qtt Documentation

Release 1.2.4

Pieter Eendebak

Mar 24, 2021

CONTENTS:

Hello, and welcome to the documentation of Quantum Technology Toolbox!

Quantum Technology Toolbox (QTT) is a Python package for performing measurements and calibration of spin-qubits. It was developed initially at QuTech, an advanced research center for quantum computing and quantum internet. QuTech is a collaboration founded by the Delft University of Technology (TU Delft) and Netherlands Organisation for Applied Scientific Research (TNO).

QuTech addresses scientific challenges as well as engineering issues in a joint center of know-how with industrial partners. One of these challenges is to automate the calibration and analysis of measurements pertaining spin qubits. For this purpose, QTT was developed. By sharing this framework with you we hope to work with you on improving it and together forward the development of quantum computers all over the world.

A more elaborate starting guide can be found in the introduction. We do include an example in here to show what QuTech Tuning is capable of:

```
import qtt
# load data
dataset = qtt.data.load_dataset('example')
# analyse
results = qtt.algorithms.gatesweep.analyseGateSweep(dataset, fig=100)
```

More examples can be found in the example notebooks.

CHAPTER

ONE

DOCUMENTATION

1.1 Introduction

Welcome to the QTT framework. This introduction will shortly introduce the framework, and it will guide you through the structure, installation process and how to contribute. We look forward to working with you!

1.1.1 Quantum Technology Toolbox

Quantum Technology Toolbox (QTT) is a Python-based framework developed initialy by QuTech for the tuning and calibration of quantum dots and spin qubits. QuTech is an advanced research center based in Delft, the Netherlands, for quantum computing and quantum internet. It is a collaboration founded by the Delft University of Technology (TU Delft) and the Netherlands Organisation for Applied Scientific Research (TNO).

The experiments done on spin-qubits at QuTech make use of the QTT framework to add automated funcionalities and algorithms to their measurement code. This paves the way to a more time-efficient, user-friendly and robust code, making more complex research on larger systems possible. We invite you to use and contribute to QTT. Below we will guide you through the installation.

QTT is the framework on which you can base your measurement and analysis scripts, and QTT itself is based on Qcodes.

1.1.2 Installation

QTT is compatible with Python 3.5+.

QTT can be installed as a pip package:

```
$ pip install --upgrade qtt
```

For development we advice to install from source. First retrieve the source code using git, and then install from the qtt source directory using the command:

```
$ python setup.py develop
```

For for Vandersypen research group there are more detailed instructions, read the file INSTALL-spinqubits.md in the spin-projects repository.

1.1.3 Updating QTT

If you registered qtt with Python via `python setup.py develop` or `pip install -e .`, all you need to do to get the latest code is open a terminal window pointing to anywhere inside the repository and run *git pull*.

If you installed qtt via the pip package you can run the pip install comment again:

```
$ pip install --upgrade qtt
```

1.1.4 Usage

In QTT, we use GitHub for combined developing and python for scientific use. If you have some experience with scientific python you will be able to understand the code fairly easily. If not, we urge you to read through some lectures before using the QTT framework. For a general introduction see:

- Introduction to Github
- Scientific python lectures

We advise to use the following settings when using QTT:

- If you use Spyder then use the following settings:
  - Use a `IPython` console and set the IPython backend graphics option to `QT`. This ensures correctly displaying the `ParameterViewer` and `DataBrowser`
  - In `Tools->Preferences->Console->Advanced settings` uncheck the box `Enable UMR`

For the usage of algorithms or calibrations we point you to the documentation of those subjects.

1.1.5 Testing

Tests for the qtt packages are contained in the subdirectory `tests` and as test functions (`test_*`) in the code. To run the tests you can run one of the commands below.

```
$ pytest
```

1.2 Example notebooks

1.2.1 Measurements

Perform simple measurements with the qtt measurement functions

This example shows how to set values on instruments (such as a voltage on a gate), how to readout values from measurement instruments (such as the voltage on a multimeter), and how to perform simple measurements loops (in this case a 2D gate scan).

[1]:
```
import qtt
import numpy as np
```

Measurements with Parameters

For the purpose of this example will use a virtual system that simulates a quantum dot measurement setup. See the `qtt.simulation.virtual_dot_array` documentation for more info.

[2]:
```
import qtt.simulation.virtual_dot_array
station=qtt.simulation.virtual_dot_array.initialize()
gates=station.gates
```

```
initialize: create virtualdot
initialized virtual dot system (2 dots)
```

We can read out instruments using a qcodes `Parameter`.

[3]:
```
value=gates.P1.get(); print(value)
```

```
-0.01483164894377098
```

Custom measurement loops

The qcodes loop is not suitable for all measurements. You can also write your own loop constructions. There are already several constructions available. For example make a 2D scan one can use the `qtt.scans.scan2D`

[4]:
```
import qtt.measurements.scans
scanjob=qtt.measurements.scans.scanjob_t({'sweepdata': {'param':'P1', 'start':20,'end
↪':28,'step': 1.75} ,
        'stepdata': {'param': 'P2', 'start': 0, 'end': 7, 'step': 1}, 'minstrument':␣
↪['keithley1.amplitude']})
dataset=qtt.measurements.scans.scan2D(station, scanjob)
```

```
scan2D: 0/7: time 00:00:00 (~00:00:00 remaining): setting P2 to 0.000
```

[5]:
```
print(dataset)
```

```
DataSet:
   location = '2018-09-05/11-50-28_qtt_scan2D'
   <Type>    | <array_id>          | <array.name>        | <array.shape>
   Measured  | keithley1_amplitude | keithley1_amplitude | (7, 5)
   Setpoint  | P2                  | P2                  | (7,)
   Setpoint  | P1                  | P1                  | (7, 5)
```

The raw data is available as a DataArray or numpy array.

[6]:
```
print(dataset.default_parameter_array())
```

```
DataArray[7,5]: keithley1_amplitude
array([[2.98960663, 2.98920055, 2.98878521, 2.99806085, 2.99676836],
       [2.99280892, 2.99730119, 2.99056696, 2.99518558, 2.99344639],
       [2.99558079, 2.98947501, 2.98971753, 2.99565561, 2.99637049],
       [2.99046836, 2.99784205, 2.98961711, 2.99544447, 2.99375562],
       [2.99459975, 2.99424155, 2.98910142, 2.99222029, 2.98887384],
       [2.99335894, 2.99296707, 2.99501929, 2.99703682, 2.99673491],
       [2.99093673, 2.99259619, 2.99469442, 2.9918319 , 2.99783992]])
```

[7]:
```
print(dataset.default_parameter_name())
print(np.array(dataset.default_parameter_array()))
```

```
keithley1_amplitude
[[2.98960663 2.98920055 2.98878521 2.99806085 2.99676836]
 [2.99280892 2.99730119 2.99056696 2.99518558 2.99344639]
 [2.99558079 2.98947501 2.98971753 2.99565561 2.99637049]
 [2.99046836 2.99784205 2.98961711 2.99544447 2.99375562]
 [2.99459975 2.99424155 2.98910142 2.99222029 2.98887384]
 [2.99335894 2.99296707 2.99501929 2.99703682 2.99673491]
 [2.99093673 2.99259619 2.99469442 2.9918319  2.99783992]]
```

Defining a station

In this example we will shortly discuss the use of a station with instruments. A station bounds instruments and has functionality to control and store the settings of the devices. We will start from a simulated station (the virtual dot array) that has a special function `initialize` to directly create the station object with simulated instruments:

[10]:
```
from qtt.simulation import virtual_dot_array
station = virtual_dot_array.initialize()
```

```
initialize: create virtualdot
initialized virtual dot system (2 dots)
```

The simulated station contain the instuments; `gates`, `keithley1`, `keithley3`, `keithley4`, `ivvi1`, `ivvi2`, `vawg` and a `digitizer`. In general, every Instrument of a station have Parameter items, which are gettable or settable. For example, using the simulated station we can get the keithley and P1 gate amplitude:

[3]:
```
gates = station.gates
print(gates)
print(gates.P1.get())

gates.P1.set(1.5)
print(gates.P1.get())

keithley1 = station.keithley1
print(keithley1)
print(keithley1.amplitude.get())
```

```
gates: gates (16 gates)
0.3166101322411219
1.5
<VirtualMeter: keithley1>
2.9964151248162425
```

Default workflow after connecting the instruments (the `initialize` function) is to use the station with the measurement functions. The measurements make use of the instruments in the station. To create a station, you need to define you own `initialize` function. An example `initialize` function is shown below. For your own setup to have to modify the code with the correct instruments and settings.

A view of all the settings of a instrument or station can be shown using the `snapshot` function. The measurements use `snapshot` to collect the settings, which are stored together with the measurement results.

[6]:
```
print(keithley1.snapshot())
```

```
{'functions': {}, 'submodules': {}, '__class__': 'qtt.instrument_drivers.virtual_
↪instruments.VirtualMeter', 'parameters': {'IDN': {'value': None, 'ts': None, 'raw_
↪value': None, '__class__': 'qcodes.instrument.parameter.Parameter', 'full_name':
↪'keithley1_IDN', 'unit': '', 'instrument': 'qtt.instrument_drivers.virtual_
↪instruments.VirtualMeter', 'instrument_name': 'keithley1', 'label': 'IDN', 'inter_
↪delay': 0, 'name': 'IDN', 'vals': '<Anything>', 'post_delay': 0}, 'amplitude': {
↪'value': 2.9964151248162425, 'ts': '2018-09-16 21:34:35', 'raw_value': 2.
↪9964151248162425, '__class__': 'qcodes.instrument.parameter.Parameter', 'full_name':
↪ 'keithley1_amplitude', 'unit': 'a.u.', 'instrument': 'qtt.instrument_drivers.
↪virtual_instruments.VirtualMeter', 'instrument_name': 'keithley1', 'label':
↪'keithley1 amplitude', 'inter_delay': 0, 'name': 'amplitude', 'post_delay': 0},
↪'readnext': {'value': None, 'ts': None, 'raw_value': None, '__class__': 'qcodes.
↪instrument.parameter.Parameter', 'full_name': 'keithley1_readnext', 'unit': '',
```

(continues on next page)

(continued from previous page)

[7]:
```
snapshot = station.snapshot()
print(snapshot.keys())
```

```
dict_keys(['instruments', 'parameters', 'components', 'default_measurement', 'metadata
↪'])
```

After all experiments are over, all devices need to be disconnected. The `virtual_dot_array` has a `close` function to stop and clean up all the instrument resources:

[8]:
```
virtual_dot_array.close()
```

```
close gates: gates (16 gates)
close <VirtualMeter: keithley1>
close <VirtualMeter: keithley3>
close <VirtualMeter: keithley4>
close VirtualIVVI: ivvi1
close VirtualIVVI: ivvi2
close <SimulationAWG: vawg>
close <SimulationDigitizer: sdigitizer>
```

For your own setup to have to write your own `close` function.

Example of videomode tuning

In this example we show how to initialize and use the videomode for 1D and 2D scans of gates. Note that the code shown below is not suitable for running via this Jupyter notebook, because videomode requires actual hardware to work with, which is probably not available from the PC this example is being run on.

Author: Sjaak van Diepen, Pieter Eendebak

[1]:
```
%gui qt
import matplotlib.pyplot as plt
import tempfile
import imageio
import qcodes
import qtt
from qtt.instrument_drivers.gates import VirtualDAC
from qtt.instrument_drivers.virtual_instruments import VirtualIVVI
from qcodes.station import Station
from qcodes_contrib_drivers.drivers.Spectrum.M4i import M4i
from qtt.measurements.videomode import VideoMode
from qtt.instrument_drivers.virtualAwg.virtual_awg import VirtualAwg
from qcodes_contrib_drivers.drivers.ZurichInstruments.ZIHDAWG8 import ZIHDAWG8
```

```
Windows found
```

Import modules and create a station with DAC modules and an AWG

Import the station and the VideoMode class. Here we use the stationV2 as an example, because there is not a virtual station available which can be used for a simulation of the videomode.

[3]:
```
ivvi = VirtualIVVI(name='ivvi0', model=None)
gates = VirtualDAC('gates', [ivvi], {'P1': (0, 1), 'P2': (0, 2), 'P3': (0,3), 'P4':␣
↪(0,4)})

station = Station(ivvi, gates)

m4i = qcodes.find_or_create_instrument(M4i, name='m4i')
m4i.timeout(15*1e3)  # set timeout of 10 seconds
m4i.sample_rate(1e6)

station.add_component(m4i)
```

```
WARNING:qcodes.instrument.base:[m4i(M4i)] Snapshot: Could not update parameter:␣
↪channel_0
WARNING:qcodes.instrument.base:[m4i(M4i)] Snapshot: Could not update parameter:␣
↪channel_1
WARNING:qcodes.instrument.base:[m4i(M4i)] Snapshot: Could not update parameter:␣
↪channel_2
WARNING:qcodes.instrument.base:[m4i(M4i)] Snapshot: Could not update parameter:␣
↪channel_3
```

[3]:
```
'm4i'
```

[4]:
```
address = 'DEV8049'
awg = qcodes.find_or_create_instrument(ZIHDAWG8, 'awg8', device_id=address)
station.add_component(awg)
```

```
Connected to: None awg8 (serial:None, firmware:None) in 12.44s
```

[4]:
```
'awg8'
```

Initialize the virtual AWG

[5]:
```
from qcodes.utils.validators import Numbers

class HardwareType(qcodes.Instrument):

    def __init__(self, name, awg_map, awg_scalings={}, **kwargs):
        super().__init__(name, **kwargs)

        self.awg_map = awg_map
        for gate in self.awg_map.keys():
            p = 'awg_to_%s' % gate
            self.add_parameter(p, parameter_class=qcodes.ManualParameter,
                               initial_value=awg_scalings.get(gate, 1),
                               label='{} (factor)'.format(p), unit='mV/V',
                               vals=Numbers(0, 400))

    def get_idn(self):
        ''' Overrule because the default VISA command does not work '''
        IDN = {'vendor': 'QuTech', 'model': 'hardwareV2',
               'serial': None, 'firmware': None}
```

(continues on next page)

(continued from previous page)

```
        return IDN

awg_map = {'P1': (0, 5 - 1), 'P2': (0, 6 - 1), 'P3': (0, 7-1), 'm4i_mk': (0, 5 - 1,␣
↪0)}
awg_scalings = {f'P{ii}': 300. for ii in range(1,4)}

hardware = HardwareType(qtt.measurements.scans.instrumentName('hardware'), awg_map,␣
↪awg_scalings)
station.add_component(hardware)

virtual_awg = VirtualAwg([awg], hardware, qtt.measurements.scans.instrumentName(
↪'virtual_awg'))
virtual_awg.digitizer_marker_delay(17e-6)
virtual_awg.digitizer_marker_uptime(50e-6)

station.add_component(virtual_awg)
```

[5]:
```
'virtual_awg'
```

1D - videomode

First set which parameter to sweep, this must be a parameter of the gates instrument in the station, over which range (in milliVolt) to sweep and which FPGA channel to acquire the data from. Then we would run the last line in the cell below, which starts a GUI that looks like the image below. This image is a print screen taken of a 1D videomode.

[6]:
```
sweepparams = 'P1'
sweepranges = .8

vm = VideoMode(station, sweepparams=sweepparams, sweepranges=sweepranges,␣
↪minstrument=(m4i, [0,1]), resolution=[32,64])
```

```
live_plotting: is1dscan True
live_plotting: is1dscan True
enable_averaging called, undefined: value True
enable_averaging called, undefined: value True
VideoMode: run
```

```
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'm4i_mk_
↪marker' cut down to playable length from 9375 to 9368 samples (should be a multiple␣
↪of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'P1_
↪sawtooth' cut down to playable length from 9375 to 9368 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'm4i_mk_
↪marker' cut down to playable length from 9375 to 9368 samples (should be a multiple␣
↪of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'P1_
↪sawtooth' cut down to playable length from 9375 to 9368 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'm4i_mk_
↪marker' cut down to playable length from 9375 to 9368 samples (should be a multiple␣
↪of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'P1_
↪sawtooth' cut down to playable length from 9375 to 9368 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'm4i_mk_
↪marker' cut down to playable length from 9375 to 9368 samples (should be a multiple␣
↪of 8 samples for single channel or 4 samples for dual channel waveforms)
```

(continues on next page)

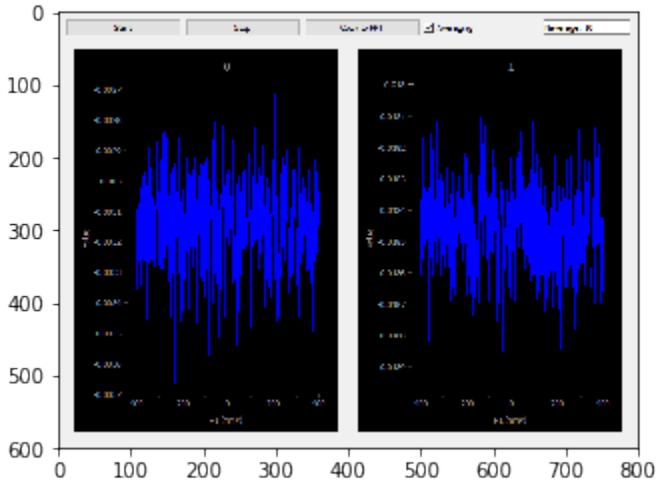
(continued from previous page)

```
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 6): waveform 'P1_
↪sawtooth' cut down to playable length from 9375 to 9368 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
```

```
VideoMode: run: startreadout
VideoMode: start readout
```

We show the VideoMode window in the notebook by taking a screenshot.

[7]:
```
def make_screenshot(vm):
    fname=tempfile.mktemp(suffix='.png')
    vm.mainwin.grab().save(fname)
    im=imageio.read(fname)
    im=im.get_data(0)
    plt.figure()
    _=plt.imshow(im)
make_screenshot(vm)
```



We can also get the data acquired in DataSet format.

[ ]:
```
dataset=vm.get_dataset()[0]
qtt.data.plot_dataset(dataset)
vm.stop()
```

About the GUI

The framerate of the videomode is shown in the title of the window. When this print screen was taken the framerate was 16.02 frames per second. The "Start" and "Stop" buttons can be used for starting and stopping the videomode. Note that these buttons control both the plotting and the AWG sweep. The recorded data is available via the method `get_dataset` the videomode object. In the Naverage field the user can adjust the averaging.

2D - videomode

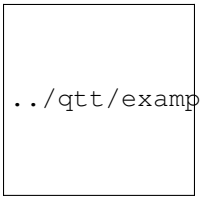
[12]:
```
sweepparams = ['P2', 'P3']
sweepranges = [100, 100]
vm = VideoMode(station, sweepparams=sweepparams, sweepranges=sweepranges,␣
↪minstrument=(m4i, [0,1]))
```

```
live_plotting: is1dscan False
live_plotting: is1dscan False
enable_averaging called, undefined: value True
enable_averaging called, undefined: value True
VideoMode: run
```

```
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'm4i_mk_
↪marker' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P2_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P3_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'm4i_mk_
↪marker' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P2_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P3_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'm4i_mk_
↪marker' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P2_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P3_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'm4i_mk_
↪marker' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P2_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
WARNING:qcodes.instrument.base:[awg8(ZIHDAWG8)] Warning (line: 9): waveform 'P3_
↪sawtooth' cut down to playable length from 88473 to 88472 samples (should be a␣
↪multiple of 8 samples for single channel or 4 samples for dual channel waveforms)
```

```
VideoMode: run: startreadout
VideoMode: start readout
```

[ ]:
```
make_screenshot(vm)
```



Using the Virtual AWG

The virtual AWG is an abstraction layer, which lets you define waveform sequences. The sequences can be upload to gates and played. The virtual AWG takes care of gates to AWG channel relation and creates the markers for the digitizer and interconnecting AWG's.

In this example we will discuss the use of the virtual AWG in combination with a digitizer. We used the V1 setup with one Tektronix 5014 AWG's and one Spectrum M4i digitizer card as hardware.

[5]:
```
import numpy as np
import matplotlib.pyplot as plt

from qupulse.pulses import FunctionPT
from qupulse.pulses import SequencePT

from qcodes.utils.validators import Numbers
from qcodes.instrument.base import Instrument
from qcodes.instrument.parameter import ManualParameter
from qcodes_contrib_drivers.drivers.Spectrum.M4i import M4i
from qcodes.instrument_drivers.tektronix.AWG5014 import Tektronix_AWG5014

from qtt.instrument_drivers.virtualAwg.sequencer import DataTypes
from qtt.instrument_drivers.virtualAwg.virtual_awg import VirtualAwg
from qtt.measurements.scans import measuresegment as measure_segment
```

First we need to create a HardwareSettings class that contains the awg map and the awg to gates. The awg map contains the relation between the AWG channels, markers and gates. The awg to gate defines the voltage loss caused by attenuation, filtering, etc. of the AWG channels. This is, the ratio between AWG channel output value and the voltage on the gate. In the past the class HardwareSettings was called hardware.

[4]:
```
class HardwareSettings(Instrument):

    def __init__(self, name='settings'):
        """ Contains the quantum chip settings:
                awg_map: Relation between gate name and AWG number and AWG channel.
                awg_to_gate: Scaling ratio between AWG output value and the voltoge␣
↪on the gate.
        """
        super().__init__(name)
        awg_gates = {'X2': (0, 1), 'P7': (0, 2), 'P6': (0, 3), 'P5': (0, 4),
                     'P2': (1, 1), 'X1': (1, 2), 'P3': (1, 3), 'P4': (1, 4)}
        awg_markers = {'m4i_mk': (0, 4, 1)}
        self.awg_map = {**awg_gates, **awg_markers}
```

(continues on next page)

(continued from previous page)

```
        for awg_gate in awg_gates:
            parameter_name = 'awg_to_{}'.format(awg_gate)
            parameter_label = '{} (factor)'.format(parameter_name)
            self.add_parameter(parameter_name, parameter_class=ManualParameter,
                               initial_value=1000.0, label=parameter_label,␣
↪vals=Numbers(1, 1000))
```

Last we define some functions to quickly plot the digitizer output data.

[6]:
```
def update_awg_settings(virtual_awg, sampling_rate, amplitude, marker_low, marker_
↪high):
    for awg_number in range(len(virtual_awg.awgs)):
        virtual_awg.update_setting(awg_number, 'sampling_rate', sampling_rate)
        virtual_awg.update_setting(awg_number, 'amplitudes', amplitude)
        virtual_awg.update_setting(awg_number, 'marker_low', marker_low)
        virtual_awg.update_setting(awg_number, 'marker_high', marker_high)

def plot_data_1d(digitizer_data, label_x_axis='', label_y_axis=''):
    plt.figure();
    plt.clf();
    plt.xlabel(label_x_axis)
    plt.ylabel(label_y_axis)
    plt.plot(digitizer_data.flatten(),'.b')
    plt.show()

def plot_data_2d(digitizer_data, label_x_axis='', label_y_axis='', label_colorbar=''):
    plt.figure();
    plt.clf();
    im = plt.imshow(digitizer_data[0])
    cbar = plt.colorbar(im)
    plt.xlabel(label_x_axis)
    plt.ylabel(label_y_axis)
    cbar.ax.set_ylabel(label_colorbar)
    plt.show()
```

Initializing the virtual AWG and digitizer

Before we can start using the virtual AWG, an AWG output channel should be directly connected to the digitizer channel. The virtual AWG can be tested using this connection. We connected gate 'X2' to digitizer channel 0 for the XLD setup.

After the hardware change the connection to the AWG's and digitizer. The hardware needs specific settings in order to work properly.

[7]:
```
digitizer = M4i(name='digitizer')

sample_rate_in_Hz = 2e6
digitizer.sample_rate(sample_rate_in_Hz)

timeout_in_ms = 10 * 1000
digitizer.timeout(timeout_in_ms)

millivolt_range = 2000
digitizer.initialize_channels(mV_range=millivolt_range)
```

(continues on next page)

(continued from previous page)

```
import pyspcm
external_clock_mode = pyspcm.SPC_CM_EXTREFCLOCK
digitizer.clock_mode(external_clock_mode)

reference_clock_10mHz = int(1e7)
digitizer.reference_clock(reference_clock_10mHz)

# Initialize Tektronix AWG's
trigger_level_in_Volt = 0.5
clock_frequency_in_Hz = 1.0e7

address_awg1 = 'GPIB1::5::INSTR'
awg1 = Tektronix_AWG5014(name='awg1', address=address_awg1)
awg1.clock_freq(clock_frequency_in_Hz)
awg1.trigger_level(trigger_level_in_Volt)
```

```
Connected to: TEKTRONIX AWG5014 (serial:B010106, firmware:SCPI:99.0 FW:3.1.141.647)␣
↪in 0.48s
```

The virtual AWG and the settings object can be created once the hardware is connected.

[8]:
```
settings = HardwareSettings()
virtual_awg = VirtualAwg([awg1], settings)

uptime_in_seconds = 1.0e-5
marker_delay_in_sec = 3.5e-5

virtual_awg.update_digitizer_marker_settings(uptime_in_seconds, marker_delay_in_sec)

output_amplitude = 0.5
marker_low_level = 0.0
marker_high_level = 2.6

update_awg_settings(virtual_awg, clock_frequency_in_Hz, output_amplitude, marker_low_
↪level, marker_high_level)
```

Sawtooth Sequence

The most simple sequence to create is a sawtooth with a certain width. First upload the sawtooth to the AWG using the sweep_gates function on gate X2.

[9]:
```
output_gate = 'X2'
mV_sweep_range = 50
sec_period = 1.0e-3
sweep_data = virtual_awg.sweep_gates({output_gate: 1}, mV_sweep_range, sec_period)
```

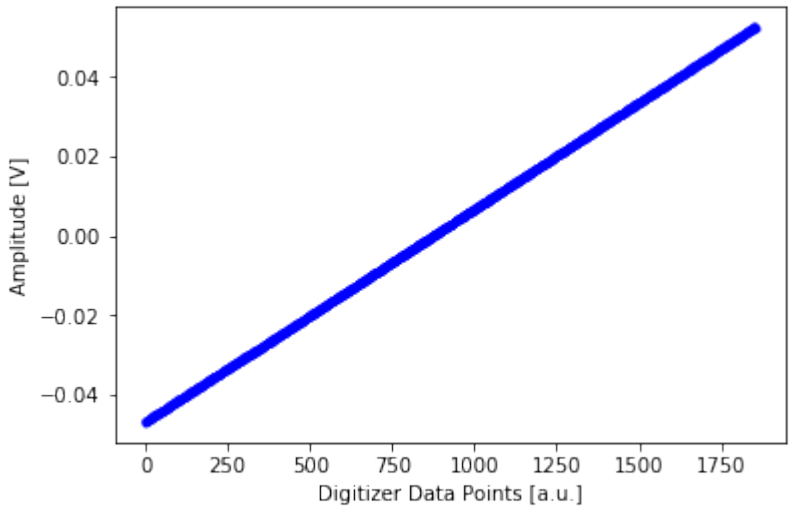
Then enable the AWG channel and run the sequence. Start the recording of the digitizer with averaging a 100 times. Afterwards the AWG is stopped and the channel disabled.

[10]:
```
virtual_awg.enable_outputs([output_gate])
virtual_awg.run()

readout_channels = [2]
number_of_averages = 100
```

(continues on next page)

(continued from previous page)

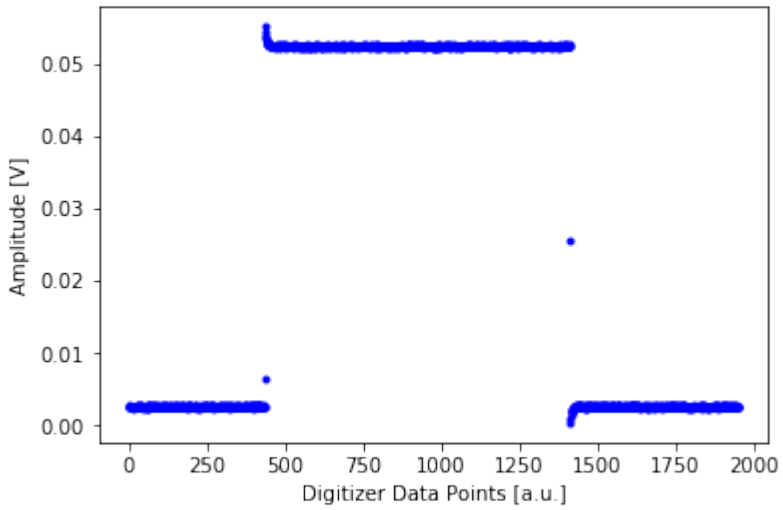
```
data = measure_segment(sweep_data, number_of_averages, digitizer, readout_channels)

virtual_awg.stop()
virtual_awg.disable_outputs([output_gate])

plot_data_1d(data, 'Digitizer Data Points [a.u.]', 'Amplitude [V]')
```



Pulse Sequence

Run a square wave on gate X2. Collect the data using a digitizer with 100 times averaging.

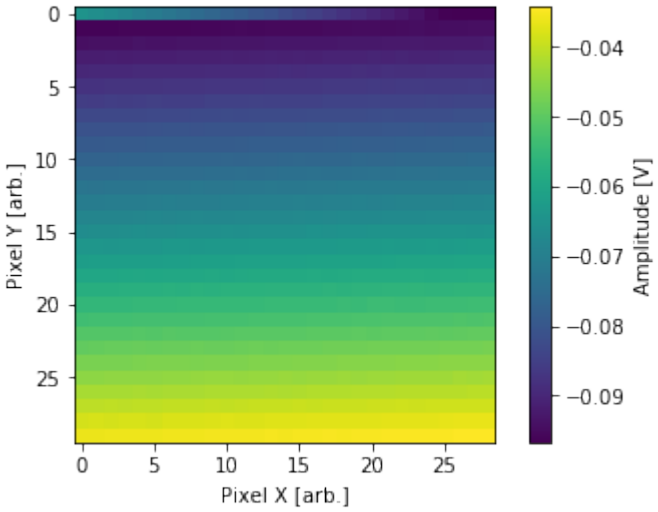
[11]:
```
output_gate = 'X2'
mV_sweep_range = 50
sec_period = 1.0e-3
sweep_data = virtual_awg.pulse_gates({output_gate: 1}, mV_sweep_range, sec_period)

virtual_awg.enable_outputs([output_gate])
virtual_awg.run()

readout_channels = [2]
number_of_averages = 100
data = measure_segment(sweep_data, number_of_averages, digitizer, readout_channels,␣
↪process=False)

virtual_awg.stop()
virtual_awg.disable_outputs([output_gate])

plot_data_1d(data, 'Digitizer Data Points [a.u.]', 'Amplitude [V]')
```



Pulse combination on Two Gates

Supplies sawtooth signals to a linear combination of gates, which effectively does a 2D scan. Collect the data of one axes using the digitizer with 100 times averaging.

[12]:
```
sec_period = 5.0e-5
resolution = [32, 32]
mV_sweep_ranges = [100, 100]
output_gates = [{'X2': 1}, {'P7': 1}]
sweep_data = virtual_awg.sweep_gates_2d(output_gates, mV_sweep_ranges, sec_period,␣
↪resolution)

virtual_awg.enable_outputs(['X2', 'P7'])
virtual_awg.run()

number_of_averages = 100
readout_channels = [2]
data = measure_segment(sweep_data, number_of_averages, digitizer, readout_channels)

virtual_awg.disable_outputs(['X2', 'P7'])
virtual_awg.stop()

plot_data_2d(data, 'Pixel X [a.u.]', 'Pixel Y [a.u.]', 'Amplitude [V]')
```



Apply custom Sequence on Gate

Create a decaying sine-wave sequence using QuPulse

[13]:
```
sec_to_ns = 1.0e9

mV_amplitude = 25
sec_period = 1.0e-3
sine_decay = 5e5
sine_period = 2 * np.pi * 1e-2 * sec_period

pulse_function = FunctionPT('alpha*exp(-t/tau)*sin(phi*t)', 'duration')
input_variables = {'alpha': mV_amplitude, 'tau': sine_decay, 'phi': sine_period,
↪'duration': sec_period * sec_to_ns}

other = (pulse_function, input_variables)
sequence_data = {'name': 'test', 'wave': SequencePT(*(other,)), 'type': DataTypes.QU_
↪PULSE}
sequence = {'X2': sequence_data}

sequence.update(virtual_awg._VirtualAwg__make_markers(sec_period))
```

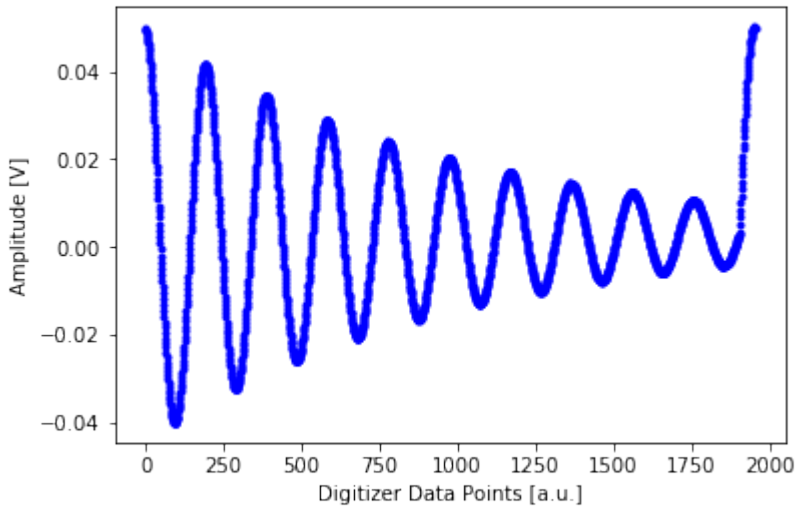
Upload the QuPulse sequence onto the AWG. Play the sequence and collect the data using the digitizer with 100 averaging.

[15]:
```
sweep_data = virtual_awg.sequence_gates(sequence)
sweep_data.update({
    'period': sec_period,
    'samplerate': virtual_awg.awgs[0].retrieve_setting('sampling_rate'),
    'markerdelay': virtual_awg.digitizer_marker_delay()
})

virtual_awg.enable_outputs(['X2'])
virtual_awg.run()

number_of_averages = 100
```

(continues on next page)

(continued from previous page)

```
readout_channels = [2]
data = measure_segment(sweep_data, number_of_averages, digitizer, readout_channels,␣
↪process=False)

virtual_awg.disable_outputs(['X2'])
virtual_awg.stop()

plot_data_1d(data, 'Digitizer Data Points [a.u.]', 'Amplitude [V]')
```



[ ]:

Spin qubit measurement software

This example shows a typical workflow of tasks undertaken when performing spin qubit measurements.

[2]:
```
import sys, os, tempfile
import numpy as np
%matplotlib inline
%gui qt
import matplotlib.pyplot as plt
import qcodes
from qcodes.data.data_set import DataSet
import qtt
from qtt.measurements.scans import scanjob_t

# set data directory
datadir = os.path.join(tempfile.mkdtemp(), 'qdata')
DataSet.default_io = qcodes.data.io.DiskIO(datadir)
```

Load your station

For the purpose of this example we will use a virtual system that simulates the device and instruments in a 2-dot spin-qubit dot setup. The hardware consists of a virtual multimeter (`keithley`), voltage sources (`ivvi`) and a virtual gates object (`gates`) that applies voltages to the virtual device gates.

[3]:
```
import qtt.simulation.virtual_dot_array as virtual_dot

nr_dots = 2
station = virtual_dot.initialize(nr_dots=nr_dots)

keithley1 = station.keithley1
keithley3 = station.keithley3

# virtual gates for the model
gates = station.gates
```

```
initialize: create virtualdot
initialized virtual dot system (2 dots)
```

Setup measurement windows

- Parameter viewer (`pv`): gui for reading and changing the values of the instruments
- Live plotting window (`plotQ`): for on-going measurements

[4]:
```
pv = qtt.createParameterWidget([gates, ])
mwindows = qtt.gui.live_plotting.setupMeasurementWindows(station, create_parameter_
↪widget=False)
plotQ = mwindows['plotwindow']
```

Read out instruments

We can, for example, readout a gate voltage or readout the voltage measured by the multimeter. We can also retrieve the full state of our measurement station using `station.snapshot()`, which returns a dictionary with every parameter of every instrument in the station.

[5]:
```
print('gate P1: %.1f, amplitude: %f' % (gates.P1.get(),  keithley3.readnext()) )
snapshotdata = station.snapshot()
```

```
gate P1: -0.1, amplitude: 1.001941
```

Simple 1D scan loop

We use the `scan1D` function to measurements as we sweep one parameter. This function has `scanjob` as argument, where the parameters of the scan are set.

[6]:
```
scanjob = scanjob_t({'sweepdata': dict({'param': 'P1', 'start': -500, 'end': 1, 'step
↪': .8, 'wait_time': 1e-2}), 'minstrument': [keithley3.amplitude]})
data1d = qtt.measurements.scans.scan1D(station, scanjob, location=None, verbose=1)
```

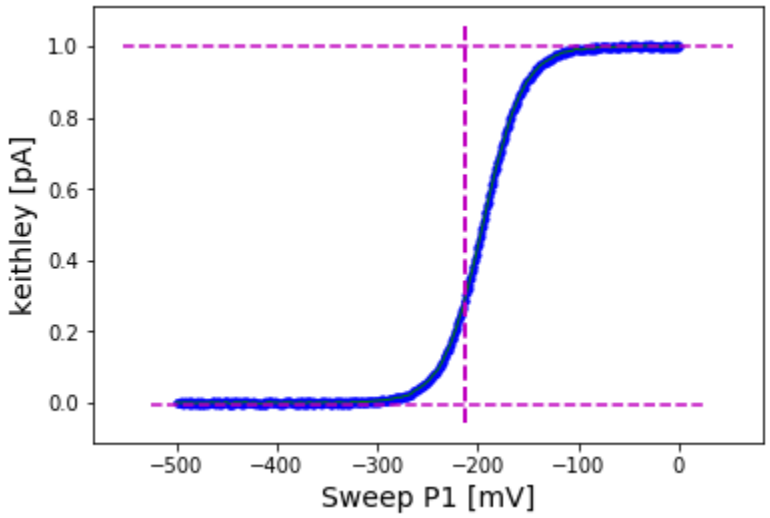
```
scan1D: 0/627: time 0.4
scan1D: 132/627: time 1.9
scan1D: 262/627: time 3.4
scan1D: 394/627: time 5.0
scan1D: 525/627: time 6.5
```

Analyse the scan

We have scripts for performing various analysis on our data (see the Analysis examples). In this example we use a script that determines the pinch-off voltage from our 1D scan.

[7]:
```
print( data1d )
adata = qtt.algorithms.gatesweep.analyseGateSweep(data1d, fig=100)
```

```
DataSet:
   location = '2018-09-05/15-45-03_qtt_scan1D'
   <Type>    | <array_id>           | <array.name>         | <array.shape>
   Measured | keithley3_amplitude | keithley3_amplitude | (627,)
   Setpoint | P1                   | P1                   | (627,)
analyseGateSweep: pinch-off point -212.000, value 0.297
```

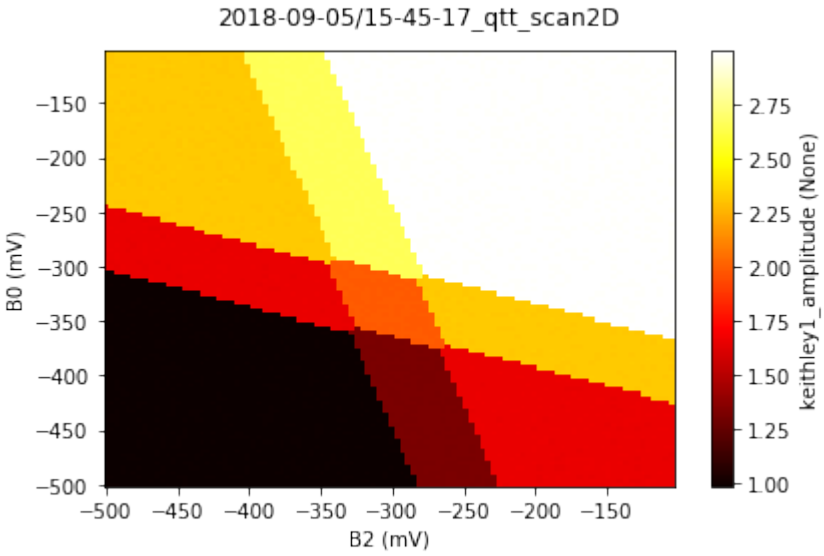


Make a 2D scan

We can also perfom scans of a 2-dimensional gate space. In this example the scan reveals the charge stability diagram of our 2-dot system.

[8]:
```
start=-500
scanjob = scanjob_t({'sweepdata': dict({'param': 'B2', 'start': start, 'end':␣
↪start+400, 'step': 4.}), 'minstrument': ['keithley1.amplitude'], 'wait_time': 0.})
scanjob['stepdata'] = dict({'param': 'B0', 'start': start, 'end': start+400, 'step':␣
↪4.})
data = qtt.measurements.scans.scan2D(station, scanjob, liveplotwindow=plotQ)

_=qcodes.MatPlot(data.default_parameter_array())
```

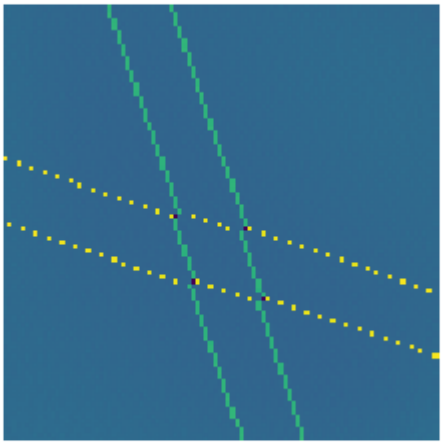
```
scan2D: 0/100: time 00:00:00 (~00:00:00 remaining): setting B0 to -500.000
scan2D: 65/100: time 00:00:02 (~00:00:01 remaining): setting B0 to -240.000
```



More analysis

[9]:
```
from qtt.utilities.imagetools import cleanSensingImage
im, tr = qtt.data.dataset2image(data)

imx=cleanSensingImage(im)
plt.imshow(imx)
plt.axis('off')
```

[9]:
```
(-0.5, 99.5, 99.5, -0.5)
```



Send data to powerpoint

We can copy data from a dataset to an open Powerpoint presentation using `qtt.utilities.tools.addPPT_dataset(dataset)`.

[10]:
```
_=qtt.utilities.tools.addPPT_dataset(data, verbose=1)
```

```
could not open active Powerpoint presentation, opening blank presentation
addPPTslide: presentation name: Presentation1, adding slide 1
 image aspect ratio 1.67, slide aspect ratio 1.78
adjust width 720->583
slide width height: [960.0, 540.0]
image width height: 583, 350
```

Browse the recorded data

We have a GUI for easy browsing of our saved datasets.

[11]:
```
logviewer = qtt.gui.dataviewer.DataViewer(datadir=datadir, verbose=1)
```

```
findfilesR: C:\Users\EENDEB~1\AppData\Local\Temp\tmpxtbpablc\qdata: 0.0%
DataViewer: found 2 files
```

We can fetch the active dataset from the viewer:

[13]:
```
dataset = logviewer.dataset
print( dataset )
```

```
DataSet:
   location = 'C:\\Users\\EENDEB~1\\AppData\\Local\\Temp\\tmpxtbpablc\\qdata\\2018-09-
↪05\\15-45-03_qtt_scan1D'
   <Type>    | <array_id>          | <array.name> | <array.shape>
   Setpoint  | P1                  | None         | (627,)
   Measured  | keithley3_amplitude | None         | (627,)
```

[ ]:
```
```

Using the virtual dot array

In this example we use the virtual dot array to show how to perofrm measurements and analysis using QTT. For a real device the measurements and analysis will go exactly the same (although you will have more noise and the scans will have more distortions).

[1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
import tempfile

import qcodes
from qcodes.plots.pyqtgraph import QtPlot
from qcodes.plots.qcmatplotlib import MatPlot
from qcodes.data.data_set import DataSet

import qtt
from qtt import createParameterWidget
```

(continues on next page)

(continued from previous page)

```
from qtt.algorithms.gatesweep import analyseGateSweep
from qtt.measurements.scans import scanjob_t
from qtt.instrument_drivers.virtual_gates import virtual_gates, create_virtual_matrix_
↪dict

from qtt import save_state, load_state
import qtt.measurements.videomode

import qtt.simulation.virtual_dot_array

np.set_printoptions(precision=2, suppress=True)
datadir = tempfile.mkdtemp(prefix='qtt_example')
DataSet.default_io = qcodes.data.io.DiskIO(datadir)
```

Create a virtual model for testing

The model resembles the spin-qubit dot setup. The hardware consists of a virtual keithley, IVVI racks and a virtual gates object

[2]:
```
nr_dots = 3
station = qtt.simulation.virtual_dot_array.initialize(reinit=True, nr_dots=nr_dots,␣
↪maxelectrons=2)
print(station.components.keys())
```

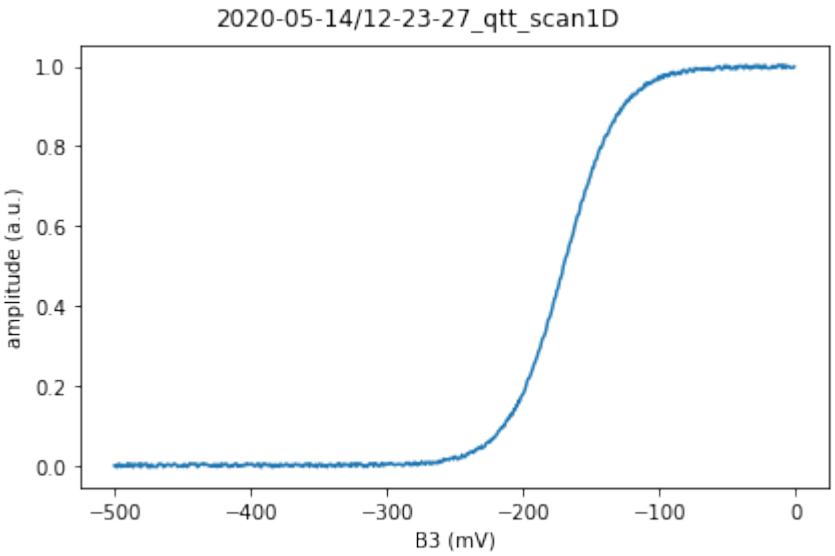
```
initialize: create virtual dot system
initialized virtual dot system (3 dots)
dict_keys(['gates', 'keithley1', 'keithley3', 'keithley4', 'ivvi1', 'ivvi2', 'vawg',
↪'sdigitizer', 'dotmodel'])
```

[3]:
```
keithley1 = station.keithley1
keithley3 = station.keithley3

gates = station.gates
```

Simple 1D scan loop

[4]:
```
param_left=station.model.bottomgates[0]
param_right=station.model.bottomgates[-1]
scanjob = scanjob_t({'sweepdata': dict({'param': param_right, 'start': -500, 'end': 0,
↪ 'step': .8, 'wait_time': 3e-3}), 'minstrument': ['keithley3.amplitude']})
data1d = qtt.measurements.scans.scan1D(station, scanjob, location=None, verbose=1)

print(data1d)

_ = MatPlot(data1d.default_parameter_array())
```

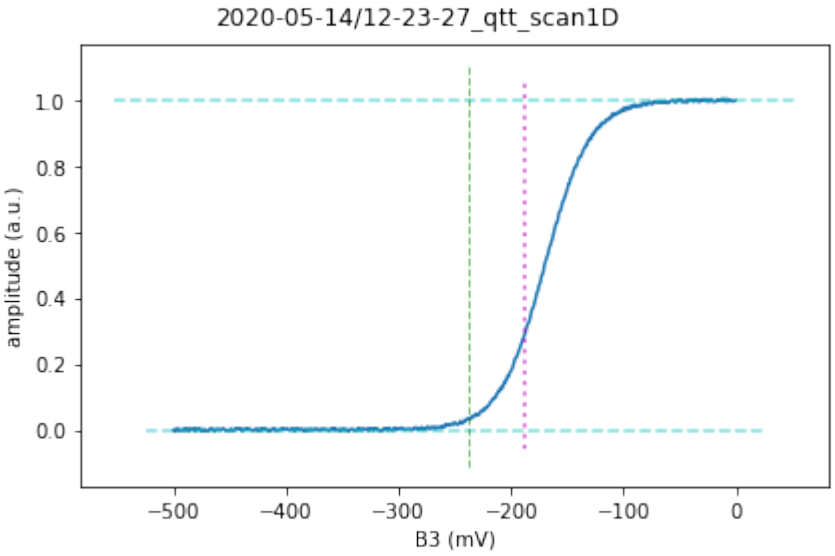
```
scan1D: 0/625: time 0.0
scan1D: 381/625: time 1.5
DataSet:
   location = '2020-05-14/12-23-27_qtt_scan1D'
   <Type>    | <array_id> | <array.name> | <array.shape>
   Measured | amplitude  | amplitude    | (625,)
   Setpoint | B3         | B3           | (625,)
```



Fit 1D pinch-off scan:

[5]:
```
adata = analyseGateSweep(data1d, fig=100)
```

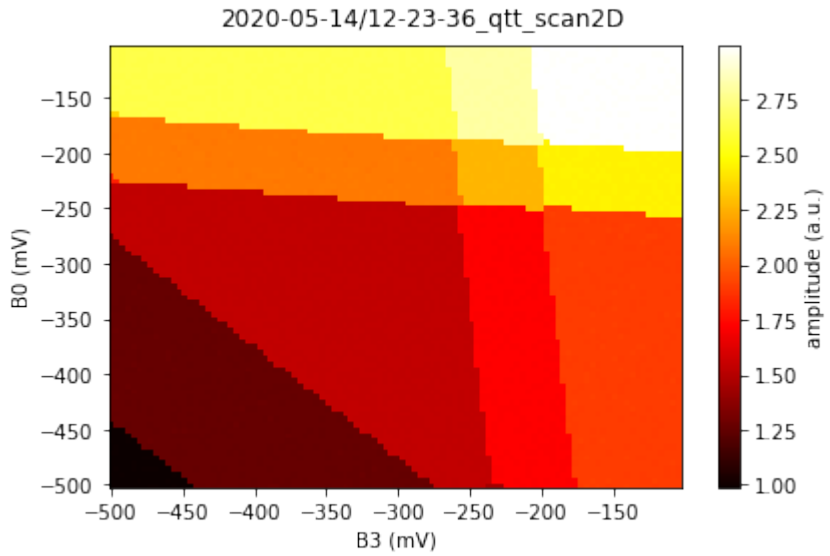
```
analyseGateSweep: pinch-off point -187.200, value 0.297
```



Make a 2D scan

[6]:
```
start = -500
scanjob = scanjob_t({'sweepdata': dict({'param': param_right, 'start': start, 'end':␣
↪start + 400, 'step': 4., 'wait_time': 0.}), 'minstrument': ['keithley1.amplitude']})
scanjob['stepdata'] = dict({'param': param_left, 'start': start, 'end': start + 400,
↪'step': 5.})
data2d = qtt.measurements.scans.scan2D(station, scanjob)

_ = MatPlot(data2d.default_parameter_array())
```

```
scan2D: 0/80: time 00:00:00 (~00:00:00 remaining): setting B0 to -500.000
```



[7]:
```
gv={'B0': -300.000,'B1': 0.145,'B2': -0.357,'B3': -300.000,'D0': 0.085,'O1': 0.222,'O2
↪': -0.403,'O3': 0.117,'O4': -0.275,'O5': -0.163,'P1': 30.,'P2': -40,'P3': -0.072,
↪'SD1a': 0.254,'SD1b': -0.442,'SD1c': 0.252,'bias_1': 0.337,'bias_2': -0.401}
gates.resetgates(gv, gv)
```

```
resetgates: setting gates to default values
  setting gate B0 to -300.0 [mV]
  setting gate B1 to 0.1 [mV]
  setting gate B2 to -0.4 [mV]
  setting gate B3 to -300.0 [mV]
  setting gate D0 to 0.1 [mV]
  setting gate O1 to 0.2 [mV]
  setting gate O2 to -0.4 [mV]
  setting gate O3 to 0.1 [mV]
  setting gate O4 to -0.3 [mV]
  setting gate O5 to -0.2 [mV]
  setting gate P1 to 30.0 [mV]
  setting gate P2 to -40.0 [mV]
  setting gate P3 to -0.1 [mV]
  setting gate SD1a to 0.3 [mV]
  setting gate SD1b to -0.4 [mV]
  setting gate SD1c to 0.3 [mV]
  setting gate bias_1 to 0.3 [mV]
  setting gate bias_2 to -0.4 [mV]
```

Make virtual gates

Instread of scanning physical gates, we can also scan linear combinations of gates. We use the `virtual_gates` object to define linear combinations and make scans.

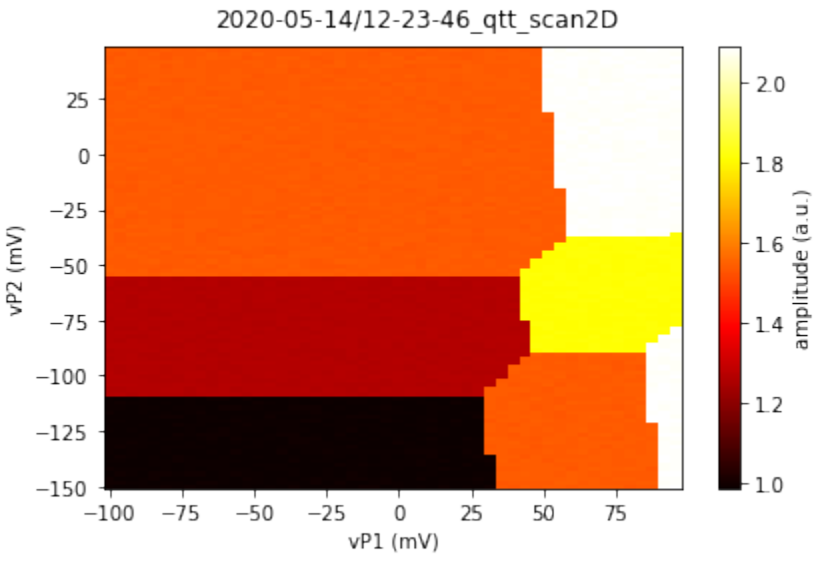
[8]:
```
gates.resetgates(gv, gv, 0)

c = np.array([[1,.56,.15],[.62,1,.593],[.14,.62,1.]])
crosscap_map = create_virtual_matrix_dict(['vP1', 'vP2', 'vP3'], ['P1', 'P2', 'P3'],␣
↪c=c)
virts = virtual_gates(qtt.measurements.scans.instrumentName('vgates'), gates,␣
↪crosscap_map)
virts.print_matrix()
```

```
create_virtual_matrix_dict: adding vP1
create_virtual_matrix_dict: adding vP2
create_virtual_matrix_dict: adding vP3
        P1      P2      P3
vP1     1       0.56    0.15
vP2     0.62    1       0.593
vP3     0.14    0.62    1
```

```
d:\dev\qtt_release\qtt\src\qtt\instrument_drivers\virtual_gates.py:-1: UserWarning:␣
↪Call to deprecated function VirtualGates.
```

[9]:
```
r=100
scanjob = scanjob_t({'sweepdata': dict({'param': virts.vP1, 'start': -r, 'end': r,
↪'step': 4.}), 'minstrument': ['keithley1.amplitude']})
scanjob['stepdata'] = dict({'param': virts.vP2, 'start': -50 - r, 'end': -50 +r, 'step
↪': 2.})
data_virtual_gates = qtt.measurements.scans.scan2D(station, scanjob)

_ = MatPlot(data_virtual_gates.default_parameter_array())
```

```
scan2D: 0/100: time 00:00:00 (~00:00:00 remaining): setting vP2 to -150.000
```



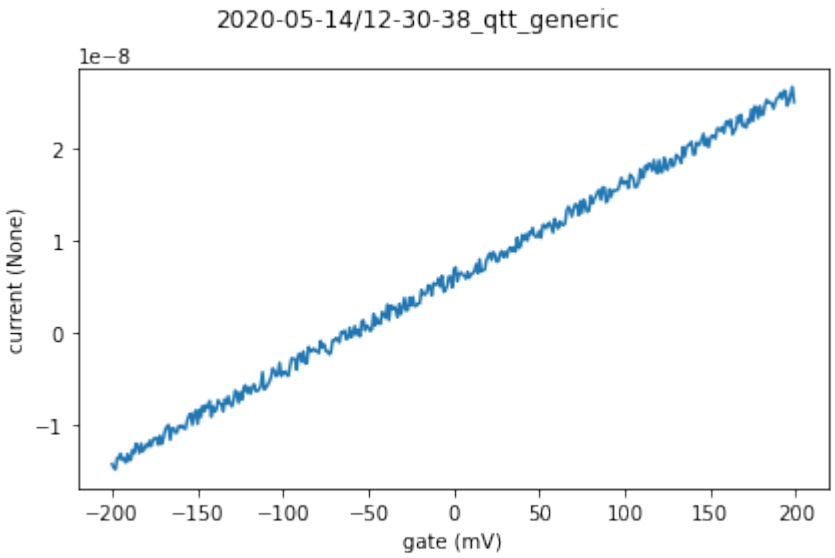
[ ]:

1.2.2 Analysis

Fit scan of Ohmic contact

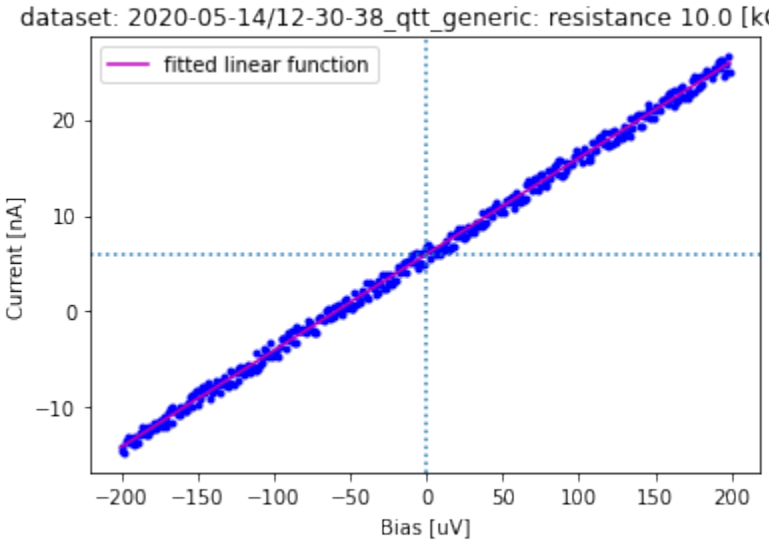
The core function used in this example is `fitOhmic`. The input for this function is a resistance plot; the linear relation between the bias voltage over and the current through the device. In this example a sample dataset is created; a linear relation with some random noise.

[1]:
```
%matplotlib inline
import qcodes.tests.legacy.data_mocks
from qcodes.plots.qcmatplotlib import MatPlot
import numpy as np
import qtt
from qtt.algorithms.ohmic import fitOhmic

ds = qcodes.tests.legacy.data_mocks.DataSet1D()
x = np.arange(-200, 200)
y = 1e-10 * (x + 50 + 20 * np.random.rand(x.size))
ds = qtt.data.makeDataSet1Dplain('gate', x, xunit='mV', yname='current', y=y)
_ = MatPlot(ds.default_parameter_array())
```



Fit the data with a linear function and plot the results. The 'fitOhmic' function will return a dictionary with the fit parameters (directional coefficient, intersection), the ohmic resistance and the biascurrent.

[2]:
```
r = fitOhmic(ds, fig=300, gainx=1e-6, gainy=1)
```



[3]:
```
print(r)
```

```
{'fitparam': array([1.00426032e-04, 5.95574163e-09]), 'resistance': 9957.577568364948,
↪ 'biascurrent': 5.955741625514289e-09, 'description': 'ohmic'}
```

[ ]:

Fitting PAT measurements

Authors: Pieter Eendebak, Sjaak van Diepen

The core package for this example is `qtt.algorithms.pat_fitting`. We use this package to analyse the data from a photon-assisted-tunneling measurement performed on 2-dot system, and extract the tunnel coupling and lever arm from the measurement.

For more information on PAT measurements and fitting see "Automated tuning of inter-dot tunnel coupling in double quantum dots", https://doi.org/10.1063/1.5031034

Import the modules used in this program:

[1]:
```
import os, sys
import qcodes
import scipy.constants
import matplotlib.pyplot as plt
import numpy as np

from qcodes.plots.qcmatplotlib import MatPlot
import qtt
from qtt.data import load_example_dataset
from qtt.algorithms.tunneling import fit_pol_all, polmod_all_2slopes
from qtt.algorithms.pat_fitting import fit_pat, plot_pat_fit, pre_process_pat, show_
↪traces, detect_peaks
%matplotlib inline
```

Load dataset

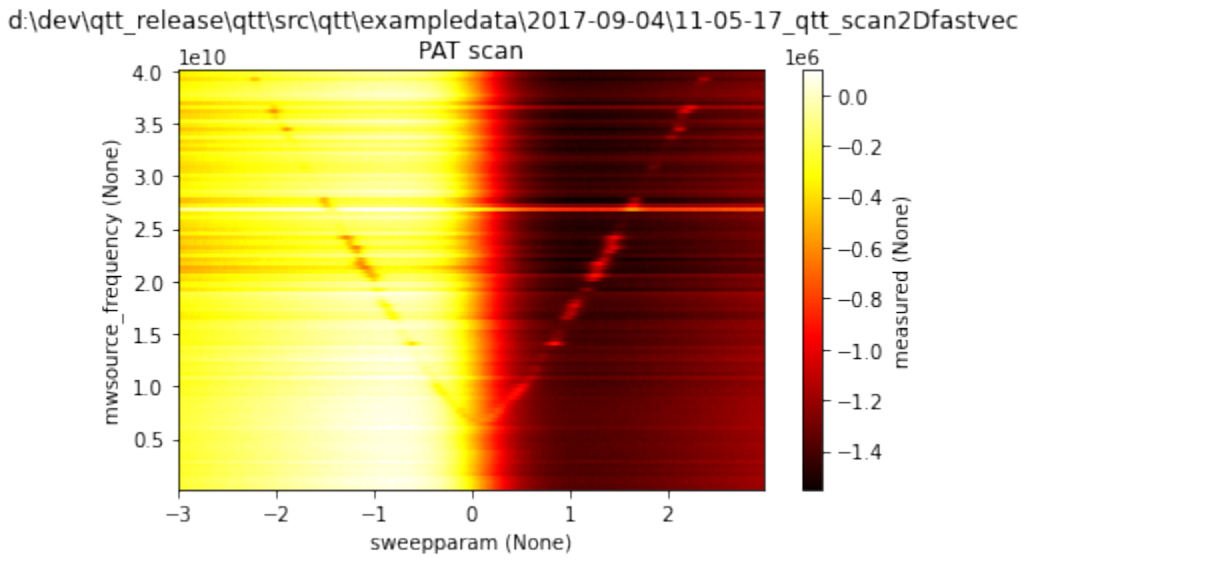
[2]:
```
dataset_pat = load_example_dataset('PAT_scan') # main dataset for PAT analysis
dataset_pol = load_example_dataset('PAT_scan_background') # 1D trace of the␣
↪background data
```

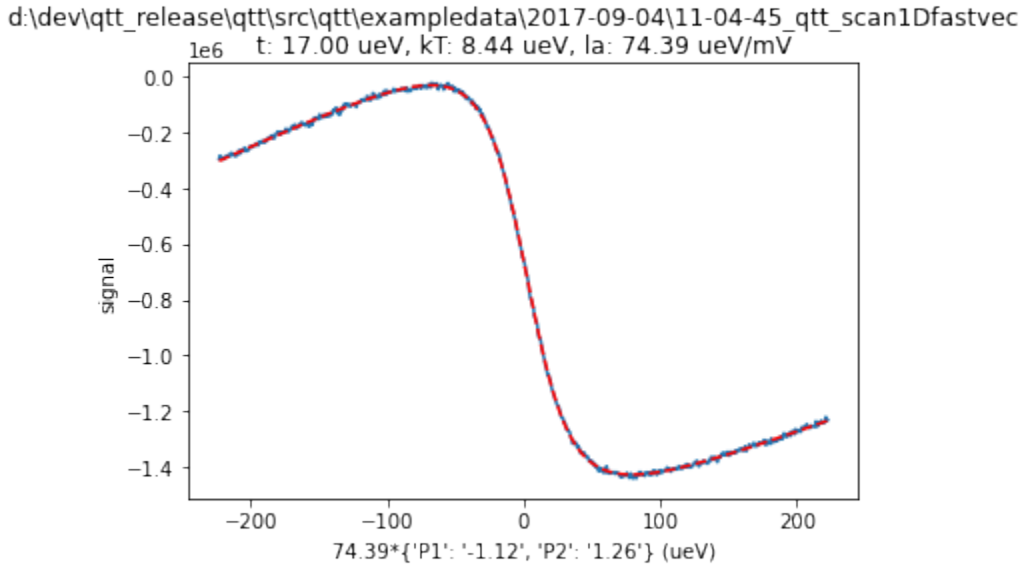
Set some parameters from the data.

[3]:
```
la = 74.39 # [ueV/mV], lever arm
sweep_detun = {'P1': -1.1221663904980717, 'P2': 1.262974805193041} # [mV on gate / mV␣
↪swept], sweep_detun * la = detuning in ueV
kb = scipy.constants.physical_constants['Boltzmann constant in eV/K'][0]*1e6  # [ueV/
↪K]
Te = 98e-3*kb # [ueV], electron temperature
ueV2GHz = 1e15*scipy.constants.h/scipy.constants.elementary_charge # [GHz/ueV]
```

Show the PAT scan and the background data.

[4]:
```
MatPlot(dataset_pat.default_parameter_array(), num=5)
plt.title('PAT scan')

pol_fit, pol_guess, _ = fit_pol_all(la*dataset_pol.sweepparam.ndarray, dataset_pol.
↪measured1, kT=Te)  # 1 indicates fpga channel

fig_pol = plt.figure(10)
plt.plot(la*dataset_pol.sweepparam.ndarray, dataset_pol.measured1)
plt.plot(la*dataset_pol.sweepparam.ndarray, polmod_all_2slopes(la*dataset_pol.
↪sweepparam.ndarray, pol_fit, kT=Te), 'r--')
plt.xlabel('%.2f*%s (ueV)' % (la,str({plg: '%.2f' % sweep_detun[plg] for plg in sweep_
↪detun})))
plt.ylabel('signal')
plt.title('t: %.2f ueV, kT: %.2f ueV, la: %.2f ueV/mV' % (np.abs(pol_fit[0]), Te, la))
_=plt.suptitle(dataset_pol.location)
```

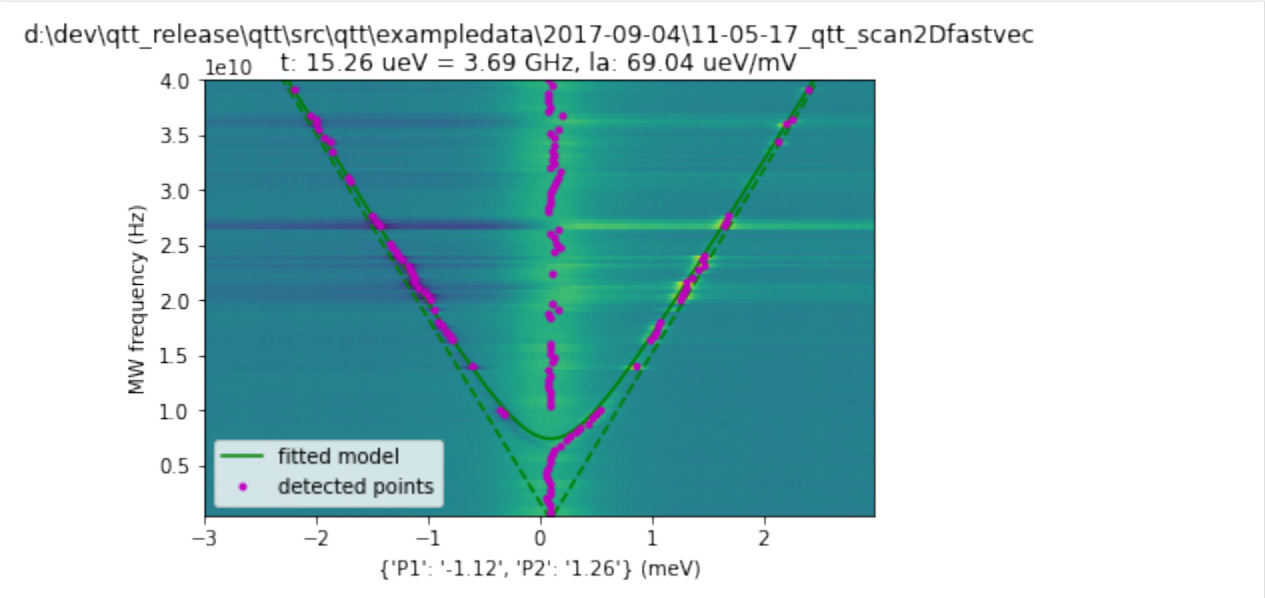




Fit PAT model

[5]:
```
x_data = dataset_pat.sweepparam.ndarray[0]
y_data = np.array(dataset_pat.mwsource_frequency)
z_data = np.array(dataset_pat.measured)
background = np.array(dataset_pol.default_parameter_array())

pp, pat_fit = fit_pat(x_data, y_data, z_data, background)
imq=pat_fit['imq']
```

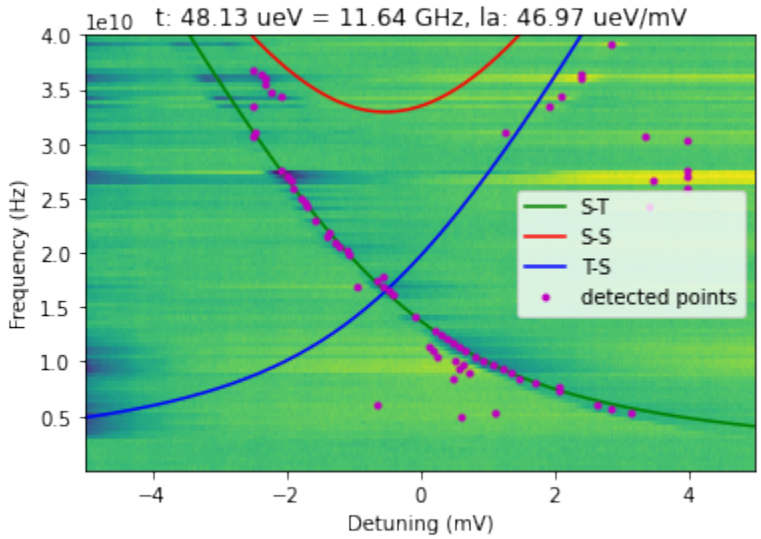
[6]:
```
pat_fit_fig = plt.figure(100); plt.clf()
plot_pat_fit(x_data, y_data, imq, pp, fig=pat_fit_fig.number, label='fitted model')
plt.plot(pat_fit['xd'], pat_fit['yd'], '.m', label='detected points')
plt.title('t: %.2f ueV = %.2f GHz, la: %.2f ueV/mV' % (np.abs(pp[2]), np.abs(pp[2]/
↪ueV2GHz), pp[1]))
plt.suptitle(dataset_pat.location)
plt.xlabel('%s (meV)' % (str({plg: '%.2f' % sweep_detun[plg] for plg in sweep_detun}
↪)))
plt.ylabel('MW frequency (Hz)')
_=plt.legend()
```



Fit 2-electron model

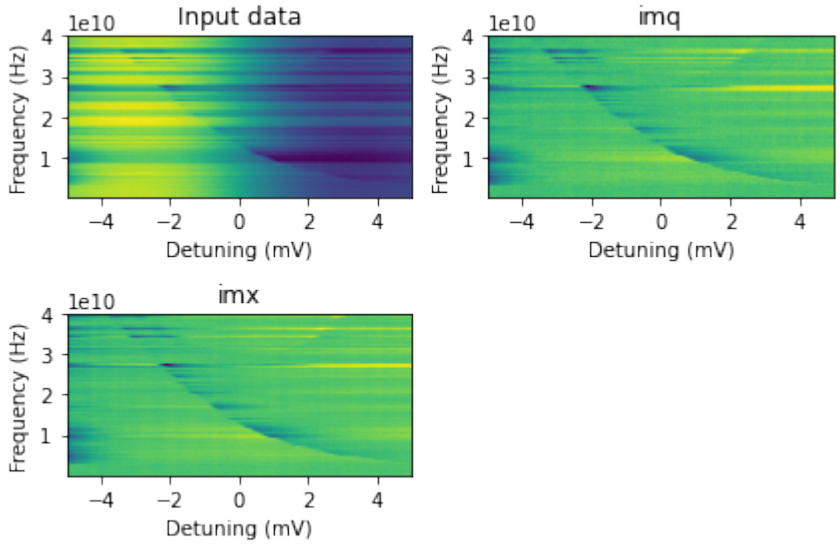
[7]:
```
dataset_pat = load_example_dataset(r'2electron_pat/pat')
dataset_pol = load_example_dataset(r'2electron_pat/background')
```

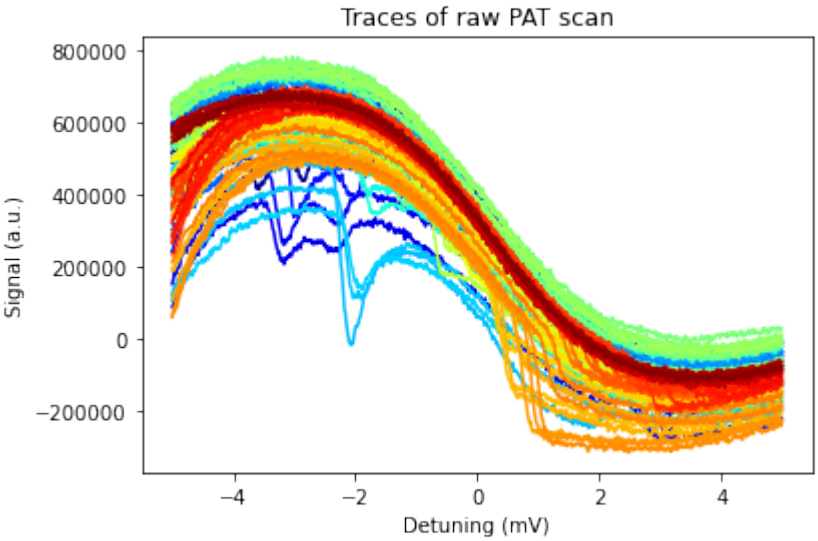
[8]:
```
x_data = dataset_pat.sweepparam.ndarray[0]
y_data = np.array(dataset_pat.mwsource_frequency)
z_data = np.array(dataset_pat.measured)
background = np.array(dataset_pol.default_parameter_array())

pp, pat_fit = fit_pat(x_data, y_data, z_data, background, trans='two_ele', even_
↪branches=[True, False, False])
imq=pat_fit['imq']

plot_pat_fit(x_data, y_data, imq, pp, fig=pat_fit_fig.number, label='fitted model',␣
↪trans='two_ele')
plt.plot(pat_fit['xd'], pat_fit['yd'], '.m', label='detected points')
plt.title('t: %.2f ueV = %.2f GHz, la: %.2f ueV/mV' % (np.abs(pp[2]), np.abs(pp[2]/
↪ueV2GHz), pp[1]))
_=plt.legend()
```



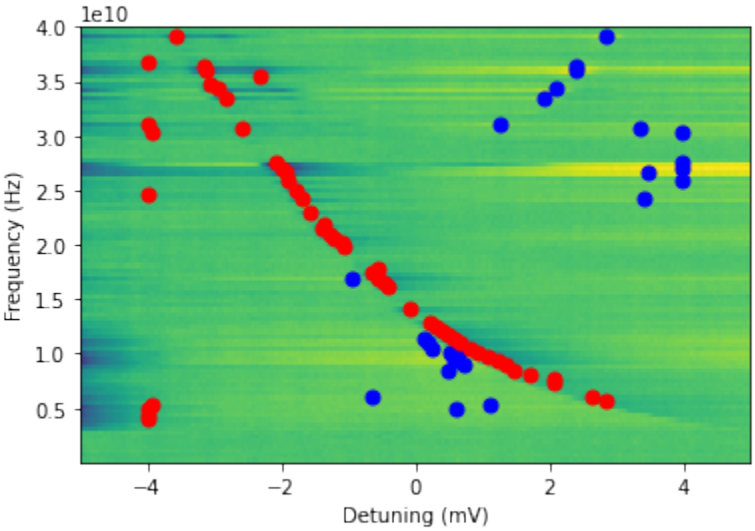
Show pre-processing and intermediate steps

[9]:
```
imx, imq, _ = pre_process_pat(x_data, y_data, background, z_data, fig=100)
show_traces(x_data, z_data, fig=101, direction='h', title='Traces of raw PAT scan')
```





[10]:
```
xx, _ = detect_peaks(x_data, y_data, imx, sigmamv=.05, fig=200)
```



[ ]:
```
```

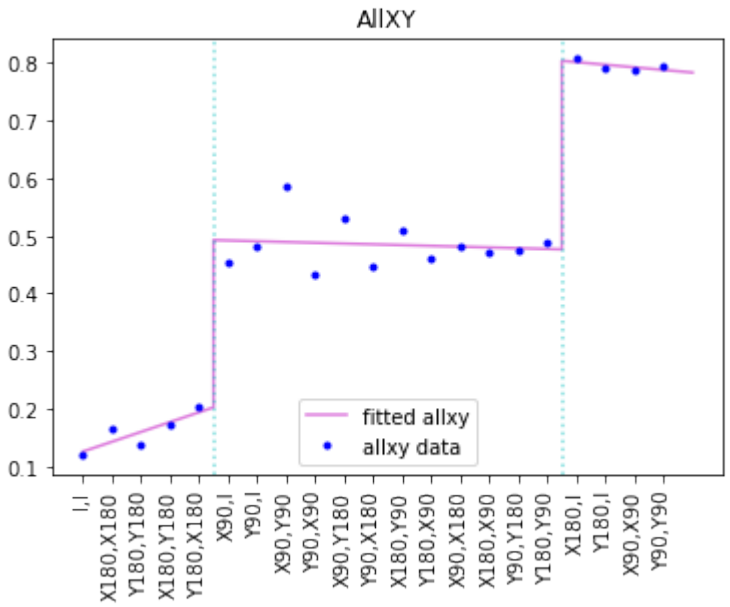
Fitting an AllXY experiment

The AllXY experiment was introduced by M. Reed in "Entanglement and Quantum Error Correction with Superconducting Qubits". It allows for a quick analysis of the power, frequency detuning of qubit gates.

[1]:
```
import numpy as np
import qtt
from qtt.algorithms.allxy import fit_allxy, plot_allxy, allxy_model
```

We define example data for an AllXY experiment and fit the data to the `allxy_model`.

[2]:
```
dataset = qtt.data.makeDataSet1Dplain('index', np.arange(21), 'allxy', [0.12      , 0.
↪16533333, 0.136     , 0.17066666, 0.20266667,
           0.452     , 0.48133334, 0.58666666, 0.43199999, 0.52933334,
           0.44533333, 0.51066667, 0.46      , 0.48133334, 0.47066667,
           0.47333333, 0.488     , 0.80799999, 0.78933333, 0.788     ,
           0.79333333])
result = fit_allxy(dataset)
plot_allxy(dataset, result, fig=1)
```



[3]:
```
print(result)
```

```
{'fitted_parameters': array([ 0.15893333,  0.01706667,  0.48422222, -0.00134266,  0.
↪7924    ,
       -0.00453333]), 'initial_parameters': array([0.15893333, 0.        , 0.48422222,
↪ 0.        , 0.79466666,
       0.        ]), 'reduced_chi_squared': 0.001369275257254969, 'type':
↪'Model(allxy_model)', 'fitted_parameter_dictionary': {'offset0': 0.158933332,
↪'slope0': 0.017066667, 'offset1': 0.4842222225, 'slope1': -0.0013426571678321682,
↪'offset2': 0.792399997, 'slope2': -0.004533331000000019}, 'fitted_parameters_
↪covariance': array([2.73855051e-04, 1.36927526e-04, 1.14106271e-04, 9.57535145e-06,
       4.10782577e-04, 2.73855051e-04]), 'description': 'allxy fit'}
```

The `fitted_parameters` are the parameters of the `allxy_model` function.

[4]:
```
help(allxy_model)
```

```
Help on function allxy_model in module qtt.algorithms.allxy:

allxy_model(indices:Union[float, numpy.ndarray], offset0:float, slope0:float, offset1:
↪float, slope1:float, offset2:float, slope2:float) -> Union[float, numpy.ndarray]
    Model for AllXY experiment

    The model consists of three linear segments. The segments correspond to the pairs␣
↪of gates that result in
    fraction 0, 0.5 and 1 in the AllXY experiment.
```

(continues on next page)

(continued from previous page)

```
    Args:
        index: Indices of the allxy pairs or a single index
        offset0: Offset of first segment
        slope0: Slope of first segment
        offset1: Offset of second segment
        slope1: Slope of second segment
        offset2: Offset of last segment
        slope2: Slope of last segment
    Returns:
        Fractions for the allxy pairs
```

[ ]:

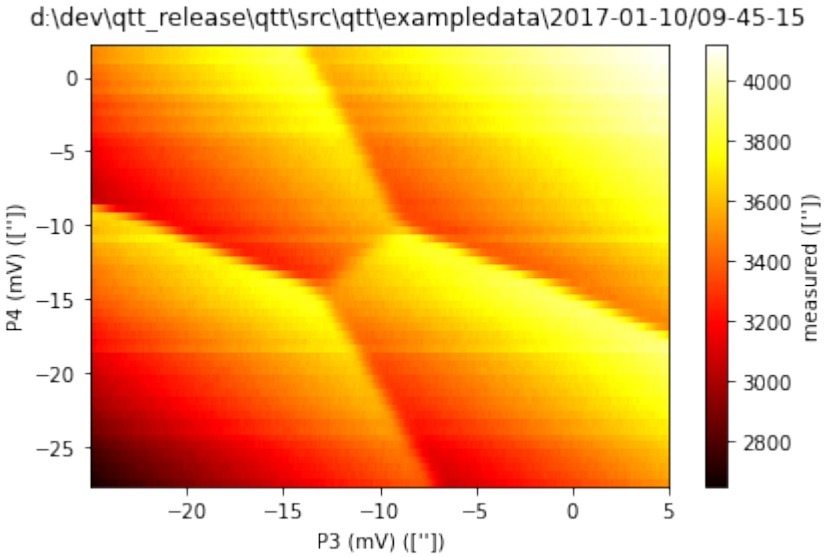
Example of automatic fitting of anti-crossing

Pieter Eendebak pieter.eendebak@tno.nl

[1]:
```
# import the modules used in this program:
import sys, os, time
import qcodes
import numpy as np

import matplotlib.pyplot as plt
%matplotlib inline

import scipy.optimize
import cv2
import qtt
import qtt.measurements
from qtt.algorithms.anticrossing import fit_anticrossing, plot_anticrossing
from qtt.data import load_example_dataset
```

Load dataset

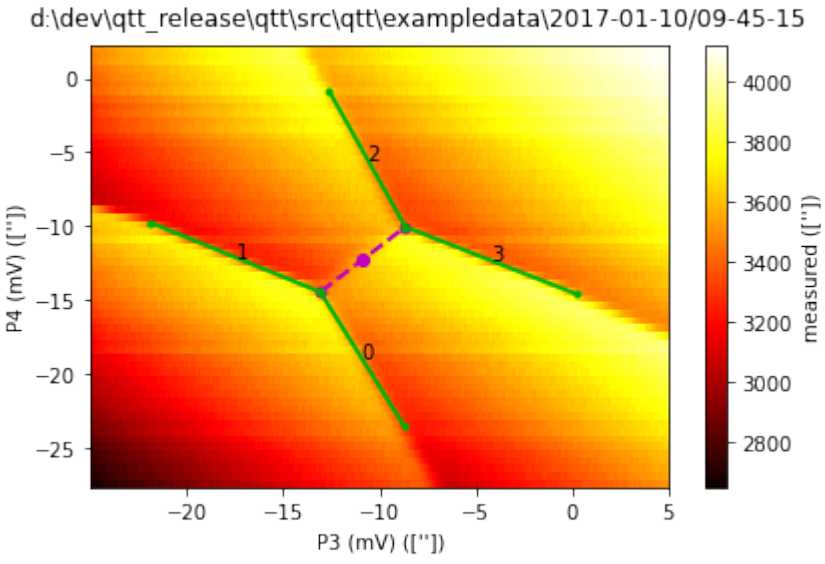
[2]:
```
data = load_example_dataset('charge_stability_diagram_anti_crossing')

qtt.data.plot_dataset(data, fig=10)
```



[3]:
```
fit_results = fit_anticrossing(data)
```

```
straightenImage: size (60, 928) fx 0.1294 fy 2.0000
straightenImage: result size (120, 120) mvx 0.2500 mvy 0.2500
fitModel: score 1618.90 -> 1297.85
fitModel: score 1297.85 -> 1297.85
fit_anticrossing: patch size (60, 60)
```

[4]:
```
plot_anticrossing(data, fit_results)
```



[5]:
```
print(fit_results)
```

```
{'labels': ['P4', 'P3'], 'centre': array([[-10.87200927],
       [-12.26911426]]), 'fitpoints': {'centre': array([[-10.87200927],
       [-12.26911426]]), 'left_point': array([[-13.08187867],
       [-14.47898366]]), 'right_point': array([[ -8.66213987],
       [-10.05924485]]), 'inner_points': array([[-13.08187867, -13.08187867,  -8.
↪66213987,  -8.66213987,
```

(continues on next page)

(continued from previous page)

```
       -24.87058186],
      [-14.47898366, -14.47898366, -10.05924485, -10.05924485,
         2.15483141]]), 'outer_points': array([[ -8.75020813, -21.89998627, -12.
↪61622885,   0.23561125],
       [-23.4921202 ,  -9.76294947,  -0.87419285, -14.62323707]])}, 'fit_params':␣
↪array([13.99857259, 14.42394567,  3.12522728,  1.12279268,  3.63270088,
        4.30587599,  0.47394414]), 'params': {}}
```

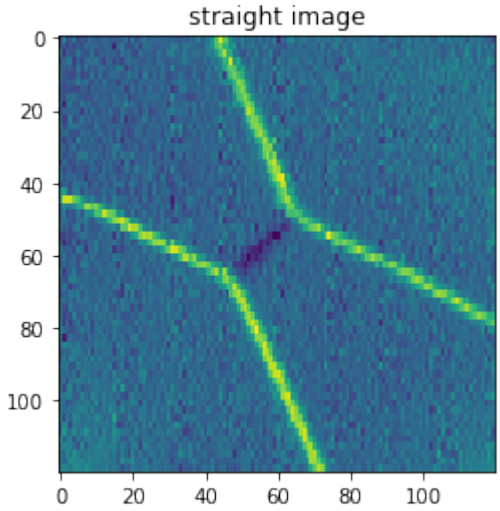
Detailed steps (mainly for debugging)

Pre-process image to a honeycomb

[6]:
```
from qtt.algorithms.images import straightenImage
from qtt.utilities.imagetools import cleanSensingImage
from qtt.utilities.tools import showImage as showIm
from qtt.measurements.scans import fixReversal
from qtt.utilities.imagetools import fitModel, evaluateCross
```

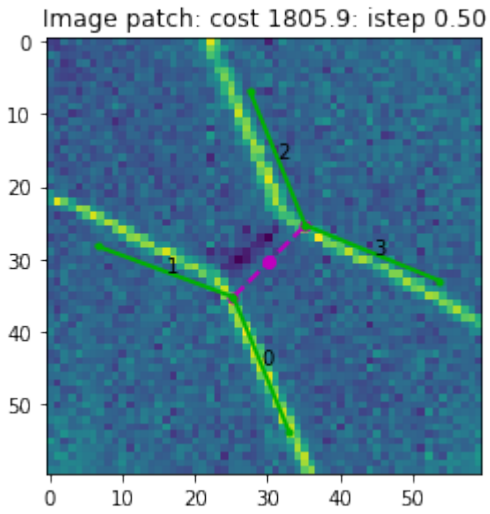
[7]:
```
im, tr = qtt.data.dataset2image(data)
imextent = tr.scan_image_extent()
mpl_imextent =tr.matplotlib_image_extent()
istep=.25

imc = cleanSensingImage(im, sigma=0.93, verbose=1)
imx, (fw, fh, mvx, mvy, Hstraight) = straightenImage(imc, imextent, mvx=istep,␣
↪verbose=2)

imx = imx.astype(np.float64)*(100./np.percentile(imx, 99)) # scale image

showIm(imx, fig=100, title='straight image')
```

```
fitBackground: is1d 0, order 3
 checkReversal: 1 (mval 0.1, thr -0.4)
straightenImage: size (60, 928) fx 0.1294 fy 2.0000
straightenImage: result size (120, 120) mvx 0.2500 mvy 0.2500
```



Initial input

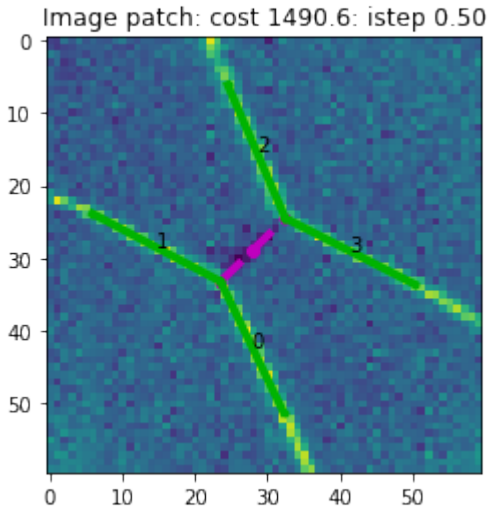
[8]:
```
istepmodel = .5
ksizemv = 31
param0 = [(imx.shape[0] / 2 + .5) * istep, (imx.shape[0] / 2 + .5) * istep, \
          3.5, 1.17809725, 3.5, 4.3196899, 0.39269908]
param0e = np.hstack((param0, [np.pi / 4]))
cost, patch, r, _ = evaluateCross(param0e, imx, verbose=0, fig=21, istep=istep,␣
↪istepmodel=istepmodel)
```

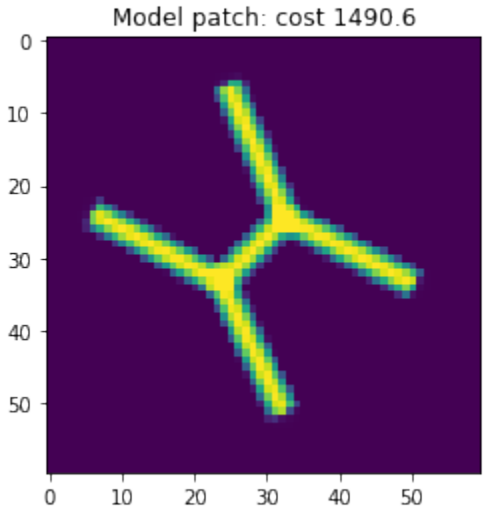


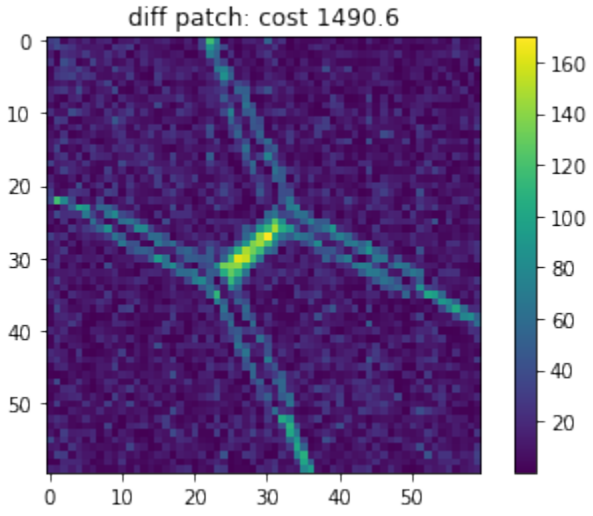
Find the anti-crossing

[9]:
```
t0 = time.time()
res = qtt.utilities.imagetools.fitModel(param0e, imx, verbose=1, cfig=10, istep=istep,
                    istepmodel=istepmodel, ksizemv=ksizemv, use_abs=True)
param = res.x
dt = time.time() - t0
print('calculation time: %.2f [s]' % dt)

cost, patch, cdata, _ = evaluateCross(param, imx, verbose=1, fig=25, istep=istep,␣
↪istepmodel=istepmodel, linewidth=4)
```

```
fitModel: score 1618.90 -> 1297.57
calculation time: 2.48 [s]
evaluateCross: patch shape (60, 60)
  add cost for image cc: 2.0
```



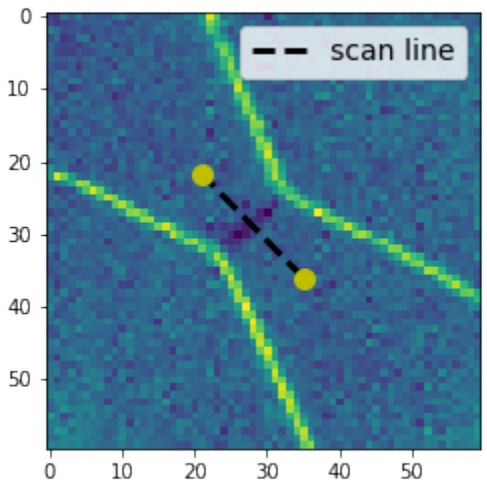





```
evaluateCross: cost 1490.5538
```

Show orthogonal line (for polarization scan)

[10]:
```
showIm(patch, fig=25)
ppV, ccV, slopeV = qtt.utilities.imagetools.Vtrace(cdata, param, fig=25)
```



[ ]:
```
```

Example fitting AWG to plunger factor

Many systems have a bias T with attenuation on the fast lines. We need to have the correct factor for the AWG to the gate voltage, to make sure we are able to apply the same voltage to the gates with both the fast and the slow lines.

The data for this test can be acquired by making a scan2Dfast measurement, with the same gate on both axis, where the one axis is sweeped with the awg and one axis is stepped with the DAC's. Measurement should roughly be centred around an addition line. The measurement can be performed with the function `measure_awg_to_plunger`, which can be found in qtt.algorithms.awg_to_plunger.

From the slope of the addition line the awg to plunger conversion factor can be checked with the function `analyse_awg_to_plunger`. If the awg_to_plunger factor is correct, the addition line in a scan2dfast measurement should have a slope of -1. The code will fit the addition line, calculate the slope and from there the correction to the awg to plunger factor.

Importing packages:

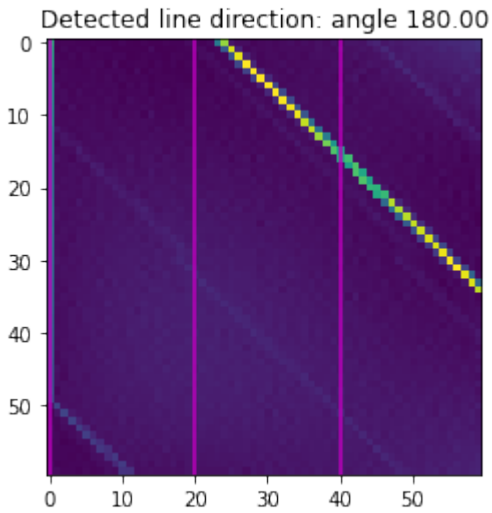
[1]:
```
%matplotlib inline
import os
import qtt
import pickle
exampledatadir=os.path.join(qtt.__path__[0], 'exampledata')

from qtt.algorithms.awg_to_plunger import get_dataset, analyse_awg_to_plunger
```

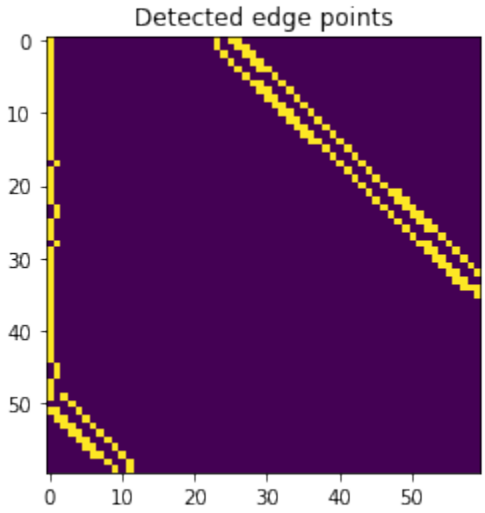
Import the data and give it the shape it would have as a result of running the 'measure_awg_to_plunger' function.

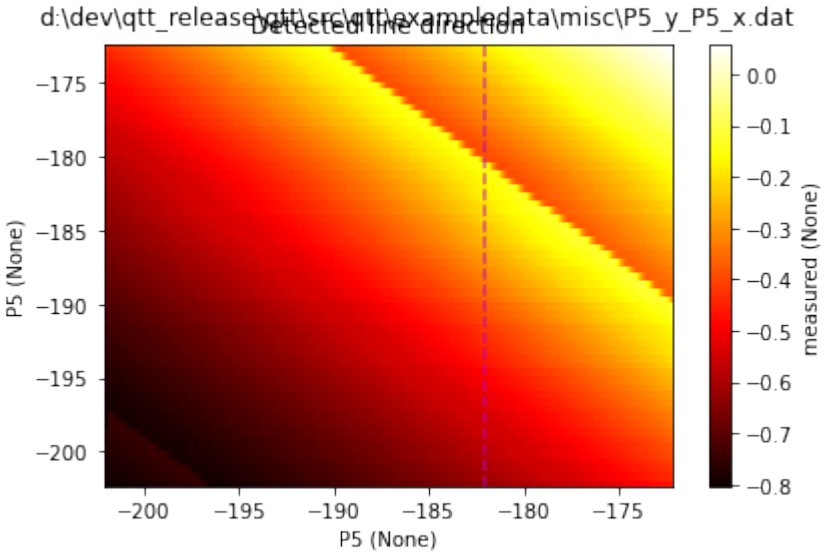
[2]:
```
dfile=os.path.join(exampledatadir, 'charge_stability_diagram_dac_vs_awg', 'charge_
↪stability_diagram_dac_vs_awg.dat')
ds = get_dataset(dfile)
result = {'type': 'awg_to_plunger', 'awg_to_plunger': None, 'dataset': ds.location}
```

Running the function 'analyse_awg_to_plunger'. This function has two options for method: 'hough' (demonstrated in this example) and 'click'. The 'hough' method finds the addition line automatically using the function cv2.HoughLines. This method is prefered, since it is most acurate. If it does not work for some reason, the method 'click' offers the posibility to indicate where the addition line is by clicking on it twice.

[3]:
```
aresult = analyse_awg_to_plunger(result, method='hough', fig=100)
```







[4]:
```
print('detected angle:  %.2f degrees' % aresult['angle_degrees'] + '\n' + 'correction␣
↪of awg to plunger factor: %.5f'% aresult['correction of awg_to_plunger'])
```

```
detected angle:  180.00 degrees
correction of awg to plunger factor: 8165619676597685.00000
```

[ ]:

Compensation for a non-linear charge sensor

We analyse the effect of a non-linear sensing dot on the value for the tunnel coupling obtained from the fitting of an inter-dot transition line. The sensing dot shape is simulated based on a Gaussian, while the data of the inter-dot transition is experimental data.

First we load all necessary packages

[1]:
```
import os
import qcodes
import matplotlib.pyplot as plt
import time
import numpy as np
import scipy

from qcodes.data.hdf5_format import HDF5Format

import qtt
import qtt.pgeometry
from qtt.data import load_example_dataset
from qtt.algorithms.functions import gaussian
from qtt.algorithms.tunneling import polmod_all_2slopes, fit_pol_all
from qtt.algorithms.chargesensor import DataLinearizer, correctChargeSensor
%matplotlib inline

np.set_printoptions(suppress=True, precision=3)
```

[2]:
```
def show_pol_fit(delta, signal, par_fit, fig=1):
    """ Show data of a polarization fit """
    plt.figure(fig)
    plt.clf()
    plt.plot(delta, signal, 'bo')
    plt.plot(delta, polmod_all_2slopes(delta, par_fit, kT), 'r')
    plt.title('Tunnel coupling: %.2f (ueV) = %.2f (GHz)' %
              (par_fit[0], par_fit[0] / h))
    plt.xlabel('Difference in chemical potentials (ueV)')
    _ = plt.ylabel('Signal (a.u.)')
```

Define physical constants and parameters

[3]:
```
h = 1e9*scipy.constants.h/(1e-6*scipy.constants.elementary_charge)  # Planck's␣
↪constant in units [ueV/GHz]
kb = scipy.constants.k/(1e-6*scipy.constants.elementary_charge)  # [ueV/K], Boltzmann␣
↪constant
kT = 10e-3 * kb  # effective electron temperature in ueV
```

Load example dataset and define signal of charge sensor

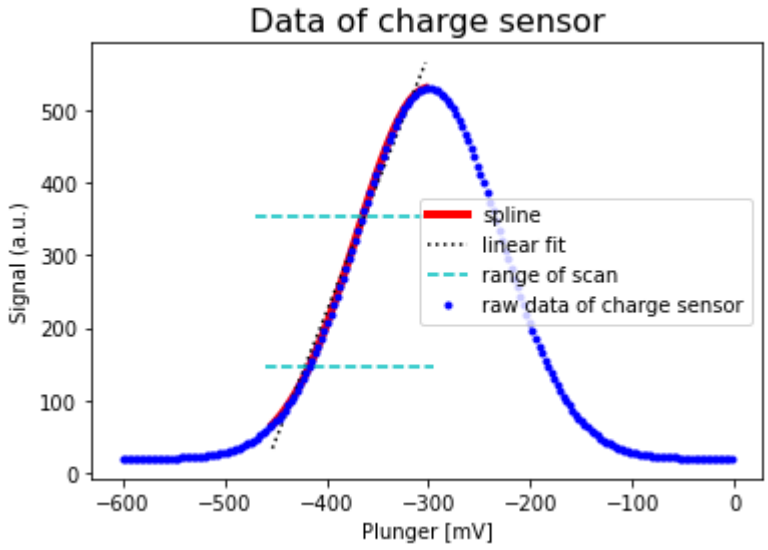
[4]:
```
dataset = load_example_dataset('polarization_line')

signal = np.array(dataset.default_parameter_array('signal'))
delta = np.array(dataset.default_parameter_array('signal').set_arrays[0])

# Define signal of the sensing dot
xs = np.arange(-600, 0, 3.)
ys = gaussian(xs, -300, std=70, amplitude=510, offset=2)
ys = gaussian(xs, -300, std=70, amplitude=510, offset=20)
```

Find range of sensing dot used

The correction to the non-linearity of the charge sensor is done by fitting a linear function in the region of interest.

[5]:
```
dl, results = correctChargeSensor(delta, signal, xs, ys, fig=100)
plt.plot(xs, ys, '.b', label='raw data of charge sensor')
plt.legend()
plt.title('Data of charge sensor', fontsize=16)
_ = plt.xlabel('Plunger [mV]')
_ = plt.ylabel('Signal (a.u.)')
```

```
fitCoulombPeaks: peak 0: position -300.00 max 530.00 valid 1
filterPeaks: 1 -> 1 good peaks
peakScores: noise factor 1.00
peakScores: 0: height 268.3 halfwidth 27.0, score 144.87
```



Determine the corrected data points

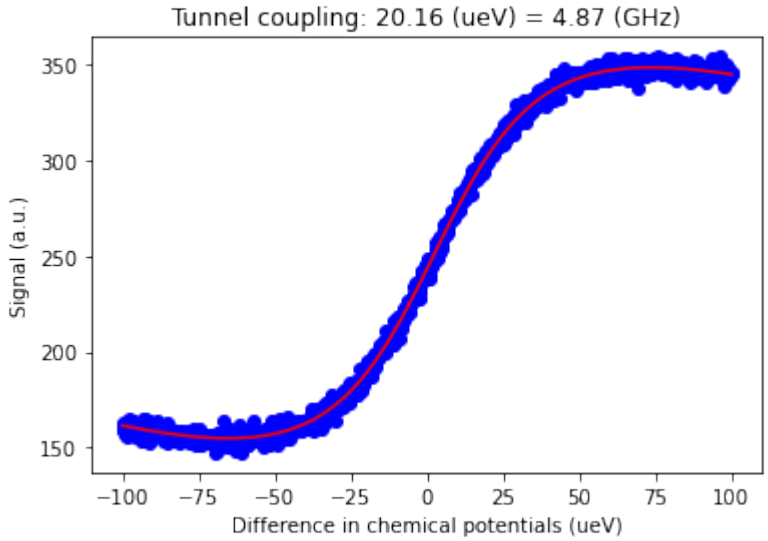
[6]:
```
xsignal = dl.backward_curve(signal)
signal_corrected = dl.forward(xsignal)  # make sure data is in similar range
```

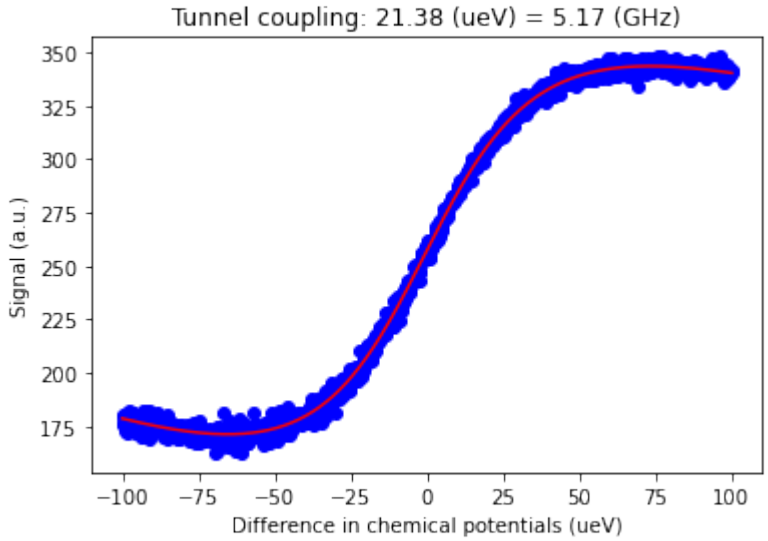
Fit the polarization line

The effect of the non-linear charge sensor is a 5% error in the estimated tunnel coupling.

[7]:
```
par_fit,_ , _ = fit_pol_all(delta, signal, kT, par_guess=None)
show_pol_fit(delta, signal, par_fit, fig=1)

par_fit_corrected, _, _ = fit_pol_all(delta, signal_corrected, kT)
show_pol_fit(delta, signal_corrected, par_fit_corrected, fig=2)

print('tunnel coupling: %.1f [GHz]' % (par_fit[0] / h))
print('tunnel coupling with compensation: %.1f [GHz]' % (par_fit_corrected[0] / h))
```

```
tunnel coupling: 4.9 [GHz]
tunnel coupling with compensation: 5.2 [GHz]
```





[8]:
```
print('### fitted parameters ###')
print(par_fit)
print(par_fit_corrected)
```

```
### fitted parameters ###
[ 20.16    1.969  99.64   -0.505  -0.442 300.373]
[ 21.38   -1.59  112.843  -0.558  -0.406 280.184]
```

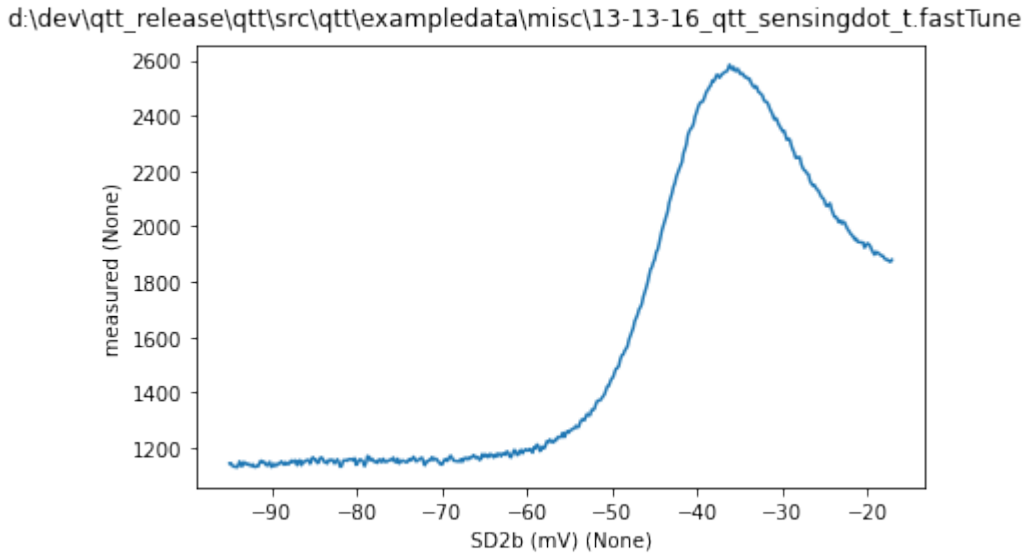
[ ]:

Analyse and fit Coulomb peaks

Load packages

[1]:
```
import os
import numpy as np
import qcodes
from qcodes.plots.qcmatplotlib import MatPlot
import matplotlib.pyplot as plt
%matplotlib inline

import qtt
import qtt.algorithms.coulomb
from qtt.algorithms.coulomb import analyseCoulombPeaks
from qtt.data import load_example_dataset
```

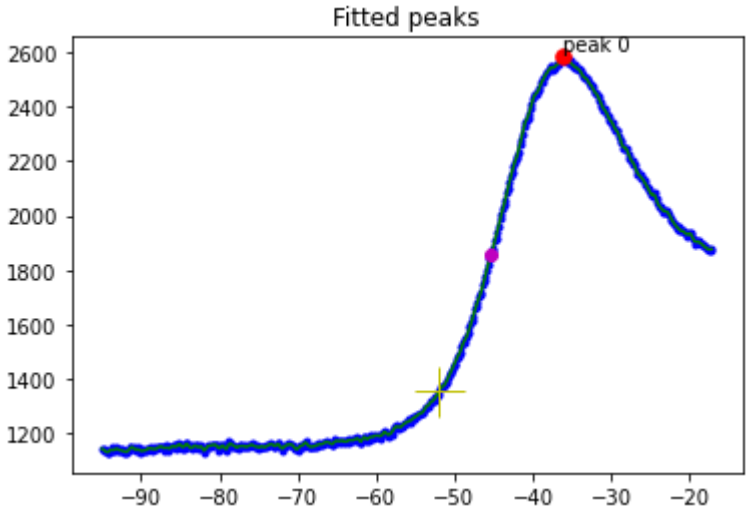
[2]:
```
dataset=load_example_dataset('coulomb_peak')
qtt.data.plot_dataset(dataset)
```



Fit Coulomb peaks

[3]:
```
peaks=qtt.algorithms.coulomb.analyseCoulombPeaks(dataset, fig=10)
_=plt.plot(peaks[0]['xbottom'], peaks[0]['ybottoml'], '+y', markersize=25)
```

```
fitCoulombPeaks: peak 0: position -36.25 max 2583.48 valid 1
filterPeaks: 1 -> 1 good peaks
peakScores: noise factor 1.00
peakScores: 0: height 1225.8 halfwidth 9.0, score 1287.70
```



[4]:
```
peaks[0]
```

[4]:
```
{'p': 348,
 'x': -36.2474,
 'y': 2583.48,
 'gaussfit': array([-3.57655832e+01,  1.22339747e+01,  7.86964941e+04]),
 'halfvaluelow': 1857.8959,
 'height': 1451.1682,
 'valid': 1,
 'lowvalue': 1132.3118,
 'type': 'peak',
 'phalf0': 294,
 'phalfl': None,
 'xhalfl': -45.286672018348625,
 'xfoot': -53.90643419525066,
 'yhalfl': 1860.27,
 'pbottomlow': 223,
 'pbottom': 255,
 'pbottoml': 255,
 'xbottom': -51.9487,
 'xbottoml': -51.9487,
 'vbottom': 1357.64,
 'ybottoml': 1357.64,
 'score': 1287.6962930290897,
 'slope': 101.20512200893356,
 'heightscore': 0.8574826172722056,
 'scorerelative': 0.9007514745793096,
 'noisefactor': 0.9998830914752265}
```

[ ]:

Fitting an RTS histogram and extracting the tunnel rates

The core function for this example is `tunnelrates_RTS` from `qtt.algorithms.random_telegraph_signal`. The function takes a time-resolved dataset with a random telegraph signal and fits the histogram of signal values to a double-gaussian function, from which the SNR based visibility of the signal can be obtained. It then searches dataset for the RTS transitions (up and down) and from the histograms of transitions, calculates the tunnel rates for both transitions.

[1]:
```
import os
import qcodes
from qcodes.data.data_set import DataSet
import qtt

%matplotlib inline
import numpy as np
from qtt.algorithms.random_telegraph_signal import tunnelrates_RTS
```

Load a sample dataset with a time-resolved trace with RTS:

[2]:
```
exampledatadir=os.path.join(qtt.__path__[0], 'exampledata')
DataSet.default_io = qcodes.data.io.DiskIO(exampledatadir)
dataset = qtt.data.load_dataset('rts_signal')

rtsdata = dataset.measured.ndarray
samplerate = 1 / (dataset.time[1] - dataset.time[0])
```

Run the core function and obtain the tunnel rates:

[3]:
```
tunnelrates_RTS(rtsdata, samplerate=samplerate, min_sep = 1.0, min_duration = 20, num_
↪bins = 40, fig=1, verbose=1)
```

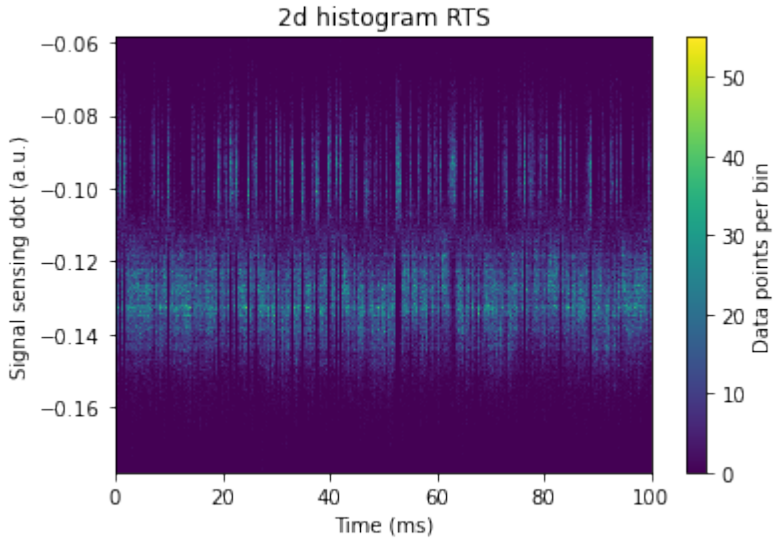
```
Fit parameters double gaussian:
 mean down: -0.130 counts, mean up: -0.097 counts, std down: 0.010 counts, std up:0.
↪010 counts
Separation between peaks gaussians: 1.653 std
Split between two levels: -0.114
Tunnel rate down to up: 94.3 kHz
Tunnel rate up to down: 72.1 kHz
```

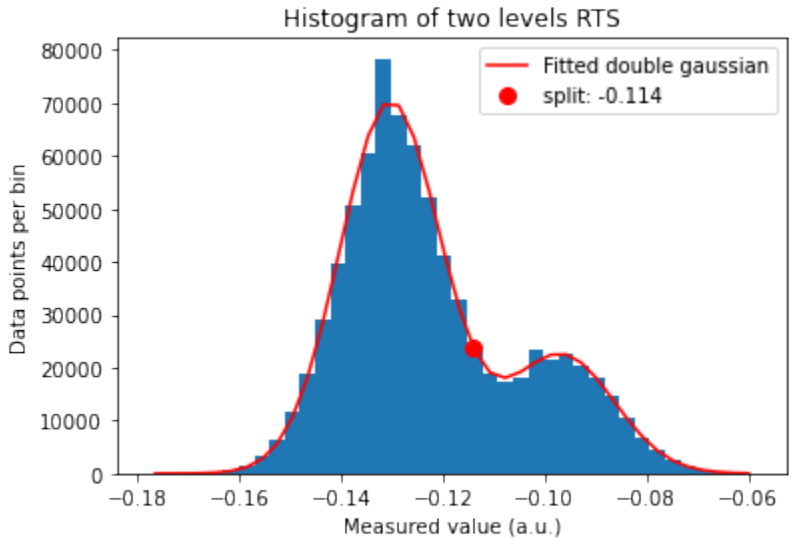
[3]:
```
(94.25667715824251,
 72.09803029068505,
 {'sampling rate': 7812630.000000018,
  'fit parameters double gaussian': array([ 7.02695960e+04,  2.24117976e+04,  9.
↪90102748e-03,  1.03982288e-02,
         -1.30233303e-01, -9.66862750e-02]),
  'separations between peaks gaussians': 1.6526234967095543,
  'split between the two levels': -0.1138706321452195,
  'down_segments': {'mean': 9.850112516891924e-06,
   'p50': 5.759904155194844e-06,
   'mean_filtered': 76.95528455284553},
  'up_segments': {'mean': 3.696660340979315e-06,
   'p50': 7.67987220692646e-07,
   'mean_filtered': 28.88063947974529},
  'tunnelrate_down_to_up': 101521.68295388539,
  'tunnelrate_up_to_down': 270514.4394562042,
  'fraction_down': 0.7271498494746974,
  'fraction_up': 0.2728501505253026,
  'fit parameters exp. decay down': [1.9993002376362625,
```

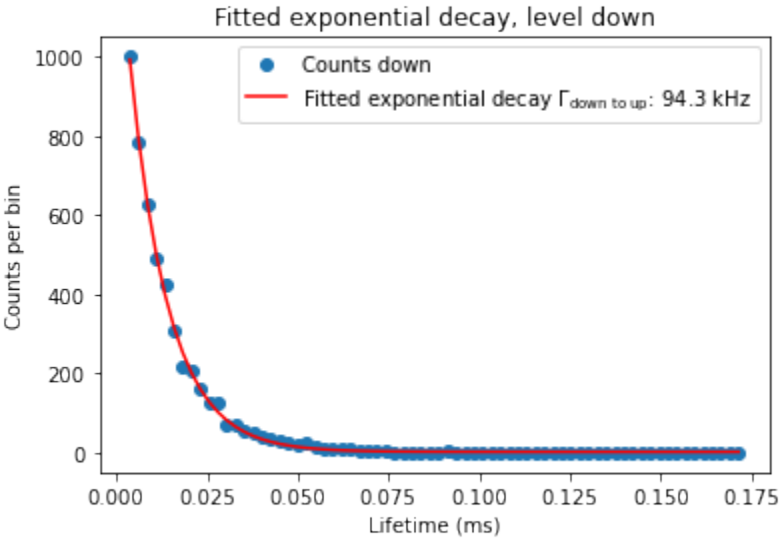
(continues on next page)

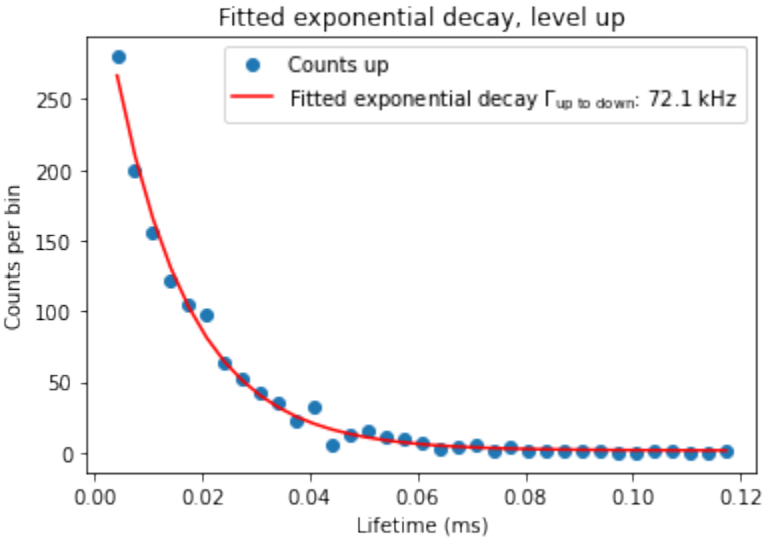
(continued from previous page)

```
  1406.8284745097944,
  94256.67715824251],
 'fit parameters exp. decay up': [1.4958935334632635,
  357.8806097674549,
  72098.03029068504],
 'tunnelrate_down_exponential_fit': 94.25667715824251,
 'tunnelrate_up_exponential_fit': 72.09803029068505})
```









[ ]:

Simulation of Elzerman readout

In this notebook we simulate the measurement traces generated by an electron tunneling on or off a quantum dot, using a continuous-time Markov model. A Markov chain (according to Wikipedia) is a stochastic model describing a sequence of possible events in which the provavility of each event depends only on the state attained in the previous event. For more information: https://www.probabilitycourse.com/chapter11/11_3_1_introduction.php and https://vknight.org/unpeudemath/code/2015/08/01/simulating_continuous_markov_chains.html

This simulation is used to investigate ways to analyse the data of random telegraph signal (RTS) and Elzerman readout. For the latter we also calculate the readout fidelity for our model.

[1]:
```
import warnings
import random
import matplotlib.pyplot as plt
```

(continues on next page)

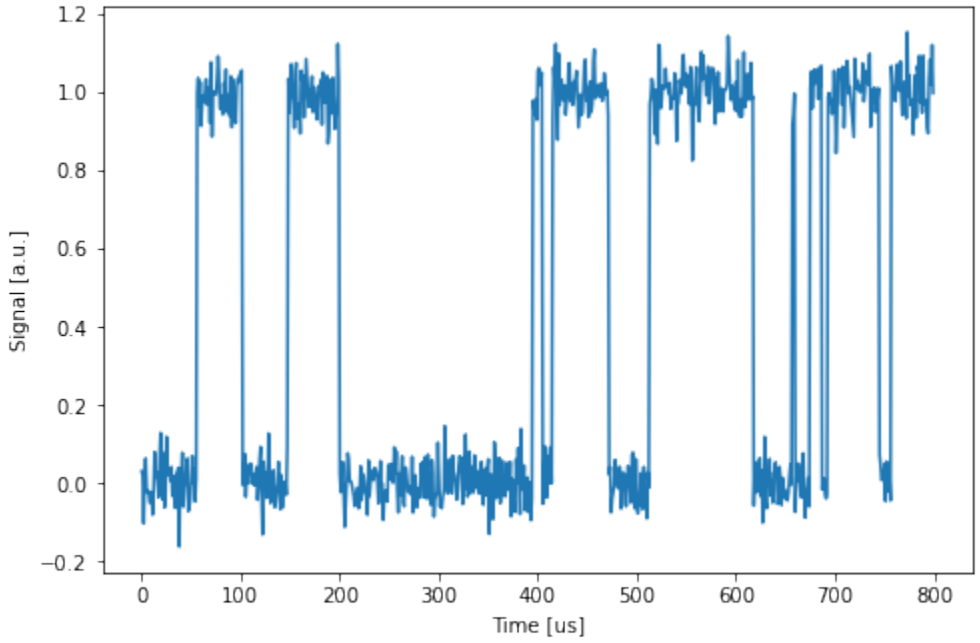
(continued from previous page)

```
import numpy as np
import scipy
import matplotlib
matplotlib.rcParams['figure.figsize']=[1.3*size for size in matplotlib.rcParams[
↪'figure.figsize'] ]

import qtt
from qtt.algorithms.random_telegraph_signal import generate_RTS_signal
from qtt.algorithms.markov_chain import ContinuousTimeMarkovModel, generate_traces
from qtt.algorithms.random_telegraph_signal import tunnelrates_RTS
from qtt.algorithms.random_telegraph_signal import fit_double_gaussian
from qtt.utilities.visualization import plot_vertical_line, plot_double_gaussian_fit,␣
↪plot_single_traces
np.random.seed(1)
```

Random telegraph signal

We start with a model for a random telegraph signal. This model is valid for an electron tunneling into and out of a quantum dot with zero magnetic field. The figure shows a measurement signal which is typical for RTS.

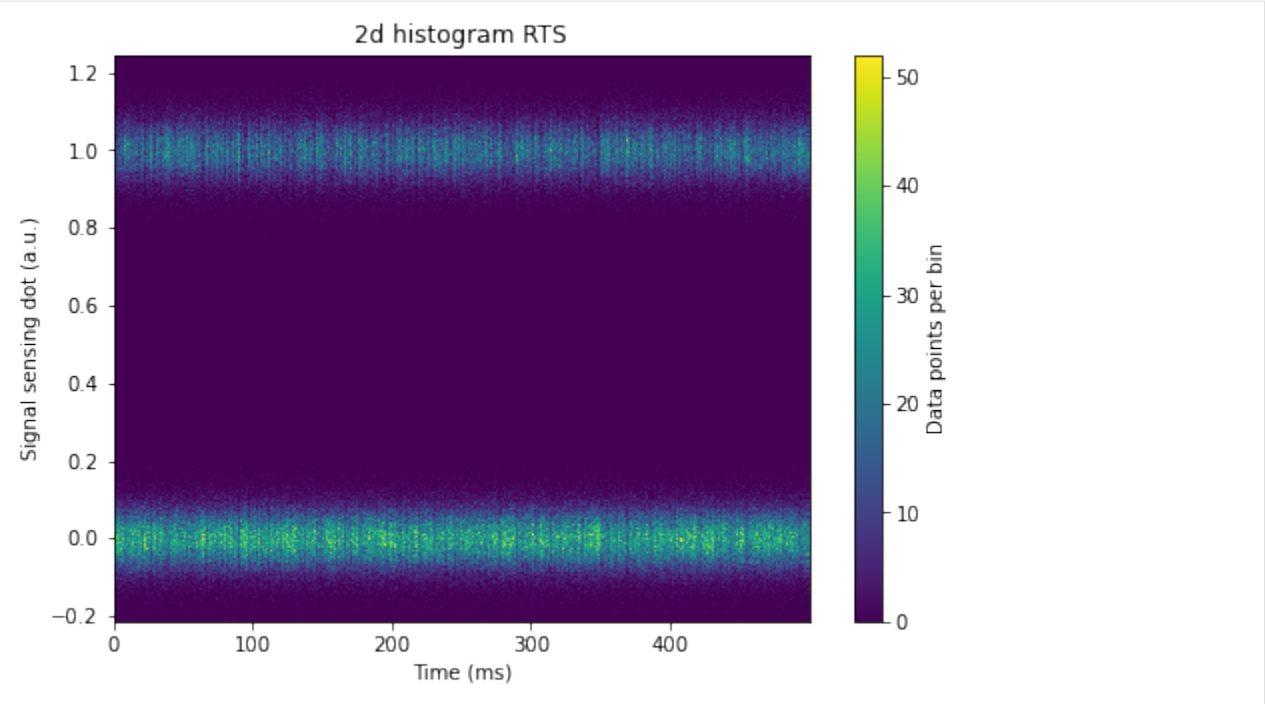
[2]:
```
model_unit = 1e-6 # we work with microseconds as the base unit
rate_up = 15e3 # kHz
rate_down = 25e3 # kHz
rts_model = ContinuousTimeMarkovModel(['zero', 'one'], [rate_up*model_unit,rate_
↪down*model_unit], np.array([[0.,1],[1,0]]) )

rts_data = generate_traces(rts_model, number_of_sequences=1, length=500000, std_
↪gaussian_noise=.05, delta_time=1)

plt.figure(100); plt.clf()
plt.plot(1e6*model_unit*np.arange(800), rts_data.T[0:800,:])
plt.xlabel('Time [us]')
_=plt.ylabel('Signal [a.u.]')
```

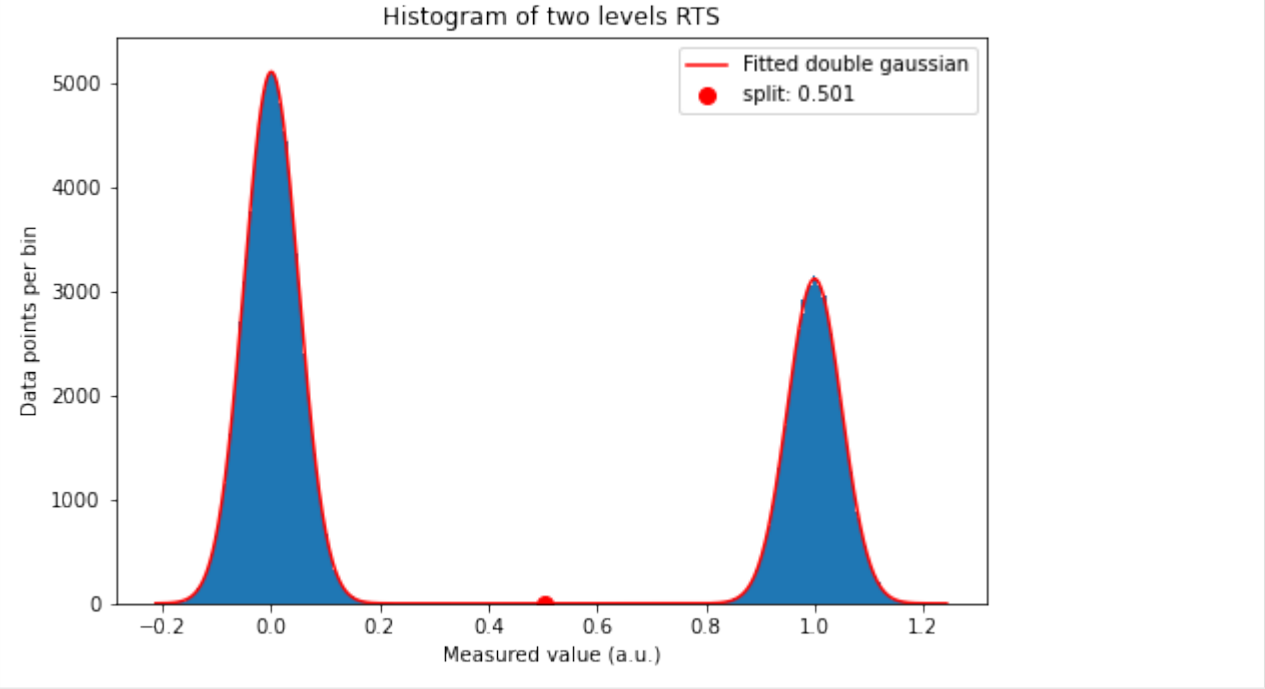


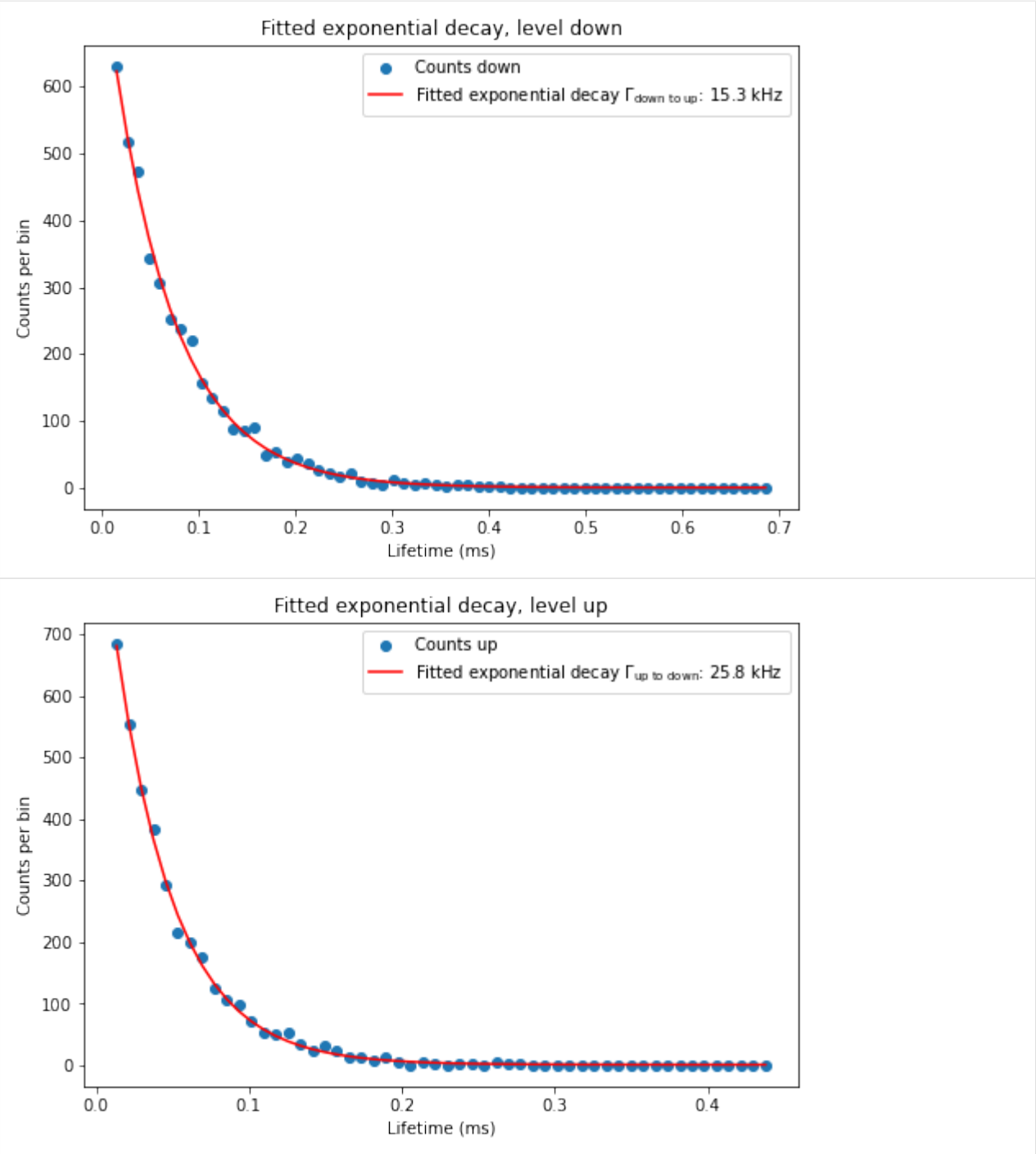
We analyse the signal to determine the tunnel rates and the separation between the two levels.

[3]:
```
samplerate=1e6
tunnelrate_dn, tunnelrate_up, results = tunnelrates_RTS(rts_data.flatten(),␣
↪samplerate=samplerate, min_sep = 1.0, max_sep=2222, min_duration = 10, fig=1,␣
↪verbose=1)
```

```
Fit parameters double gaussian:
 mean down: 0.000 counts, mean up: 1.000 counts, std down: 0.050 counts, std up:0.050␣
↪counts
Separation between peaks gaussians: 10.011 std
Split between two levels: 0.501
Tunnel rate down to up: 15.3 kHz
Tunnel rate up to down: 25.8 kHz
```


2d histogram RTS
Signal sensing dot (a.u.)
Time (ms)
Data points per bin
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
100
200
300
400
50
40
30
20
10
0


Histogram of two levels RTS
Fitted double gaussian
split: 0.501
Data points per bin
Measured value (a.u.)
5000
4000
3000
2000
1000
0
−0.2
0.0
0.2
0.4
0.6
0.8
1.0
1.2


Fitted exponential decay, level down
Counts down
Fitted exponential decay $\Gamma_{\text{down to up}}$: 15.3 kHz
Counts per bin
Lifetime (ms)
Fitted exponential decay, level up
Counts up
Fitted exponential decay $\Gamma_{\text{up to down}}$: 25.8 kHz
Counts per bin
Lifetime (ms)

More efficient calculation of tunnel rates

The tunnel rates are calculated by fitting an exponential to a histogram of segment lengths. The mean segment length contains roughly the same information. Fitting the exponantial is more accurate when the tunnelrate approximates the sampling rate. Calculating the mean segment length is more robust for low number of datapoints.

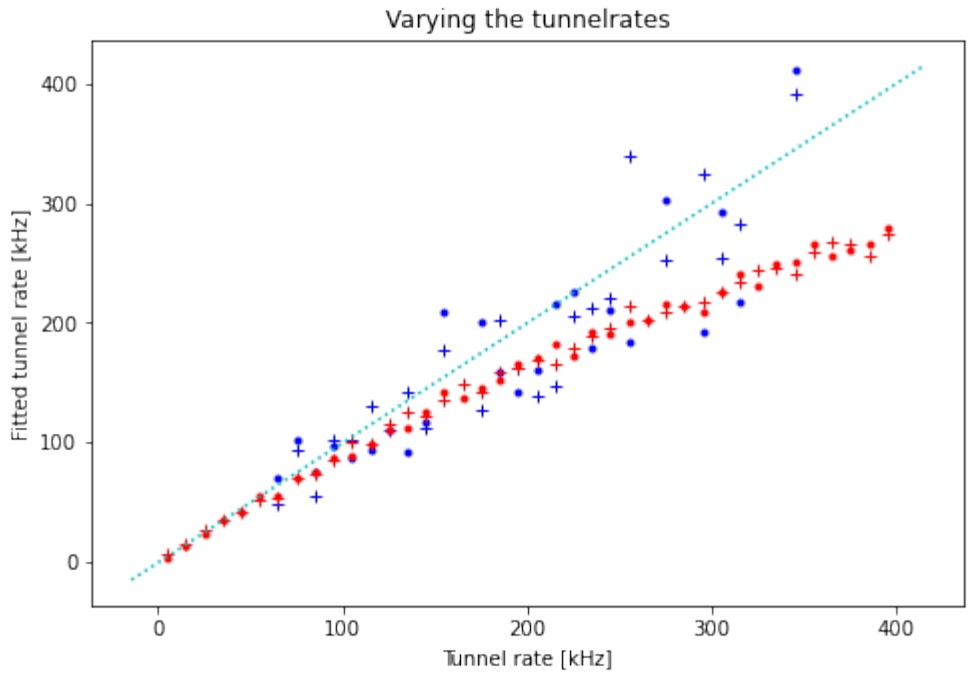
Comparing the performance of the two analysis methods, varying the tunnelrates and lowering the number of datapoints. Blue: fitted tunnelrate, red: 1 / mean segment length.

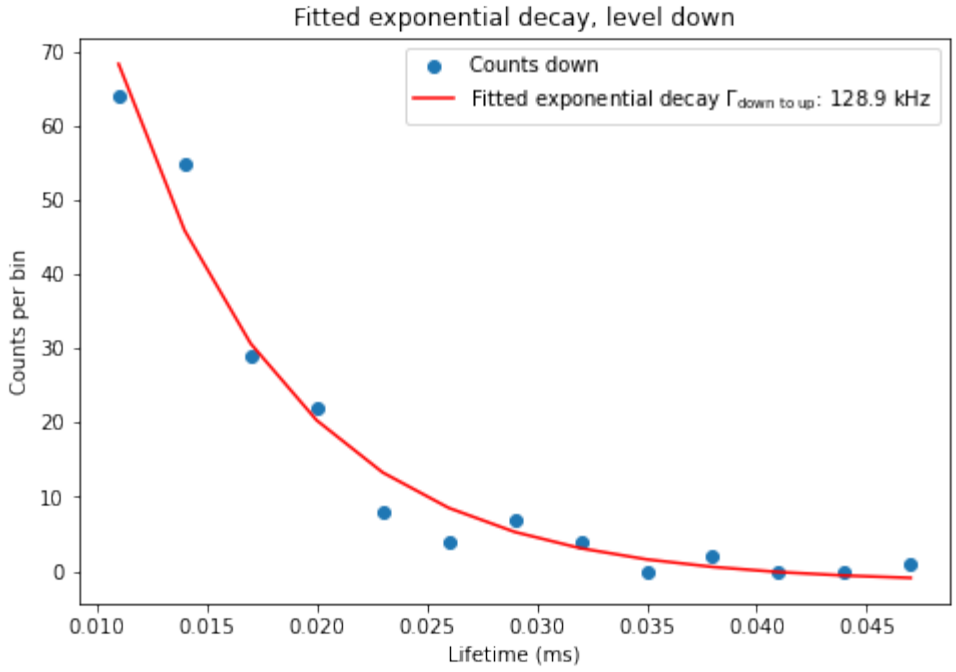
[4]:
```
def generate_RTS_results(tunnel_rate, model_unit, length):
    rts_model = ContinuousTimeMarkovModel(['down', 'up'], [tunnel_rate*model_unit,
↪tunnel_rate*model_unit], np.array([[0.,1],[1,0]]) )

    rtsdata = generate_traces(rts_model, number_of_sequences=1, length=10000, std_
↪gaussian_noise=.15)[0]
    with warnings.catch_warnings():
        warnings.filterwarnings("ignore")
        tunnelrate_dn, tunnelrate_up, results = tunnelrates_RTS(rtsdata,␣
↪samplerate=samplerate, min_sep = 1.0, max_sep=2222, min_duration = 10, num_bins =␣
↪40, fig=0, verbose=0)
    return tunnelrate_dn, tunnelrate_up, results

def plot_RTS_results(results, model_unit, fig):
    tunnelrate_dn = results['tunnelrate_down_exponential_fit']
    tunnelrate_up = results['tunnelrate_up_exponential_fit']
    plt.figure(fig)
    if tunnelrate_dn is not None:
        plt.plot(tunnel_rate/1e3, tunnelrate_dn, '.b')
        plt.plot(tunnel_rate/1e3, tunnelrate_up, '+b')

    x_factor = 1e-3
    y_factor = (1./model_unit)*x_factor
    plt.plot(tunnel_rate*x_factor, y_factor/(samplerate*results['down_segments']['mean
↪']), '.r')
    plt.plot(tunnel_rate*x_factor, y_factor/(samplerate*results['up_segments']['mean
↪']), '+r')

samplerate = 1e6

plt.figure(1002); plt.clf(); plt.xlabel('Tunnel rate [kHz]'); plt.ylabel('Fitted␣
↪tunnel rate [kHz]')
for jj, tunnel_rate in enumerate(np.arange(5, 405, 10)*1e3): #varying the tunnelrate␣
↪from 5 to 400 kHz
    tunnelrate_dn, tunnelrate_up, results = generate_RTS_results(tunnel_rate, model_
↪unit, length = 155000)
    plot_RTS_results(results, model_unit, fig = 1002)

plt.figure(1002)
qtt.pgeometry.plot2Dline([1,-1,0], ':c', label='')
plt.title('Varying the tunnelrates')

plt.figure(1010); plt.clf(); plt.xlabel('Tunnel rate [kHz]'); plt.ylabel('Fitted␣
↪tunnel rate [kHz]')
for jj, tunnel_rate in enumerate(np.arange(5, 150, 5)*1e3):
    tunnelrate_dn, tunnelrate_up, results = generate_RTS_results(tunnel_rate, model_
↪unit, length = 10000)
    plot_RTS_results(results, model_unit, fig = 1010)
```

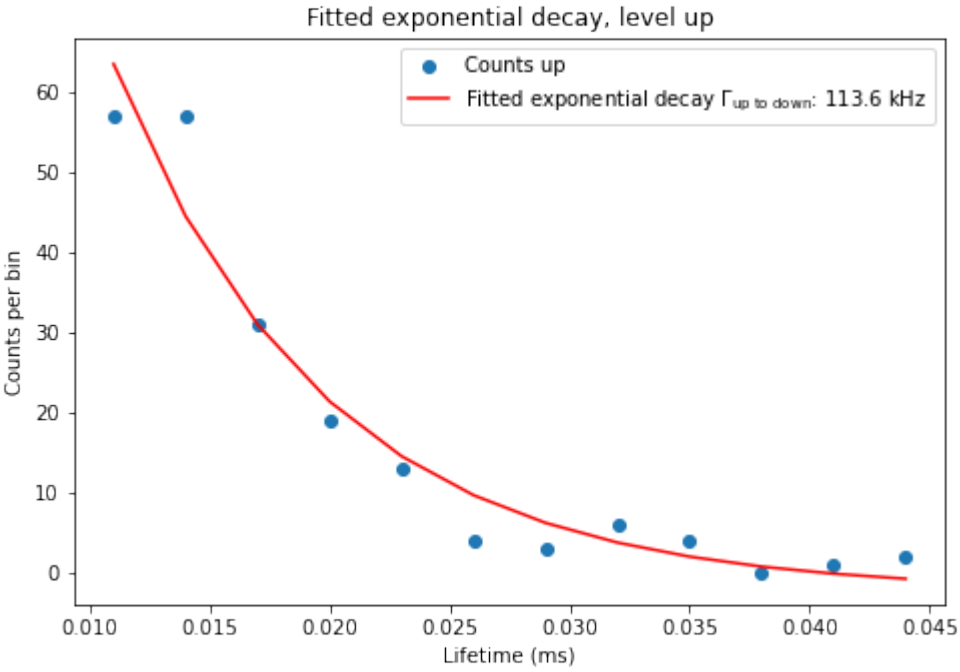
(continues on next page)

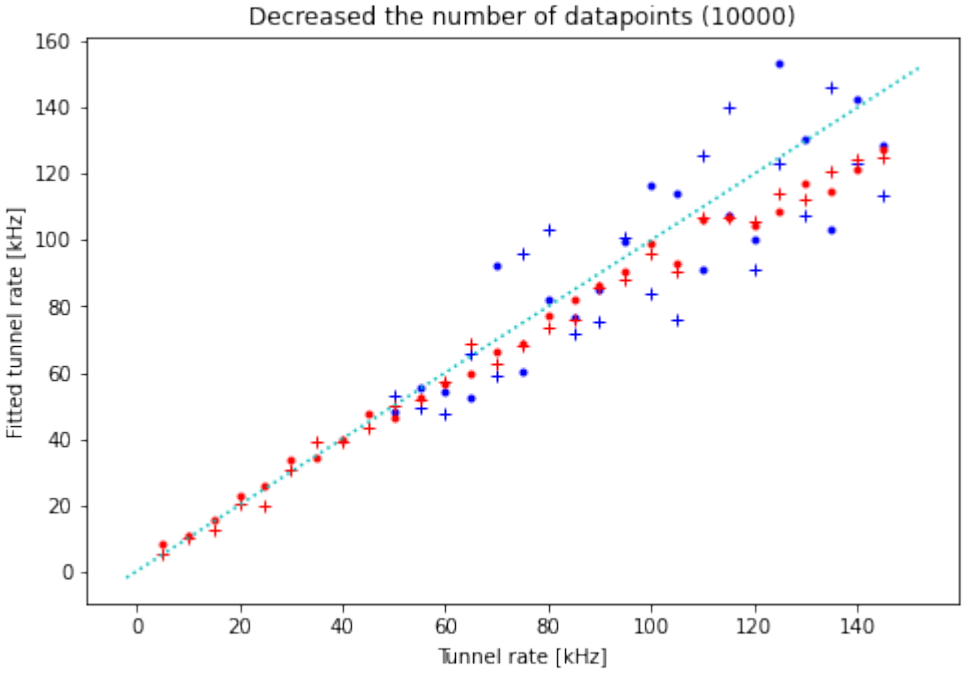
(continued from previous page)

```
plt.figure(1010)
qtt.pgeometry.plot2Dline([1,-1,0], ':c', label='')
_ = plt.title('Decreased the number of datapoints (10000)')
```






Fitted exponential decay, level up
Counts up
Fitted exponential decay $\Gamma_{\text{up to down}}$: 113.6 kHz
Counts per bin
Lifetime (ms)
0
10
20
30
40
50
60
0.010
0.015
0.020
0.025
0.030
0.035
0.040
0.045


Decreased the number of datapoints (10000)
Fitted tunnel rate [kHz]
Tunnel rate [kHz]
0
20
40
60
80
100
120
140
160

Elzerman readout

We model Elzerman readout with a Markov model with three states: empty, dot filled with a spin-up electron, dot filled with a spin-dot electron. The transitions possible are tunneling of a spin-up or spin-down electron out of the system, tunneling from an electron into the down state and decay of spin-up to spin-down (T1 decay).

[5]:
```
model_unit = 1e-6 # we work with microseconds as the baseunit

gamma_up_out = 10e3
gamma_down_out = .1e3
gamma_empty_down = 2e3
T1 = 3e-3 # [s]
gamma_up_down = 1./T1

G = np.array( [[-gamma_down_out, 0, gamma_down_out], [gamma_up_down, -(gamma_up_
↪down+gamma_up_out), gamma_up_out], [gamma_empty_down, 0, -gamma_empty_down]] ).T
holding_parameters = -np.diag(G).reshape( (-1,1))
jump_chain= (1./holding_parameters.T)*G
jump_chain[np.diag_indices(G.shape[0])]=0

elzerman_model = ContinuousTimeMarkovModel(['spin-down', 'spin-up', 'empty'], holding_
↪parameters*model_unit, jump_chain )
print(elzerman_model)
```

```
ContinuousTimeMarkovModel(id=0x23e1ad1f6d8, states=['spin-down', 'spin-up', 'empty'],␣
↪generator=[[-1.00000000e-04  3.33333333e-04  2.00000000e-03]
 [ 0.00000000e+00 -1.03333333e-02  0.00000000e+00]
 [ 1.00000000e-04  1.00000000e-02 -2.00000000e-03]])
```

We generate a number of traces with the model. We shown the generated states (first plot) and the corresponding signal of the charge sensor (second plot). We calculate the signal of the charge sensor from the states with the `sensor_values` map and add noise to the signal. This gives us the opportunity to compare the states as simulated (dot empty, dot occupied with spin-up electron, dot occupied with spin-down electron), with the corresponding measurement traces.

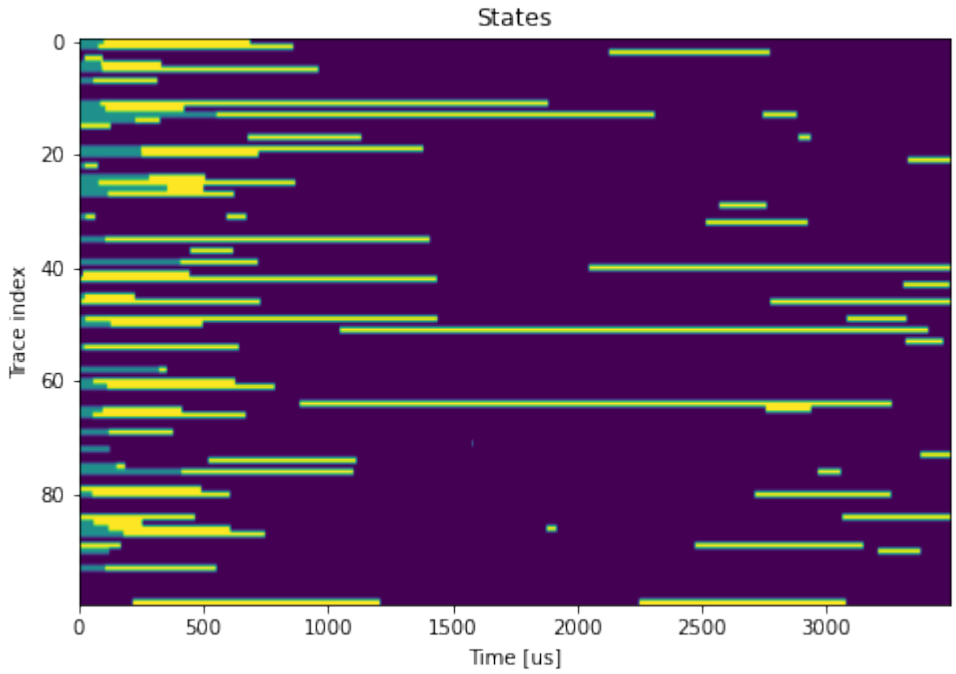
[6]:
```
sensor_values = {'spin-down': 0, 'spin-up':0, 'empty': 1}
```

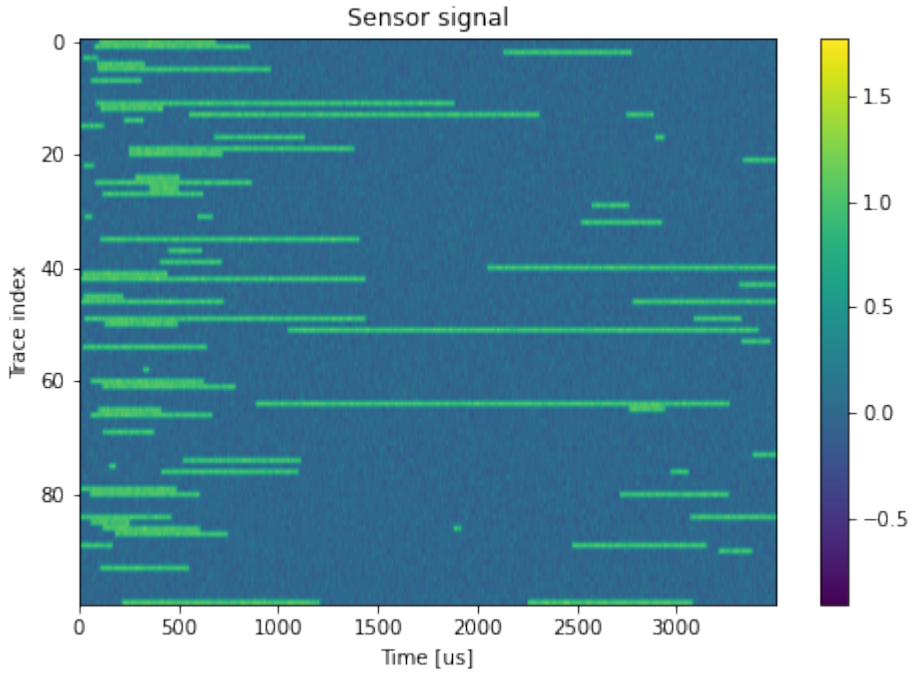
[7]:
```
def generate_model_sequences(elzerman_model, sensor_values=sensor_values, std_
↪gaussian_noise = 0.2,
                                 number_of_samples=3500, number_of_traces=1000,␣
↪initial_state=[.5, .5, 0] ):
    state_traces = generate_traces(elzerman_model, std_gaussian_noise=0,␣
↪length=number_of_samples, initial_state=initial_state, number_of_sequences=number_
↪of_traces)

    state_mapping=np.array([ sensor_values.get(state, np.NaN) for state in elzerman_
↪model.states])
    traces = state_traces.copy()
    traces=np.array(state_mapping)[traces]
    if std_gaussian_noise != 0:
        traces = traces + np.random.normal(0, std_gaussian_noise, traces.size).
↪reshape(traces.shape)

    initial_states=state_traces[:,0]
    return traces, state_traces, initial_states

traces, state_traces, initial_states = generate_model_sequences(elzerman_model,␣
↪number_of_traces=300)
```

(continues on next page)

(continued from previous page)

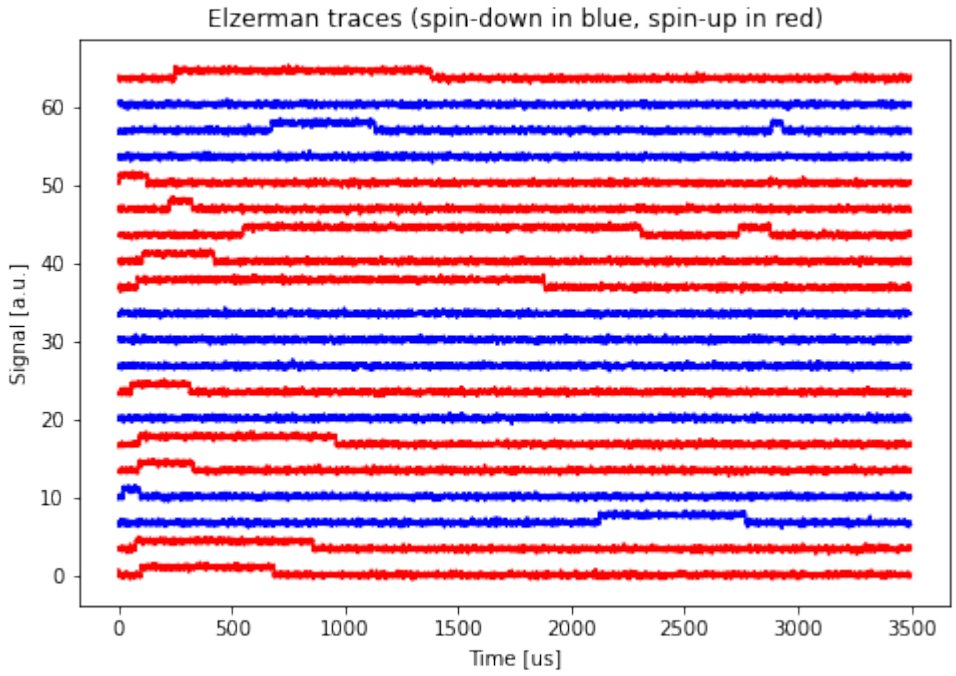
```
max_number_traces=100

plt.figure();
plt.imshow(state_traces[:max_number_traces,:])
plt.axis('tight')
plt.xlabel('Time [us]'); plt.ylabel('Trace index')
plt.title('States')

plt.figure();
plt.imshow(traces[:max_number_traces,:])
plt.axis('tight')
plt.xlabel('Time [us]'); plt.ylabel('Trace index')
plt.title('Sensor signal')
_=plt.colorbar()
```





We can also plot the individual traces. For reference we color the traces according to the initial-state of the traces.

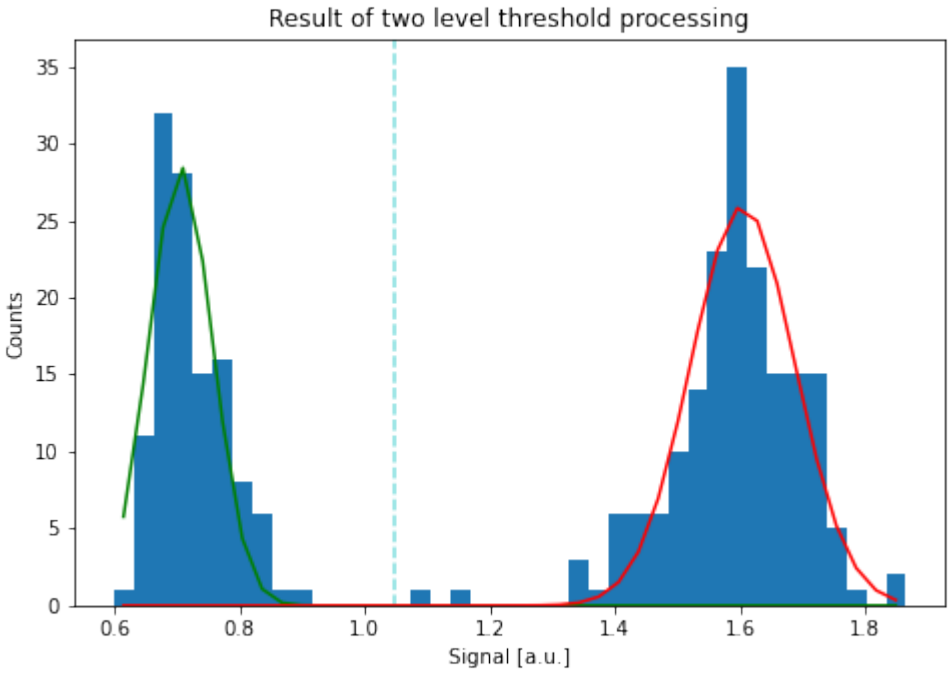
[8]:
```
plot_single_traces(traces, trace_color=initial_states, maximum_number_of_traces=20)

plt.xlabel('Time [us]')
plt.ylabel('Signal [a.u.]')
_=plt.title('Elzerman traces (spin-down in blue, spin-up in red)')
```



Determination of readout fidelity with max of trace

For each trace we termine the maximum value. We then label the traces according to whether this maximum value exceeds a given threshold.

[9]:
```
from qtt.algorithms.random_telegraph_signal import two_level_threshold, plot_two_
↪level_threshold

elzermann_threshold_result = two_level_threshold(np.max(traces, axis=1))
plot_two_level_threshold(elzermann_threshold_result)
```



For a given readout threshold and readout length we can determine the fidelity by counting the number of traces that is correctly labelled as either up or down.

[10]:
```
def calculate_fidelity(traces, initial_states, readout_threshold, readout_length):
    traces_smooth = scipy.ndimage.filters.convolve(traces, np.array([[1,1,1.]])/3,␣
↪mode='nearest')
    measured_states = np.max(traces_smooth[:, :readout_length], axis=1)>readout_
↪threshold
    F= np.sum(initial_states==measured_states) /   measured_states.size
    return F

readout_threshold=elzermann_threshold_result['signal_threshold']

F=calculate_fidelity(traces, initial_states, readout_threshold, 800)
print('readout fidelity F %.2f' % F)
```

```
readout fidelity F 0.93
```

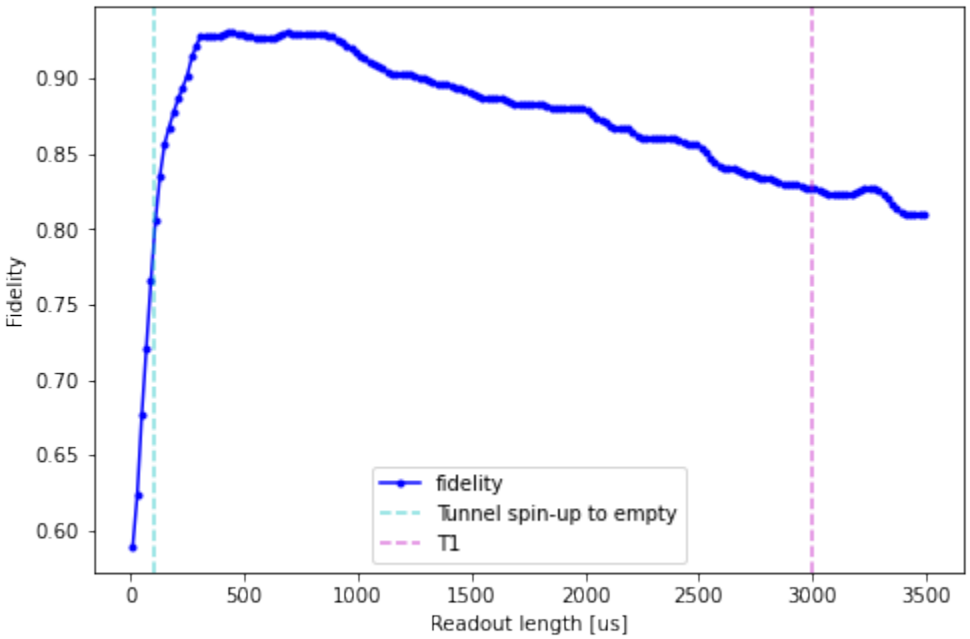
The optimal fidelity is a trade-off between longer measurement (so that a spin-up state can tunnel out) and shorter measurement (no accidental tunneling out of the ground state, or decay from spin up to spin down).

[11]:
```
readout_lengths=np.arange(10, traces.shape[1], 20)
fidelities=np.zeros(readout_lengths.size)
```

(continues on next page)

(continued from previous page)

```
for ii, readout_length in enumerate(readout_lengths):
    fidelities[ii]=calculate_fidelity(traces, initial_states, readout_threshold,␣
↪readout_length)

fidelities=qtt.algorithms.generic.smoothImage(fidelities)
plt.figure(1000);
plt.clf()
plt.plot(readout_lengths, fidelities,'.-b', label='fidelity')
plt.xlabel('Readout length [us]')
_=plt.ylabel('Fidelity')

plot_vertical_line(1.e6/gamma_up_out, label = 'Tunnel spin-up to empty')
plot_vertical_line(1.e6/gamma_up_down, label = 'T1', color='m')
_=plt.legend(numpoints=1)
```



Pauli spin blockade or readout with a resonator

Taking the maximum of the trace has the disadvantage that a lot of information from the trace is discarded. An alternative method is to take the mean of the trace (over the readout period). This does not work for Elzerman readout, as the length of the blips can be either short or long with respect to the measurement interval.

For Pauli spin-blockade (PSB) or resonator spin readout (Rapid high-fidelity gate-based spin read-out in silicon) we can average over the traces, as the signal is different for both spin-up and spin-down directly after pulsing to the measurement point.

[12]:
```
model_unit = 1e-6 # we work with microseconds as the baseunit
T1 = 3e-3 # [s]
gamma_up_down = 1./T1  # Hz
gamma_down_up = 1e-5 # Hz

psb_model = ContinuousTimeMarkovModel(['singlet', 'triplet'], [gamma_up_down*model_
↪unit,gamma_down_up*model_unit], np.array([[0.,1],[1,0]]) )
```

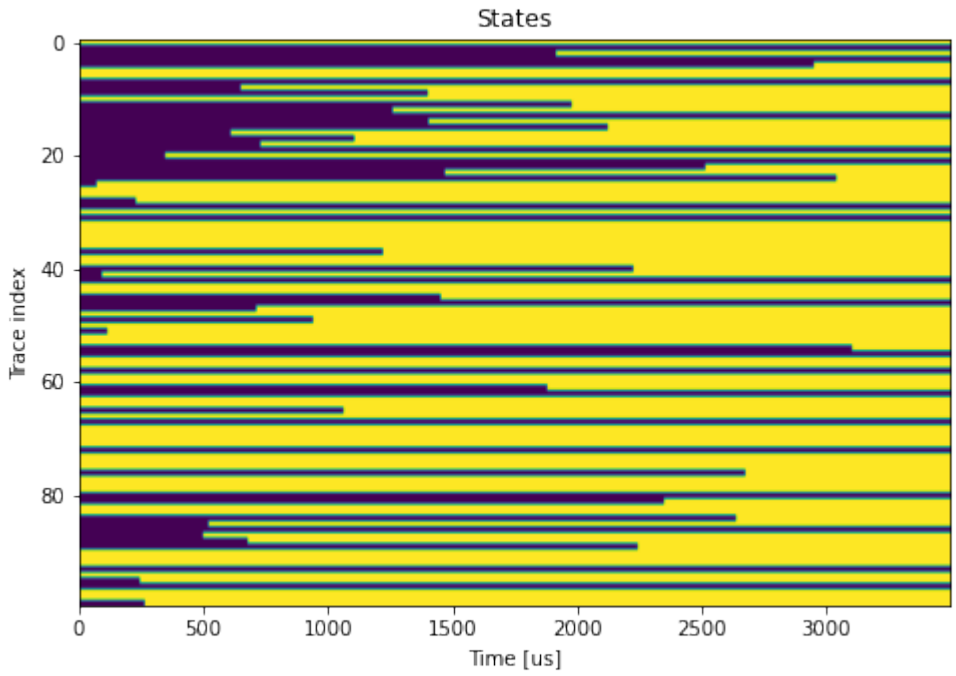
(continues on next page)

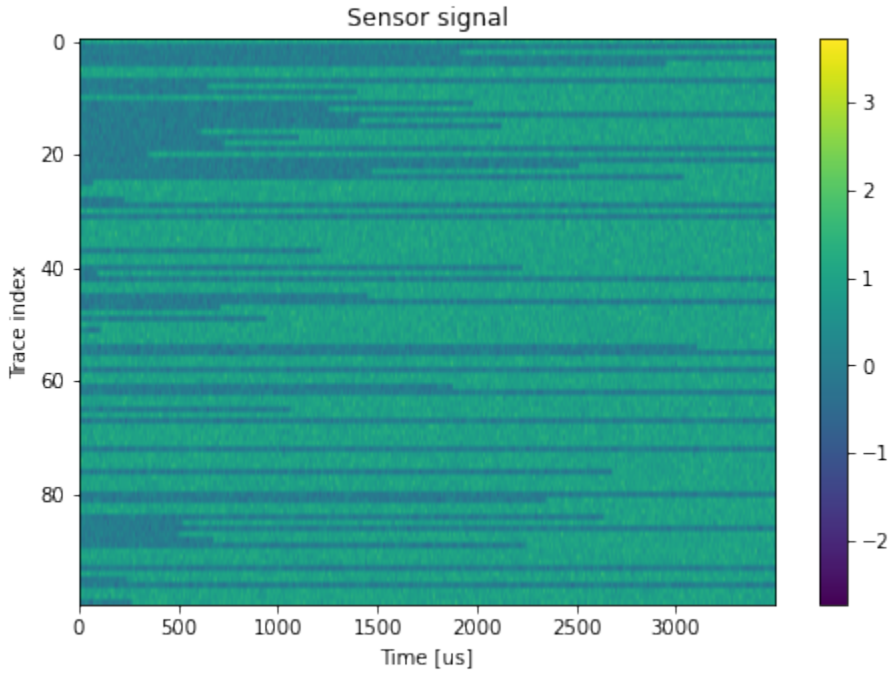
(continued from previous page)

```
print(psb_model)
```

```
ContinuousTimeMarkovModel(id=0x23e1af1ae10, states=['singlet', 'triplet'],␣
↪generator=[[-3.33333333e-04  1.00000000e-11]
 [ 3.33333333e-04 -1.00000000e-11]])
```

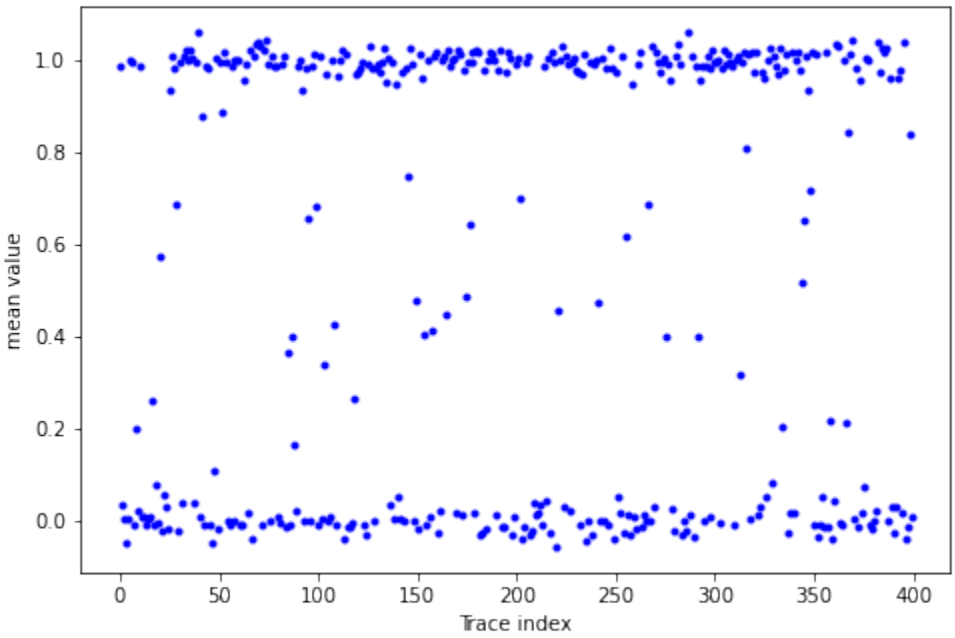
[13]:
```
sensor_values = {'singlet': 0, 'triplet':1}
traces, state_traces, initial_states = generate_model_sequences(psb_model, sensor_
↪values=sensor_values,
                                                std_gaussian_noise=.6, number_of_traces=400,␣
↪initial_state=[0.5,0.5])
max_number_traces=100

plt.figure();
plt.imshow(state_traces[:max_number_traces,:])
plt.axis('tight')
plt.xlabel('Time [us]'); plt.ylabel('Trace index')
plt.title('States')

plt.figure();
plt.imshow(traces[:max_number_traces,:])
plt.axis('tight')
plt.xlabel('Time [us]'); plt.ylabel('Trace index')
plt.title('Sensor signal')
_=plt.colorbar()
```

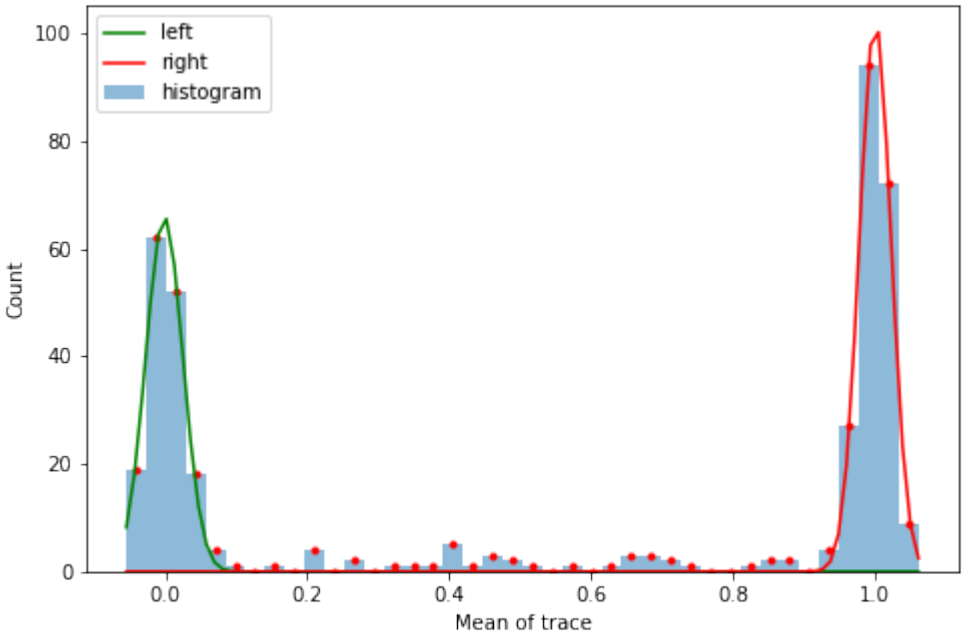




[14]:
```
readout_length = 800
trace_means = np.mean(traces[:, :readout_length], axis=1)

number_of_bins = 40
counts, bins = np.histogram(trace_means, bins=number_of_bins)
bincentres = np.array([(bins[i] + bins[i + 1]) / 2 for i in range(0, len(bins) - 1)])
par_fit, result_dict = fit_double_gaussian(bincentres, counts)
print('fitted parameters : %s' % (par_fit,))

plt.figure(50); plt.clf()
plt.plot(trace_means, '.b')
plt.xlabel('Trace index'); plt.ylabel('mean value')

plt.figure(100); plt.clf()
plt.bar(bincentres, counts, width=bincentres[1]-bincentres[0], alpha=.5, label=
↪'histogram')
_=plt.plot(bincentres, counts, '.r')
plt.xlabel('Mean of trace'); plt.ylabel('Count')

signal_range=np.linspace(trace_means.min(), trace_means.max(), 100 )

plot_double_gaussian_fit(result_dict, signal_range)
_ =plt.legend()
```

```
fitted parameters : [ 6.59287840e+01  1.02317930e+02  2.62157463e-02  2.23601538e-02
 -1.54710375e-03  1.00116972e+00]
```





[15]:
```
psb_threshold = 0.5
```

[16]:
```
def calculate_fidelity_mean(traces, initial_states, readout_threshold, readout_
↪length):
    trace_means = np.mean(traces[:, :readout_length], axis=1)
    measured_states = trace_means > readout_threshold
    F= np.sum(initial_states==measured_states) /   measured_states.size
    return F

F=calculate_fidelity_mean(traces, initial_states, psb_threshold, readout_length = 800)
```

(continues on next page)

(continued from previous page)

```
print('readout fidelity F %.2f' % F)
```

```
readout fidelity F 0.95
```

From the fitted double Gaussian the readout fidelity can also be determined (for details including a model where the T1 decay is taken into account, see "Rapid Single-Shot Measurement of a Singlet-Triplet Qubit", Barthel et al., 2009, https://arxiv.org/abs/0902.0227). This is usefull for real measurement data where the true spin-states of the traces are unknown.

[ ]:

Fitting a Fermi distribution to a quantum dot addition line

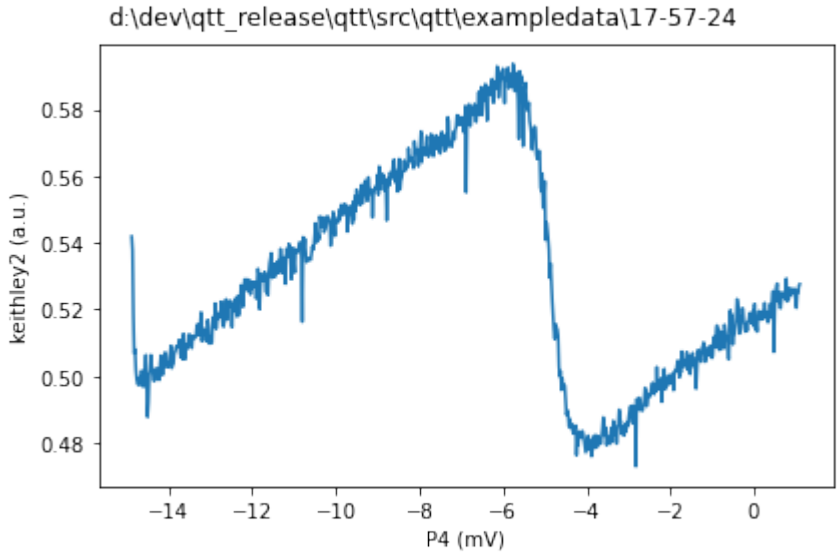
A quantum dot addition line shows the energy boundary for adding a new electron from a reservoir into the quantum dot system. A 1D trace measurement across the addition line can be used to extract the electron temperature of the reservoir, by fitting a Fermi distribution to the signal. Note that an accurate electron temperature measurement requires accurate gate-to-dot lever arms (this example shows how to obtain them), and negligible lifetime broadening (i.e. low dot-reservoir coupling compared to temperature).

This example uses the core function `fitFermiLinear` from `qtt.algorithms.fitting`.

[1]:
```
import os
import numpy as np
import scipy

import qcodes
from qcodes.plots.qcmatplotlib import MatPlot
from qcodes.data.data_array import DataArray
import matplotlib.pyplot as plt
%matplotlib inline

import qtt
from qtt.algorithms.fitting import FermiLinear, fitFermiLinear
from qtt.data import load_example_dataset
```

Load and plot a sample dataset of a 1D trace across an quantum dot addition line

[2]:
```
dataset = load_example_dataset('addition_line_scan')
_=MatPlot(dataset.default_parameter_array())
```



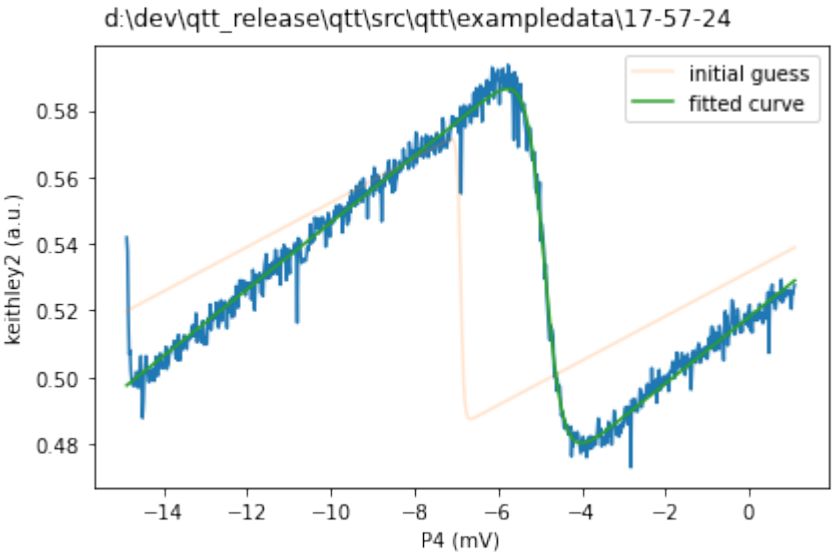
Fit Fermi function to data

[3]:
```
y_array = dataset.default_parameter_array()
setarray = y_array.set_arrays[0]
xdata = np.array(setarray)
ydata = np.array(y_array)
kb = scipy.constants.physical_constants['Boltzmann constant in eV/K'][0]*1e6  # [ueV/
↪K]
la = 100 # [ueV/mV] gate-to-dot lever arm

# fit
pp = fitFermiLinear(xdata, ydata, lever_arm=la/kb, verbose=1, fig=None)
fitted_parameters = pp[0]
initial_parameters = pp[1]['initial_parameters']

y0 = FermiLinear(xdata, *list(initial_parameters))
y = FermiLinear(xdata, *list(fitted_parameters))

print('Estimated electron temperature: %.1f mK' % (1e3 * fitted_parameters[4]))
```

```
Estimated electron temperature: 253.9 mK
```

Plot the fit to check the accuracy

[4]:
```
p = MatPlot(dataset.default_parameter_array())
v0 = DataArray(name='initial', label='initial guess', preset_data=y0,  set_
↪arrays=(setarray,))
p.add(v0, alpha=.2, label='initial guess')
v = DataArray(name='fitted', label='fitted curve', preset_data=y,  set_
↪arrays=(setarray,))
p.add(v, label='fitted curve')
_=plt.legend()
```



[ ]:

Fitting the data from a Ramsey experiment

In this notebook we analyse data from a Ramsey experiment. Using the method and data from:

Watson, T. F., Philips, S. G. J., Kawakami, E., Ward, D. R., Scarlino, P., Veldhorst, M., ... Vandersypen, L. M. K. (2018). A programmable two-qubit quantum processor in silicon. Nature, 555(7698), 633–637. https://doi.org/10.1038/nature25766

The signal that results from a Ramsey experiment oscillates at a frequency corresponding to the difference between the qubit frequency and the MW source frequency. Therefore, it can be used to accurately calibrate the MW source to be on-resonance with the qubit. Additionally, the decay time of the Ramsey signal corresponds to the free-induction decay or T2* of the qubit.

This example takes a Ramsey dataset and uses the core function `qtt.algorithms.functions.fit_gauss_ramsey` to fit it, returning the frequency and decay of the signal.

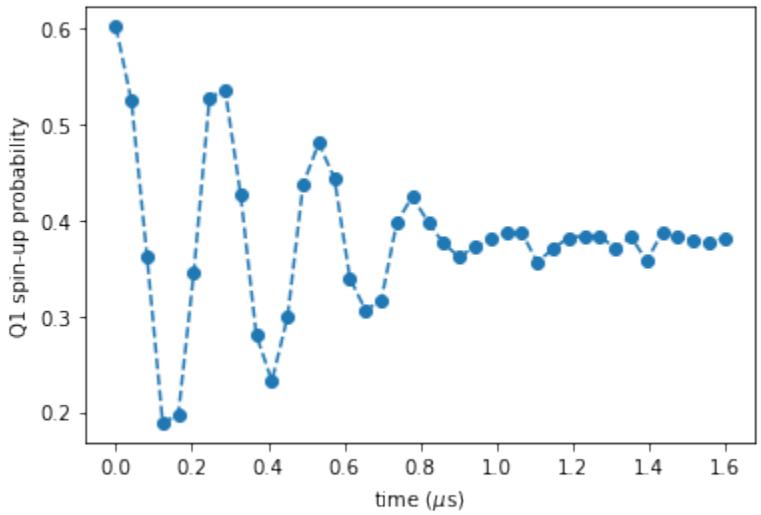
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from qtt.algorithms.functions import gauss_ramsey, fit_gauss_ramsey, plot_gauss_
↪ramsey_fit
```

Test data, based on the data acquired by Watson et all.

[2]:
```
y_data = np.array([0.6019, 0.5242, 0.3619, 0.1888, 0.1969, 0.3461, 0.5276, 0.5361,
       0.4261, 0.28  , 0.2323, 0.2992, 0.4373, 0.4803, 0.4438, 0.3392,
       0.3061, 0.3161, 0.3976, 0.4246, 0.398 , 0.3757, 0.3615, 0.3723,
       0.3803, 0.3873, 0.3873, 0.3561, 0.37  , 0.3819, 0.3834, 0.3838,
       0.37  , 0.383 , 0.3573, 0.3869, 0.3838, 0.3792, 0.3757, 0.3815])

total_wait_time = 1.6e-6
x_data = np.linspace(0, total_wait_time, len(y_data))
```

Plotting the data:

[3]:
```
plt.figure()
plt.plot(x_data * 1e6,y_data, '--o')
plt.xlabel(r'time ($\mu$s)')
plt.ylabel('Q1 spin-up probability')
plt.show()
```

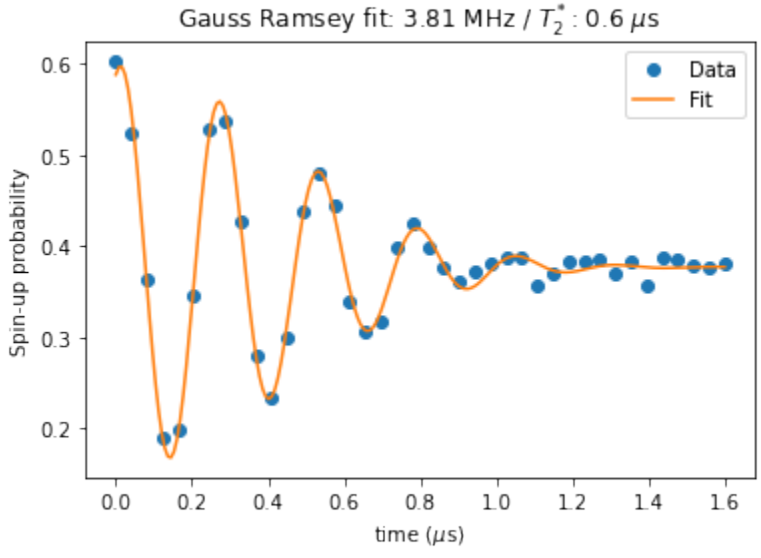


Applying the `fit_gauss_ramsey` function to fit the data:

[4]:
```
fit_parameters, _ = fit_gauss_ramsey(x_data, y_data)
freq_fit = abs(fit_parameters[2] * 1e-6)
t2star_fit = abs(fit_parameters[1] * 1e6)
```

Plotting the data and the fit:

[5]:
```
plot_gauss_ramsey_fit(x_data, y_data, fit_parameters, fig=1)
```



Note that for the Ramsey experiment, the frequency is of the MW source is offset by 4 MHz. Therefore, this experiment shows that the qubit was off-resonance from the source by -200 kHz.

[ ]:

Determining the pinch-off value of a gate

Pieter Eendebak pieter.eendebak@tno.nl

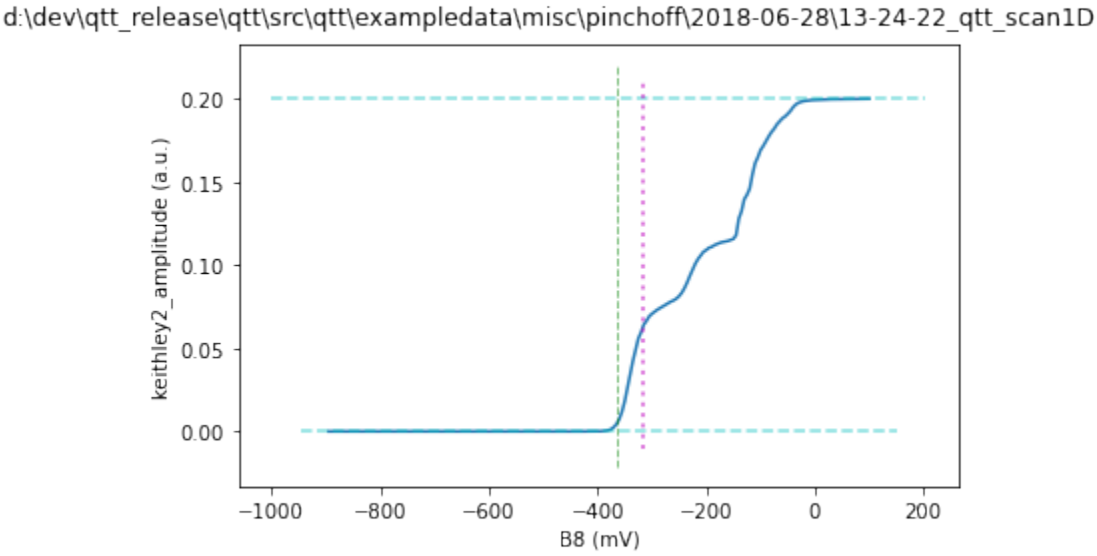
We determine the pinch-off value of a 1D gate scan by means of the function `analyseGateSweep`.

[1]:
```
import os
import pprint
from qtt.data import load_example_dataset
from qtt.algorithms.gatesweep import analyseGateSweep, plot_pinchoff

dataset=load_example_dataset('pinchoff_scan_barrier_gate')
```

Determine the pinchoff value and plot the analysis. The `analyseGateSweep` function will return a dictionary containing (among other things) the pinch-off point, the y-value at pinch-off, the y values for the saturation lines for the curve, both high and low and the name of the swept gate.

[2]:
```
result=analyseGateSweep(dd=dataset)
plot_pinchoff(result, ds=dataset, fig=10)
```

```
analyseGateSweep: pinch-off point -315.000, value 0.060
```



[3]:
```
pprint.pprint(result)
```

```
{'_debug': {'midpoint1': -225.0, 'midpoint2': -315.0},
 '_pinchvalueX': -275.0,
 'dataset': 'd:\\dev\\qtt_release\\qtt\\src\\qtt\\exampledata\\pinchoff_scan_barrier_
↪gate',
 'description': 'pinchoff analysis',
 'goodgate': True,
 'highvalue': 0.19978344039999998,
 'lowvalue': -0.00018635139222000001,
 'midpoint': -315.0,
 'midvalue': 0.059804586145445995,
```

(continues on next page)

(continued from previous page)

```
 'pinchoff_point': -365.0,
 'pinchoff_value': 0.181798285,
 'pinchvalue': 'use pinchoff_point instead',
 'type': 'gatesweep',
 'xlabel': 'Sweep B8 [mV]'}
```

[ ]:

Example of extracting the lever arm and the charging energy from bias triangles and addition lines

Authors: Anne-Marije Zwerver and Pieter Eendebak

More details on non-equilibrium charge stability measurements can be found in https://doi.org/10.1103/RevModPhys.75.1 (section B) and https://doi.org/10.1063/1.3640236

The core functions used in the example are `perpLineIntersect`, `lever_arm` and `E_charging`. Input needed for the code are the non-equilibrium charge stability diagrams of a 1,0-0,1 interdot transition for the lever arm, and a charge stability diagram with the 0-1 and 1-2 charge transitions.

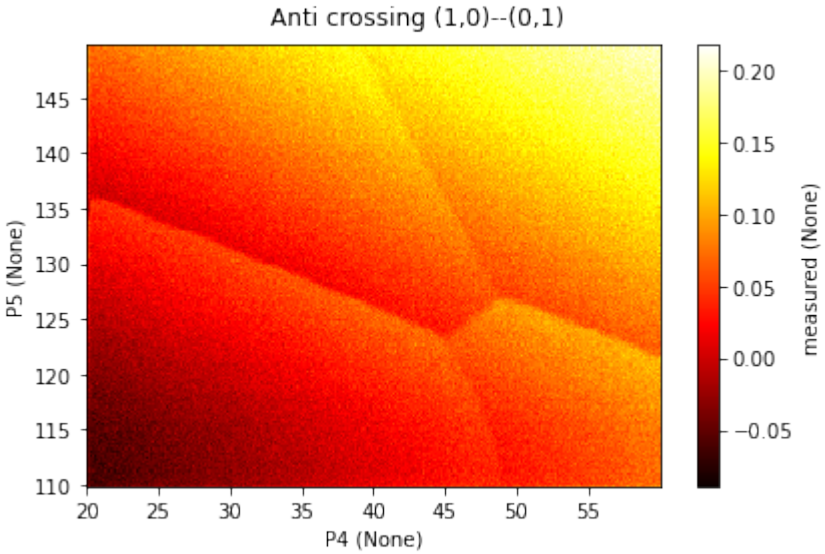
[1]:
```
%matplotlib inline
import os, sys
import qcodes
import qtt
import matplotlib.pyplot as plt
import numpy as np

from qcodes.plots.qcmatplotlib import MatPlot
from qcodes.data.data_set import DataSet
from qtt.data import diffDataset
from qtt.algorithms.bias_triangles import perpLineIntersect, lever_arm, E_charging
```

Load datasets

[2]:
```
exampledatadir=os.path.join(qtt.__path__[0],'exampledata')
DataSet.default_io = qcodes.data.io.DiskIO(exampledatadir)
dataset_anticrossing = qcodes.data.data_set.load_data('charge_stability_diagram_
↪double_dot_system_detail')
dataset_la = qcodes.data.data_set.load_data('charge_stability_diagram_double_dot_
↪system_bias_triangle')
dataset_Ec = qcodes.data.data_set.load_data('charge_stability_diagram_double_dot_
↪system')
```

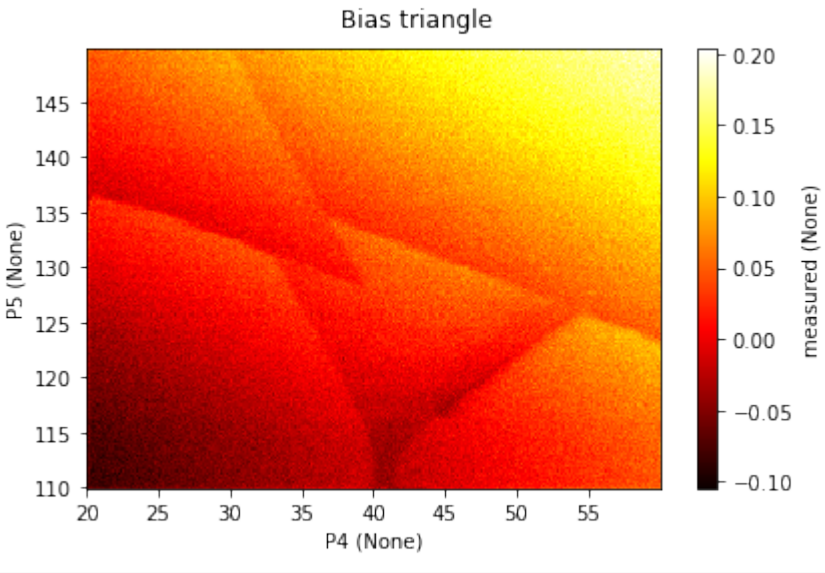
First, make a double dot and find the (1,0) – (0,1) anticrossing:

[3]:
```
plt.figure(1); plt.clf()
MatPlot([dataset_anticrossing.measured], num = 1)
_=plt.suptitle('Anti crossing (1,0)--(0,1)')
```



Then, apply a bias across the Fermi reservoirs (in the example -800 uV) and scan the anti crossing again. This non-equilibrium regime shows "bias triangles", which can be used to extract the gate-to-dot lever arms. More information on these measurements can be found in the references cited in the introduction of this example.

[4]:
```
plt.figure(1); plt.clf()
MatPlot([dataset_la.measured], num = 1)
_=plt.suptitle('Bias triangle')
```



Lever arm

The function perpLineIntersect guides you through the process of extracting the lever arm from the bias triangles. To do this, you must include description = 'lever_arm' as input to the function.

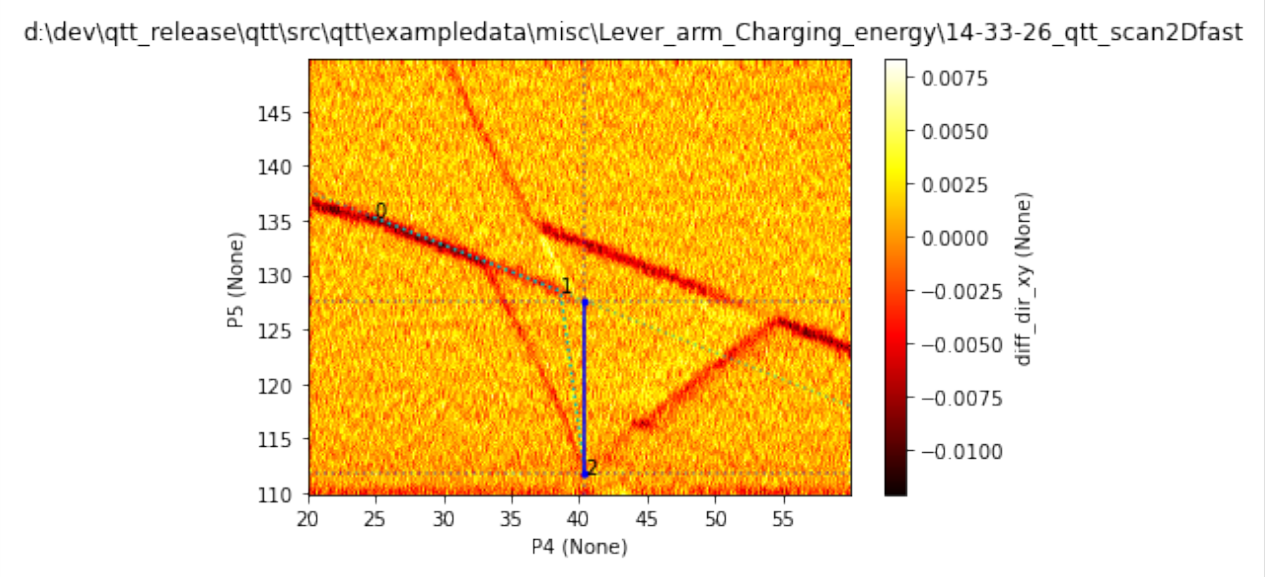
The function instructs you to click on 3 points in the figure. Point 1 and 2 along the addition line for the dot of which you want to determine the lever arm, the third point on the triple point where both dot levels and reservoir are aligned. The perpLineIntersect function will return a dictionary containing the coordinates of these three points, the intersection point of a horizontal/vertical line of the third point with the (extended) line through point 1 and 2 and the line length from the third point to the intersection.

It is important to set the vertical input based on the dot for which the lever arm is being measured. vertical = True (False) to measure the lever arm of the gate in vertical (horizontal) axis.

NB: perpLineIntersect makes use of clickable interactive plots. However, the inline plots in this notebook are not interactive, therefore, in this example we provide the function the clicked points as an input. If you want to try and click, restart the notebook and please use '%pylab tk' instead of '%matplotlib inline' and remove the points input from the function call.

[5]:
```
dot = 'P5'

if dot == 'P4':
    vertical = False
elif dot == 'P5':
    vertical = True
else:
    print("Please choose either dot 4 or dot 5")

clicked_pts = np.array([[  24.87913077,   38.63388728,   40.44875099],
       [ 135.28934654,  128.50469446,  111.75508464]])

lev_arm_fit = perpLineIntersect(dataset_la, description = 'lever_arm', vertical =␣
↪vertical, points = clicked_pts)
```

```
Please click three points;
            Point 1: on the addition line for the dot represented on the vertical axis
            Point 2: further on the addition line for the dot represented on the␣
↪vertical axis
            Point 3: on the triple point at the addition line for the dot represented␣
↪on the horizontal axis
            where both dot levels are aligned
```



Determine the lever arm (μV/mV) by dividing the applied bias for the bias triangles by the voltage span determined by perpLineIntersect

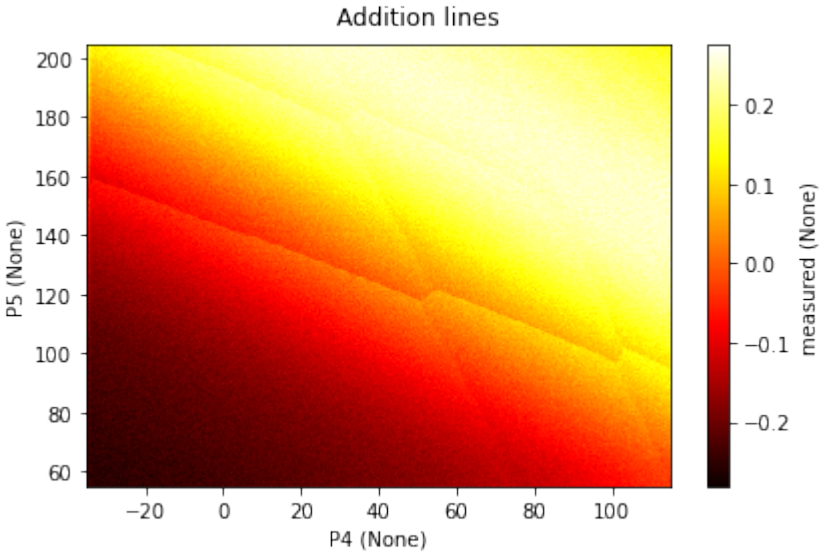
[6]:
```
bias = dataset_la.snapshot()['allgatevalues']['O5'] # bias voltage extracted from the␣
↪dataset
print(bias)
lev_arm = lever_arm(bias, lev_arm_fit, fig = True)
print('''The lever arm of gate %s to dot %s is %.2f ueV/mV'''%(dot, dot[1], lev_arm))
```

```
-800
The lever arm of gate P5 to dot 5 is 50.46 ueV/mV
```

Extract addition energy

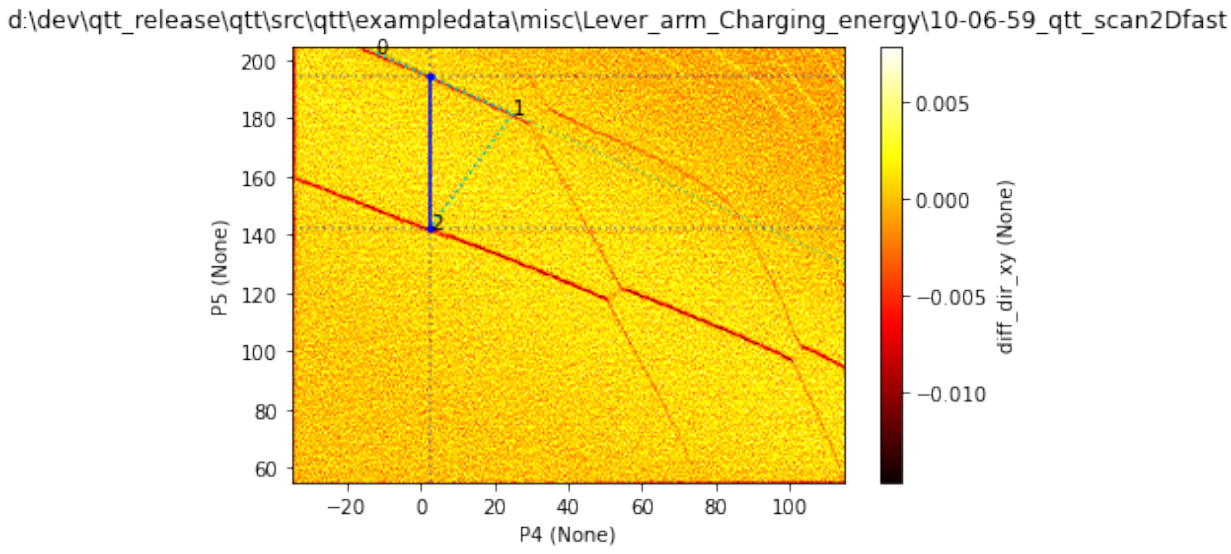
Once the lever arm is known, the addition energy can be extracted from a charge stability diagram showing 2 addition lines. Again, use the function perpLineIntersect, this time using description = 'E_charging'. The function instructs you to click on the 3 relevant points from which the distance between the 2 addition lines can be measured, and converted to meV using the lever arm.

[7]:
```
plt.figure(3); plt.clf()
MatPlot([dataset_Ec.measured], num = 3)
ax = plt.gca()
_=plt.suptitle('Addition lines')
```



[8]:
```
clicked_pts = np.array([[ -11.96239499,   24.89272409,    2.56702695],
        [ 202.62140281,  181.56972616,  142.246783  ]])

Ec_fit = perpLineIntersect(dataset_Ec, description = 'E_charging', vertical =␣
↪vertical, points = clicked_pts)
```

```
Please click three points;
            Point 1: on the (0, 1) - (0,2) addition line
            Point 2: further on the (0, 1) - (0,2) addition line
            Point 3: on the (0, 0) - (0, 1) addition line
```



[9]:
```
E_c = E_charging(lev_arm, results = Ec_fit, fig = True)
print('The charging energy of dot %s is %.2f meV' % (dot[1], E_c/1000))
```

```
The charging energy of dot 5 is 2.63 meV
```

[ ]:

Example for analysis a barrier-barrier scan of a single quantum dot

Pieter Eendebak pieter.eendebak@tno.nl

[1]:
```
import qtt
import qtt.simulation.virtual_dot_array
import qtt.algorithms.onedot

nr_dots = 3
station = qtt.simulation.virtual_dot_array.initialize(reinit=True, nr_dots=nr_dots,␣
↪maxelectrons=2, verbose=0)
gates = station.gates

gv={'B0': -300.000,'B1': 0.487,'B2': -0.126,'B3': 0.000,'D0': 0.111,'O1': -0.478,'O2':
↪ 0.283,'O3': 0.404,'O4': 0.070,'O5': 0.392,'P1': 0.436,'P2': 0.182,'P3': 39.570,
↪'SD1a': -0.160,'SD1b': -0.022,'SD1c': 0.425,'bias_1': -0.312,'bias_2': 0.063}
gates.resetgates(gv,gv, verbose=0)
```

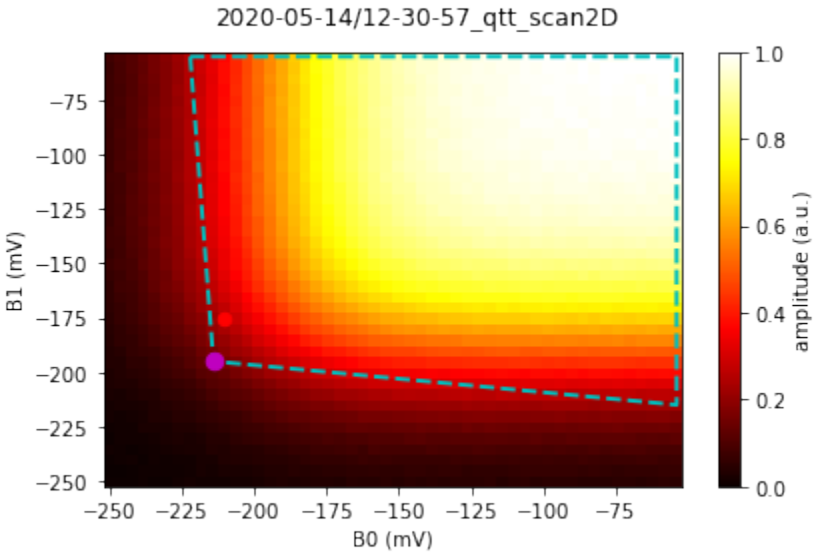
Make a 2D scan of the barrier gates.

[2]:
```
start = -250
scanjob = qtt.measurements.scans.scanjob_t({'sweepdata': dict({'param': 'B0', 'start':
↪ start, 'end': start + 200, 'step': 4., 'wait_time': 0.}), 'minstrument': [
↪'keithley3.amplitude']})
scanjob['stepdata'] = dict({'param': 'B1', 'start': start, 'end': start + 200, 'step':
↪ 5.})
data = qtt.measurements.scans.scan2D(station, scanjob)
```

```
scan2D: 0/40: time 00:00:00 (~00:00:00 remaining): setting B1 to -250.000
```

[3]:
```
results, ptv= qtt.algorithms.onedot.onedotGetBalance(dataset=data, verbose=1,␣
↪fig=None)
qtt.algorithms.onedot.plot_onedot(results, ds = data, fig=1000, verbose=2)
```

```
onedotGetBalance one-dot: balance point 0 at: -214.0 -195.0 [mV]
onedotGetBalance: balance point at: -214.0 -195.0 [mV]
```



For scans where Coulomb diamonds are visible, one can use `onedotGetBalanceFine` to improve the fitting.

[4]:
```
ptfine, rfine=qtt.algorithms.onedot.onedotGetBalanceFine(impixel=None, dd=data,␣
↪fig=None)
```

```
onedotGetBalanceFine: point/best filter value: 0.01/11.55
```

[5]:
```
print(ptfine)
```

```
[[-224.02391035]
 [-162.47011206]]
```

[ ]:
```
```

Example of polarization line fitting

In this example we demonstrate the fitting of an inter-dot transition line (also known as polarization line), by using the functions `fit_pol_all` and `polmod_all_2slopes`. This fitting is useful for determining the tunnel coupling between two quantum dots. More theoretical background about this can be found in L. DiCarlo et al., Phys. Rev. Lett. 92, 226801 (2004) and Diepen et al., Appl. Phys. Lett. 113, 033101 (2018).

Sjaak van diepen - sjaak.vandiepen@tno.nl

Import the modules used in this example.

[1]:
```
import os
import scipy.constants
import matplotlib.pyplot as plt
%matplotlib inline

import qcodes
from qcodes.data.hdf5_format import HDF5Format
import qtt
```

(continues on next page)

(continued from previous page)

```
from qtt.algorithms.tunneling import fit_pol_all, polmod_all_2slopes, plot_
↪polarization_fit
from qtt.data import load_example_dataset
```

Define some physical constants.

The fitting needs some input values: Plancks constan, the Boltzmann constant and the effective electron temperature. The effective electron temperature is the temperature of the electrons in the quantum dots. A method to determine this temperature is to do the polarization line scan at very low tunnel coupling and then fit the polarization line relative to the temperature. Here, we estimate the electron temperature to be 75 mK.

[2]:
```
h = scipy.constants.physical_constants['Planck constant in eV s'][0]*1e15  # ueV/GHz;␣
↪Planck's constant in eV/Hz*1e15 -> ueV/GHz
kb = scipy.constants.physical_constants['Boltzmann constant in eV/K'][0]*1e6  # ueV/K;
↪ Boltzmann constant in eV/K*1e6 -> ueV/K
kT = 75e-3 * kb  # effective electron temperature in ueV
```

Load example data.

Here we load an example dataset. The array 'delta' contains the difference in chemical potential between the two dots. The values for this array are in units of ueV. The fitting is not linear in the values of delta, hence to do the fitting, it is the easiest to convert the voltages on the gates to energies using the leverarm. The lever arm can be detmined in several ways, e.g. by using photon-assisted-tunneling (see example PAT), or by means of bias triangles (see example bias triangles). The array 'signal' contains the data for the sensor signal, usually measured using RF reflectometry on a sensing dot. The units for this array are arbitrary.

[3]:
```
dataset = load_example_dataset('polarization_line')
detuning = dataset.delta.ndarray
signal = dataset.signal.ndarray
```

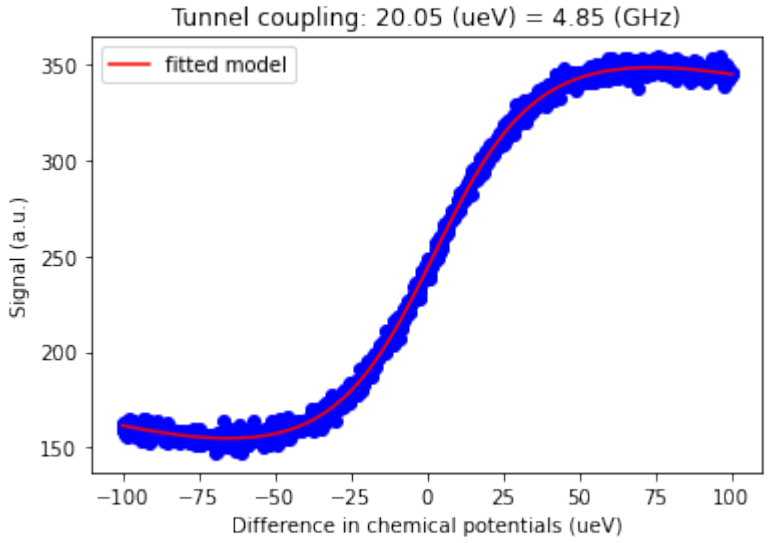
Fit.

The `fit_pol_all` function returns an array with the following parameters: - fitted_parameters[0]: tunnel coupling in ueV - fitted_parameters[1]: offset in x_data for center of transition - fitted_parameters[2]: offset in background signal - fitted_parameters[3]: slope of sensor signal on left side - fitted_parameters[4]: slope of sensor signal on right side - fitted_parameters[5]: height of transition, i.e. sensitivity for electron transition

[4]:
```
fitted_parameters, _, fit_results = fit_pol_all(detuning, signal, kT)
```

Plot the fit and the data.

[5]:
```
plot_polarization_fit(detuning, signal, fit_results, fig = 100)
```



[6]:
```
print(fit_results)
```

```
{'fitted_parameters': array([ 20.04952093,   1.96624788, 100.2481608 ,  -0.49990929,
        -0.43654573, 299.13966912]), 'initial_parameters': array([ 6.66666667e+00,  5.
↪50550556e+00,  1.58893888e+02, -1.44537853e-01,
       -1.44537853e-01,  2.14806828e+02]), 'type': 'polarization fit', 'kT': 6.
↪462999946499999}
```

The values of the model can be calculated as with the method `polmod_all_2slopes`. For example to calculate the value of the sensor at detuning zero:

[7]:
```
polmod_all_2slopes([0], fitted_parameters, kT)
```

[7]:
```
array([243.44568532])
```

[ ]:
```

```

1.2.3 Data

Dataset processing

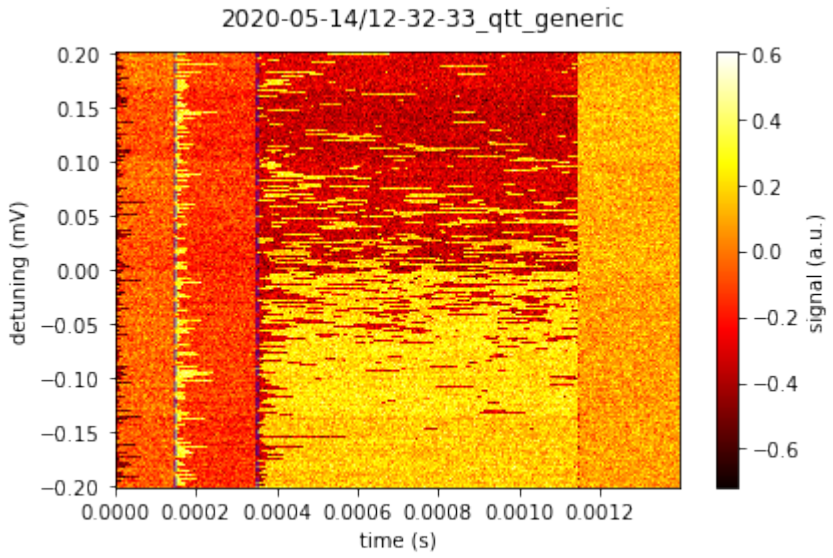
In this notebook we show several methods that can be used to process datasets (currently: `qcodes.DataSet`).

[1]:
```
import copy

import qtt
import qtt.utilities.visualization
from qtt.data import plot_dataset
from qtt.data import load_example_dataset

from qtt.dataset_processing import slice_dataset, dataset_dimension, average_dataset,␣
↪process_dataarray
```

Load and plot a dataset

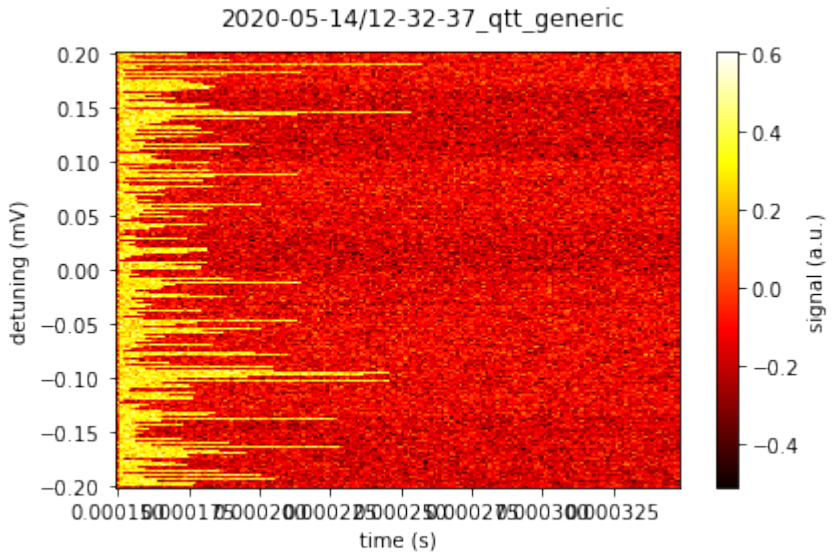
[2]:
```
dataset = load_example_dataset('elzerman_detuning_scan.json')
```

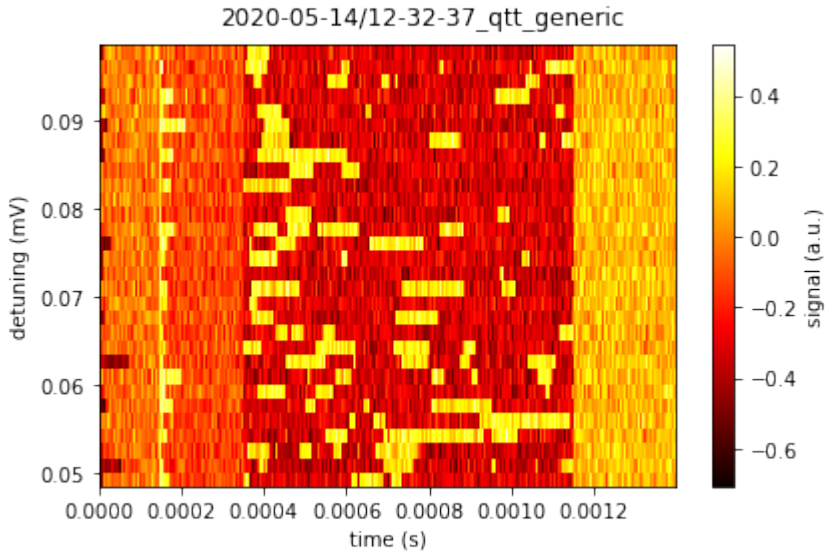
[3]:
```
qtt.data.plot_dataset(dataset)
_=qtt.utilities.visualization.plot_vertical_line(150e-6, color='b')
_=qtt.utilities.visualization.plot_vertical_line(350e-6, color='b')
```



Slice a dataset

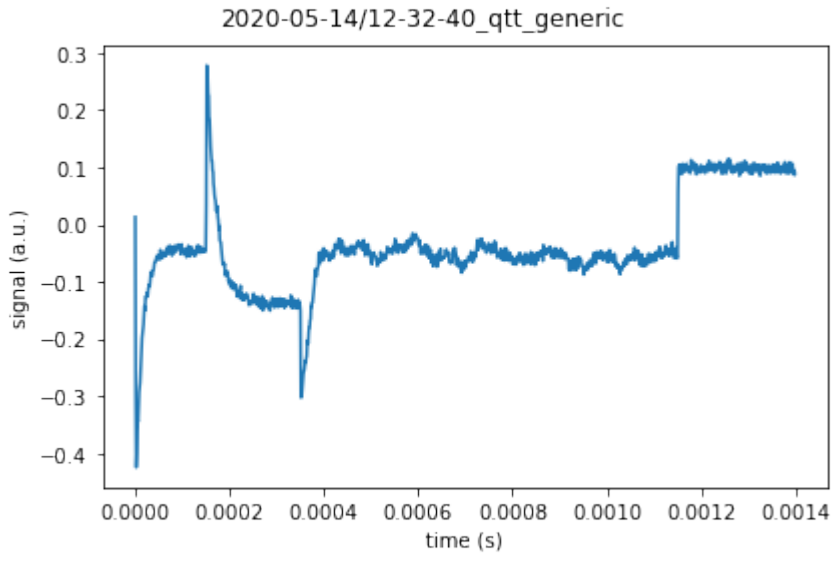
[4]:
```
loading_window = slice_dataset(dataset, [150e-6, 350e-6 ], axis=1)
plot_dataset(loading_window, fig=1)

detuning_window = slice_dataset(dataset, [0.05, 0.10 ], axis=0)
plot_dataset(detuning_window, fig=2)
```





Average over a dimension

[5]:
```
averaged_dataset = average_dataset(dataset, axis=0)
plot_dataset(averaged_dataset)
```



Convert to dictionary format

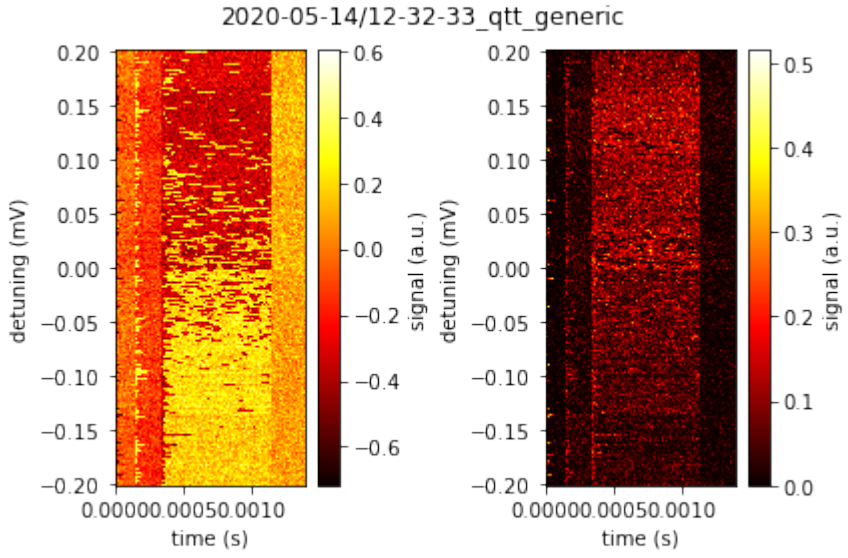
[6]:
```
dataset_dictionary=qtt.data.dataset_to_dictionary(averaged_dataset)
print(dataset_dictionary)
```

```
{'extra': {'location': '2020-05-14/12-32-40_qtt_generic', '_version': '1.2.2'},
↪'metadata': {}, 'arrays': {'signal': {'label': 'signal', 'name': 'signal', 'unit':
↪'a.u.', 'is_setpoint': False, 'full_name': 'signal', 'array_id': 'signal', 'shape':␣
↪(1367,), 'ndarray': array([ 0.01288668, -0.29996808, -0.42453639, ...,  0.09534709,
        0.08637174,  0.08866183]), 'set_arrays': ('time',)}, 'time': {'label': 'time',
↪ 'name': 'time', 'unit': 's', 'is_setpoint': True, 'full_name': 'time', 'array_id':
↪'time', 'shape': (1367,), 'ndarray': array([0.00000000e+00, 1.02400054e-06, 2.
↪04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03]), 'set_arrays': ()}}}
```

Process a DataArray

[7]:
```
process_dataarray(dataset, 'signal', 'signal_squared', lambda x: x**2)
print(dataset)

plot_dataset(dataset, parameter_names=['signal', 'signal_squared'])
```

```
DataSet:
   location = '2020-05-14/12-32-33_qtt_generic'
   <Type>    | <array_id>      | <array.name>    | <array.shape>
   Setpoint  | detuning        | detuning        | (240,)
   Measured  | signal          | signal          | (240, 1367)
   Setpoint  | time            | time            | (240, 1367)
   Measured  | signal_squared  | signal_squared  | (240, 1367)
```

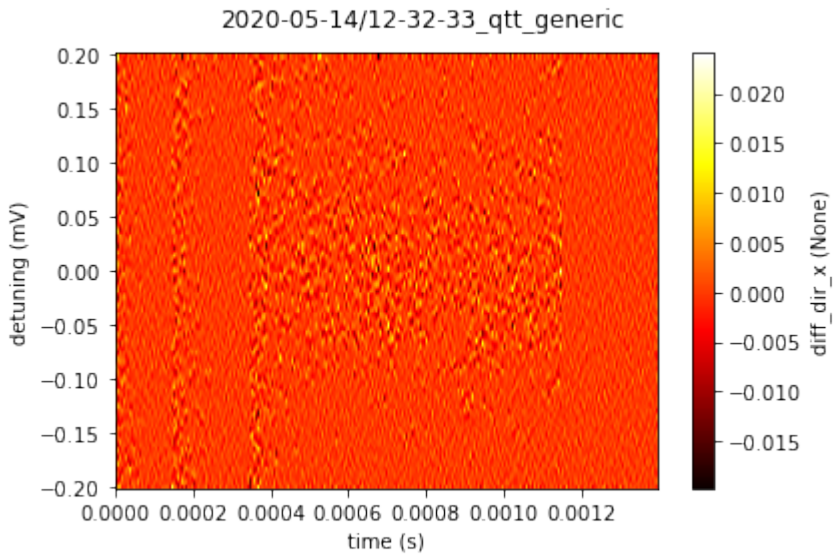


Differentiate

This currently adds a new DataArray to the dataset.

[8]:
```
differentiated_dataset = qtt.data.diffDataset(dataset)
qtt.data.diffDataset(differentiated_dataset, 'x')
print(differentiated_dataset)

plot_dataset(differentiated_dataset, parameter_names=['diff_dir_x'])
```

```
DataSet:
   location = '2020-05-14/12-32-33_qtt_generic'
   <Type>    | <array_id>      | <array.name>    | <array.shape>
   Setpoint  | detuning        | detuning        | (240,)
   Measured  | signal          | signal          | (240, 1367)
   Setpoint  | time            | time            | (240, 1367)
   Measured  | signal_squared  | signal_squared  | (240, 1367)
   Measured  | diff_dir_y      | diff_dir_y      | (240, 1367)
   Measured  | diff_dir_x      | diff_dir_x      | (240, 1367)
```



Serialization

We can convert a `DataSet` to a plain dictionary and back.

[9]:
```
dataset_dictionary = qtt.data.dataset_to_dictionary(dataset)
print(dataset_dictionary)
dataset2 = qtt.data.dictionary_to_dataset(dataset_dictionary)
```

```
{'extra': {'location': '2020-05-14/12-32-33_qtt_generic', '_version': '1.2.2'},
↪'metadata': {}, 'arrays': {'detuning': {'label': 'detuning', 'name': 'detuning',
↪'unit': 'mV', 'is_setpoint': True, 'full_name': 'detuning', 'array_id': 'detuning',
↪'shape': (240,), 'ndarray': array([-0.2       , -0.19832636, -0.19665273, -0.
↪19497909, -0.19330543,
       -0.19163179, -0.18995816, -0.18828452, -0.18661088, -0.18493724,
       -0.1832636 , -0.18158996, -0.17991632, -0.17824268, -0.17656904,
       -0.17489539, -0.17322175, -0.17154811, -0.16987447, -0.16820084,
       -0.1665272 , -0.16485356, -0.16317992, -0.16150628, -0.15983264,
```

(continues on next page)

(continued from previous page)


```
       -0.158159  , -0.15648535, -0.15481171, -0.15313807, -0.15146443,
       -0.14979079, -0.14811715, -0.14644352, -0.14476988, -0.14309624,
       -0.1414226 , -0.13974896, -0.13807531, -0.13640167, -0.13472803,
       -0.13305439, -0.13138075, -0.12970711, -0.12803347, -0.12635984,
       -0.12468619, -0.12301255, -0.12133891, -0.11966527, -0.11799163,
       -0.11631799, -0.11464435, -0.11297071, -0.11129707, -0.10962343,
       -0.10794979, -0.10627615, -0.10460251, -0.10292887, -0.10125523,
       -0.09958159, -0.09790795, -0.09623431, -0.09456067, -0.09288703,
       -0.09121339, -0.08953975, -0.08786611, -0.08619247, -0.08451883,
       -0.08284519, -0.08117155, -0.07949791, -0.07782426, -0.07615063,
       -0.07447699, -0.07280335, -0.07112971, -0.06945607, -0.06778242,
       -0.06610879, -0.06443515, -0.06276151, -0.06108787, -0.05941423,
       -0.05774058, -0.05606695, -0.05439331, -0.05271966, -0.05104603,
       -0.04937239, -0.04769874, -0.0460251 , -0.04435147, -0.04267782,
       -0.04100418, -0.03933054, -0.0376569 , -0.03598326, -0.03430962,
       -0.03263598, -0.03096234, -0.0292887 , -0.02761506, -0.02594142,
       -0.02426778, -0.02259414, -0.0209205 , -0.01924686, -0.01757322,
       -0.01589958, -0.01422594, -0.0125523 , -0.01087866, -0.00920502,
       -0.00753138, -0.00585774, -0.0041841 , -0.00251046, -0.00083682,
        0.00083682,  0.00251046,  0.0041841 ,  0.00585774,  0.00753138,
        0.00920502,  0.01087866,  0.0125523 ,  0.01422594,  0.01589958,
        0.01757322,  0.01924686,  0.0209205 ,  0.02259414,  0.02426778,
        0.02594142,  0.02761506,  0.0292887 ,  0.03096234,  0.03263598,
        0.03430962,  0.03598326,  0.0376569 ,  0.03933054,  0.04100418,
        0.04267782,  0.04435147,  0.0460251 ,  0.04769874,  0.04937239,
        0.05104603,  0.05271966,  0.05439331,  0.05606695,  0.05774058,
        0.05941423,  0.06108787,  0.06276151,  0.06443515,  0.06610879,
        0.06778242,  0.06945607,  0.07112971,  0.07280335,  0.07447699,
        0.07615063,  0.07782426,  0.07949791,  0.08117155,  0.08284519,
        0.08451883,  0.08619247,  0.08786611,  0.08953975,  0.09121339,
        0.09288703,  0.09456067,  0.09623431,  0.09790795,  0.09958159,
        0.10125523,  0.10292887,  0.10460251,  0.10627615,  0.10794979,
        0.10962343,  0.11129707,  0.11297071,  0.11464435,  0.11631799,
        0.11799163,  0.11966527,  0.12133891,  0.12301255,  0.12468619,
        0.12635984,  0.12803347,  0.12970711,  0.13138075,  0.13305439,
        0.13472803,  0.13640167,  0.13807531,  0.13974896,  0.1414226 ,
        0.14309624,  0.14476988,  0.14644352,  0.14811715,  0.14979079,
        0.15146443,  0.15313807,  0.15481171,  0.15648535,  0.158159  ,
        0.15983264,  0.16150628,  0.16317992,  0.16485356,  0.1665272 ,
        0.16820084,  0.16987447,  0.17154811,  0.17322175,  0.17489539,
        0.17656904,  0.17824268,  0.17991632,  0.18158996,  0.1832636 ,
        0.18493724,  0.18661088,  0.18828452,  0.18995816,  0.19163179,
        0.19330543,  0.19497909,  0.19665273,  0.19832636,  0.2       ]), 'set_arrays
↪': ()}, 'signal': {'label': 'signal', 'name': 'signal', 'unit': 'a.u.', 'is_setpoint
↪': False, 'full_name': 'signal', 'array_id': 'signal', 'shape': (240, 1367),
↪'ndarray': array([[-0.04470825, -0.46432495, -0.56243896, ...,  0.08163452,
         0.04119873,  0.07385254],
       [-0.00152588, -0.33325195, -0.3894043 , ...,  0.15380859,
        -0.00259399,  0.08255005],
       [ 0.18203735, -0.2507019 , -0.31784058, ...,  0.00106812,
         0.04760742,  0.0227356 ],
       ...,
       [ 0.06408691, -0.4637146 , -0.40924072, ...,  0.2116394 ,
         0.16784668,  0.21575928],
       [ 0.01800537, -0.32394409, -0.49179077, ...,  0.12954712,
         0.03738403,  0.16204834],
       [ 0.21179199, -0.02929688, -0.46127319, ..., -0.06973267,
```


(continues on next page)

(continued from previous page)


```
        0.10284424, -0.05752563]]), 'set_arrays': ('detuning', 'time')}, 'time': {
↪'label': 'time', 'name': 'time', 'unit': 's', 'is_setpoint': True, 'full_name':
↪'time', 'array_id': 'time', 'shape': (240, 1367), 'ndarray': array([[0.00000000e+00,
↪ 1.02400054e-06, 2.04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03],
       [0.00000000e+00, 1.02400054e-06, 2.04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03],
       [0.00000000e+00, 1.02400054e-06, 2.04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03],
       ...,
       [0.00000000e+00, 1.02400054e-06, 2.04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03],
       [0.00000000e+00, 1.02400054e-06, 2.04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03],
       [0.00000000e+00, 1.02400054e-06, 2.04800108e-06, ...,
       1.39673671e-03, 1.39776070e-03, 1.39878469e-03]]), 'set_arrays': ('detuning',
↪)}, 'signal_squared': {'label': 'signal', 'name': 'signal_squared', 'unit': 'a.u.',
↪'is_setpoint': False, 'full_name': 'signal_squared', 'array_id': 'signal_squared',
↪'shape': (240, 1367), 'ndarray': array([[1.99882779e-03, 2.15597660e-01, 3.
↪16337589e-01, ...,
       6.66419510e-03, 1.69733539e-03, 5.45419753e-03],
       [2.32830644e-06, 1.11056864e-01, 1.51635706e-01, ...,
       2.36570835e-02, 6.72880560e-06, 6.81451056e-03],
       [3.31375981e-02, 6.28514448e-02, 1.01022632e-01, ...,
       1.14087015e-06, 2.26646662e-03, 5.16907312e-04],
       ...,
       [4.10713255e-03, 2.15031230e-01, 1.67477969e-01, ...,
       4.47912375e-02, 2.81725079e-02, 4.65520658e-02],
       [3.24193388e-04, 1.04939775e-01, 2.41858163e-01, ...,
       1.67824561e-02, 1.39756594e-03, 2.62596644e-02],
       [4.48558480e-02, 8.58306885e-04, 2.12772959e-01, ...,
       4.86264471e-03, 1.05769373e-02, 3.30919866e-03]]), 'set_arrays': ('detuning',
↪'time')}, 'diff_dir_y': {'label': 'diff_dir_y', 'name': 'diff_dir_y', 'unit': None,
↪'is_setpoint': False, 'full_name': 'diff_dir_y', 'array_id': 'diff_dir_y', 'shape':␣
↪(240, 1367), 'ndarray': array([[ 0.02467441,  0.02317244,  0.02378851, ..., -0.
↪00312518,
       -0.00098616, -0.00793283],
       [ 0.0235369 ,  0.02001656,  0.01507552, ..., -0.00694358,
       -0.00068102, -0.007965  ],
       [ 0.01254013,  0.01410833,  0.0027335 , ..., -0.00931459,
       -0.00212355, -0.00249565],
       ...,
       [ 0.0170326 ,  0.01147505, -0.02129182, ..., -0.0197943 ,
         0.00859477, -0.00990452],
       [ 0.01980447,  0.03466332, -0.01652782, ..., -0.03326314,
         0.00596913, -0.02698303],
       [ 0.02312505,  0.04703598, -0.009843  , ..., -0.03772412,
         0.0043794 , -0.03545641]]), 'set_arrays': ('detuning', 'time')}, 'diff_dir_x
↪': {'label': 'diff_dir_x', 'name': 'diff_dir_x', 'unit': None, 'is_setpoint': False,
↪ 'full_name': 'diff_dir_x', 'array_id': 'diff_dir_x', 'shape': (240, 1367), 'ndarray
↪': array([[-6.75441402e-03, -1.24756511e-02, -1.79746588e-02, ...,
       -2.15791755e-03, -2.39050588e-03, -2.04945350e-03],
       [-6.93263137e-03, -1.22906849e-02, -1.75441497e-02, ...,
       -2.50083812e-03, -2.57659379e-03, -2.06919546e-03],
       [-4.35955638e-03, -7.98710219e-03, -1.18201167e-02, ...,
       -1.93977093e-03, -1.46484033e-03, -7.57196835e-04],
       ...,
```


(continues on next page)

(continued from previous page)

```
       [-7.32889263e-03, -9.56584479e-03, -9.78552642e-03, ...,
         1.50182692e-03,  8.72919536e-04, -7.45698221e-05],
       [-5.63340545e-03, -8.27244296e-03, -9.04882311e-03, ...,
        -7.22117700e-05, -1.01094416e-03, -2.03497390e-03],
       [-4.53192539e-03, -7.33912314e-03, -8.56147986e-03, ...,
        -8.64665655e-04, -2.00935238e-03, -3.08634345e-03]]), 'set_arrays': ('detuning
↪', 'time')}}}
```

[ ]:

Example MongoDB storage interface

With the storage interface we can store documents into a database.

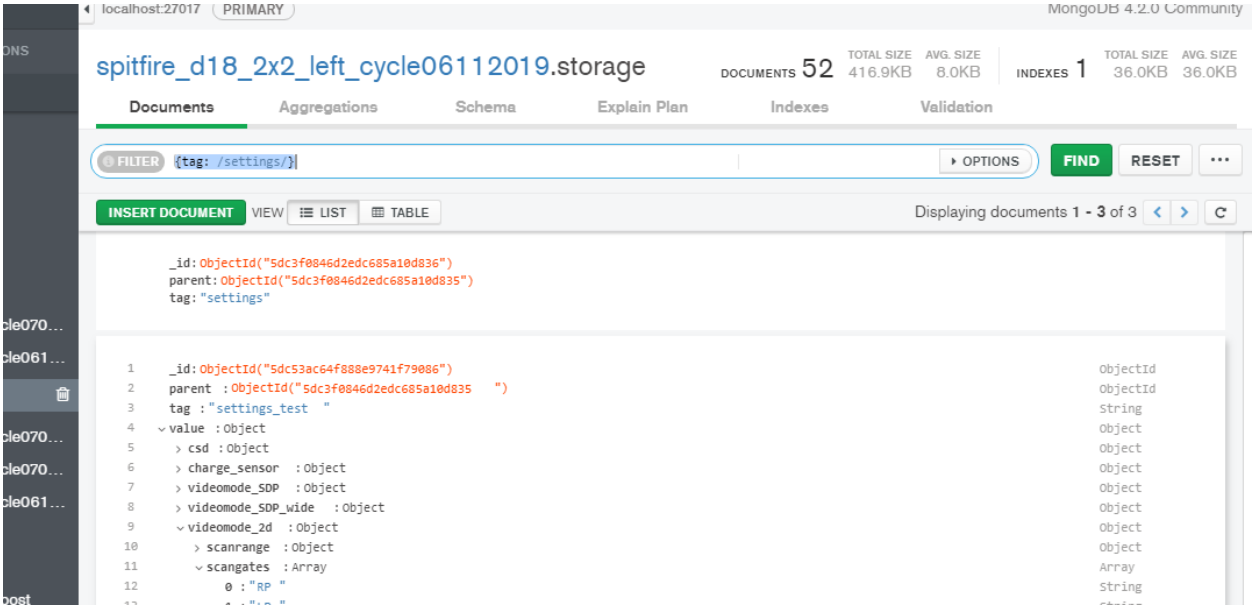
[1]:
```
import qilib.utils.storage

try:
    storage = qilib.utils.storage.StorageMongoDb('testdb', connection_timeout=.5)
except qilib.utils.storage.mongo.ConnectionTimeoutError:
    # when no MongoDB is running, we use StorageMemory as an example
    storage = qilib.utils.storage.StorageMemory('testdb')

storage.save_data({'string': 'hello', 'float': 1.7}, ['stored_document'])
storage.save_data('hello alice', ['list', 'subdocument1'])
storage.save_data('hello bob', ['list', 'subdocument2'])
```

The documents are stored in a tree-linke structure. We can list stored documents using `list_data_subtags`.

[2]:
```
print(storage.list_data_subtags([]))
print(storage.list_data_subtags(['list']))
```

```
['stored_document', 'list']
['subdocument1', 'subdocument2']
```

A GUI for the database is MongoDB Compass. To search for documents in the database, enter a query in the `FILTER` field. For example to search for all documents with tag containing the text `settings` use the query `{tag: /settings/}`.



[ ]:

[ ]:

Data serialization

The qtt package contains methods to serialize objects to JSON.

[1]:
```
from dataclasses import dataclass
from functools import partial

import numpy as np
from dataclasses_json import dataclass_json

from qtt.utilities.json_serializer import decode_json, encode_json, qtt_serializer
```

[2]:
```
json = encode_json([1.0, 'hello'])
print(f'list of items: {json}\n')

json = encode_json(np.array([1.,2.]))
print(f'numpy array: {json}')
```

```
list of items: [1.0, "hello"]

numpy array: {"__object__": "array", "__content__": {"__ndarray__":
↪"AAAAAAAA8D8AAAAAAAAAQA==", "__data_type__": "<f8", "__shape__": [2]}}
```

[3]:
```
decoded_array = decode_json(json)
print(decoded_array)
```

```
[1. 2.]
```

Custom data types

Custom data objects can be serialized by creating an encoder and decoder. For example to serialize `dataclass` objects with JSON we can do the following.

[4]:
```
@dataclass_json
@dataclass
class CustomClass():

    x : float
    y : str

mydata = CustomClass(x=1., y='hi')
print(mydata)
```

```
CustomClass(x=1.0, y='hi')
```

We can create custom encoding methods to make the serialization possible.

[5]:
```
qtt_serializer.register_dataclass(CustomClass)

json = qtt_serializer.serialize(mydata)
print(f'encoded json: {json}')

decoded_object = qtt_serializer.unserialize(json)
print(f'decoded_object: {decoded_object}')
```

```
encoded json: {"__object__": "_dataclass_CustomClass", "__content__": {"x": 1.0, "y":
↪"hi"}}
decoded_object: CustomClass(x=1.0, y='hi')
```

[ ]:
```

```

1.2.4 Simulation

Classical simulation of triple dot

This classical simulation of a triple dot system investigates the behaviour of the dot system at a fixed total electron number n=3. It specifically investigates the behaviour of the honeycomb measurements around different charge distribution states.

Import packages

[1]:
```
%matplotlib inline
import matplotlib.pyplot as plt

import numpy as np
from qtt.simulation.classicaldotsystem import ClassicalDotSystem, TripleDot
```

Initialize dot system

[2]:
```
DotSystem = TripleDot(maxelectrons=3)

DotSystem.alpha = np.array([[1.0, 0.5, 0.25],
                            [0.5, 1.0, 0.5],
                            [0.25, 0.5, 1.0]])
DotSystem.W = 2*np.array([5.0, 1.0, 5.0])
DotSystem.Eadd = np.array([54.0, 54.0, 54.0])
DotSystem.mu0 = np.array([-25.0, -25.0, -25.0])
```

1. Standard Honeycomb example

Help functions for calculating gate planes

[3]:
```
def create_linear_gate_matrix(gate_points, steps_x, steps_y):
    x_y_start = gate_points[0]
    x_end = gate_points[1]
    y_end = gate_points[2]
    step_x = ((x_end-x_y_start) * 1.0 / (steps_x-1))
    step_y = ((y_end-x_y_start) * 1.0 / (steps_y-1))
    return [[start_x+i*step_x for i in range(steps_x)] for start_x in [x_y_
↪start+i*step_y for i in range(steps_y)]]

def calculate_end_points(ref_point,ref_value,dirVecX,dirVecY,rangeX,rangeY):
    gate_points = []
    gate_points.append(ref_value-(rangeX*(1-ref_point[0])*dirVecX)-(rangeY*(1-ref_
↪point[1])*dirVecY))
    gate_points.append(gate_points[0]+rangeX*dirVecX)
    gate_points.append(gate_points[0]+rangeY*dirVecY)
    return gate_points

def create_all_gate_matrix(ref_point, ref_value, dirVecX, dirVecY, rangeX, rangeY,␣
↪pointsX, pointsY):
    gate_matrix=np.zeros((3,pointsX,pointsY))
    for gate in range(len(ref_value)):
        end_points = calculate_end_points(ref_point,ref_value[gate],dirVecX[gate],
↪dirVecY[gate],rangeX,rangeY)
        gate_matrix[gate]= create_linear_gate_matrix(end_points,pointsX,pointsY)
    return gate_matrix
```

Define gate plane

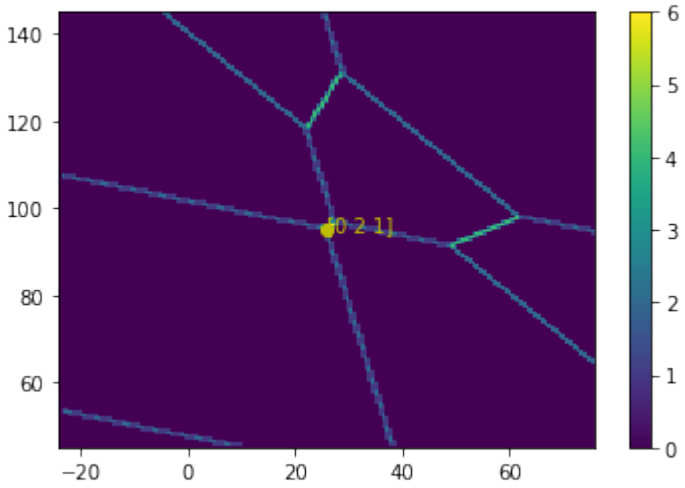
[4]:
```
P1 = 25.987333
P2 = 83.0536667
P3 = 94.987333
ref_pt = [0.5, 0.5]
ref_value = [P1, P2, P3]
dirVecY = [0.0, 0.0, 1.0]
dirVecX = [1.0, 0.0, 0.0]
rangeX = 100
rangeY = 100
pointsX = 150
pointsY = 150
```

(continues on next page)

(continued from previous page)

```
end_points_x = calculate_end_points(ref_pt,ref_value[0],dirVecX[0],dirVecY[0],rangeX,
↪rangeY)
sweepx = np.linspace(end_points_x[0], end_points_x[1], pointsX)

end_points_y = calculate_end_points(ref_pt,ref_value[2],dirVecX[2],dirVecY[2],rangeX,
↪rangeY)
sweepy = np.linspace(end_points_y[0], end_points_y[2], pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)

centre_value = np.array([P1,P2,P3])
charge_state = DotSystem.calculate_ground_state(centre_value)

DotSystem.simulate_honeycomb(gate_matrix)

plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.plot(centre_value[0], centre_value[2], 'yo')
plt.annotate(np.array_str(charge_state), xy = (centre_value[0], centre_value[2]),␣
↪color = 'y')
plt.show()
```

```
simulatehoneycomb: 12.17 [s]
```



2. N=3 plane

Let's now look at the N=3 plane around the 111 charge state. We will define two axis ϵ and δ for changing relative chemical potentials.

Virtual gates

[5]:
```
L = np.linalg.solve(DotSystem.alpha, np.array([1,0,0]))
M = np.linalg.solve(DotSystem.alpha, np.array([0,1,0]))
R = np.linalg.solve(DotSystem.alpha, np.array([0,0,1]))

# \epsilon = L - R
dirVecX = L - R
# \delta = M - (L+R)/2
dirVecY = M - L/2 - R/2
```

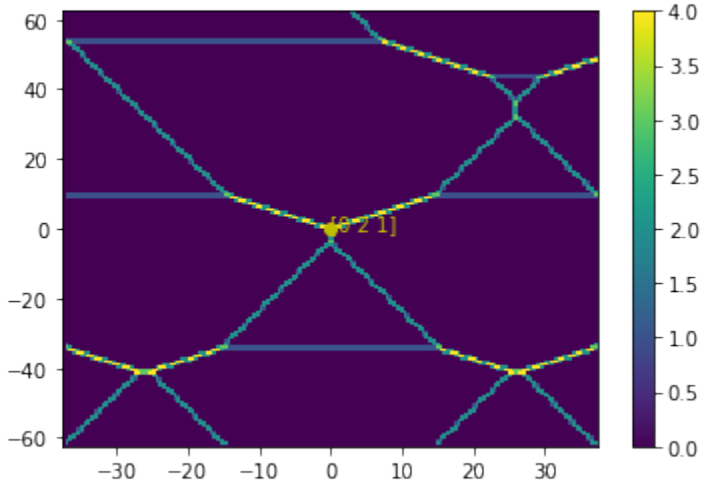
Use virtual gates to make gate plane

[6]:
```
rangeX = 75
rangeY = 125

sweepx = np.linspace(-rangeX/2,rangeX/2,pointsX)
sweepy = np.linspace(-rangeY/2,rangeY/2,pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)
```

Run simulation

[7]:
```
DotSystem.simulate_honeycomb(gate_matrix)

plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.plot(0, 0, 'yo')
plt.annotate(np.array_str(charge_state), xy = (0, 0), color = 'y')
plt.show()
```

```
simulatehoneycomb: 12.09 [s]
```



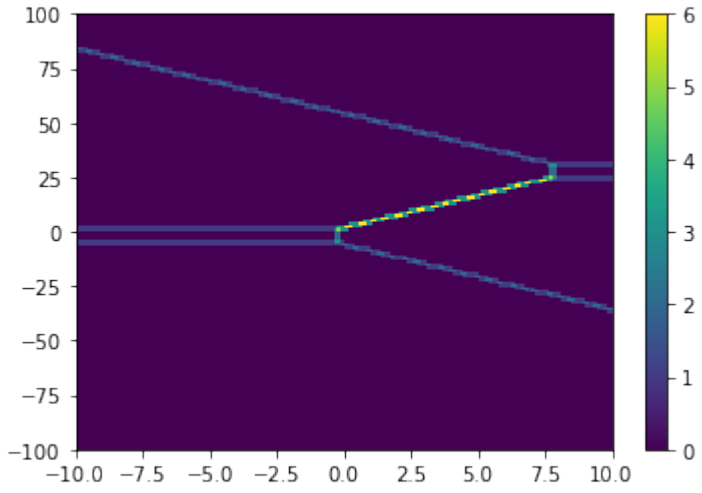
Middle dot alignment

Intersection: 102, 201, 111

[8]:
```
P1 = 115.0
P2 = -14.0
P3 = 115.0
ref_pt = [0.5, 0.5]
ref_value = [P1, P2, P3]

L = np.linalg.solve(DotSystem.alpha, np.array([1,0,0]))
M = np.linalg.solve(DotSystem.alpha, np.array([0,1,0]))
R = np.linalg.solve(DotSystem.alpha, np.array([0,0,1]))

# M
dirVecX = M
# filling: (L + M + R)/3
dirVecY = (L + M + R)/3

rangeX = 20
rangeY = 200

pointsX = 150
pointsY = 150

sweepx = np.linspace(-rangeX/2,rangeX/2,pointsX)
sweepy = np.linspace(-rangeY/2,rangeY/2,pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)
```

[9]:
```
DotSystem.simulate_honeycomb(gate_matrix)

plt.pcolor(sweepx,sweepy,DotSystem.honeycomb)
plt.colorbar()
plt.show()
```

```
simulatehoneycomb: 10.03 [s]
```



[10]:
```
corner1 = ref_value - 0.2*M - 5/3 * (L+M+R)
```

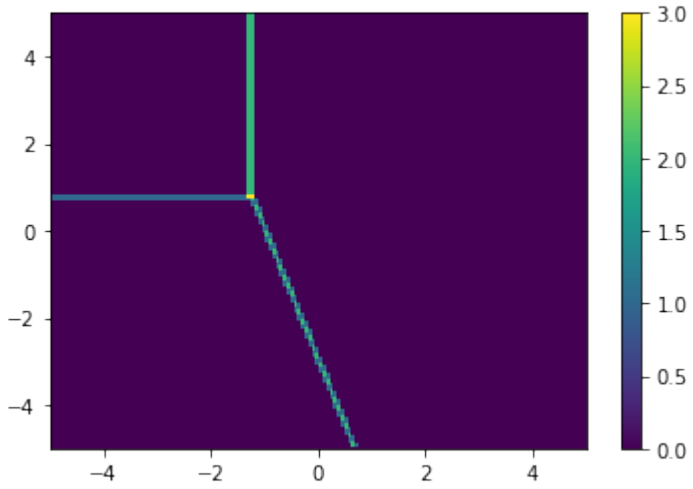
[11]:
```
corner1
```

[11]:
```
array([114.02222222, -14.88888889, 114.02222222])
```

Intersection: 012, 021, 111

[12]:
```
P1 = 45-30+10
P2 = 103-18
P3 = 45+30+10+8

ref_pt = [0.5, 0.5]
ref_value = [P1, P2, P3]

L = np.linalg.solve(DotSystem.alpha, np.array([1,0,0]))
M = np.linalg.solve(DotSystem.alpha, np.array([0,1,0]))
R = np.linalg.solve(DotSystem.alpha, np.array([0,0,1]))

# M
dirVecX = M
# filling: (L + M + R)/3
dirVecY = (L + M + R)/3

rangeX = 10
rangeY = 10

pointsX = 150
pointsY = 150

sweepx = np.linspace(-rangeX/2,rangeX/2,pointsX)
sweepy = np.linspace(-rangeY/2,rangeY/2,pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)

DotSystem.simulate_honeycomb(gate_matrix)
plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.show()
```

```
simulatehoneycomb: 9.89 [s]
```

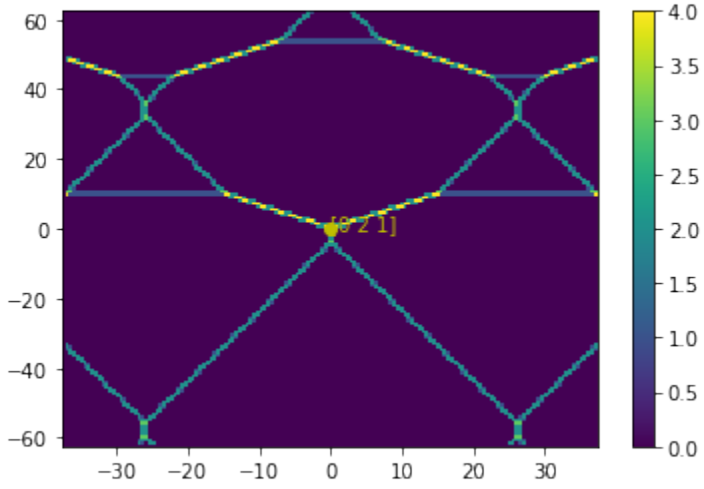


[13]:
```
corner3 = ref_value - 1.22*M + 0.783/3 * (L+M+R)
corner3
```

[13]:
```
array([25.98733333, 83.05366667, 93.98733333])
```

[14]:
```
ref_value = corner1
rangeX = 75
rangeY = 125
pointsX = 150
pointsY = 150
# \epsilon = L - R
dirVecX = L - R
# \delta = M - (L+R)/2
dirVecY = M - L/2 - R/2

#end_points_x = calculate_end_points(ref_pt,ref_value[0],dirVecX[0],dirVecY[0],rangeX,
↪rangeY)
#sweepx = np.linspace(end_points_x[0], end_points_x[1], pointsX)

#end_points_y = calculate_end_points(ref_pt,ref_value[2],dirVecX[2],dirVecY[2],rangeX,
↪rangeY)
#sweepy = np.linspace(end_points_y[0], end_points_y[2], pointsY)

sweepx = np.linspace(-rangeX/2,rangeX/2,pointsX)
sweepy = np.linspace(-rangeY/2,rangeY/2,pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)

DotSystem.simulate_honeycomb(gate_matrix)

plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.plot(0, 0, 'yo')
plt.annotate(np.array_str(charge_state), xy = (0, 0), color = 'y')
plt.show()
```

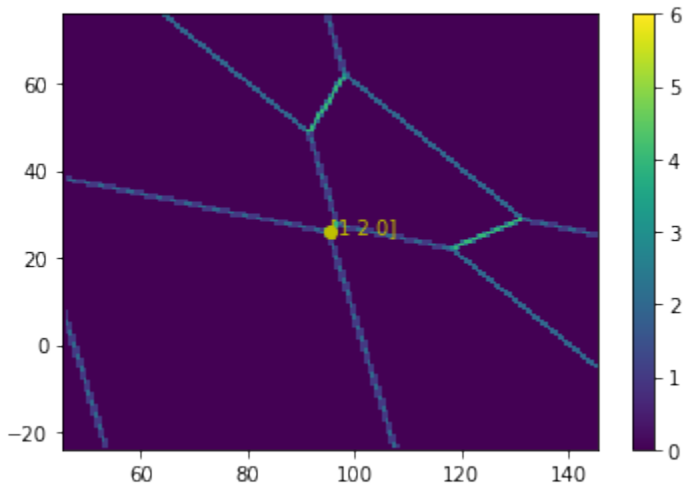
```
simulatehoneycomb: 10.09 [s]
```



Intersection: 120, 210, 111

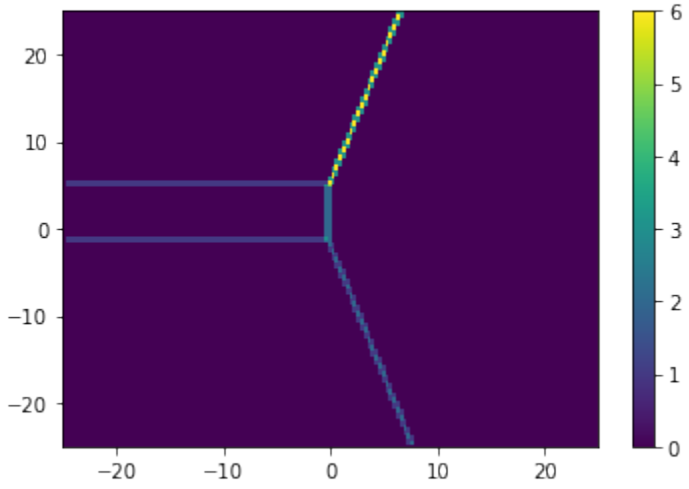
[15]:
```
P1 = 95.5
P2 = 83
P3 = 26
ref_pt = [0.5, 0.5]
ref_value = [P1, P2, P3]
dirVecY = [0.0, 0.0, 1.0]
dirVecX = [1.0, 0.0, 0.0]
rangeX = 100
rangeY = 100
pointsX = 150
pointsY = 150

end_points_x = calculate_end_points(ref_pt,ref_value[0],dirVecX[0],dirVecY[0],rangeX,
↪rangeY)
sweepx = np.linspace(end_points_x[0], end_points_x[1], pointsX)

end_points_y = calculate_end_points(ref_pt,ref_value[2],dirVecX[2],dirVecY[2],rangeX,
↪rangeY)
sweepy = np.linspace(end_points_y[0], end_points_y[2], pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)

centre_value = np.array([P1,P2,P3])
charge_state = DotSystem.calculate_ground_state(centre_value)

DotSystem.simulate_honeycomb(gate_matrix)

plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.plot(centre_value[0], centre_value[2], 'yo')
plt.annotate(np.array_str(charge_state), xy = (centre_value[0], centre_value[2]),␣
↪color = 'y')
plt.show()
```

```
simulatehoneycomb: 10.35 [s]
```



[16]:
```
# M
dirVecX = M
# filling: (L + M + R)/3
dirVecY = (L + M + R)/3

rangeX = 50
rangeY = 50

pointsX = 150
pointsY = 150

sweepx = np.linspace(-rangeX/2,rangeX/2,pointsX)
sweepy = np.linspace(-rangeY/2,rangeY/2,pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)

DotSystem.simulate_honeycomb(gate_matrix)
plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.show()
```

```
simulatehoneycomb: 10.19 [s]
```



[17]:
```
corner2 = ref_value - 0.228*M - 1.516/3 * (L+M+R)
corner2
```

[17]:
```
array([95.31511111, 82.45155556, 25.81511111])
```

3 corners -> plane

[18]:
```
corner1 = np.array([ 114.02222222,  -14.88888889,  114.02222222])
corner2 = np.array([ 95.31511111,  82.45155556,  25.81511111])
corner3 = np.array([ 25.98733333,  83.05366667,  93.98733333])

np.cross(corner2-corner1,corner3-corner1)
```

[18]:
```
array([6689.02489162, 7390.50833334, 6737.1329424 ])
```

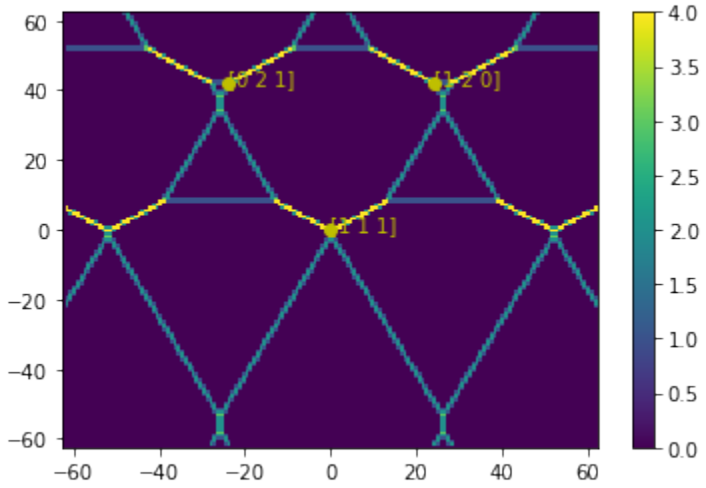
[19]:
```
corner1 = np.array([ 114.02222222,  -13,  114.02222222])
ref_value = corner1
rangeX = 125
rangeY = 125
pointsX = 150
pointsY = 150
# \epsilon = L - R
dirVecX = L - R
# \delta = M - (L+R)/2
dirVecY = M - L/2 - R/2

sweepx = np.linspace(-rangeX/2,rangeX/2,pointsX)
sweepy = np.linspace(-rangeY/2,rangeY/2,pointsY)

gate_matrix=create_all_gate_matrix(ref_pt, ref_value, dirVecX, dirVecY, rangeX,␣
↪rangeY, pointsX, pointsY)

DotSystem.simulate_honeycomb(gate_matrix)

plt.pcolor(sweepx,sweepy,DotSystem.honeycomb, shading='auto')
plt.colorbar()
plt.plot(0, 0, 'yo')
```

(continues on next page)

(continued from previous page)

```
plt.plot(24, 42, 'yo')
plt.plot(-24, 42, 'yo')
charge_state1 = DotSystem.calculate_ground_state(corner1)
charge_state2 = DotSystem.calculate_ground_state(corner2)
charge_state3 = DotSystem.calculate_ground_state(corner3)
plt.annotate(np.array_str(charge_state1), xy = (0, 0), color = 'y')
plt.annotate(np.array_str(charge_state2), xy = (24, 42), color = 'y')
plt.annotate(np.array_str(charge_state3), xy = (-24, 42), color = 'y')
plt.show()
```

```
simulatehoneycomb: 9.26 [s]
```



[20]:
```
corner3[0] - 58.02222222
```

[20]:
```
-32.034888890000005
```

[21]:
```
-32.035/(L[0] - R[0])
```

[21]:
```
-24.026249999999997
```

[22]:
```
corner1+42*(M-L/2-R/2)-24*(L-R)
```

[22]:
```
array([26.02222222, 85.        , 90.02222222])
```

[23]:
```
corner1+42*(M-L/2-R/2)+24*(L-R)
```

[23]:
```
array([90.02222222, 85.        , 26.02222222])
```

[ ]:
```

```

Simulated charge stability diagrams for a 2x2 quantum dot system

This example shows how to use `qtt.simulation.dotsystem` to define a Hubbard-based model system of a 4 quantum dot array in a 2x2 plaquette configuration. Here we will use this model system to reproduce the Fig 1c plot from https://aip.scitation.org/doi/10.1063/1.5025928

[1]:
```
%matplotlib inline

import os
import numpy as np
import matplotlib.pyplot as plt

import qtt.simulation.dotsystem as dotsystem
```

Define some extra helper functions:

[2]:
```
def gates_from_det(dot_system, det_values=None):
    """ Sets the correct gate voltages. Run this function after setting the detuning␣
↪variables."""
    if det_values:
        return np.dot(np.linalg.inv(dot_system.la_matrix), det_values)
    det_values = [getattr(dot_system, 'det%d' % (i + 1)) for i in range(dot_system.
↪ndots)]
    gate_values = np.dot(np.linalg.inv(dot_system.la_matrix), det_values)
    for i in range(dot_system.ndots):
        setattr(dot_system, 'P%d' % (i + 1), gate_values[i])
    return gate_values


def det_from_gates(dot_system, plunger_values=None):
    """ Sets the correct detuning variables that matches the gate combination.
        Run this function after setting the gate voltages.
    """
    if plunger_values:
        return np.dot(dot_system.la_matrix, plunger_values)
    plunger_values = np.array([getattr(dot_system, 'P%d' % (i + 1)) for i in␣
↪range(dot_system.ndots)])
    det_values = np.dot(dot_system.la_matrix, plunger_values)
    for i in range(dot_system.ndots):
        setattr(dot_system, 'det%d' % (i + 1), det_values[i])
    return det_values


def parse_scan_parameters(dot_system, scan_parameters, scan_steps, scan_range):
    """ Used to parse the input to the simulate_honeycomb function."""
    half_range = scan_range/2
    scan_steps_x, scan_steps_y = scan_steps
    scan_min_max = [[-half_range, half_range, -half_range, half_range],
                    [-half_range, -half_range, half_range, half_range]]
    dot_system.makeparamvalues2D(scan_parameters, scan_min_max, scan_steps_x, scan_
↪steps_y)

    if scan_parameters[0].startswith('det'):
        for parameter in dot_system.scan_parameters:
            dot_system.vals2D[pn] += getattr(dot_system, parameter)
            parameters = dot_system.vals2D.copy()
        return parameters
```

(continues on next page)

(continued from previous page)

```
    initial_values = dot_system.getall('det')
    det = [np.zeros(dot_system.vals2D[scan_parameters[0]].shape) for i in range (dot_
↪system.ndots)]
    params = dot_system.vals2D.copy()
    dict_params = {}
    for name in scan_parameters:
        if '{' in name:
            dict_prop = eval(name)
            for name2, prop in dict_prop.items():
                dict_params[name2] = getattr(dot_system, name2) + params[name] * prop
        else:
            dict_params[name] = getattr(dot_system, name) + params[name]
    for step_x in range(scan_steps_x):
        for step_y in range(scan_steps_y):
            for pn, pv in dict_params.items():
                setattr(dot_system, pn, pv[step_x, step_y])
            det_temp = det_from_gates(dot_system)
            for k in range(len(det_temp)):
                det[k][step_x, step_y] = det_temp[k]

    dot_system.setall('det', initial_values)

    dot_system.vals2D = {}
    for i in range(len(det)):
        dot_system.vals2D['det%i' % (i + 1)] = det[i]

    return params


def show_charge_occupation_numbers_on_click(dot_system, x_data, y_data, number_of_
↪clicks=1):
    """ Shows the charge occupation numbers at the clicked points in the plotted␣
↪charge stability diagram.

    Args:
        dot_system (dot_system): The simulated dot system.
        x_data (np.array): The parsed result data from the independent gate variable.
        y_data (np.array): The parsed result data from the dependent gate variable.
        number_of_clicks (int): The number of times the occupation numbers should be␣
↪printed.
    """
    mV_minimum_x = x_data.min()
    mV_minimum_y = y_data.min()
    mV_range_x = x_data.max() - mV_minimum_x
    mV_range_y = y_data.max() - mV_minimum_y
    pixels_range_x, pixels_range_y = np.shape(x_data)

    if not 'QTT_UNITTEST' in os.environ:
        for i in range(number_of_clicks):
            mouse_clicks = plt.ginput()
            if mouse_clicks:
                (mV_coordinate_x, mV_coordinate_y) = mouse_clicks[0]

                x_index = int((mV_coordinate_x - mV_minimum_x) / mV_range_x * pixels_
↪range_x)
                y_index = int((mV_coordinate_y - mV_minimum_y) / mV_range_y * pixels_
↪range_y)
```

(continues on next page)

(continued from previous page)

```
                charge_occupation_numbers = str(dot_system.hcgs[y_index, x_index])
                plt.text(mV_coordinate_x, mV_coordinate_y, charge_occupation_numbers,␣
↪color='white')
```

Initialize the model system with the experimental parameters

[3]:
```
def initialize_two_by_two_system():
    """ Creates the two by two quantum model. The parameters are set according to the␣
↪experimental setup."""
    two_by_two = dotsystem.TwoXTwo()

    # cross-capacitance matrix and lever arms
    #                                      P1      P2      P3      P4
    cross_capacitance_matrix = np.array([[ 1.00,  0.45,  0.54,  0.87],  # Dot 1
                                         [ 0.65,  1.00,  0.47,  0.50],  # Dot 2
                                         [ 0.17,  0.47,  1.00,  0.24],  # Dot 3
                                         [ 0.44,  0.35,  0.88,  1.00]]) # Dot 4

    det_to_plunger = np.array([0.039 * np.ones(4), 0.041 * np.ones(4),
                               0.054 * np.ones(4), 0.031 * np.ones(4)]) # meV/mV

    two_by_two.la_matrix = cross_capacitance_matrix * det_to_plunger

    # All the following values in meV
    # On-site interaction per dot
    two_by_two.osC1 = 2.5
    two_by_two.osC2 = 2.3
    two_by_two.osC3 = 3
    two_by_two.osC4 = 1.8

    # Intersite interaction per pairs of dots
    two_by_two.isC1 = 0.47 # 1-2
    two_by_two.isC2 = 0.35 # 2-3
    two_by_two.isC3 = 0.43 # 3-4
    two_by_two.isC4 = 0.30 # 4-1
    two_by_two.isC5 = 0.28 # 1-3
    two_by_two.isC6 = 0.18 # 2-4

    # Tunnel coupling per pairs of dots
    two_by_two.tun1 = 0.02 # 1-2
    two_by_two.tun2 = 0.02 # 2-3
    two_by_two.tun3 = 0.02 # 3-4
    two_by_two.tun4 = 0.02 # 4-1

    # Energy offsets per dot (0 is the boundary for adding 1 electron)
    two_by_two.det1 = 1
    two_by_two.det2 = 1
    two_by_two.det3 = 0
    two_by_two.det4 = 0

    gate_voltages = gates_from_det(two_by_two) # This adds the gate voltages (tbt.P#,␣
↪in mV) that result in the above detuning
    print('Current gate voltages: P1={:.2f} mV, P2={:.2f} mV, P3={:.2f} mV, P4={:.2f}␣
↪mV'.format(*gate_voltages))
```

(continues on next page)

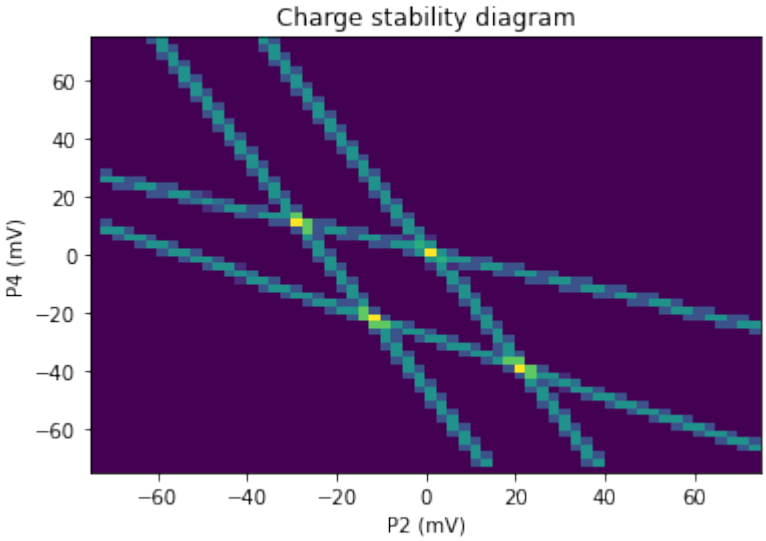
(continued from previous page)

```
    return two_by_two
```

Run a 2D gate scan simulation and plot the charge stability diagram

[4]:
```
two_by_two = initialize_two_by_two_system()

scan_parameters = ['P2', 'P4']
parameter_x, parameter_y = scan_parameters
scan_steps = [61, 61]
scan_range = 150

parsed_results = parse_scan_parameters(two_by_two, scan_parameters, scan_steps, scan_
↪range)
two_by_two.simulatehoneycomb()

x_values = parsed_results[parameter_x]
y_values = parsed_results[parameter_y]

plt.figure()
plt.pcolor(x_values, y_values, two_by_two.honeycomb, shading='auto')
plt.xlabel("{0} (mV)".format(parameter_x))
plt.ylabel("{0} (mV)".format(parameter_y))
_ = plt.title('Charge stability diagram')
```

```
Current gate voltages: P1=35.83 mV, P2=11.19 mV, P3=-8.40 mV, P4=-12.29 mV
simulatehoneycomb: 0/61
simulatehoneycomb: 4/61
simulatehoneycomb: 8/61
simulatehoneycomb: 12/61
simulatehoneycomb: 16/61
simulatehoneycomb: 20/61
simulatehoneycomb: 24/61
simulatehoneycomb: 28/61
simulatehoneycomb: 32/61
simulatehoneycomb: 36/61
simulatehoneycomb: 40/61
simulatehoneycomb: 44/61
simulatehoneycomb: 48/61
simulatehoneycomb: 52/61
simulatehoneycomb: 56/61
simulatehoneycomb: 60/61
simulatehoneycomb: 17.52 [s] (multiprocess False)
```



If you would like to check the charge occupation states at different points in the charge stability diagram, you can do that using the method below. The module matplotlib is set to interactive mode using `%pylab tk`. This will show up a new window that allows for clicking functionality.

[5]:
```
if not 'QTT_UNITTEST' in os.environ:
    %pylab tk

two_by_two = initialize_two_by_two_system()

scan_parameters = ['P2', 'P4']
parameter_x, parameter_y = scan_parameters
scan_steps = [61, 61]
scan_range = 150

parsed_results = parse_scan_parameters(two_by_two, scan_parameters, scan_steps, scan_
↪range)
two_by_two.simulatehoneycomb()

x_values = parsed_results[parameter_x]
y_values = parsed_results[parameter_y]

plt.figure()
plt.pcolor(x_values, y_values, two_by_two.honeycomb, shading='auto')
plt.xlabel("{0} (mV)".format(parameter_x))
plt.ylabel("{0} (mV)".format(parameter_y))
_ = plt.title('Charge stability diagram')

show_charge_occupation_numbers_on_click(two_by_two, x_values, y_values, number_of_
↪clicks=4)
```

```
Populating the interactive namespace from numpy and matplotlib
Current gate voltages: P1=35.83 mV, P2=11.19 mV, P3=-8.40 mV, P4=-12.29 mV
simulatehoneycomb: 0/61
simulatehoneycomb: 4/61
simulatehoneycomb: 8/61
simulatehoneycomb: 12/61
simulatehoneycomb: 16/61
simulatehoneycomb: 20/61
```

(continues on next page)

(continued from previous page)

```
simulatehoneycomb: 24/61
simulatehoneycomb: 28/61
simulatehoneycomb: 32/61
simulatehoneycomb: 36/61
simulatehoneycomb: 40/61
simulatehoneycomb: 44/61
simulatehoneycomb: 48/61
simulatehoneycomb: 52/61
simulatehoneycomb: 56/61
simulatehoneycomb: 60/61
simulatehoneycomb: 18.50 [s] (multiprocess False)
```

```
---------------------------------------------------------------------------
TclError                                  Traceback (most recent call last)
d:\dev\qtt_release\env\lib\site-packages\matplotlib\blocking_input.py in __call__
↪(self, n, timeout)
     91             # Start event loop.
---> 92             self.fig.canvas.start_event_loop(timeout=timeout)
     93         finally:  # Run even on exception like ctrl-c.

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backend_bases.py in start_event_
↪loop(self, timeout)
   2287         while self._looping and counter * timestep < timeout:
-> 2288             self.flush_events()
   2289             time.sleep(timestep)

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backends\_backend_tk.py in flush_
↪events(self)
    404         # docstring inherited
--> 405         self._master.update()
    406

~\AppData\Local\Programs\Python\Python36\lib\tkinter\__init__.py in update(self)
   1176         """Enter event loop until all pending events have been processed by␣
↪Tcl."""
-> 1177         self.tk.call('update')
   1178     def update_idletasks(self):

TclError: can't invoke "update" command: application has been destroyed

During handling of the above exception, another exception occurred:

TclError                                  Traceback (most recent call last)
d:\dev\qtt_release\env\lib\site-packages\matplotlib\backend_bases.py in _wait_cursor_
↪for_draw_cm(self)
   2784             try:
-> 2785                 self.set_cursor(cursors.WAIT)
   2786                 yield

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backends\_backend_tk.py in set_
↪cursor(self, cursor)
    543         window = self.canvas.get_tk_widget().master
--> 544         window.configure(cursor=cursord[cursor])
    545         window.update_idletasks()

~\AppData\Local\Programs\Python\Python36\lib\tkinter\__init__.py in configure(self,␣
↪cnf, **kw)
```

(continues on next page)

(continued from previous page)

```
   1484         """
-> 1485         return self._configure('configure', cnf, kw)
   1486     config = configure

~\AppData\Local\Programs\Python\Python36\lib\tkinter\__init__.py in _configure(self,␣
↪cmd, cnf, kw)
   1475             return self._getconfigure1(_flatten((self._w, cmd, '-'+cnf)))
-> 1476         self.tk.call(_flatten((self._w, cmd)) + self._options(cnf))
   1477     # These used to be defined in Widget:

TclError: invalid command name "."

During handling of the above exception, another exception occurred:

TclError                                  Traceback (most recent call last)
<ipython-input-5-3b8135c8f477> in <module>
     21 _ = plt.title('Charge stability diagram')
     22
---> 23 show_charge_occupation_numbers_on_click(two_by_two, x_values, y_values,␣
↪number_of_clicks=4)

<ipython-input-2-144fae537620> in show_charge_occupation_numbers_on_click(dot_system,␣
↪x_data, y_data, number_of_clicks)
     82     if not 'QTT_UNITTEST' in os.environ:
     83         for i in range(number_of_clicks):
---> 84             mouse_clicks = plt.ginput()
     85             if mouse_clicks:
     86                 (mV_coordinate_x, mV_coordinate_y) = mouse_clicks[0]

d:\dev\qtt_release\env\lib\site-packages\matplotlib\pyplot.py in ginput(n, timeout,␣
↪show_clicks, mouse_add, mouse_pop, mouse_stop)
   2320         n=n, timeout=timeout, show_clicks=show_clicks,
   2321         mouse_add=mouse_add, mouse_pop=mouse_pop,
-> 2322         mouse_stop=mouse_stop)
   2323
   2324

d:\dev\qtt_release\env\lib\site-packages\matplotlib\figure.py in ginput(self, n,␣
↪timeout, show_clicks, mouse_add, mouse_pop, mouse_stop)
   2329                                                     mouse_stop=mouse_stop)
   2330         return blocking_mouse_input(n=n, timeout=timeout,
-> 2331                                     show_clicks=show_clicks)
   2332
   2333     def waitforbuttonpress(self, timeout=-1):

d:\dev\qtt_release\env\lib\site-packages\matplotlib\blocking_input.py in __call__
↪(self, n, timeout, show_clicks)
    261         self.clicks = []
    262         self.marks = []
--> 263         BlockingInput.__call__(self, n=n, timeout=timeout)
    264         return self.clicks
    265

d:\dev\qtt_release\env\lib\site-packages\matplotlib\blocking_input.py in __call__
↪(self, n, timeout)
     93         finally:  # Run even on exception like ctrl-c.
     94             # Disconnect the callbacks.
```

(continues on next page)

(continued from previous page)

```
---> 95             self.cleanup()
     96         # Return the events in this case.
     97         return self.events

d:\dev\qtt_release\env\lib\site-packages\matplotlib\blocking_input.py in cleanup(self,
↪ event)
    250                 mark.remove()
    251             self.marks = []
--> 252             self.fig.canvas.draw()
    253         # Call base class to remove callbacks.
    254         BlockingInput.cleanup(self)

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backends\backend_tkagg.py in␣
↪draw(self)
      7 class FigureCanvasTkAgg(FigureCanvasAgg, FigureCanvasTk):
      8     def draw(self):
----> 9         super(FigureCanvasTkAgg, self).draw()
     10         _backend_tk.blit(self._tkphoto, self.renderer._renderer, (0, 1, 2, 3))
     11         self._master.update_idletasks()

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backends\backend_agg.py in␣
↪draw(self)
    390         with RendererAgg.lock, \
    391              (self.toolbar._wait_cursor_for_draw_cm() if self.toolbar
--> 392               else nullcontext()):
    393             self.figure.draw(self.renderer)
    394             # A GUI class may be need to update a window using this draw, so

~\AppData\Local\Programs\Python\Python36\lib\contextlib.py in __enter__(self)
     79     def __enter__(self):
     80         try:
---> 81             return next(self.gen)
     82         except StopIteration:
     83             raise RuntimeError("generator didn't yield") from None

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backend_bases.py in _wait_cursor_
↪for_draw_cm(self)
   2786                 yield
   2787             finally:
-> 2788                 self.set_cursor(self._lastCursor)
   2789         else:
   2790             yield

d:\dev\qtt_release\env\lib\site-packages\matplotlib\backends\_backend_tk.py in set_
↪cursor(self, cursor)
    542     def set_cursor(self, cursor):
    543         window = self.canvas.get_tk_widget().master
--> 544         window.configure(cursor=cursord[cursor])
    545         window.update_idletasks()
    546

~\AppData\Local\Programs\Python\Python36\lib\tkinter\__init__.py in configure(self,␣
↪cnf, **kw)
   1483         the allowed keyword arguments call the method keys.
   1484         """
-> 1485         return self._configure('configure', cnf, kw)
   1486     config = configure
```

(continues on next page)

(continued from previous page)

```
   1487     def cget(self, key):

~\AppData\Local\Programs\Python\Python36\lib\tkinter\__init__.py in _configure(self,␣
↪cmd, cnf, kw)
   1474         if isinstance(cnf, str):
   1475             return self._getconfigure1(_flatten((self._w, cmd, '-'+cnf)))
-> 1476         self.tk.call(_flatten((self._w, cmd)) + self._options(cnf))
   1477     # These used to be defined in Widget:
   1478     def configure(self, cnf=None, **kw):

TclError: invalid command name "."
```

```
ERROR:tornado.application:Exception in callback functools.partial(<function Kernel.
↪enter_eventloop.<locals>.advance_eventloop at 0x00000223CF62FC80>)
Traceback (most recent call last):
  File "d:\dev\qtt_release\env\lib\site-packages\tornado\ioloop.py", line 743, in _
↪run_callback
    ret = callback()
  File "d:\dev\qtt_release\env\lib\site-packages\ipykernel\kernelbase.py", line 314,␣
↪in advance_eventloop
    eventloop(self)
  File "d:\dev\qtt_release\env\lib\site-packages\ipykernel\eventloops.py", line 232,␣
↪in loop_tk
    app.tk.createfilehandler(stream.getsockopt(zmq.FD), READABLE, notifier)
AttributeError: '_tkinter.tkapp' object has no attribute 'createfilehandler'
```

[ ]:

One and two electron Hamiltonian

This model is valid for a double-dot system tuned to the transition from (1,0) to (0,1) or with two electrons for (1,1) to (2,0).

Author: Pieter Eendebak (pieter.eendebak@gmail.com), Bruno Buijtendorp (brunobuijtendorp@gmail.com)

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sympy as sp
%matplotlib inline
sp.init_printing(use_latex='latex')
```

One electron Hamiltonian

Define 1-electron double dot Hamiltonian e is detuning, t is tunnel coupling. The basis we work in is (1,0) and (0,1).

[2]:
```
e, t = sp.symbols('e t')
H = sp.Matrix([[e/2, t],[t, -e/2]])
sp.pprint(H)

#%% Get normalized eigenvectors and eigenvalues
eigvec_min = H.eigenvects()[0][2][0].normalized()
eigval_min = H.eigenvects()[0][0]

eigvec_plus = H.eigenvects()[1][2][0].normalized()
```

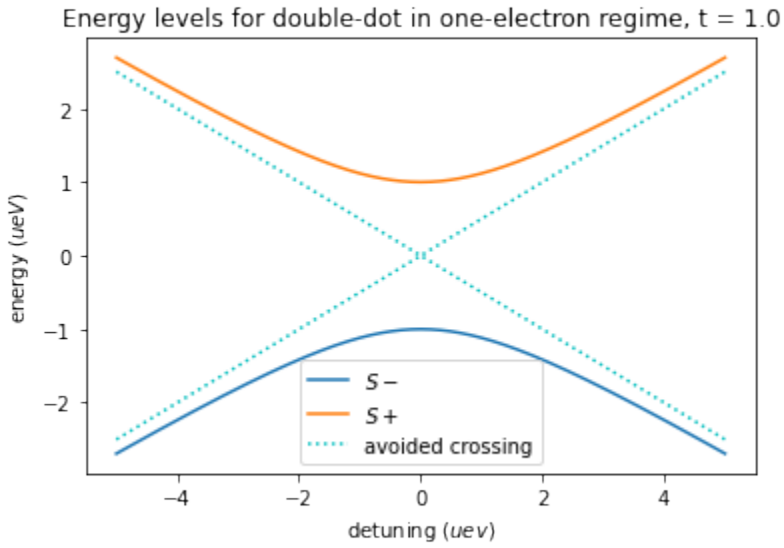
(continues on next page)

(continued from previous page)

```
eigval_plus = H.eigenvects()[1][0]

#%% Lambdify eigenvalues to make them numerical functions of e and t (nicer plotting)
eigval_min_func = sp.lambdify((e,t), eigval_min , 'numpy')
eigval_plus_func = sp.lambdify((e,t), eigval_plus, 'numpy')

#%% Plot energy levels
t_value = 1
plot_x_limit = 5
Npoints_x = 1000
```

```
e
─   t
2

   -e
t  ──
    2
```

[3]:
```
erange = np.linspace(-plot_x_limit, plot_x_limit, Npoints_x)
levelfig, levelax = plt.subplots()
levelax.plot(erange, eigval_min_func(erange , t_value), label='$S-$')
levelax.plot(erange, eigval_plus_func(erange, t_value), label ='$S+$')
levelax.set_title('Energy levels for double-dot in one-electron regime, t = %.1f' % t_
↪value)
plt.plot(erange, erange/2, ':c', label='avoided crossing')
plt.plot(erange, -erange/2, ':c')
plt.legend()
levelax.set_xlabel('detuning $(uev)$')
levelax.set_ylabel('energy $(ueV)$')
_=plt.axis('tight')
```

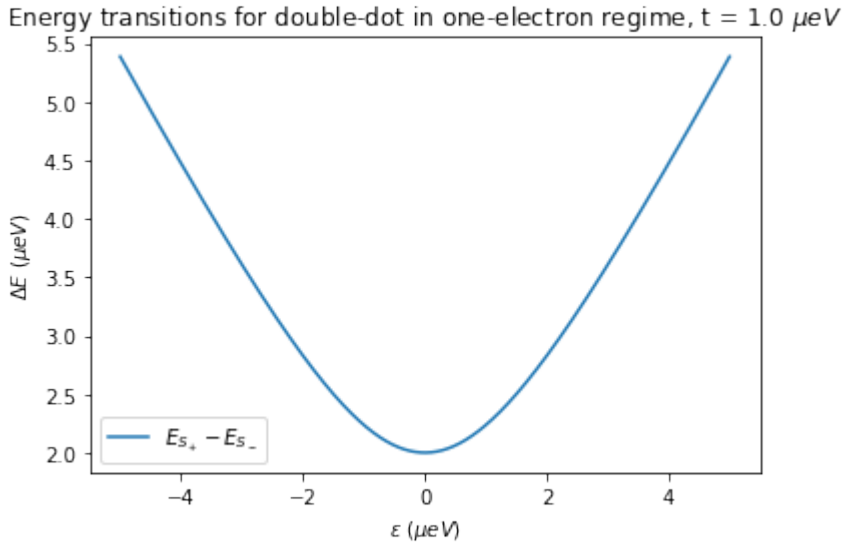


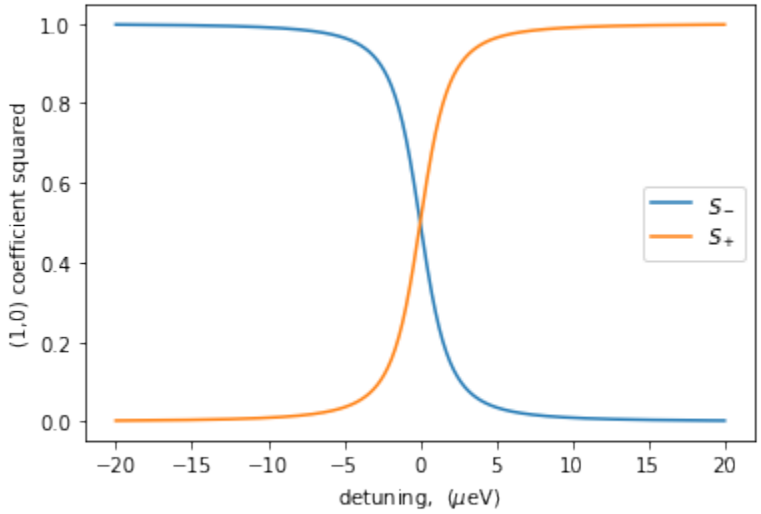
[4]:
```
#%% Plot energy level differences
SminS = eigval_plus_func(erange , t_value) - eigval_min_func(erange, t_value)

plt.figure()
```

(continues on next page)

(continued from previous page)

```
plt.plot(erange, SminS, label='$E_{S_+} - E_{S_-}$')
plt.title('Energy transitions for double-dot in one-electron regime, t = %.1f $\mu eV$
↪' % (t_value))
plt.legend()
plt.ylabel('$\Delta E$ $ (\mu eV)$')
plt.xlabel('$\epsilon$ $ (\mu eV)$')

#%% Get S(1,0) component of eigenvectors
eigcomp_min = eigvec_min[0]
eigcomp_plus = eigvec_plus[0]

#%% Plot S(1,0) components squared (probabilities) of eigenvectors as function of␣
↪detuning
t_value = 1
erange = np.linspace(-20,20,500)
plot_x_limit = 20

# Lambdify eigenvector components to make them functions of e and t
eigcompmin_func = sp.lambdify((e,t), eigcomp_min , 'numpy')
eigcompplus_func = sp.lambdify((e,t), eigcomp_plus, 'numpy')

fig2, ax2 = plt.subplots()
ax2.plot(erange,eigcompmin_func(erange, t_value)**2, label='$S_-$')
ax2.plot(erange,eigcompplus_func(erange, t_value)**2, label='$S_+$')
ax2.set_xlabel('detuning,  ($\mu$eV)')
ax2.set_ylabel('(1,0) coefficient squared')
_=plt.legend()
```





Two-electron Hamiltonian

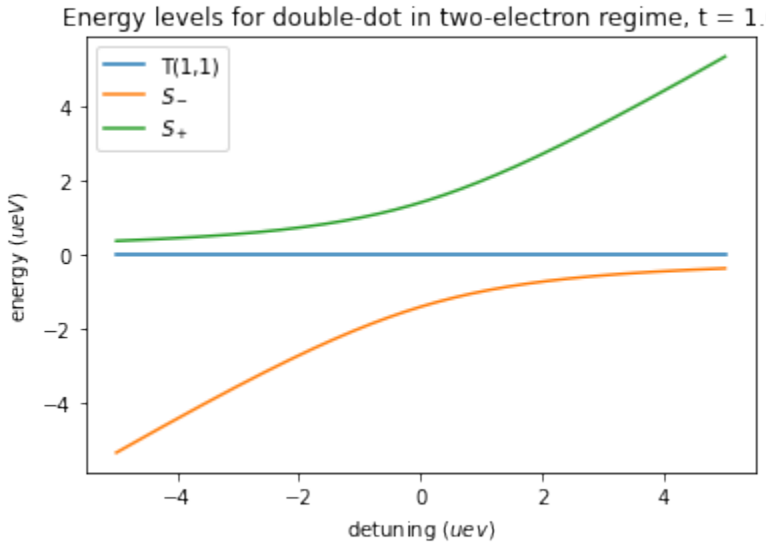
Define 2-electron double dot Hamiltonian e is detuning, t is tunnel coupling. The basis we work in is: {S(2,0), S(1,1), T(1,1)}

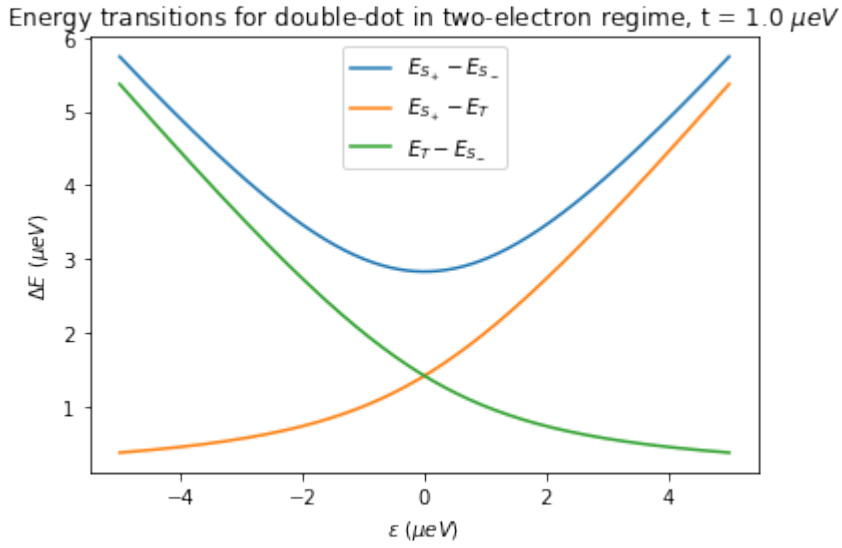
[5]:
```
e, t = sp.symbols('e t')
# Basis: {S(2,0), S(1,1), T(1,1)}
H = sp.Matrix([[e, sp.sqrt(2)*t, 0],[sp.sqrt(2)*t, 0, 0],[0, 0, 0]])

#%% Get normalized eigenvectors and eigenvalues
eigvec_min = H.eigenvects()[1][2][0].normalized()
eigval_min = H.eigenvects()[1][0]

eigvec_plus = H.eigenvects()[2][2][0].normalized()
eigval_plus = H.eigenvects()[2][0]

eigvec_T = H.eigenvects()[0][2][0].normalized()
eigval_T = H.eigenvects()[0][0]

#%% Lambdify eigenvalues to make them numerical functions of e and t (nicer plotting)
eigval_min_func = sp.lambdify((e,t), eigval_min , 'numpy')
eigval_plus_func = sp.lambdify((e,t), eigval_plus, 'numpy')

#%% Plot energy levels
t_value = 1
plot_x_limit = 5
Npoints_x = 1000

erange = np.linspace(-plot_x_limit, plot_x_limit, Npoints_x)
levelfig, levelax = plt.subplots()
levelax.plot(erange, [eigval_T]*len(erange), label='T(1,1)')
levelax.plot(erange, eigval_min_func(erange , t_value), label='$S_-$')
levelax.plot(erange, eigval_plus_func(erange, t_value), label ='$S_+$')
levelax.set_title('Energy levels for double-dot in two-electron regime, t = %.1f' % t_
↪value)
plt.legend()
```

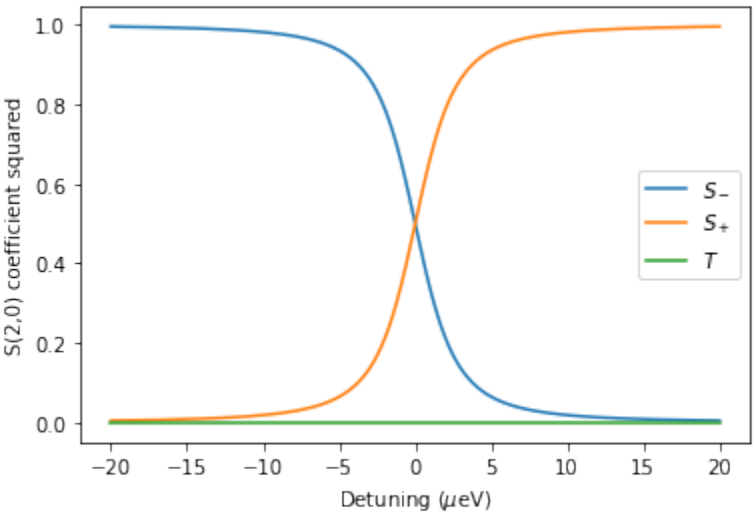
(continues on next page)

(continued from previous page)

```
levelax.set_xlabel('detuning $(uev)$')
levelax.set_ylabel('energy $(ueV)$')
plt.axis('tight')

#%% Plot energy level differences
SminS = eigval_plus_func(erange , t_value) - eigval_min_func(erange, t_value)
S20minT = eigval_plus_func(erange, t_value)
TminS11 = -eigval_min_func(erange, t_value)

plt.figure()
plt.plot(erange, SminS, label='$E_{S_+} - E_{S_-}$')
plt.plot(erange, S20minT, label = '$E_{S_+} - E_T$')
plt.plot(erange, TminS11, label = '$E_T - E_{S_-}$')
plt.title('Energy transitions for double-dot in two-electron regime, t = %.1f $\mu eV$
↪' % (t_value))
plt.legend()
plt.ylabel('$\Delta E$ $ (\mu eV)$')
plt.xlabel('$\epsilon$ $ (\mu eV)$')

#%% Get S(2,0) component of eigenvectors
eigcomp_min = eigvec_min[0]
eigcomp_plus = eigvec_plus[0]
eigcomp_T = eigvec_T[0]

#%% Plot S(2,0) components squared (probabilities) of eigenvectors as function of␣
↪detuning
t_value = 1
erange = np.linspace(-20,20,500)
plot_x_limit = 20

# Lambdify eigenvector components to make them functions of e and t
eigcompmin_func = sp.lambdify((e,t), eigcomp_min , 'numpy')
eigcompplus_func = sp.lambdify((e,t), eigcomp_plus, 'numpy')

fig2, ax2 = plt.subplots()
ax2.plot(erange,eigcompmin_func(erange, t_value)**2, label='$S_-$')
ax2.plot(erange,eigcompplus_func(erange, t_value)**2, label='$S_+$')
ax2.plot(erange,[eigcomp_T]*len(erange), label='$T$')
ax2.set_xlabel('Detuning ($\mu$eV)')
ax2.set_ylabel('S(2,0) coefficient squared')
_=plt.legend()
```


Energy levels for double-dot in two-electron regime, t = 1.0
T(1,1)
S_-
S_+
energy (μeV)
detuning (μev)


Energy transitions for double-dot in two-electron regime, t = 1.0 μeV
$E_{S_+} - E_{S_-}$
$E_{S_+} - E_T$
$E_T - E_{S_-}$
ΔE (μeV)
ε (μeV)


S(2,0) coefficient squared
S_-
S_+
T
Detuning (μeV)

[ ]:

Classical simulation of quantum dots

In this file we show some of the capabilities of the `ClassicalDotSystem`, which can simulate simple quantum dot systems.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from qtt.simulation.classicaldotsystem import ClassicalDotSystem, TripleDot
```

We define a linear triple-dot system using the `TripleDot` class, which inherits from the `ClassicalDotSystem`.

[2]:
```
test_dot = TripleDot()
```

The function `calculate_ground_state` calculates the number of electrons in the ground state for given values of the voltages on the gates forming the dots.

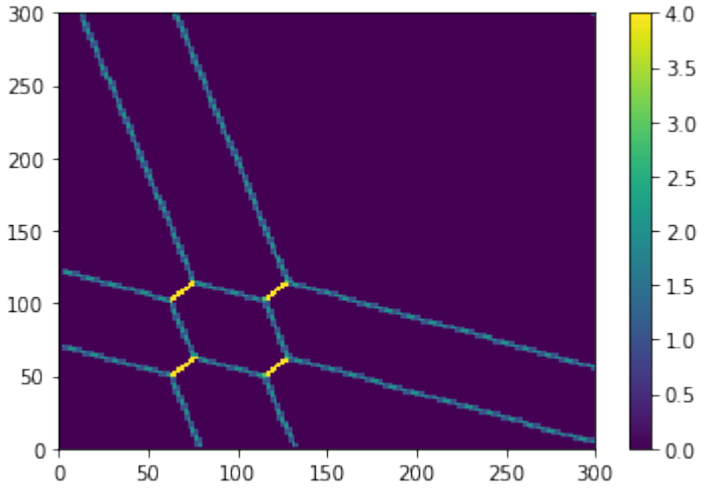
[3]:
```
temp_state = test_dot.calculate_ground_state(np.array([0,0,0]));
print(temp_state)
temp_state = test_dot.calculate_ground_state(np.array([120,0,100]));
print(temp_state)
```

```
[0 0 0]
[1 0 1]
```

[4]:
```
# make a test gate plane
nx = 150
ny = 150
test2Dparams = np.zeros((3,nx,ny))
sweepx = np.linspace(0, 300, nx)
sweepy = np.linspace(0, 300, ny)
xv, yv = np.meshgrid(sweepx, sweepy)
test2Dparams[0] = xv+.1*yv
xv, yv = np.meshgrid(sweepy, sweepx)
test2Dparams[1] = .1*xv+yv

# run the honeycomb simulation
test_dot.simulate_honeycomb(test2Dparams, multiprocess=False, verbose=1)
```

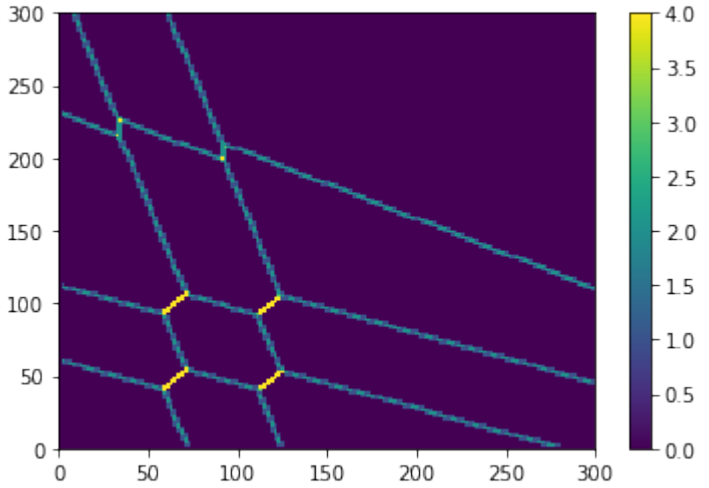
```
simulatehoneycomb: 0/150
simulatehoneycomb: 0.53 [s]
```

[5]:
```
plt.pcolor(sweepx,sweepy,test_dot.honeycomb, shading='auto')
plt.colorbar()
plt.show()
```



[6]:
```
test2Dparams[2] = 103

# run the honeycomb simulation again
test_dot.simulate_honeycomb(test2Dparams, multiprocess=False, verbose=1)
plt.pcolor(sweepx,sweepy,test_dot.honeycomb, shading='auto')
plt.colorbar()
plt.show()
```

```
simulatehoneycomb: 0/150
simulatehoneycomb: 0.61 [s]
```



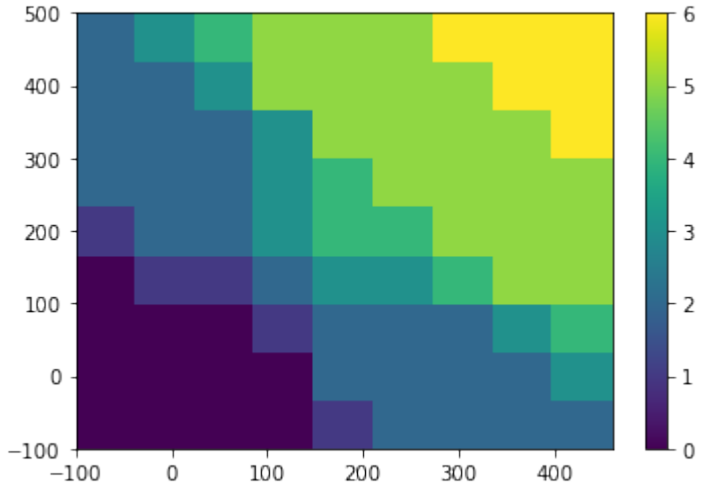
Defining your own system

[7]:
```
class TestDot(ClassicalDotSystem):
    def __init__(self, name='testdot', **kwargs):
        super().__init__(name=name, ndots=3, ngates=3, maxelectrons=2, **kwargs)

        self.makebasis(ndots=3)

        vardict = {}

        vardict["mu0_values"] = np.array([-27.0, -20.0, -25.0])  # chemical potential␣
↪at zero gate voltage
        vardict["Eadd_values"] = np.array([54.0, 52.8, 54.0])  # addition energy
        vardict["W_values"] = np.array([12.0, 5.0, 10.0])  # inter-site Coulomb␣
↪repulsion (!order is important: (1,2), (1,3), (2,3)) (lexicographic ordering)
        vardict["alpha_values"] = np.array([[1.0, 0.25, 0.1],
                                    [0.25, 1.0, 0.25],
                                    [0.1, 0.25, 1.0]])

        for name in self.varnames:
            setattr(self, name, vardict[name+'_values'])
```

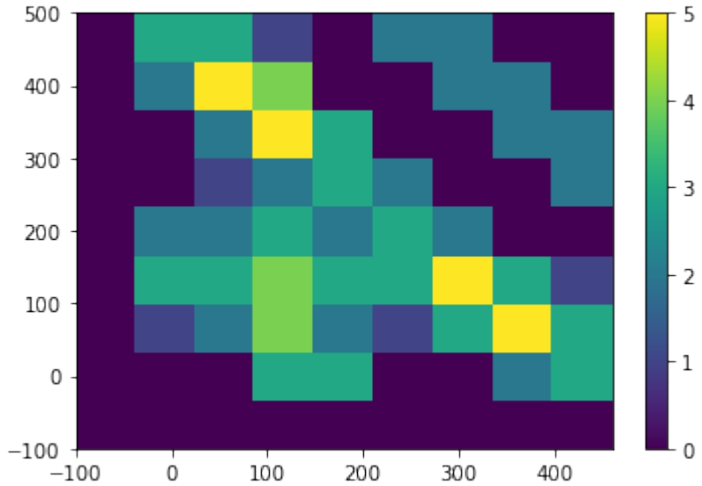
[8]:
```
test_dot_2 = TestDot()

# make a test gate plane
nx = 10
ny = 10
test2Dparams = np.zeros((test_dot_2.ngates,nx,ny))

sweepx = np.linspace(-100, 460, nx)
sweepy = np.linspace(-100, 500, ny)
xv, yv = np.meshgrid(sweepx, sweepy)
test2Dparams[0] = xv+.1*yv
xv, yv = np.meshgrid(sweepy, sweepx)
test2Dparams[2] = .1*xv+yv

# run the honeycomb simulation
test_dot_2.simulate_honeycomb(test2Dparams, multiprocess=False)
```

```
simulatehoneycomb: 0/10
simulatehoneycomb: 0.00 [s]
```

[9]:
```
plt.clf()
plt.pcolor(sweepx,sweepy,test_dot_2.hcgs.sum(axis=2), shading='auto')
plt.colorbar()
plt.show()
```



[10]:
```
plt.clf()
plt.pcolor(sweepx,sweepy,test_dot_2.honeycomb, shading='auto')
plt.colorbar()
plt.show()
```



[ ]:

Back to the *main page*.

1.3 Measurements

To perform measurements several tools are available.

1.3.1 Scan functions

For basic scanning the following functions are available:

qtt.measurements.scans.scan1D(station, scanjob)	Simple 1D scan.
qtt.measurements.scans.scan2D(station, scanjob)	Make a 2D scan and create dictionary to store on disk.
qtt.measurements.scans.scan2Dfast(station, ...)	Make a 2D scan and create qcodes dataset to store on disk.
qtt.measurements.scans.scan2Dturbo(station, ...)	Perform a very fast 2d scan by varying two physical gates with the AWG.

For more advanced measurements, write your own data acquisition loop.

1.3.2 Plotting data

For plotting a *qcodes.DataSet* one can use

qtt.data.plot_dataset(dataset[, ...])	Plot a dataset to matplotlib figure window

To automatically copy a plotted figure to PowerPoint one can add a button:

qtt.utilities.tools.create_figure_ppt_callback(fig)	Create a callback on a matplotlib figure to copy data to PowerPoint slide.

1.3.3 Parameter viewer

The ParameterViewer widget allow to view and set numeric parameters of instruments. To start the ParameterViewer pass the instruments to be monitored as the first argument.

```
parameter_viewer = qtt.gui.parameterviewer.ParameterViewer([gates, keithley1], start_
↪timer=True)
parameter_viewer.setGeometry(100,100, 400, 800)
```

qtt.gui.parameterviewer.ParameterViewer(...)	Simple class to show qcodes parameters

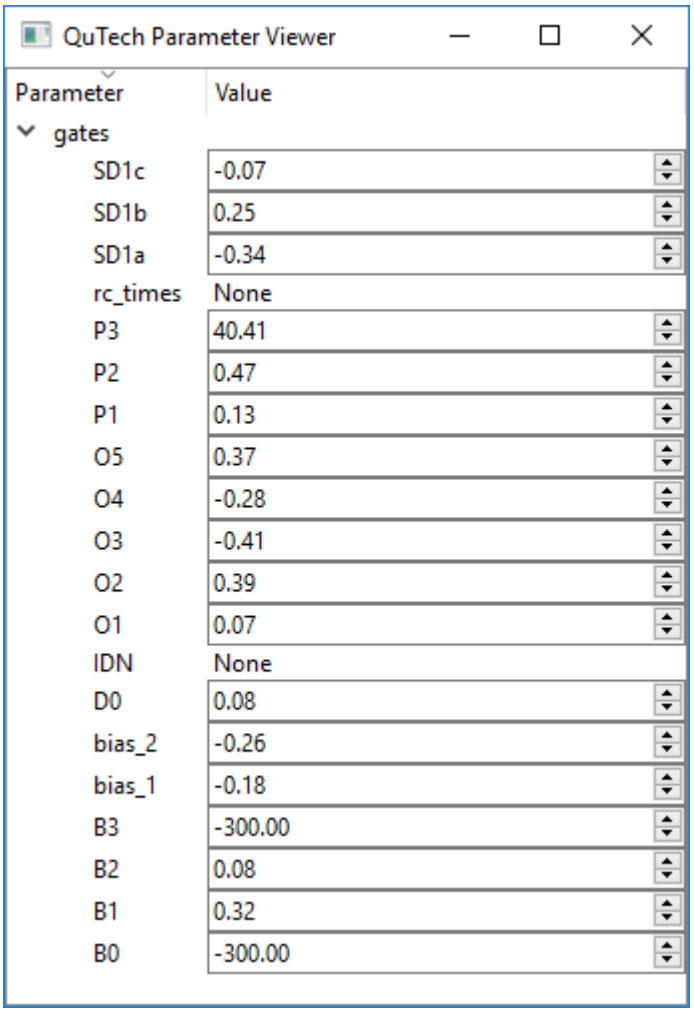


Fig. 1: ParameterViewer widget

1.3.4 VideoMode

The videomode tool can create fast charge stability diagrams using a 1D or 2D sawtooth wave. For more information see the code or the example notebooks.

`qtt.measurements.videomode.VideoMode(station)`	Controls the videomode tool.

1.3.5 Data browser

The data browser can be used to inspect recorded data.

`qtt.gui.dataviewer.DataViewer([...])`

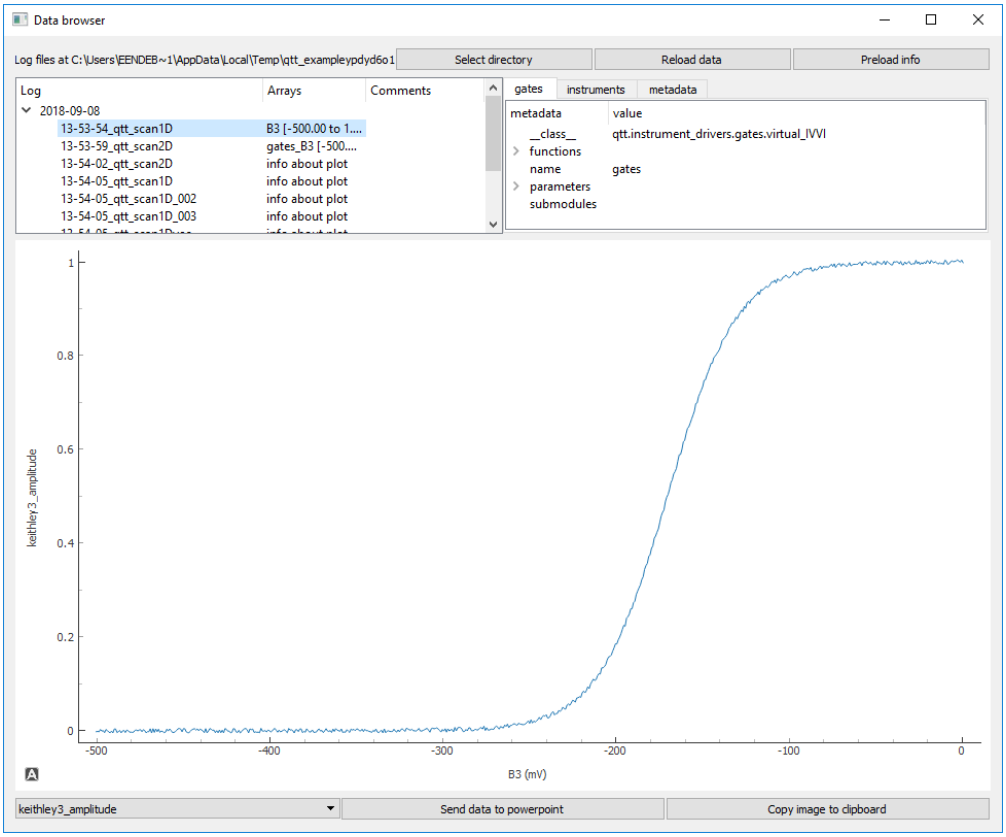


Fig. 2: DataViewer widget

1.3.6 Live plotting

Live plotting is done using a `qcodes.QtPlot` window. The window can be setup with:

`qtt.gui.live_plotting.setupMeasurementWindows([...])`	Create liveplot window and parameter widget (optional)

1.3.7 Named gates

The `VirtualDAC` object can translate gate names into the proper channels of a DAC (or multiple DACs). This is convenient because gate names are easier to work with than a number. Also when a device is controlled with multiple DACs.

`qtt.instrument_drivers.gates.VirtualDAC(...)`	This class maps the dacs of IVVI('s) to named gates.

1.3.8 Measurement control

A simple GUI to abort running measurements without interfering with the instrument communication.

`qtt.gui.live_plotting.MeasurementControl([...])`	Simple control for running measurements

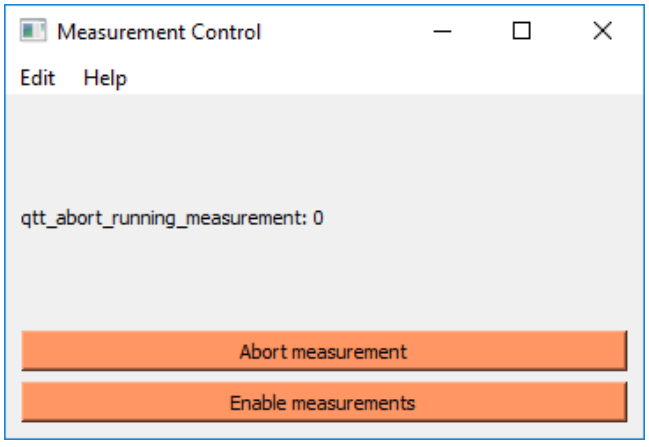


Fig. 3: Measurement control widget

1.3.9 Virtual gates

qtt.instrument_drivers.virtual_gates.virtual_gates	A virtual gate instrument to control linear combinations of gates.

1.3.10 Storing the system state

To store the system state one can store the `station.snapshot()`. In order to restore the state, it is often sufficient for spin-qubits to restore the DAC values.

qtt.measurements.storage.save_state(station)	Save current state of the system to disk
qtt.measurements.storage.load_state([tag, ...])	Load state of the system from disk
qtt.measurements.storage.list_states([verbose])	List available states of the system

For example:

```
import qtt.simulation.virtual_dot_array
station = qtt.simulation.virtual_dot_array.initialize(reinit=True, nr_dots=2,␣
↪maxelectrons=2, verbose=0)

tag = save_state(station, virtual_gates = None)
# do some work
r = load_state(station=station, tag=tag, verbose=1)
```

1.3.11 Copying data to Powerpoint

To copy a dataset or a figure window to Powerpoint, including the scan metadata one can use:

—

1.4 Algorithms

Algorithms have as input data (typically as a QCoDeS dataset or a numpy array) and parameters of the algorithm. The output is a dictionary with the main results as keys of the dictionary.

1.4.1 Example algorithms in notebooks

Most of the algorithms are documented in the *Example notebooks* section.

1.4.2 Source code

The source code for the algoritms can be found below:

`qtt.algorithms.allxy`	
`qtt.algorithms.anticrossing`	Functions to analyse anti-crossings in charge stability diagrams.
`qtt.algorithms.awg_to_plunger`	Functionality to determine the awg_to_plunger ratio.
`qtt.algorithms.bias_triangles`	Functionality to analyse bias triangles
`qtt.algorithms.chargesensor`	Analyse effect of sensing dot on fitting of tunnel barrier
`qtt.algorithms.coulomb`	Functions to fit and analyse Coulomb peaks
`qtt.algorithms.fitting`	Fitting of various models.
`qtt.algorithms.functions`	Mathematical functions and models
`qtt.algorithms.gatesweep`	Functionality to analyse pinch-off scans
`qtt.algorithms.generic`	Various functions
`qtt.algorithms.images`	Functionality to analyse and pre-process images
`qtt.algorithms.misc`	Misc algorithms
`qtt.algorithms.ohmic`	Functionality to fit scans of ohmic contacts
`qtt.algorithms.onedot`	Functionality for analysis of single quantum dots
`qtt.algorithms.pat_fitting`	Functionality to fit PAT models
`qtt.algorithms.random_telegraph_signal`	Functionality to analyse random telegraph signals
`qtt.algorithms.tunneling`	Functionality for analysing inter-dot tunnel frequencies.

Back to the *main page*.

1.5 Simulation

1.5.1 Simulation code

The package contains software to simulation quantum dot systems and plot charge stability diagrams. The basic approach is

1. Define parameters of the system (e.g. number of dots, additional energy, etc.)
2. Generate the Hamiltonian from the parameters
3. Calculate the eigenvalues and eigenstatus of the the system
4. Calculate derived properties such as the occupation numbers and charge stability diagrams.

Example notebooks: *Classical simulation of quantum dots*, *Classical simulation of triple dot*.

The documentation is in `qtt.simulation.dotsystem`

1.5.2 The virtual dot

The virtual dot is a simulation model of a linear dot array, used for testing code and learning the system. The simulation is not a good physical simulation, but is sufficient to run some of the measurement and analysis functions.

For a complete example, see the notebook *Using the virtual dot array*.

The documentation is in `qtt.simulation.virtual_dot_array`

1.5.3 Continuous-time Markov model

For details see the code in `qtt.algorithms.markov_chain`. An example notebook is *Simulation of Elzerman readout*.

1.6 Calibrations

This document contains some guidelines for creating robust and re-usable calibration and tuning routines.

1. Separate your measurement, analysis and visualization part of the code
2. Each calibration result should be a dictionary with the calibration results. A couple of fields have a special meaning:
   - `type`: Contains a string with the type of calibration. E.g. T1 or pinchoff
   - `description`: A string with a more elaborate description of the calibration procedure
   - `dataset`: Contains either the measurement data or a reference (string) to the measurement data
   - `status`: A string that can be "failed", "good" or "unknown"

   The calibration framework will automatically add the following fields to the dictionary:
   - `timestamp` (string): String with the date and time of calibration
   - `tag` (string): Contains identifier of the calibration
3. The calibration results are stored in a central database. The calibration results are identified by tags which are lists of strings, e.g. `['calibration', 'qubit1', 'T1']`.

An example of a calibration result

```
# measure a pinchoff-scan
...

# analyse the scan
$ result = analyseGateSweep(dataset)
analyseGateSweep: leftval 0.0, rightval 0.3
$ print(result)

{'_mp': 392,
 '_pinchvalueX': -450.0,
 'dataset': '2018-08-18/16-33-50_qtt_generic',
 'description': 'pinchoff analysis',
 'goodgate': True,
 'highvalue': 0.9999999999999998,
 'lowvalue': 9.445888759986548e-18,
 'midpoint': -408.0,
 'midvalue': 0.2999999999999993,
```

(continues on next page)

(continued from previous page)

```
 'pinchvalue': -458.0,
 'type': 'gatesweep',
 'xlabel': 'Sweep plunger [mV]'}
```

1.6.1 Storage

The DataSet objects are stored with the qcodes `qcodes.data.gnuplot_format.GNUPlotFormat` or the `qcodes.data.hdf5_format.HDF5FormatMetadata`. For storage of other objects we recommend to use HDF5.

1.7 Contributing

We welcome all of you to contribute to QTT, your input is valuable so that we together can continue improving it. To keep the framework useable for everyone, we ask all contributors to keep to the guidelines laid out in this section. If there are issues you cannot solve yourself or if there are any other questions, contacting the main developers of QuTech Tuning can be done via GitHub issues.

1.7.1 Code development

When contributing to QTT you will want to write a new piece of code. Please keep in mind the guidelines below to allow us to work together in an efficient way:

- Before starting contributing make a new branch on GitHub, where you can push your contributions to. You will not be able to push directly to the master branch.
- Make regular commits and clearly explain what you are doing.
- If you run into a problem you cannot solve yourself, please take up contact with our main developers via GitHub issues.
- Always include a test function in the file of your function

1.7.2 Code style

Because QTT is a framework used by many people, we need it to follow certain style guidelines. We try to follow the PEP 8 style guide, except that we allow lines to be up to 120 characters. Many editors support autopep8 that can help with coding style. Below we list some basic coding style guidelines, including examples. Please follow them when contributing to QTT. We also try to follow PEP 20.

- Docstrings are required for classes, attributes, methods, and functions (if public i.e no leading underscore).
- Document your functions before making a pull request into the QuTech Tuning main branch. An example of a well documented function is shown below:

```
def _cost_double_gaussian(signal_amp, counts, params):
        """ Cost function for fitting of double Gaussian.

        Args:
                signal_amp (array): x values of the data
                counts (array): y values of the data
                params (array): parameters of the two gaussians, [A_dn, A_up,␣
↪sigma_dn, sigma_up, mean_dn, mean_up]
```

(continues on next page)

(continued from previous page)

```
                        amplitude of first (second) gaussian = A_dn (A_up)
                        standard deviation of first (second) gaussian = sigma_dn␣
↪(sigma_up)
                        average value of the first (second) gaussian = mean_dn␣
↪(mean_up)

        Returns:
                cost (float): value which indicates the difference between the␣
↪data and the fit

        """
        model = double_gaussian(signal_amp, params)
        cost = np.linalg.norm(counts - model)

        return cost
```

- Since we are dealing with code in development:
  - For methods implementing an algorithm return a dictionary so that we can modify the output arguments without breaking backwards compatibility
  - Add arguments `fig` or `verbose` to function to provide flexible analysis and debugging

1.7.3 Uploading code

To upload code use git commit and git push. For the qtt repository always make a branch first. After uploading a branch one can make a pull request which will be reviewed for inclusion in QTT by our main developers. If the code is up to standards we will include it in the QTT repository.

1.7.4 Bugs reports and feature requests

If you don't know how to solve a bug yourself or want to request a feature, you can raise an issue via github's issues. Please first search for existing and closed issues, if your problem or idea is not yet addressed, please open a new issue.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

LICENSE

CHAPTER

FOUR

CONTRIBUTORS

- Pieter Eendebak (*peendebak*)
- CumulonimbusCalvus (*CumulonimbusCalvus*)
- Sjaak van Diepen (*CJvanDiepen*)
- JP Dehollain (*jpdehollain*)
- lgnjanssen (*lgnjanssen*)
- dpfranke (*dpfranke*)
- fvanriggelen (*fvanriggelen*)
- qSaevar (*qSaevar*)
- AM (*azwerver*)
- Christian Volk (*Christian-Volk*)
- Laurens Janssen (*Velkit*)
- Andrea Corna (*YakBizzarro*)
- Bruno (*brunobuijtendorp*)
- takafumifujita (*takafumifujita*)
- NoraFahrenfort (*NoraFahrenfort*)
- Steven (*spmvg*)
- Laurens Hagendoorn (*TheLaurens*)
- tfwatson15 (*tfwatson15*)

and many more...