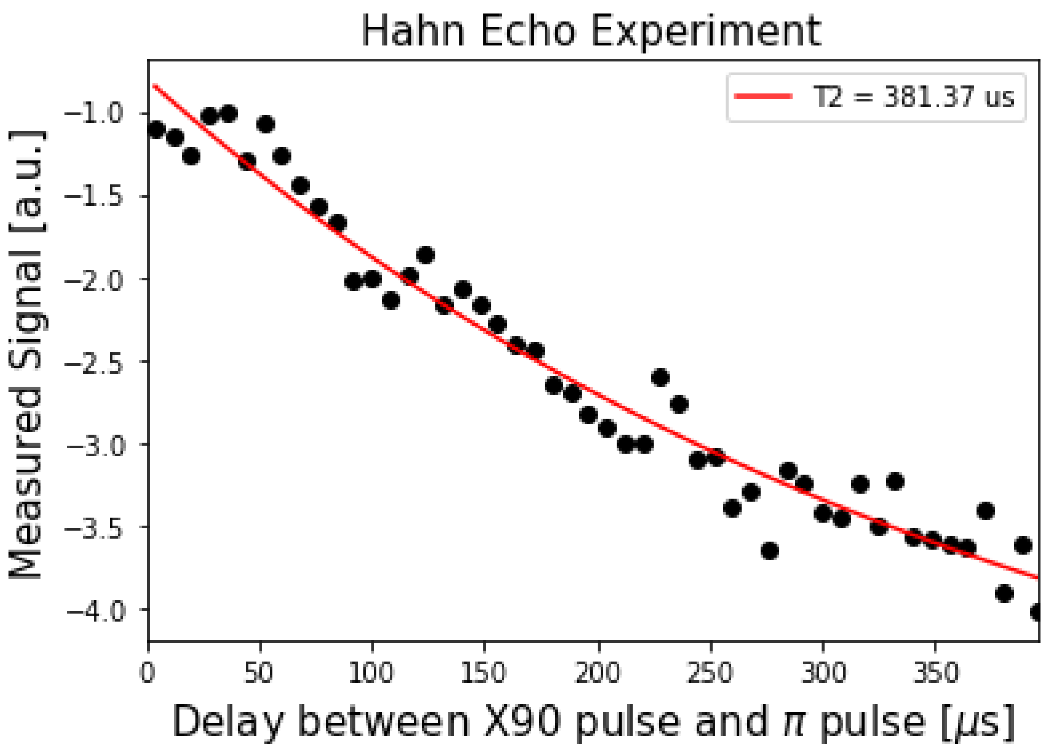
[415]:
```
T2_4 = T2
T2_4
```

[415]: 323.16515210907886

[416]:
```
t2_values = []
for i in range(len(taus_us)):
    t2_values.append(t2_results5.get_memory(i)[qubit]*scale_factor)

t2_values5 = t2_values

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.show()
```



[417]:
```
fit_params, y_fit = fit_function(2*taus_us, np.real(t2_values),
             lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
             [-3, 0, 100])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, np.real(t2_values), color='black')
plt.plot(2*taus_us, y_fit, color='red', label=f"T2 = {T2:.2f} us")
plt.xlim(0, np.max(2*taus_us))
plt.xlabel('Delay between X90 pulse and $\pi$ pulse [$\mu$s]', fontsize=15)
plt.ylabel('Measured Signal [a.u.]', fontsize=15)
plt.title('Hahn Echo Experiment', fontsize=15)
plt.legend()
plt.show()
```



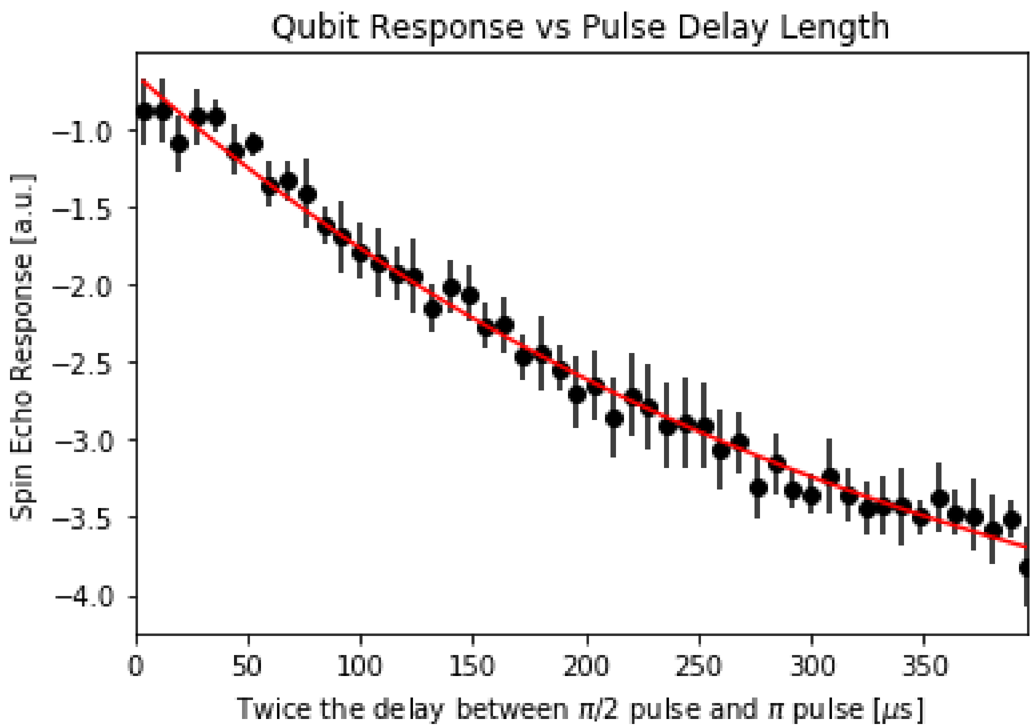
[418]:
```
T2_5 = T2
T2_5
```

[418]: 381.365005536713

[419]:
```
#avg_T2 = (T2_1 + T2_2 + T2_3 + T2_4 + T2_5) / 5
```

[620]:
```
t2_values_avg = np.mean([t2_values1, t2_values2, t2_values3, t2_values4,␣
 ↪t2_values5], axis = 0)
t2_values_std = np.std([t2_values1, t2_values2, t2_values3, t2_values4,␣
 ↪t2_values5], axis = 0)

fit_params, y_fit = fit_function(2*taus_us, np.real(t2_values_avg),
             lambda x, A, B, T2: (A * np.exp(-x / T2) + B),
             [-3, 0, 100])

_, _, T2 = fit_params
print()

plt.scatter(2*taus_us, np.real(t2_values_avg), color='black')
plt.errorbar(2*taus_us, np.real(t2_values_avg), yerr = t2_values_std, color =␣
 ↪'black', ls = 'none')
plt.plot(2*taus_us, y_fit, color='red')
plt.xlim(0, np.max(2*taus_us))
```

```
plt.xlabel('Twice the delay between $\pi/2$ pulse and $\pi$ pulse [$\mu$s]')
plt.ylabel('Spin Echo Response [a.u.]')
plt.title('Qubit Response vs Pulse Delay Length')
plt.show()
```



[578]:
```
T2
```

[578]: 345.9772465453321

[426]:
```
avg_T2 = T2
```

[ ]:

[ ]:

[ ]:

[ ]:

[427]:
```
T2_std = np.sqrt(((T2_1 - avg_T2)**2 + (T2_2 - avg_T2)**2 + (T2_3 - avg_T2)**2␣
↪+ (T2_4 - avg_T2)**2 + (T2_5 - avg_T2)**2) / 5)
```

[428]:
```
avg_T2
```

[428]: 345.9772465453321

[429]:
```
T2_std
```

[429]: 41.611933095606275

[ ]:
```
#phasing time
```

[600]:
```
avg_T2
```

[600]: 345.9772465453321

[601]:
```
rate = 1/(avg_T2*10**-6) - 1/(2*avg_T1*10**-6)
```

[602]:
```
rate
```

[602]: 332.6325609746773

[603]:
```
1/rate
```

[603]: 0.00300632023837296

[437]:
```
T_phi = 1/rate
```

[438]:
```
T_phi
```

[438]: 3006.3202383729617

[ ]:
```
#T_phi is in microseconds
```

[ ]:
```

```

[441]:
```
#Now we do Cavity QED
```

[442]:
```
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.visualization import SchedStyle
# Loading your IBM Q account(s)
provider = IBMQ.load_account()
IBMQ.get_provider(hub='ibm-q', group='open', project='main')
backend = provider.get_backend('ibmq_armonk')
```

```
ibmqfactory.load_account:WARNING:2021-05-10 08:20:37,124: Credentials are
already in use. The existing account in the session will be replaced.
```

[443]:
```
backend_config = backend.configuration()
backend_defaults = backend.defaults()
backend.configuration().parametric_pulses = []  # Will allow us to send a␣
 ↪larger waveform for our experiments
style = SchedStyle(figsize=(3, 2), title_font_size=10, axis_font_size=8) #␣
 ↪style for displaying the pulse sequence
```

[444]:
```
from scipy.optimize import curve_fit
from scipy.signal import savgol_filter

# samples need to be multiples of 16 to accommodate the hardware limitations
```

```
def get_closest_multiple_of_16(num):
    return int(num + 8 ) - (int(num + 8 ) % 16)

# process the reflective measurement results
# in a reflective measurement the data is encoded in the phase of the output␣
↪signal
def process_reflective_measurement(freqs,values):
    phase_grad = np.gradient(savgol_filter(np.unwrap(np.
↪angle(values)),3,2),freqs)
    return (phase_grad-min(phase_grad))/(max(phase_grad)-min(phase_grad)) - 1

# lorentzian function
def lorentzian(f, f0, k, a, offs):
    return -a*k/(2*np.pi)/((k/2)**2+(f-f0)**2)+offs

#fit_lorentzian takes two arrays that contain the frequencies and experimental␣
↪output values of each frequency respectively.
#returns the lorentzian parameters that best fits this output of the experiment.
#popt are the fit parameters and pcov is the covariance matrix for the fit
def fit_lorentzian(freqs,values):
    p0=[freqs[np.argmin(values)],(freqs[-1]-freqs[0])/2,min(values),0]
    bounds=([freqs[0],0,-np.inf,-np.inf],[freqs[-1],freqs[-1]-freqs[0],np.
↪inf,np.inf])
    popt,pcov=curve_fit(lorentzian, freqs, values, p0=p0, bounds=bounds)
    return popt,pcov

# exponential function
def exponential(t,tau,a,offset):
    return a*np.exp(-t/tau)+offset

# fit an exponential function
def fit_exponential(ts,values):
    p0=[np.average(ts),1,0]
    return curve_fit(exponential, ts, values,p0=p0)
```

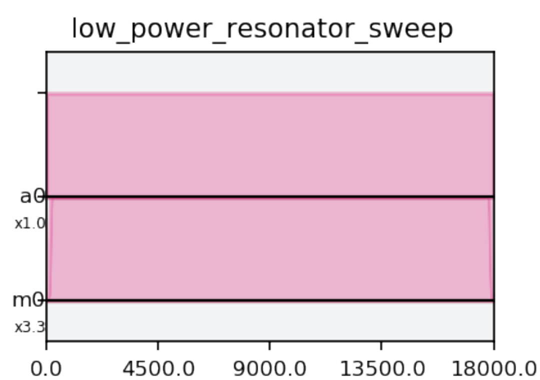
[445]:
```
#We measure kappa
```

[446]:
```
from qiskit import pulse            # This is where we access all of our Pulse␣
↪features!
from qiskit.pulse import Play, Acquire
import qiskit.pulse.library as pulse_lib
import numpy as np

dt=backend_config.dt   # hardware resolution

qubit=0   # qubit used in our experiment
```

```
readout_time = 4e-6
readout_sigma = 10e-9

# low power drive for the resonator for dispersive readout
# We use a square pulse with a Guassian rise and fall time
readout_drive_low_power=pulse_lib.GaussianSquare(duration =␣
↪get_closest_multiple_of_16(readout_time//dt),
                             amp = .3,
                             sigma = get_closest_multiple_of_16(readout_sigma//
↪dt),
                             width =␣
↪get_closest_multiple_of_16((readout_time-8*readout_sigma)//dt),
                             name = 'low power readout tone')

meas_chan = pulse.MeasureChannel(qubit) # resonator channel
acq_chan = pulse.AcquireChannel(qubit)  # readout signal acquisition channel

# readout output signal acquisition setup
acquisition_time = readout_time   # We want to acquire the readout signal for␣
↪the full duration of the readout
```

[447]:
```
# build the pulse sequence for low power resonator spectroscopy
with pulse.build(name='low_power_resonator_sweep') as pulse_low_power:
    #drive the resonator with low power
    pulse.play(readout_drive_low_power, meas_chan)
    #acquire the readout signal
    pulse.acquire(duration = get_closest_multiple_of_16(acquisition_time//dt),
                    qubit_or_channel = acq_chan,
                    register = pulse.MemorySlot(0))

pulse_low_power.draw(style=style)
```

[447]:



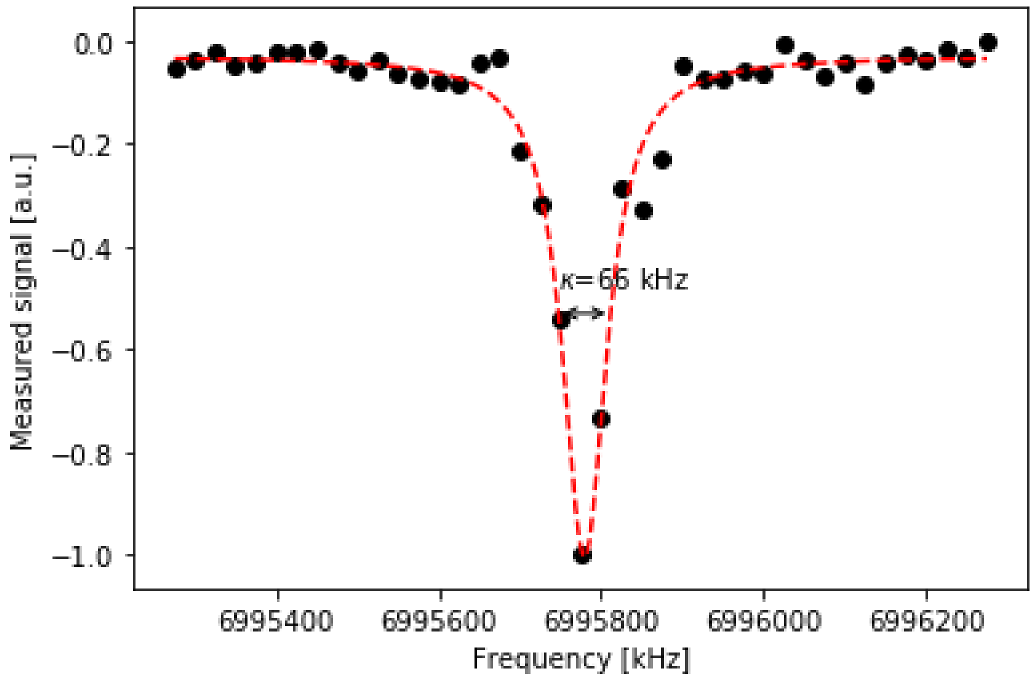
[448]:
```
center_freq = 6995775000.0  # an estimate for the resonator frequency
freq_span = 1e6 # resonator scan span. The span should be larger than the␣
 ↪resonator linewidth kappa

frequencies_range = np.linspace(center_freq-freq_span/2,center_freq+freq_span/
 ↪2,41)
# list of resonator frequencies for the experiment
schedule_frequencies = [{meas_chan: freq} for freq in frequencies_range]
```

[449]:
```
from qiskit import assemble
from qiskit.tools.monitor import job_monitor

num_shots_per_frequency = 8*1024
frequency_sweep_low_power = assemble(pulse_low_power,
                                     backend=backend,
                                     meas_level=1,
                                     meas_return='avg',
                                     shots=num_shots_per_frequency,
                                     schedule_los=schedule_frequencies)

job_low_power = backend.run(frequency_sweep_low_power)
job_monitor(job_low_power)

low_power_sweep_results1 = job_low_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[450]:
```
import matplotlib.pyplot as plt

low_power_sweep_values = []
for i in range(len(low_power_sweep_results1.results)):
    res_low_power = low_power_sweep_results1.get_memory(i)
    low_power_sweep_values.append(res_low_power[qubit])

low_power_sweep_values1 =␣
 ↪process_reflective_measurement(frequencies_range,low_power_sweep_values)

plt.scatter(frequencies_range/1e3, low_power_sweep_values1, color='black')

popt_low_power1,_=fit_lorentzian(frequencies_range,low_power_sweep_values1)

popt_low_power1,_=fit_lorentzian(frequencies_range,low_power_sweep_values1)
f0, kappa, a, offset = popt_low_power1

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power1), color='red', ls='--')
```

```python
plt.annotate("", xy=((f0-kappa/2)/1e3, offset-1/2), xytext=((f0+kappa/2)/1e3, offset-1/2), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\kappa$={:d} kHz".format(int(kappa/1e3)), xy=((f0-kappa/2)/1e3, offset-.45), color='black')

plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[451]:
```python
kappa_1 = kappa
kappa_1 #in Hertz
```

[451]: 66049.6307742988

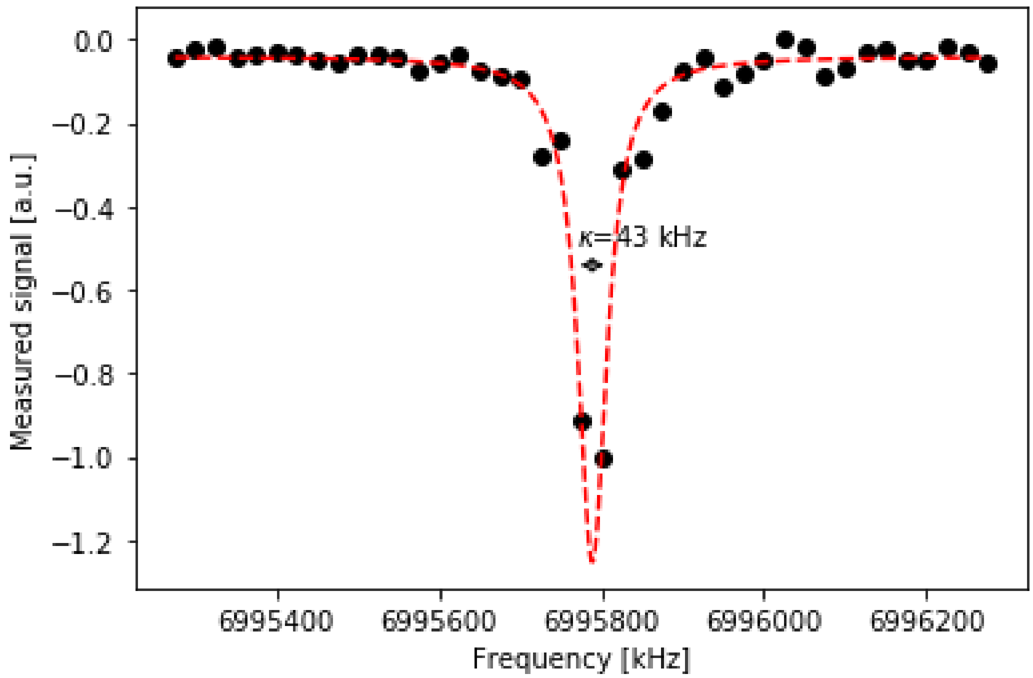
[452]:
```python
center_freq = 6995775000.0  # an estimate for the resonator frequency
freq_span = 1e6 # resonator scan span. The span should be larger than the resonator linewidth kappa

frequencies_range = np.linspace(center_freq-freq_span/2,center_freq+freq_span/2,41)
# list of resonator frequencies for the experiment
schedule_frequencies = [{meas_chan: freq} for freq in frequencies_range]
```

[453]:
```python
num_shots_per_frequency = 8*1024
frequency_sweep_low_power = assemble(pulse_low_power,
                                    backend=backend,
```

```
                                    meas_level=1,
                                    meas_return='avg',
                                    shots=num_shots_per_frequency,
                                    schedule_los=schedule_frequencies)

job_low_power = backend.run(frequency_sweep_low_power)
job_monitor(job_low_power)

low_power_sweep_results2 = job_low_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[454]:
```
import matplotlib.pyplot as plt

low_power_sweep_values = []
for i in range(len(low_power_sweep_results2.results)):
    res_low_power = low_power_sweep_results2.get_memory(i)
    low_power_sweep_values.append(res_low_power[qubit])

low_power_sweep_values2 =␣
 ↪process_reflective_measurement(frequencies_range,low_power_sweep_values)

plt.scatter(frequencies_range/1e3, low_power_sweep_values2, color='black')

popt_low_power2,_=fit_lorentzian(frequencies_range,low_power_sweep_values2)

popt_low_power2,_=fit_lorentzian(frequencies_range,low_power_sweep_values2)
f0, kappa, a, offset = popt_low_power2

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power2), color='red', ls='--')
plt.annotate("", xy=((f0-kappa/2)/1e3, offset-1/2), xytext=((f0+kappa/2)/1e3,␣
 ↪offset-1/2), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\kappa$={:d} kHz".format(int(kappa/1e3)), xy=((f0-kappa/2)/1e3,␣
 ↪offset-.45), color='black')

plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[455]:
```
kappa_2 = kappa
kappa_2
```

[455]: 43328.47816938712

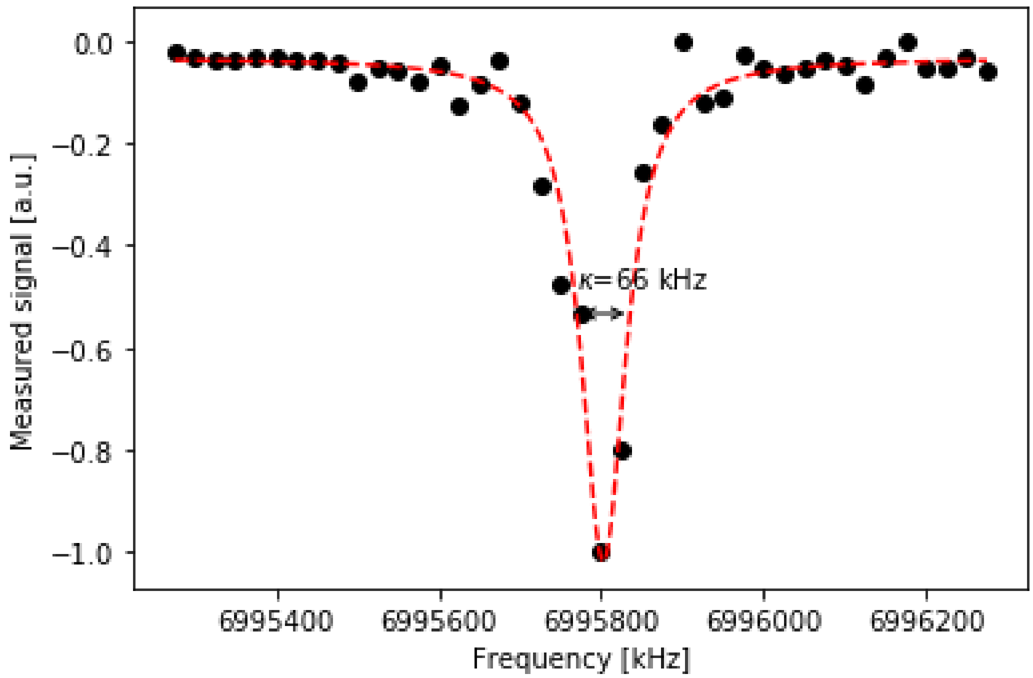
[456]:
```
center_freq = 6995775000.0  # an estimate for the resonator frequency
freq_span = 1e6 # resonator scan span. The span should be larger than the␣
 ↪resonator linewidth kappa

frequencies_range = np.linspace(center_freq-freq_span/2,center_freq+freq_span/
 ↪2,41)
# list of resonator frequencies for the experiment
schedule_frequencies = [{meas_chan: freq} for freq in frequencies_range]
```

[457]:
```
num_shots_per_frequency = 8*1024
frequency_sweep_low_power = assemble(pulse_low_power,
                                   backend=backend,
                                   meas_level=1,
                                   meas_return='avg',
                                   shots=num_shots_per_frequency,
                                   schedule_los=schedule_frequencies)

job_low_power = backend.run(frequency_sweep_low_power)
job_monitor(job_low_power)

low_power_sweep_results3 = job_low_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[458]:

```
import matplotlib.pyplot as plt

low_power_sweep_values = []
for i in range(len(low_power_sweep_results3.results)):
    res_low_power = low_power_sweep_results3.get_memory(i)
    low_power_sweep_values.append(res_low_power[qubit])

low_power_sweep_values3 =␣
 ↪process_reflective_measurement(frequencies_range,low_power_sweep_values)

plt.scatter(frequencies_range/1e3, low_power_sweep_values3, color='black')

popt_low_power3,_=fit_lorentzian(frequencies_range,low_power_sweep_values3)

popt_low_power3,_=fit_lorentzian(frequencies_range,low_power_sweep_values3)
f0, kappa, a, offset = popt_low_power3

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power3), color='red', ls='--')
plt.annotate("", xy=((f0-kappa/2)/1e3, offset-1/2), xytext=((f0+kappa/2)/1e3,␣
 ↪offset-1/2), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\kappa$={:d} kHz".format(int(kappa/1e3)), xy=((f0-kappa/2)/1e3,␣
 ↪offset-.45), color='black')

plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[459]:
```
kappa_3 = kappa
kappa_3
```

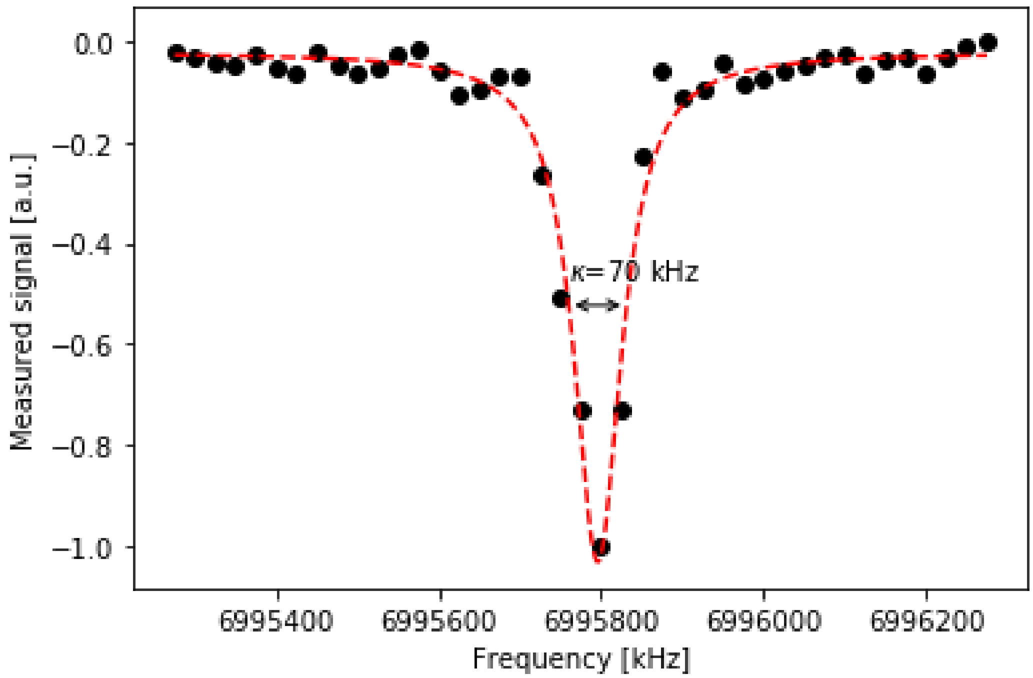
[459]: 66771.55099537665

[460]:
```
center_freq = 6995775000.0  # an estimate for the resonator frequency
freq_span = 1e6 # resonator scan span. The span should be larger than the␣
↪resonator linewidth kappa

frequencies_range = np.linspace(center_freq-freq_span/2,center_freq+freq_span/
↪2,41)
# list of resonator frequencies for the experiment
schedule_frequencies = [{meas_chan: freq} for freq in frequencies_range]
```

[461]:
```
num_shots_per_frequency = 8*1024
frequency_sweep_low_power = assemble(pulse_low_power,
                                     backend=backend,
                                     meas_level=1,
                                     meas_return='avg',
                                     shots=num_shots_per_frequency,
                                     schedule_los=schedule_frequencies)

job_low_power = backend.run(frequency_sweep_low_power)
job_monitor(job_low_power)

low_power_sweep_results4 = job_low_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[462]:
```python
import matplotlib.pyplot as plt

low_power_sweep_values = []
for i in range(len(low_power_sweep_results4.results)):
    res_low_power = low_power_sweep_results4.get_memory(i)
    low_power_sweep_values.append(res_low_power[qubit])

low_power_sweep_values4 =␣
↪process_reflective_measurement(frequencies_range,low_power_sweep_values)

plt.scatter(frequencies_range/1e3, low_power_sweep_values4, color='black')

popt_low_power4,_=fit_lorentzian(frequencies_range,low_power_sweep_values4)

popt_low_power4,_=fit_lorentzian(frequencies_range,low_power_sweep_values4)
f0, kappa, a, offset = popt_low_power4

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power4), color='red', ls='--')
plt.annotate("", xy=((f0-kappa/2)/1e3, offset-1/2), xytext=((f0+kappa/2)/1e3,␣
↪offset-1/2), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\kappa$={:d} kHz".format(int(kappa/1e3)), xy=((f0-kappa/2)/1e3,␣
↪offset-.45), color='black')

plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[463]:
```
kappa_4 = kappa
kappa_4
```

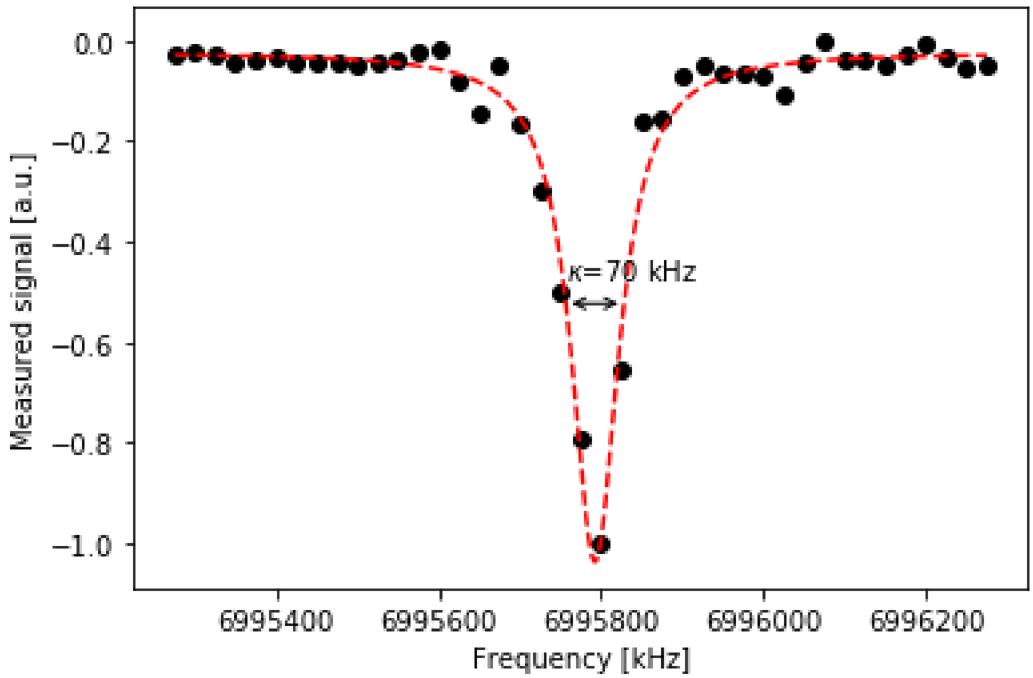
[463]: 70993.80792606095

[464]:
```
center_freq = 6995775000.0  # an estimate for the resonator frequency
freq_span = 1e6 # resonator scan span. The span should be larger than the␣
↪resonator linewidth kappa

frequencies_range = np.linspace(center_freq-freq_span/2,center_freq+freq_span/
↪2,41)
# list of resonator frequencies for the experiment
schedule_frequencies = [{meas_chan: freq} for freq in frequencies_range]
```

[465]:
```
num_shots_per_frequency = 8*1024
frequency_sweep_low_power = assemble(pulse_low_power,
                                     backend=backend,
                                     meas_level=1,
                                     meas_return='avg',
                                     shots=num_shots_per_frequency,
                                     schedule_los=schedule_frequencies)

job_low_power = backend.run(frequency_sweep_low_power)
job_monitor(job_low_power)

low_power_sweep_results5 = job_low_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[466]:
```python
import matplotlib.pyplot as plt

low_power_sweep_values = []
for i in range(len(low_power_sweep_results5.results)):
    res_low_power = low_power_sweep_results5.get_memory(i)
    low_power_sweep_values.append(res_low_power[qubit])

low_power_sweep_values5 =␣
 ↪process_reflective_measurement(frequencies_range,low_power_sweep_values)

plt.scatter(frequencies_range/1e3, low_power_sweep_values5, color='black')

popt_low_power5,_=fit_lorentzian(frequencies_range,low_power_sweep_values5)

popt_low_power5,_=fit_lorentzian(frequencies_range,low_power_sweep_values5)
f0, kappa, a, offset = popt_low_power5

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power5), color='red', ls='--')
plt.annotate("", xy=((f0-kappa/2)/1e3, offset-1/2), xytext=((f0+kappa/2)/1e3,␣
 ↪offset-1/2), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\kappa$={:d} kHz".format(int(kappa/1e3)), xy=((f0-kappa/2)/1e3,␣
 ↪offset-.45), color='black')

plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[467]:
```
kappa_5 = kappa
kappa_5
```

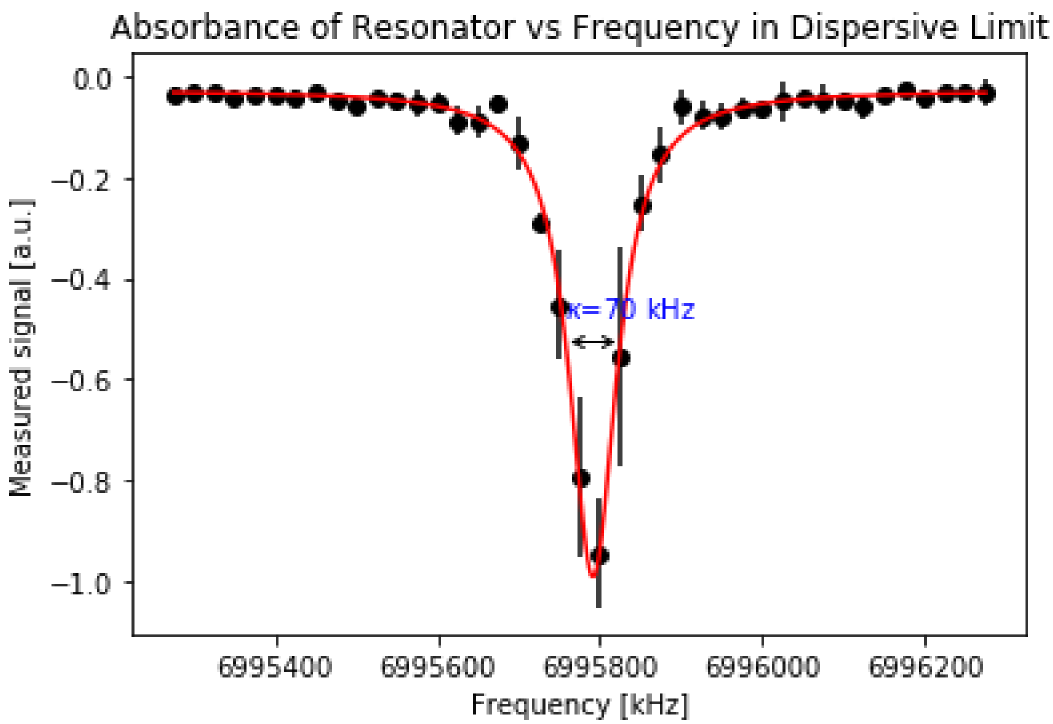
[467]: 70789.04543473611

[509]:
```
kappa_avgs = np.mean([low_power_sweep_values1, low_power_sweep_values2,␣
↪low_power_sweep_values3, low_power_sweep_values4, low_power_sweep_values5],␣
↪axis = 0)
```

[510]:
```
kappa_stds =np.std([low_power_sweep_values1, low_power_sweep_values2,␣
↪low_power_sweep_values3, low_power_sweep_values4, low_power_sweep_values5],␣
↪axis = 0)
```

[556]:
```
plt.scatter(frequencies_range/1e3, kappa_avgs, color='black')

popt_low_power,_=fit_lorentzian(frequencies_range,kappa_avgs)

popt_low_power,_=fit_lorentzian(frequencies_range,kappa_avgs)
f0, kappa, a, offset = popt_low_power

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power), color='red')
plt.annotate("", xy=((f0-kappa/2)/1e3, offset-1/2), xytext=((f0+kappa/2)/1e3,␣
↪offset-1/2), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\kappa$={:d} kHz".format(int(kappa/1e3)), xy=((f0-kappa/2)/1e3,␣
↪offset-.45), color='blue')
```

```
plt.errorbar(frequencies_range/1e3, kappa_avgs, yerr = kappa_stds, color =␣
↪'black', ls = 'none')

plt.title('Absorbance of Resonator vs Frequency in Dispersive Limit')

plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.show()
```



[527]:
```
avg_kappa = kappa
```

[ ]:
```
```

[528]:
```
#avg_kappa = (kappa_1 + kappa_2 + kappa_3 + kappa_4 + kappa_5) / 5
```

[529]:
```
kappa_std = np.sqrt(((kappa_1 - avg_kappa)**2 + (kappa_2 - avg_kappa)**2 +␣
↪(kappa_3 - avg_kappa)**2 + (kappa_4 - avg_kappa)**2 + (kappa_5 -␣
↪avg_kappa)**2)/ 5)
```

[ ]:
```
```

[530]:
```
avg_kappa
```

[530]: 70229.00276257984

[531]:
```
kappa_std
```

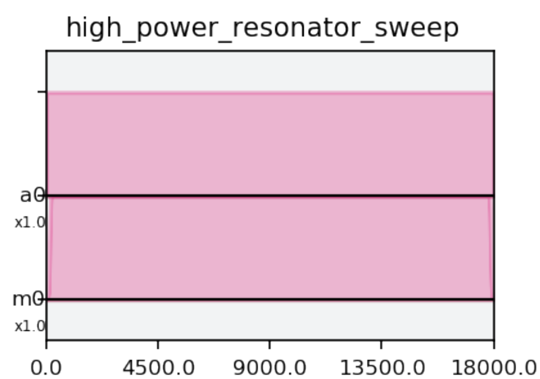
[531]: 12279.72248107142

[473]:
```
#now chi and g
```

[474]:
```
readout_drive_high_power=pulse_lib.GaussianSquare(duration =␣
 ↪get_closest_multiple_of_16(readout_time//dt),
                              amp = 1, # High drive amplitude
                              sigma = get_closest_multiple_of_16(readout_sigma//
 ↪dt),
                              width =␣
 ↪get_closest_multiple_of_16((readout_time-8*readout_sigma)//dt),
                              name = 'high power readout tone')
```

[475]:
```
# build the pulse sequence for high power resonator spectroscopy
with pulse.build(name='high_power_resonator_sweep') as pulse_high_power:
    #drive the resonator with high power
    pulse.play(readout_drive_high_power, meas_chan)
    #acquire the readout signal
    pulse.acquire(duration = get_closest_multiple_of_16(acquisition_time//dt),
                  qubit_or_channel = acq_chan,
                  register = pulse.MemorySlot(0))

pulse_high_power.draw(style=style)
```

[475]:

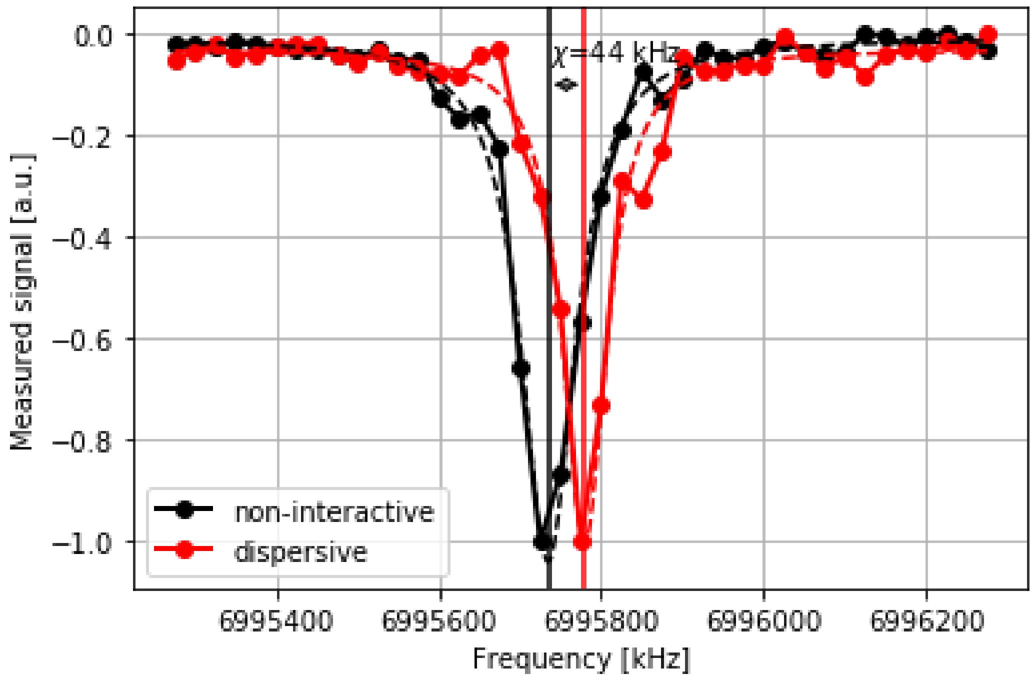


[476]:
```
frequency_sweep_high_power = assemble(pulse_high_power,
                                      backend=backend,
                                      meas_level=1,
                                      meas_return='avg',
                                      shots=num_shots_per_frequency,
                                      schedule_los=schedule_frequencies)

job_high_power = backend.run(frequency_sweep_high_power)
job_monitor(job_high_power)

high_power_sweep_results = job_high_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[477]:
```
high_power_sweep_values = []
for i in range(len(high_power_sweep_results.results)):
    res_high_power = high_power_sweep_results.get_memory(i)
    high_power_sweep_values.append(res_high_power[qubit])

high_power_sweep_values1 =␣
 ↪process_reflective_measurement(frequencies_range,high_power_sweep_values)

popt_high_power1,_=fit_lorentzian(frequencies_range,high_power_sweep_values1)
```

[478]:
```
plt.plot(frequencies_range/1e3, high_power_sweep_values1, '-o', color='black',␣
 ↪lw=2, label='non-interactive')
plt.plot(frequencies_range/1e3, low_power_sweep_values1,'-o', color='red',␣
 ↪lw=2, label='dispersive')

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_high_power1), color='black', ls='--')
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power1), color='red', ls='--')

plt.axvline(x=popt_low_power1[0]/1e3, color='red')
plt.axvline(x=popt_high_power1[0]/1e3, color='black')

chi=popt_low_power1[0]-popt_high_power1[0]
plt.annotate("", xy=(popt_low_power1[0]/1e3, -.1), xytext=(popt_high_power1[0]/
 ↪1e3, -.1), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\chi$={:d} kHz".format(int(chi/1e3)), xy=(popt_high_power1[0]/
 ↪1e3, -.05), color='black')

plt.grid()
plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.legend()
plt.show()

print(r'chi={:.1f} kHz'.format((popt_low_power1[0]-popt_high_power1[0])/1e3))
Delta=abs(backend_defaults.meas_freq_est[qubit] - backend_defaults.
 ↪qubit_freq_est[qubit])
print(r'g={:.1f} MHz'.format(np.sqrt(chi*Delta)/1e6))

chi = popt_low_power1[0]-popt_high_power1[0]
g = np.sqrt(chi*Delta)/1e6
```




```
chi=44.9 kHz
g=9.5 MHz
```

[479]:
```
chi_1 = chi
chi_1
```

[479]: 44919.64657020569

[480]:
```
g_1 = g
g_1
```

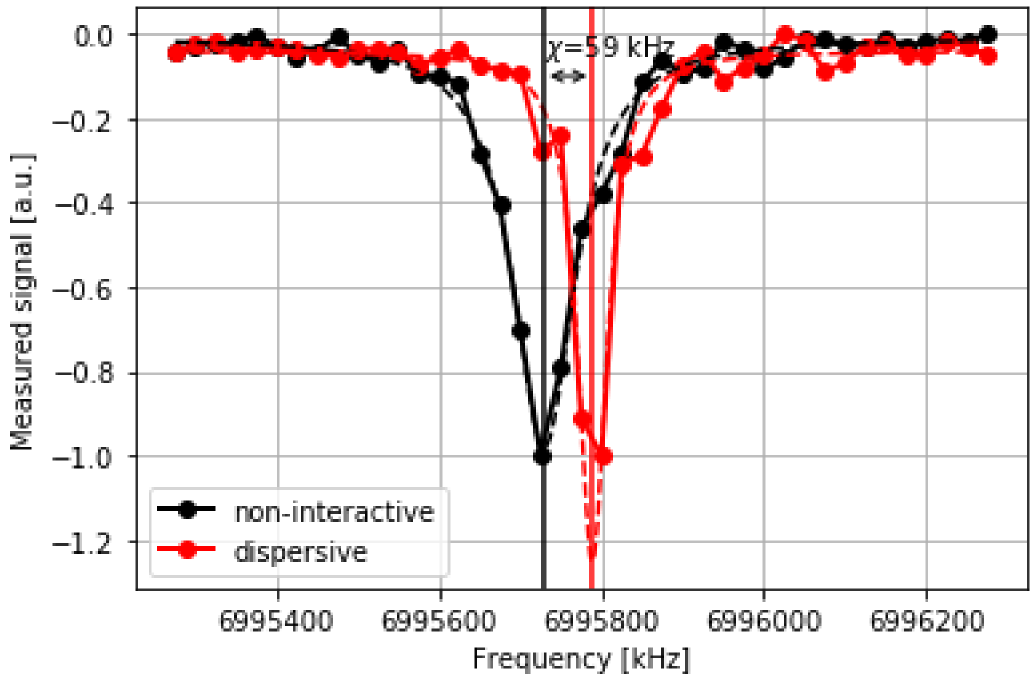
[480]: 9.529189431909755

[481]:
```
frequency_sweep_high_power = assemble(pulse_high_power,
                                      backend=backend,
                                      meas_level=1,
                                      meas_return='avg',
                                      shots=num_shots_per_frequency,
                                      schedule_los=schedule_frequencies)

job_high_power = backend.run(frequency_sweep_high_power)
job_monitor(job_high_power)

high_power_sweep_results = job_high_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[482]:
```python
high_power_sweep_values = []
for i in range(len(high_power_sweep_results.results)):
    res_high_power = high_power_sweep_results.get_memory(i)
    high_power_sweep_values.append(res_high_power[qubit])

high_power_sweep_values2 =␣
 ↪process_reflective_measurement(frequencies_range,high_power_sweep_values)

popt_high_power2,_=fit_lorentzian(frequencies_range,high_power_sweep_values2)
```

[483]:
```python
plt.plot(frequencies_range/1e3, high_power_sweep_values2, '-o', color='black',␣
 ↪lw=2, label='non-interactive')
plt.plot(frequencies_range/1e3, low_power_sweep_values2,'-o', color='red',␣
 ↪lw=2, label='dispersive')

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_high_power2), color='black', ls='--')
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power2), color='red', ls='--')

plt.axvline(x=popt_low_power2[0]/1e3, color='red')
plt.axvline(x=popt_high_power2[0]/1e3, color='black')

chi=popt_low_power2[0]-popt_high_power2[0]
plt.annotate("", xy=(popt_low_power2[0]/1e3, -.1), xytext=(popt_high_power2[0]/
 ↪1e3, -.1), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\chi$={:d} kHz".format(int(chi/1e3)), xy=(popt_high_power2[0]/
 ↪1e3, -.05), color='black')

plt.grid()
plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.legend()
plt.show()

print(r'chi={:.1f} kHz'.format((popt_low_power2[0]-popt_high_power2[0])/1e3))
Delta=abs(backend_defaults.meas_freq_est[qubit] - backend_defaults.
 ↪qubit_freq_est[qubit])
print(r'g={:.1f} MHz'.format(np.sqrt(chi*Delta)/1e6))

chi = popt_low_power2[0]-popt_high_power2[0]
g = np.sqrt(chi*Delta)/1e6
```




```
chi=59.6 kHz
g=11.0 MHz
```

[484]:
```
chi_2 = chi
chi_2
```

[484]: 59644.453300476074

[485]:
```
g_2 = g
g_2
```

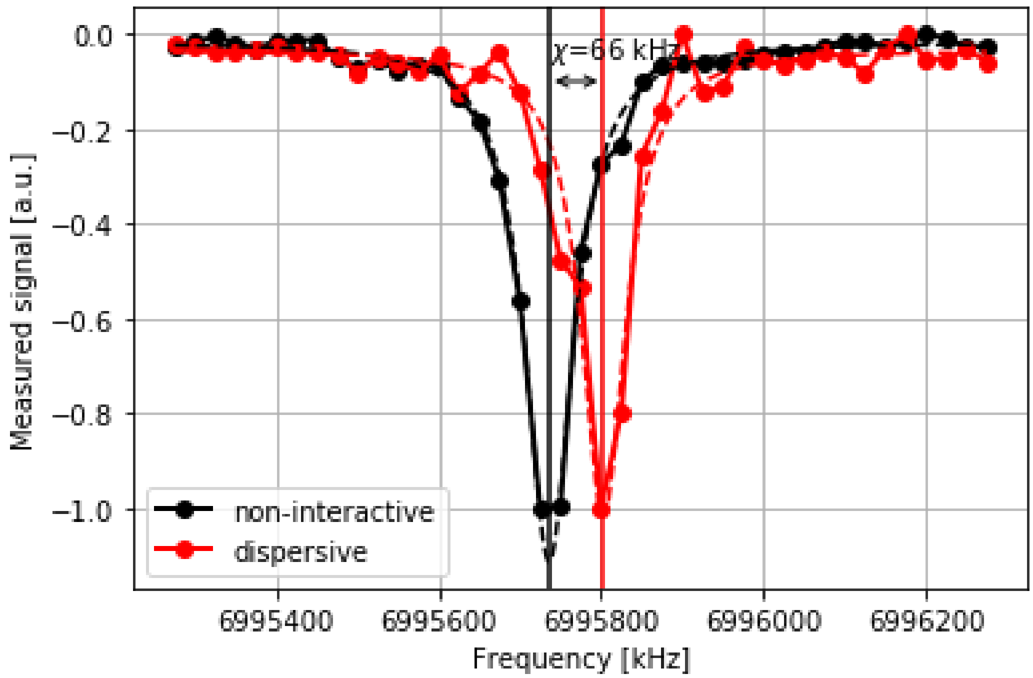
[485]: 10.98051795062963

[486]:
```
frequency_sweep_high_power = assemble(pulse_high_power,
                                      backend=backend,
                                      meas_level=1,
                                      meas_return='avg',
                                      shots=num_shots_per_frequency,
                                      schedule_los=schedule_frequencies)

job_high_power = backend.run(frequency_sweep_high_power)
job_monitor(job_high_power)

high_power_sweep_results = job_high_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[487]:
```python
high_power_sweep_values = []
for i in range(len(high_power_sweep_results.results)):
    res_high_power = high_power_sweep_results.get_memory(i)
    high_power_sweep_values.append(res_high_power[qubit])

high_power_sweep_values3 =␣
 ↪process_reflective_measurement(frequencies_range,high_power_sweep_values)

popt_high_power3,_=fit_lorentzian(frequencies_range,high_power_sweep_values3)
```

[488]:
```python
plt.plot(frequencies_range/1e3, high_power_sweep_values3, '-o', color='black',␣
 ↪lw=2, label='non-interactive')
plt.plot(frequencies_range/1e3, low_power_sweep_values3,'-o', color='red',␣
 ↪lw=2, label='dispersive')

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_high_power3), color='black', ls='--')
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power3), color='red', ls='--')

plt.axvline(x=popt_low_power3[0]/1e3, color='red')
plt.axvline(x=popt_high_power3[0]/1e3, color='black')

chi=popt_low_power3[0]-popt_high_power3[0]
plt.annotate("", xy=(popt_low_power3[0]/1e3, -.1), xytext=(popt_high_power3[0]/
 ↪1e3, -.1), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\chi$={:d} kHz".format(int(chi/1e3)), xy=(popt_high_power3[0]/
 ↪1e3, -.05), color='black')

plt.grid()
plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.legend()
plt.show()

print(r'chi={:.1f} kHz'.format((popt_low_power3[0]-popt_high_power3[0])/1e3))
Delta=abs(backend_defaults.meas_freq_est[qubit] - backend_defaults.
 ↪qubit_freq_est[qubit])
print(r'g={:.1f} MHz'.format(np.sqrt(chi*Delta)/1e6))

chi = popt_low_power3[0]-popt_high_power3[0]
g = np.sqrt(chi*Delta)/1e6
```




```
chi=66.8 kHz
g=11.6 MHz
```

[489]:
```
chi_3 = chi
chi_3
```

[489]: 66849.01218509674

[490]:
```
g_3 = g
g_3
```

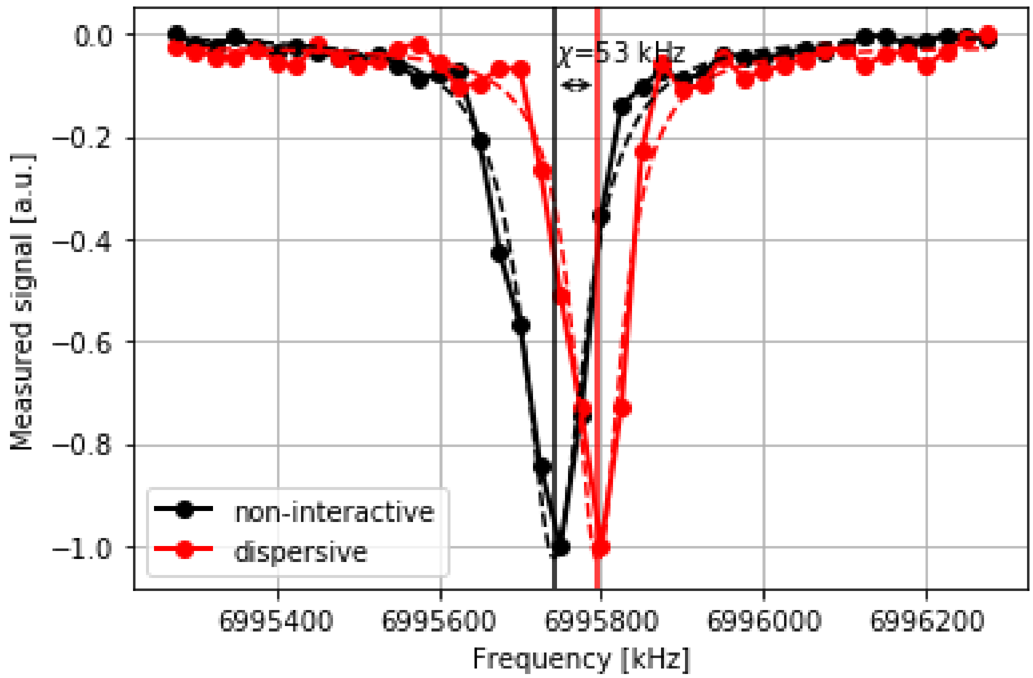
[490]: 11.62479471254869

[491]:
```
frequency_sweep_high_power = assemble(pulse_high_power,
                                     backend=backend,
                                     meas_level=1,
                                     meas_return='avg',
                                     shots=num_shots_per_frequency,
                                     schedule_los=schedule_frequencies)

job_high_power = backend.run(frequency_sweep_high_power)
job_monitor(job_high_power)

high_power_sweep_results = job_high_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[492]:
```python
high_power_sweep_values = []
for i in range(len(high_power_sweep_results.results)):
    res_high_power = high_power_sweep_results.get_memory(i)
    high_power_sweep_values.append(res_high_power[qubit])

high_power_sweep_values4 =␣
↪process_reflective_measurement(frequencies_range,high_power_sweep_values)

popt_high_power4,_=fit_lorentzian(frequencies_range,high_power_sweep_values4)
```

[493]:
```python
plt.plot(frequencies_range/1e3, high_power_sweep_values4, '-o', color='black',␣
↪lw=2, label='non-interactive')
plt.plot(frequencies_range/1e3, low_power_sweep_values4,'-o', color='red',␣
↪lw=2, label='dispersive')

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_high_power4), color='black', ls='--')
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power4), color='red', ls='--')

plt.axvline(x=popt_low_power4[0]/1e3, color='red')
plt.axvline(x=popt_high_power4[0]/1e3, color='black')

chi=popt_low_power4[0]-popt_high_power4[0]
plt.annotate("", xy=(popt_low_power4[0]/1e3, -.1), xytext=(popt_high_power4[0]/
↪1e3, -.1), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\chi$={:d} kHz".format(int(chi/1e3)), xy=(popt_high_power4[0]/
↪1e3, -.05), color='black')

plt.grid()
plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.legend()
plt.show()

print(r'chi={:.1f} kHz'.format((popt_low_power4[0]-popt_high_power4[0])/1e3))
Delta=abs(backend_defaults.meas_freq_est[qubit] - backend_defaults.
↪qubit_freq_est[qubit])
print(r'g={:.1f} MHz'.format(np.sqrt(chi*Delta)/1e6))

chi = popt_low_power4[0]-popt_high_power4[0]
g = np.sqrt(chi*Delta)/1e6
```




```
chi=53.5 kHz
g=10.4 MHz
```

[494]:
```
chi_4 = chi
chi_4
```

[494]:
```
53513.70391845703
```

[495]:
```
g_4 = g
g_4
```

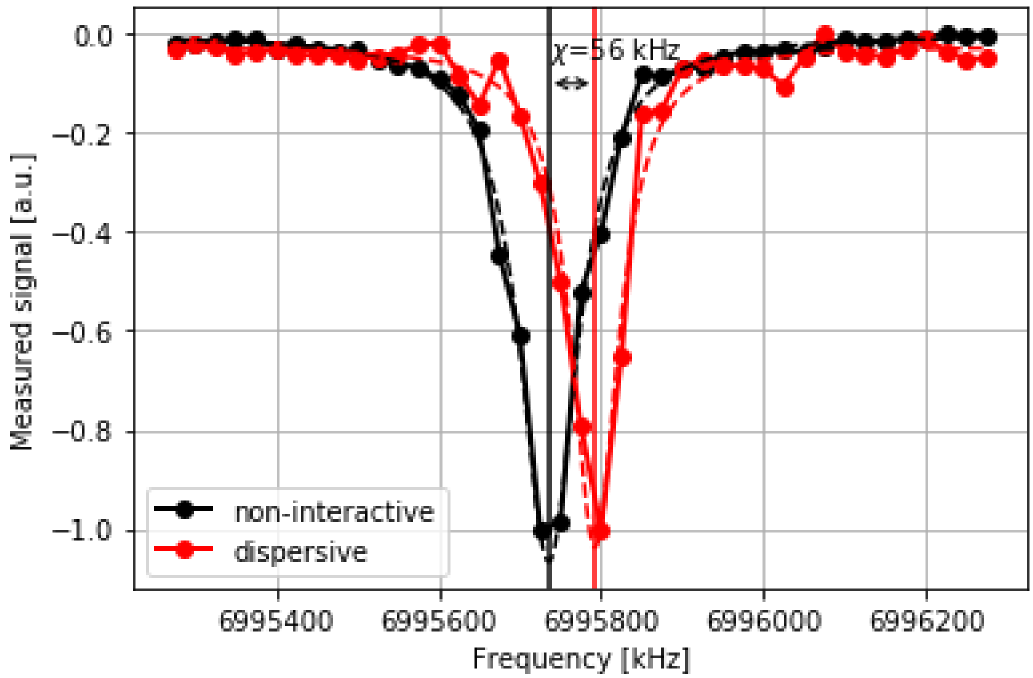
[495]:
```
10.400885152174075
```

[496]:
```
frequency_sweep_high_power = assemble(pulse_high_power,
                                      backend=backend,
                                      meas_level=1,
                                      meas_return='avg',
                                      shots=num_shots_per_frequency,
                                      schedule_los=schedule_frequencies)

job_high_power = backend.run(frequency_sweep_high_power)
job_monitor(job_high_power)

high_power_sweep_results = job_high_power.result(timeout=120)
```

```
Job Status: job has successfully run
```

[497]:
```
high_power_sweep_values = []
for i in range(len(high_power_sweep_results.results)):
    res_high_power = high_power_sweep_results.get_memory(i)
    high_power_sweep_values.append(res_high_power[qubit])

high_power_sweep_values5 =␣
↪process_reflective_measurement(frequencies_range,high_power_sweep_values)

popt_high_power5,_=fit_lorentzian(frequencies_range,high_power_sweep_values5)
```

[498]:
```
plt.plot(frequencies_range/1e3, high_power_sweep_values5, '-o', color='black',␣
↪lw=2, label='non-interactive')
plt.plot(frequencies_range/1e3, low_power_sweep_values5,'-o', color='red',␣
↪lw=2, label='dispersive')

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_high_power5), color='black', ls='--')
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power5), color='red', ls='--')

plt.axvline(x=popt_low_power5[0]/1e3, color='red')
plt.axvline(x=popt_high_power5[0]/1e3, color='black')

chi=popt_low_power5[0]-popt_high_power5[0]
plt.annotate("", xy=(popt_low_power5[0]/1e3, -.1), xytext=(popt_high_power5[0]/
↪1e3, -.1), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\chi$={:d} kHz".format(int(chi/1e3)), xy=(popt_high_power5[0]/
↪1e3, -.05), color='black')

plt.grid()
plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.legend()
plt.show()

print(r'chi={:.1f} kHz'.format((popt_low_power5[0]-popt_high_power5[0])/1e3))
Delta=abs(backend_defaults.meas_freq_est[qubit] - backend_defaults.
↪qubit_freq_est[qubit])
print(r'g={:.1f} MHz'.format(np.sqrt(chi*Delta)/1e6))

chi = popt_low_power5[0]-popt_high_power5[0]
g = np.sqrt(chi*Delta)/1e6
```




```
chi=57.0 kHz
g=10.7 MHz
```

[499]:
```
chi_5 = chi
chi_5
```

[499]: 56989.50929450989

[500]:
```
g_5 = g
g_5
```

[500]: 10.733349074741232

[534]:
```
chi_avgs = np.mean([high_power_sweep_values1, high_power_sweep_values2,␣
↪high_power_sweep_values3, high_power_sweep_values4,␣
↪high_power_sweep_values5], axis=0)
```

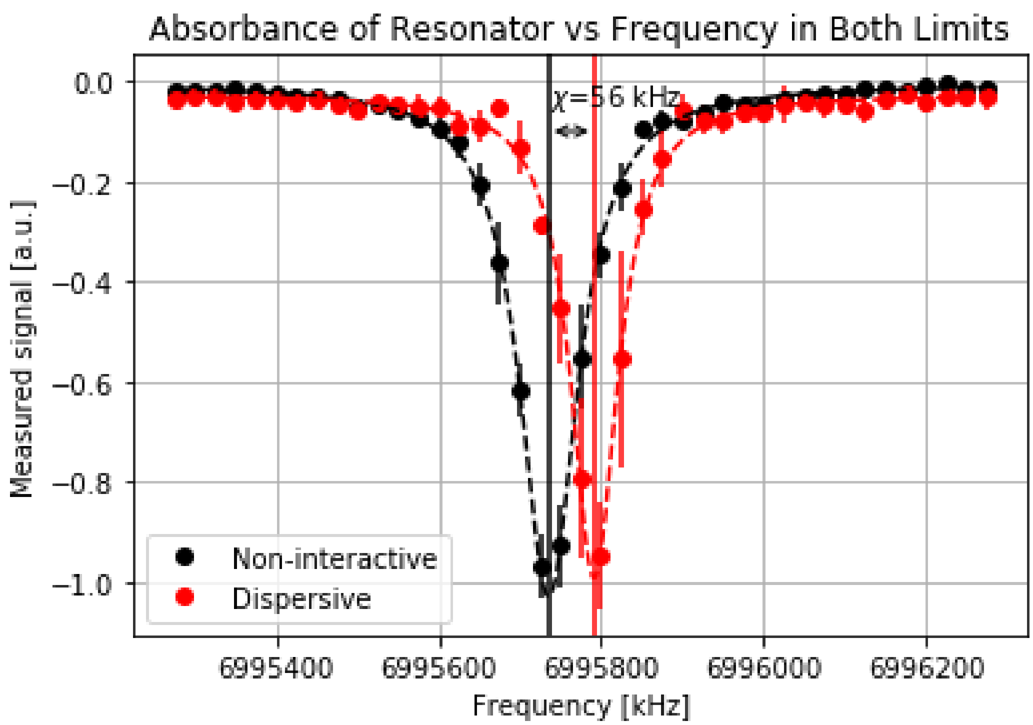
[533]:
```
chi_stds = np.std([high_power_sweep_values1, high_power_sweep_values2,␣
↪high_power_sweep_values3, high_power_sweep_values4,␣
↪high_power_sweep_values5], axis=0)
```

[543]:
```
len(frequencies_range)
```

[543]: 41

[557]:
```
plt.plot(frequencies_range/1e3, np.real(chi_avgs), '-o', color='black',␣
↪label='Non-interactive', ls = 'none')
```

```
plt.plot(frequencies_range/1e3, np.real(kappa_avgs),'-o', color='red',␣
 ↪label='Dispersive', ls = 'none')

plt.errorbar(frequencies_range/1e3, chi_avgs, yerr=chi_stds, color='black', ls␣
 ↪= 'none')
plt.errorbar(frequencies_range/1e3, kappa_avgs, yerr=kappa_stds, color='red',␣
 ↪ls = 'none')

popt_high_power,_ = fit_lorentzian(frequencies_range,chi_avgs)

fs=np.linspace(frequencies_range[0],frequencies_range[-1],1000)
plt.plot(fs/1e3, lorentzian(fs,*popt_high_power), color='black', ls='--')
plt.plot(fs/1e3, lorentzian(fs,*popt_low_power), color='red', ls='--')

plt.axvline(x=popt_low_power[0]/1e3, color='red')
plt.axvline(x=popt_high_power[0]/1e3, color='black')

chi=popt_low_power[0]-popt_high_power[0]
plt.annotate("", xy=(popt_low_power[0]/1e3, -.1), xytext=(popt_high_power[0]/
 ↪1e3, -.1), arrowprops=dict(arrowstyle="<->", color='black'))
plt.annotate("$\chi$={:d} kHz".format(int(chi/1e3)), xy=(popt_high_power[0]/
 ↪1e3, -.05), color='black')

plt.grid()
plt.xlabel("Frequency [kHz]")
plt.ylabel("Measured signal [a.u.]")
plt.legend()
plt.title('Absorbance of Resonator vs Frequency in Both Limits')
plt.show()

print(r'chi={:.1f} kHz'.format((popt_low_power[0]-popt_high_power[0])/1e3))
Delta=abs(backend_defaults.meas_freq_est[qubit] - backend_defaults.
 ↪qubit_freq_est[qubit])
print(r'g={:.1f} MHz'.format(np.sqrt(chi*Delta)/1e6))

chi = popt_low_power[0]-popt_high_power[0]
g = np.sqrt(chi*Delta)/1e6
```




```
chi=56.1 kHz
g=10.6 MHz
```

[623]:
```
avg_g = g
```

[624]:
```
avg_chi = chi
```

[627]:
```
Delta
```

[627]: 2021508586.1661701

[625]:
```
#avg_g = (g_1 + g_2 + g_3 + g_4 + g_5) / 5
avg_g
```

[625]: 10.64530030575244

[626]:
```
std_g = np.sqrt(((g_1 - avg_g)**2 + (g_2 - avg_g)**2 + (g_3 - avg_g)**2 + (g_4␣
↪- avg_g)**2 + (g_5 - avg_g)**2) / 5)
std_g
```

[626]: 0.6906482459416238

[572]:
```
#avg_chi = (chi_1 + chi_2 + chi_3 + chi_4 + chi_5) / 5
avg_chi
```

[572]: 56058.34146595001

[573]:
```
std_chi = np.sqrt(((chi_1 - avg_chi)**2 + (chi_2 - avg_chi)**2 + (chi_3 -
 ↪avg_chi)**2 + (chi_4 - avg_chi)**2 + (chi_5 - avg_chi)**2) / 5)
std_chi
```

[573]: 7220.963237855669

[ ]:

[574]:
```
detuning = - avg_g**2 / avg_chi
```

[575]:
```
detuning
```

[575]: -2021508586.16617

[579]:
```
T_phi
```

[579]: 3006.3202383729617

[585]:
```
1/(T_phi)
```

[585]: 0.00033262325609746771

[593]:
```
n = avg_amp / (detuning**2 + (avg_kappa / 2)**2)
```

[595]:
```
eta = avg_kappa**2 / (avg_kappa**2 + 4 * avg_chi**2)
```

[596]:
```
eta * n * (4 * avg_chi**2) / avg_kappa
```

[596]: 1.7390952285432757e-15

[586]:
```
avg_kappa
```

[586]: 70229.00276257984

[587]:
```
avg_chi
```

[587]: 56058.34146595001

[592]:
```
avg_amp
```

[592]: 0.14090025466819067

[598]:
```
qubit_freq + detuning
```

[598]: 2950312372.6819506

[606]:
```
detuning
```

[606]: -2021508586.16617

[ ]:

[601]:
```
last_freqs[33]
```

[601]: 6996620669.0

[604]:
```
(frequencies_range[18])
```

[604]: 6995775000.0

[ ]:

[ ]:

[557]:
```
frequencies_range[13]
```

[557]: 6991620669.0

[558]:
```
last_freqs = frequencies_range
```

[574]:
```
last_freqs[33]
```

[574]: 6996620669.0

[ ]:

[604]:
```
avg_chi
```

[604]: 56058.34146595001

[605]:
```
avg_kappa
```

[605]: 70229.00276257984

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[376]:
```python
#do rabi osc with avg freq. Get out avg params
```

[ ]:
```python
#with newfound pi pulse, do T1, T2 times
```

[ ]:
```python
#set up hamiltonian
```

[ ]:
```python
#with hamiltonian, implement gates in q-ctrl.
```

[ ]:
```python
#access higher level states?
#lind-bland dynamics?
```

[ ]:
```python
#build a multi-circuit system with copies of this qubit??? Add a twin qubit and␣
↪couple them. Make a third qubit that is tunable
#can do original system that way!!!
```

[41]:
```python
backend.configuration().hamiltonian
```

[41]:
```
{'description': 'Qubits are modeled as Duffing oscillators. In this case, the
system includes higher energy states, i.e. not just |0> and |1>. The Pauli
operators are generalized via the following set of
transformations:\n\n$(\\mathbb{I}-\\sigma_{i}^z)/2 \\rightarrow O_i \\equiv
b^\\dagger_{i} b_{i}$,\n\n$\\sigma_{+} \\rightarrow b^\\dagger$,\n\n$\\sigma_{-}
\\rightarrow b$,\n\n$\\sigma_{i}^X \\rightarrow b^\\dagger_{i} +
b_{i}$.\n\nQubits are coupled through resonator buses. The provided Hamiltonian
has been projected into the zero excitation subspace of the resonator buses
leading to an effective qubit-qubit flip-flop interaction. The qubit resonance
frequencies in the Hamiltonian are the cavity dressed frequencies and not
exactly what is returned by the backend defaults, which also includes the
dressing due to the qubit-qubit interactions.\n\nQuantities are returned in
angular frequencies, with units 2*pi*GHz.\n\nWARNING: Currently not all system
Hamiltonian information is available to the public, missing values have been
replaced with 0.\n',
 'h_latex': '\\begin{align} \\mathcal{H}/\\hbar = & \\sum_{i=0}^{0}\\left(\\frac
{\\omega_{q,i}}{2}(\\mathbb{I}-\\sigma_i^{z})+\\frac{\\Delta_{i}}{2}(O_i^2-O_i)+
\\Omega_{d,i}D_i(t)\\sigma_i^{X}\\right) \\\\ \\end{align}',
 'h_str': ['_SUM[i,0,0,wq{i}/2*(I{i}-Z{i})]',
  '_SUM[i,0,0,delta{i}/2*O{i}*O{i}]',
  '_SUM[i,0,0,-delta{i}/2*O{i}]',
  '_SUM[i,0,0,omegad{i}*X{i}||D{i}]'],
```

```
 'osc': {},
 'qub': {'0': 3},
 'vars': {'delta0': -2181477525.8495026,
  'omegad0': 118839821.64990656,
  'wq0': 31239117029.21657}}
```

[241]:
```
backend_defaults.qubit_freq_est[qubit]
```

[241]:
```
4971859893.026022
```

[242]:
```
backend_defaults.meas_freq_est[qubit]
```

[242]:
```
6993370669.0
```

[243]:
```
backend_defaults.qubit_freq_est[qubit] - backend_defaults.meas_freq_est[qubit]
```

[243]:
```
-2021510775.973978
```

[ ]:
```

```