QCoDeS in DiCarlo-Lab

July 14 2016 – Adriaan Rol (edited for PycQED)

Questions to be answered in presentation

- 1) what generally does your experiment look like (devices and what properties of them you're controlling and measuring)
- 2) what are the instruments you're using
- 3) what kind of data do you generate from those instruments
- 4) what do you need to do with that data afterward (or in the middle) to interpret it

I was imagining 1 hour for each presenter, that would encompass this intro, a presentation / demo of how they've been using (and contributing to :slightly_smiling_face:) qcodes so far, along with particular challenges / needs for the near future, and discussion.

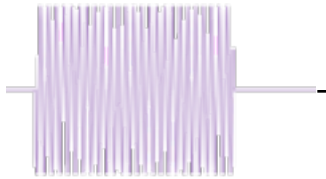
A SimpleTransmon Experiment

Flux tunable
qubit frequency

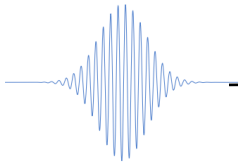
Flux bias

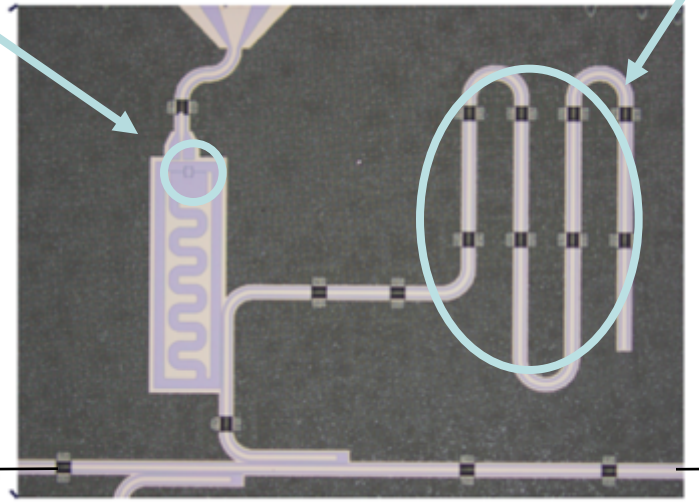
Qubit-state dependent
resonator frequency

MW-RO tone (~300ns)

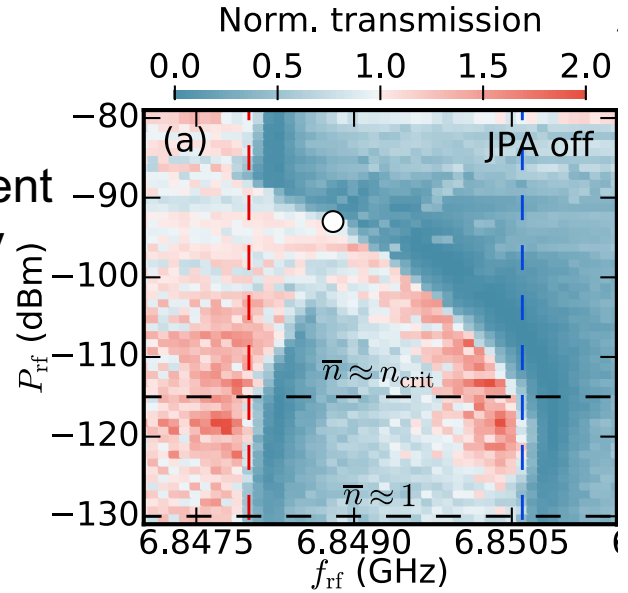


MW-Qubit control pulse (20ns)



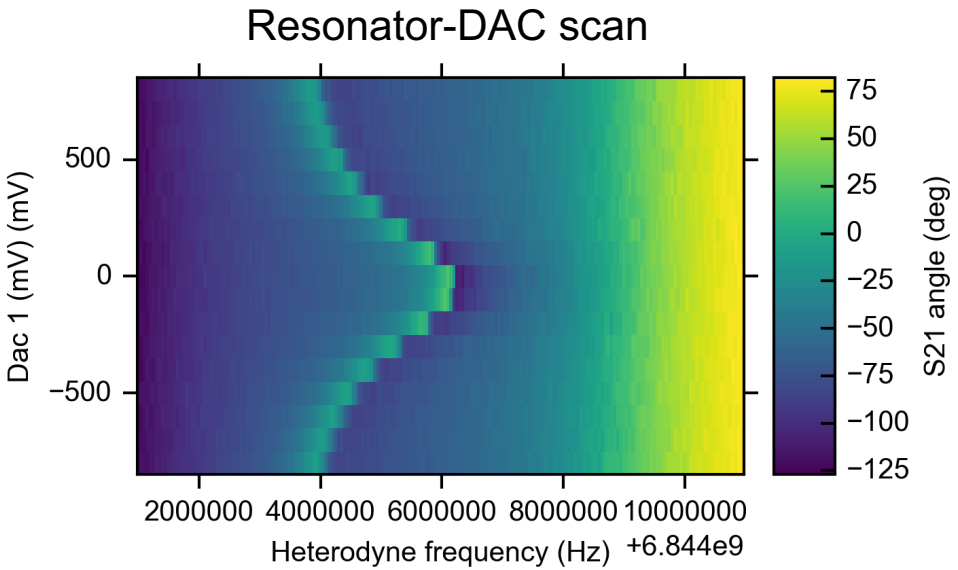


Transmitted signal to
digitizer (ATS or FPGA)

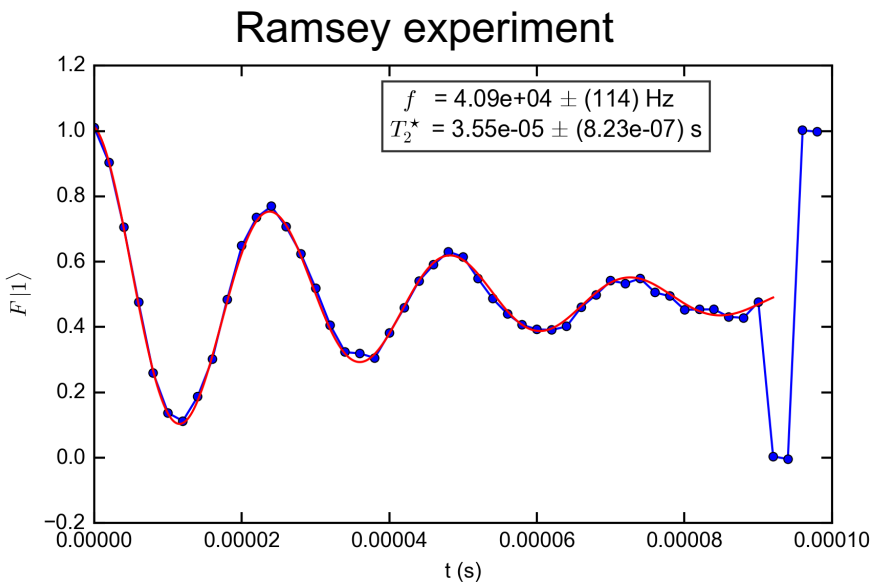


Two types of circuit QED experiments

Continuous-Wave (CW)



Time-Domain (TD)



Advances in Tuning

What is limiting us?

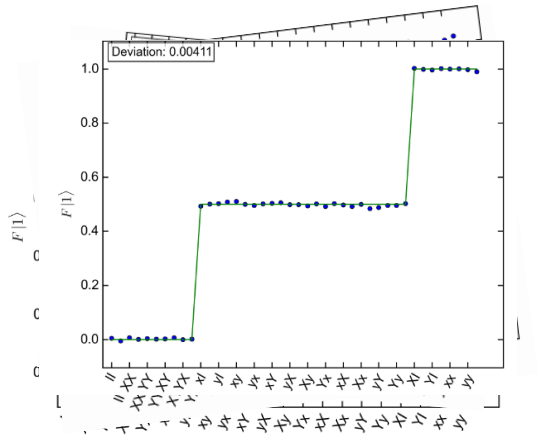
- Development time
- Measurement time
  - Time per datapoint
  - Number of datapoints

Hand Tuning ~15 mins

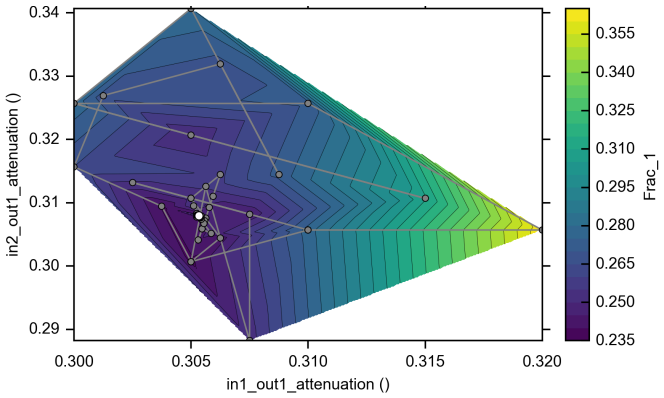
Procedural tuning ~5 mins

Adaptive tuning ~10mins

Restless tuning ~1min




```
nr_cliffords = [2, 4, 8, 16, 30, 60, 100, 200, 300, 400, 600, 800, 1200]
qubit.measure_ssro()
qubit.measure_T1(np.linspace(0, 120e-6, 41))
a = ma.T1_Analysis(label='T1', auto=True)
T1 = a.T1
qubit.find_pulse_amplitude(amps=np.linspace(-.5, .5, 31),
                           max_n=1, update=True,
                           take_fit_I=True)
qubit.find_frequency(method='ramsey', steps=[30, 100, 300], update=True)
qubit.measure_motoi_XY(motzois=np.linspace(-0.4, -0.2, 21))
qubit.find_pulse_amplitude(amps=np.linspace(-.5, .5, 31),
                           N_steps=[3, 7, 19], max_n=100, take_fit_I=True)
qubit.measure_allxy()
qubit.measure_randomized_benchmarking(nr_cliffords=nr_cliffords)
ma.RandomizedBenchmarking_Analysis(close_main_fig=False, T1=T1,
                                   pulse_delay=qubit.pulse_delay.get(),
                                   label='RB')
```



Reed, M. Entanglement and Quantum Error Correction with Superconducting Qubits. PhD. Thesis Yale (2013).

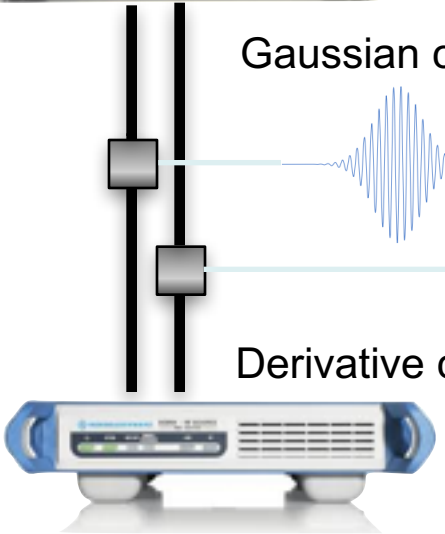
Kelly, J. et al. Optimal quantum control using randomized benchmarking. Phys. Rev. Lett. 112, 1–5 (2014).

Adaptive tuning

Fixed AWG sequence:
Randomized Benchmarking at fixed number of Cliffords

Gaussian component

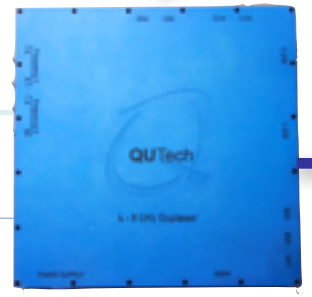
Derivative component



Qubit drive tone

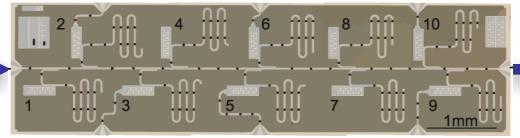
Python optimizer
Set parameters,
Minimize error fraction

QuTech CBox FPGA Digitizer
Count fraction of erroneous results



QuTech VSM
Fast control of pulse parameters

Device



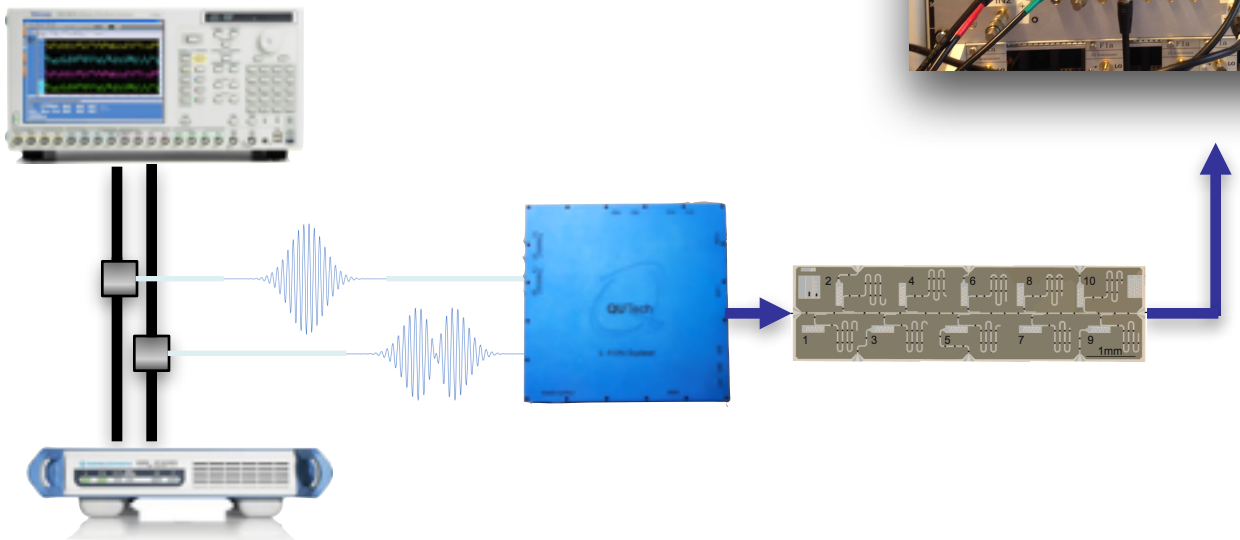
1.6s/iteration
(8000 shots)

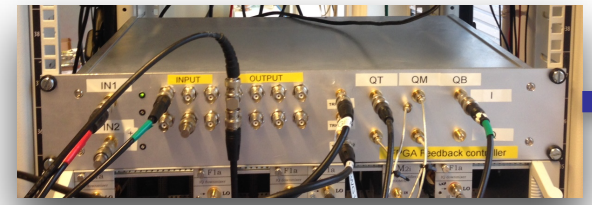
Asaad, S. *et al. Independent, extensible control of same-frequency superconducting qubits by selective broadcasting.* arXiv 1508.06676 (2015).

Data flows involved

Example experiments

- Spectroscopy
- Simple Time domain (e.g. Ramsey)
- Adaptive tuning (e.g. numerical tuning)
- QEC cycle





- Raw-transient(s) (1D-array)
- Demodulated and integrated transient (I,Q) tuple of floats
- Declared state (bool or int)

- Histogrammed and/or averaged

N = number of (demodulation) channels
M = number of repetitions
K = number of unique measurements in experiment
L = Integration lenght (in samples)

Software

QCODES + PYCQED

Goals of Software Development

Provide the tools required to control (increasingly) complex experiments

- Extensible — code is modular and reusable
- Easy to use — minimal number of concepts
- Automatable — closed loop between experiment and analysis

Software project started one and a half year ago

- Has since replaced LabView on all DCL setups
- Led to our participation in the Microsoft QCoDeS collaboration
- Used in latest publications

QUTech

QuSurf

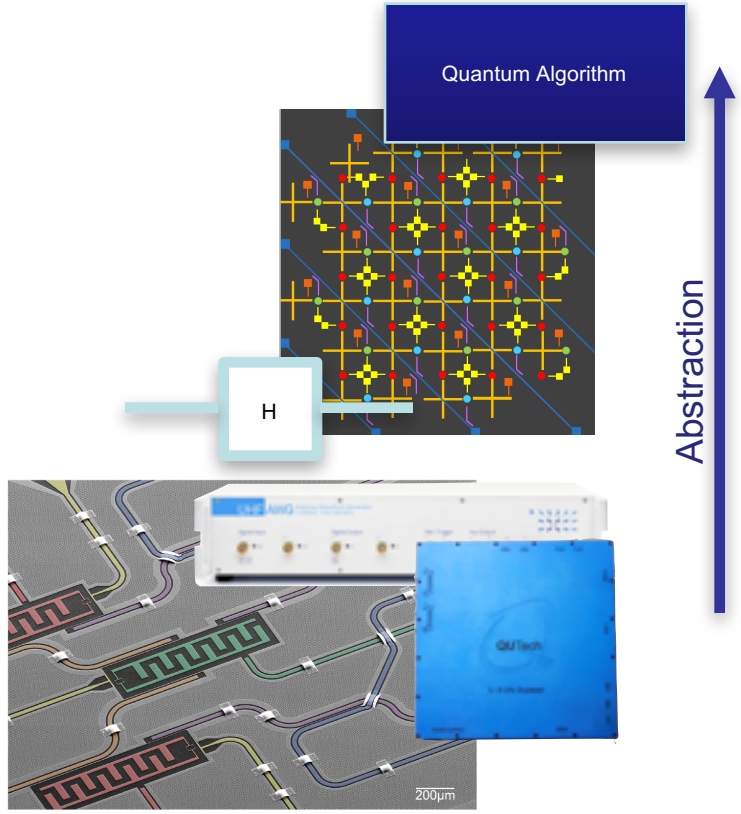
Building layers of abstraction

High Level - Quantum Compiler
- Solid

Middle level – System specific routines
- PycQED

Low level – Experiment control software
- QCoDeS

Microsoft



We should collaborate on those parts of our system that are common

- We are currently using QCoDeS as a replacement for QTLab (Instruments and plotting)
- We are committed to using QCoDeS-PycQED combination for all new experiments
- We want to replace the parts of PycQED that are superseded by QCoDeS

A minimal set of concepts

- Instrument
- Parameter
- Loop
- Analysis

Any experiment consists of a Loop

Loop:

1. Some parameter(s) is/are varied
2. Some parameter is measured
3. Data is saved and analyzed

The **Loop** takes care of standardised datasaving, logging and live plotting

Example: Heterodyne spectroscopy experiment

PycQED syntax

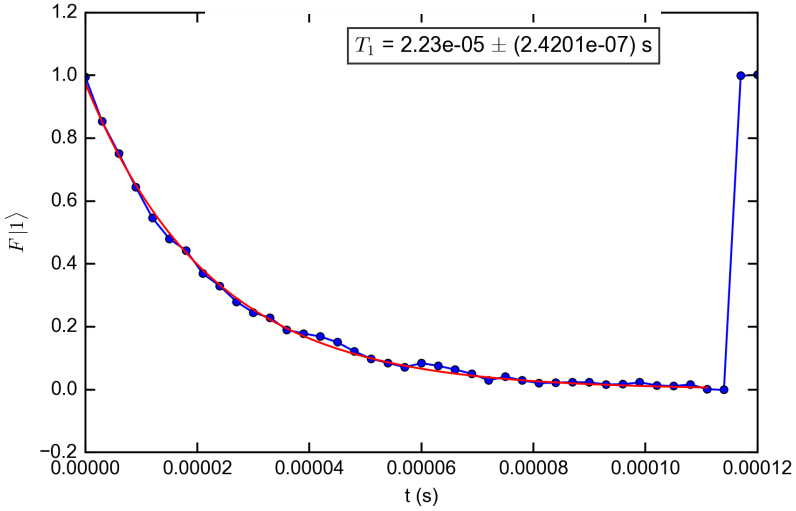
```
MC.set_sweep_function(source.frequency)
MC.set_sweep_points(np.linspace(start, stop, steps))
MC.set_detector_function(HeterodyneDetector())
MC.run(name='Heterodyne')
ma.MeasurementAnalysis()
```

QCodes syntax

```
data = qc.Loop(source.frequency[start:stop:step]).each(
    Heterodyne.IQ).run(name='Heterodyne')
qc.QTPlot(data.IQ)
```

A Loop can also be over a “hard” parameter...

Example: T1 experiment



A **hard**(ware) Loop would be a measurement in which the hardware and not python is in control of the experiment

PycQED syntax

```
MC.set_sweep_function(awg_swf.T1(
                        pulse_pars=pulse_pars, RO_pars=RO_pars))
MC.set_sweep_points(np.linspace(t_start, t_stop, t_step))
MC.set_detector_function(det.CBox_integrated_average_detector(CBox, AWG))
MC.run(name='T1_qubit_1')
a = ma.T1_Analysis(auto=True, close_fig=False)
```

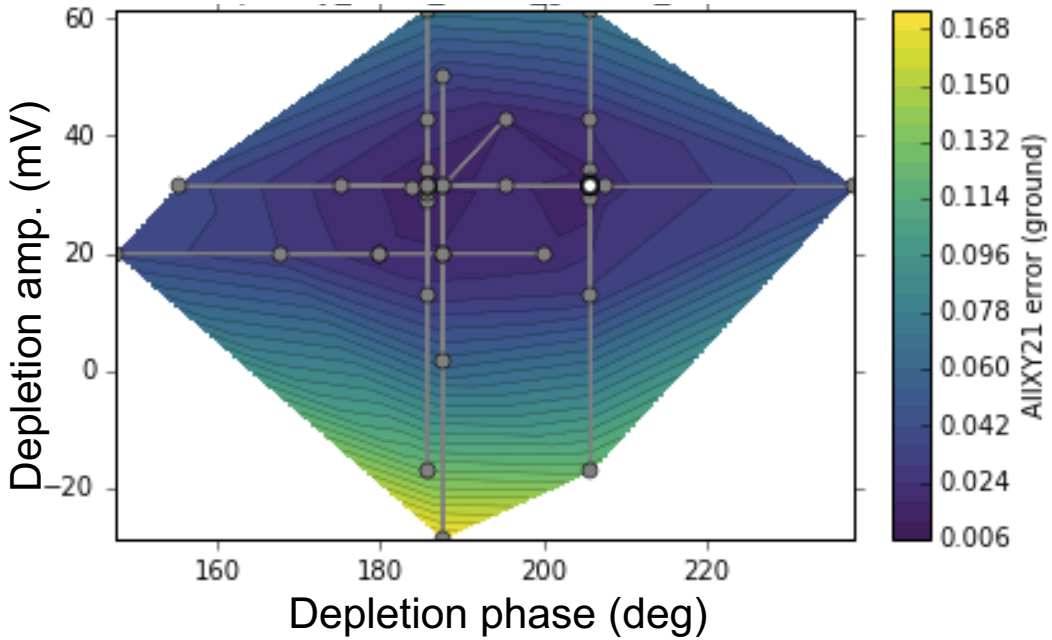
Desired QCodes syntax

```
data = qc.Loop(T1_seq(pulse_pars, RO_pars)[t_start, t_stop, t_step]).each(
    CBox.integrated_avg_result()).run('T1_qubit_1')
qca.T1_Analysis(data)
```

multi dimensional parameter help wanted question
#243 opened 21 days ago by nataliejpg

... or adaptive

Powell optimization of photon depletion pulse



Example depletion pulse optimization

```
from scipy.optimize import fmin_powell

ad_func_pars = {'adaptive_function': fmin_powell,
                'x0': start_val,
                'ftol': f_tol,
                'xtol': x_tol,
                'direc': direc, # Specifies initial steps
                'minimize': minimize}
MC.set_adaptive_function_parameters(ad_func_pars)
MC.set_sweep_functions([CB_swf.CBox_CLEAR_phase_b1(reload_pulses=True),
                        CB_swf.CBox_CLEAR_amplitude_b1(reload_pulses=False)])
MC.set_detector_function(cdet.Photon_number_detector(
    driving='AWG', CLEAR_double=False, optimize_on_ground=True,
    sequence='AllXY21'))
MC.run(name='Num_CLEAR_opt', mode='adaptive')
ma.OptimizationAnalysis(auto=True, label='Num_CLEAR_opt')
```

The adaptive loop can use **any adaptive function** on the results of **anything we can quantify**/measure

Bultink, C. C. *et al. Active resonator reset in the nonlinear dispersive regime of circuit QED*. ArXiv: 1604.00916 (2016)

An Instrument is typically a thin standardized wrapper around an instrument driver

- Provides standardization
- Provides logging (snapshot)
- Provides interface for the loop through parameters

An **instrument** is a container for parameters

Example: Rhode & Schwarz Microwave source

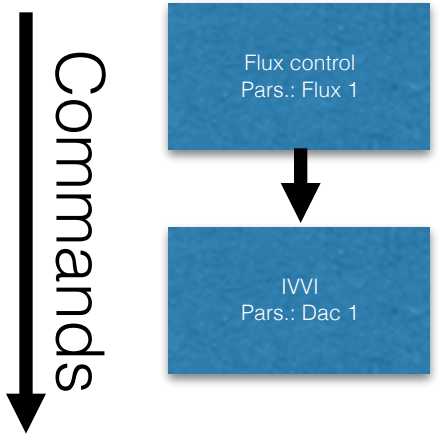
```python
from qcodes import VisaInstrument, validators as vals

class RohdeSchwarz_SGS100A(VisaInstrument):
    '''
    This is the qcodes driver for the Rohde & Schwarz SGS100A signal generator
    '''
    def __init__(self, name, address, **kwargs):
        super().__init__(name, address, **kwargs)

        self.add_parameter(name='frequency',
                           label='Frequency',
                           units='Hz',
                           get_cmd='SOUR:FREQ' + '?',
                           set_cmd='SOUR:FREQ' + ' {:.2f}',
                           get_parser=float,
                           vals=vals.Numbers(1e9, 20e9))
        self.add_function('reset', call_cmd='*RST')
        self.add_function('run_self_tests', call_cmd='*TST?')

        self.connect_message()

    def parse_on_off(self, stat):
        if stat.startswith('0'):
            stat = 'Off'
        elif stat.startswith('1'):
            stat = 'On'
        return stat

    def set_status(self, stat):
        if stat.upper() in ('ON', 'OFF'):
            self.write(':OUTP:STAT %s' % stat)
        else:
            raise ValueError('Unable to set status to %s, ' % stat +
                             'expected "ON" or "OFF"')
```

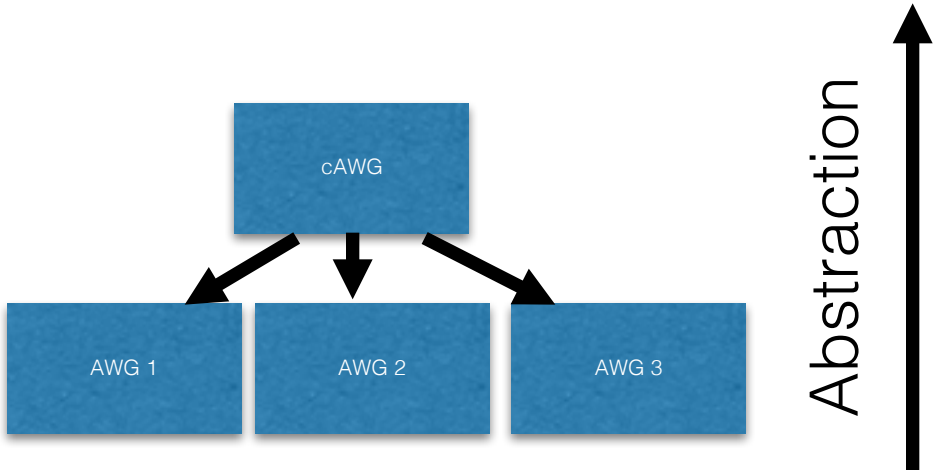
Meta-instruments allow layers of abstraction and modularity

Example: Flux-control
Converts fluxes to dac-voltages
using a calibrated correction matrix

Example: Composite AWG
Acts as a single multi-channel AWG
but talks to underlying instruments

A **meta-instrument** can contain other instruments but acts like a regular instrument





See also: virtual instruments for simulations by Damaz

An analysis object contains routines for common analyses

Example: Rabi Analysis

```python
class Rabi_Analysis(TD_Analysis):
    def __init__(self, label='Rabi', **kw):
        kw['label'] = label
        kw['h5mode'] = 'r+'
        super().__init__(**kw)

    def run_default_analysis(self, close_file=True, **kw):
        self.get_naming_and_values()
        self.fit_data(**kw)
        self.make_figures(**kw)
        if close_file:
            self.data_file.close()
        return self.fit_res

    def make_figures(self, **kw):...

    def fit_data(self, print_fit_results=False, **kw):...
```

- Inherits from a base analysis class
- standardized matplotlib plotting functions
- fitting using lmfit module with custom models and guess functions in our own module
- data extraction using single function
- analysis_tools module with convenience functions

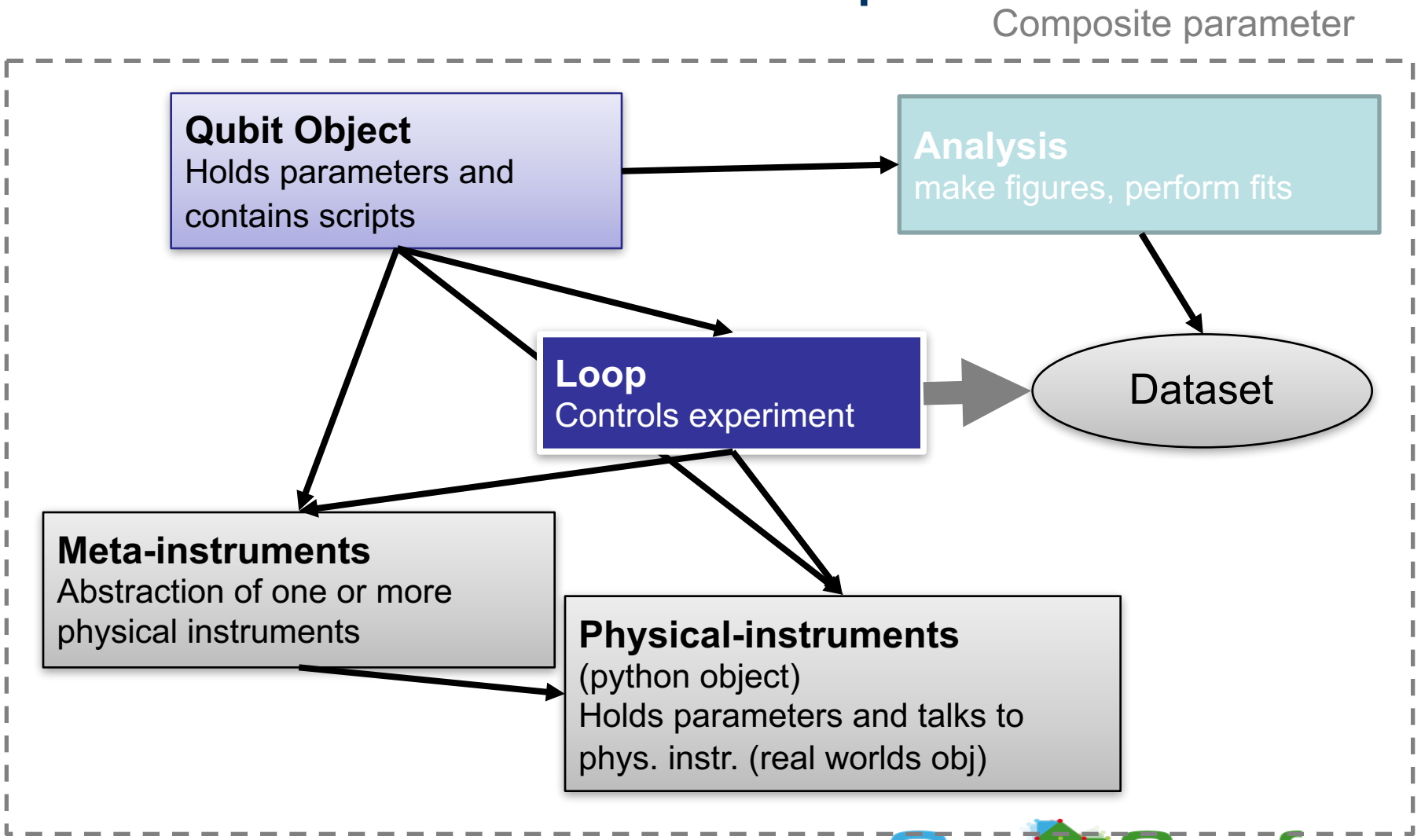
```python
def get_timestamps_in_range(timestamp_start, timestamp_end=None,
                            label=None, exact_label_match=True):
```

```python
def compare_instrument_settings(analysis_object_a, analysis_object_b):
```

The Qubit object is a special instrument that contains small measurement scripts

Typical script

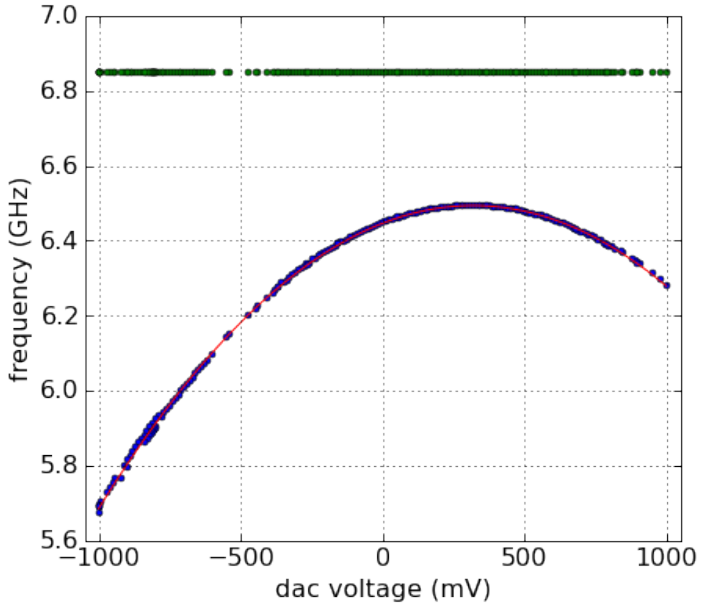
1. Prepare by setting parameters
2. Execute Loop
3. Analyze dataset

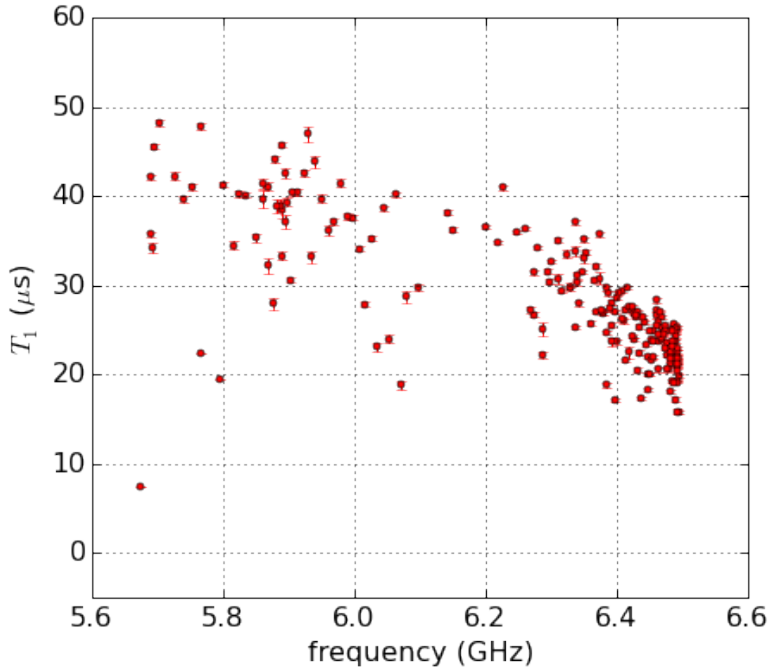


A composite parameter allows easy automation of otherwise complex measurements

Example: T1 vs qubit frequency

```
MC.set_sweep_function(swf.Flux_Control_mV(dac_channel))
MC.set_sweep_points(virtual_dac_voltages)
MC.set_detector_function(cdet.T1_Detector(qubit))
MC.run(measurement_name='T1_Mux0_Bottom_Qubit',
       debug_mode=False)
```





Consists of :

1. Finding the resonator
2. Finding the qubit
3. Calibrating the pulse amplitude
4. Performing a T1 measurement

NB! altough it is currently possible to put a Loop within a Loop this requires rewriting +testing old code. It is also not possible to get the data out midway for analysis.

AWG sequencing made easy

Specify experiment as a list of operations/gates

- Sequencer uses pulse parameters to generate correct waveforms (including modulation and phase tracking)
- Compatible with different types of pulses (e.g. 4-channel VSM pulses and conventional 2-channel pulses)
- Allowed rapid implementation of Gate Set Tomography
- Currently only for single qubit pulses

```
171  def AllXY_seq(pulse_pars, RO_pars, double_points=False,
172                verbose=False): ...
185      pulses = get_pulse_dict_from_pars(pulse_pars)
186
187      pulse_combinations = [['I', 'I'], ['X180', 'X180'], ['Y180', 'Y180'],
188                            ['X180', 'Y180'], ['Y180', 'X180'],
189  ...                       ['X180', 'I'], ['Y180', 'I'], ['X90', 'X90'],
194                            ['Y90', 'Y90']]
195  ...
199      for i, pulse_comb in enumerate(pulse_combinations):
200          pulse_list = [pulses[pulse_comb[0]],
201                        pulses[pulse_comb[1]],
202                        RO_pars]
203          el = multi_pulse_elt(i, station, pulse_list)
204          el_list.append(el)
205          seq.append_element(el, trigger_wait=True)
206
207      station.components['AWG'].stop()
208      station.pulsar.program_awg(seq, *el_list, verbose=verbose)
209      return seq_name
```


Related work:

Fu. X, et al., Quantum Instruction Set Architecture (QISA), in preparation (2016)

Häner, T., et al. Software Methodology for Compiling Quantum Programs. ArXiv: 1604.01401 (2016).

AWG sequencing made easy

Pulses(dict) containing pulses by “name” (X180 etc)

Pulse_pars (dict)

AWG segment is made by adding pulses (by key) in a for loop

Pulse-definition is contained in pulse-lib (currently only RO pulse and SSB Drag pulse)

```
MC.set_sweep_function(awg_swf.Randomized_Benchmarking(
    pulse_pars=self.pulse_pars, RO_pars=self.RO_pars,
    nr_cliffords, nr_seeds=nr_seeds))
pulses = {'I': deepc
          'X180': de
          'mX180': c
          'X
          'm
          'Y
          'm
          'Y
          'm
```

```
149     def get_pulse_pars(self):
150         self.pulse_pars = {
151             'I_channel': self.pulse_I_channel.get(),
152             'Q_channel': self.pulse_Q_channel.get(),
153             'amplitude': self.amp180.get(),
154             'sigma': self.gauss_sigma.get(),
155             'nr_sigma': 4,
156             'motzoi': self.motzoi.get(),
157             'mod_frequency': self.f_pulse_mod.get(),
158             'pulse_separation': self.pulse_separation.g
159             'phase': 0,
160             'pulse_type': 'SSB_DRAG_pulse'}
```

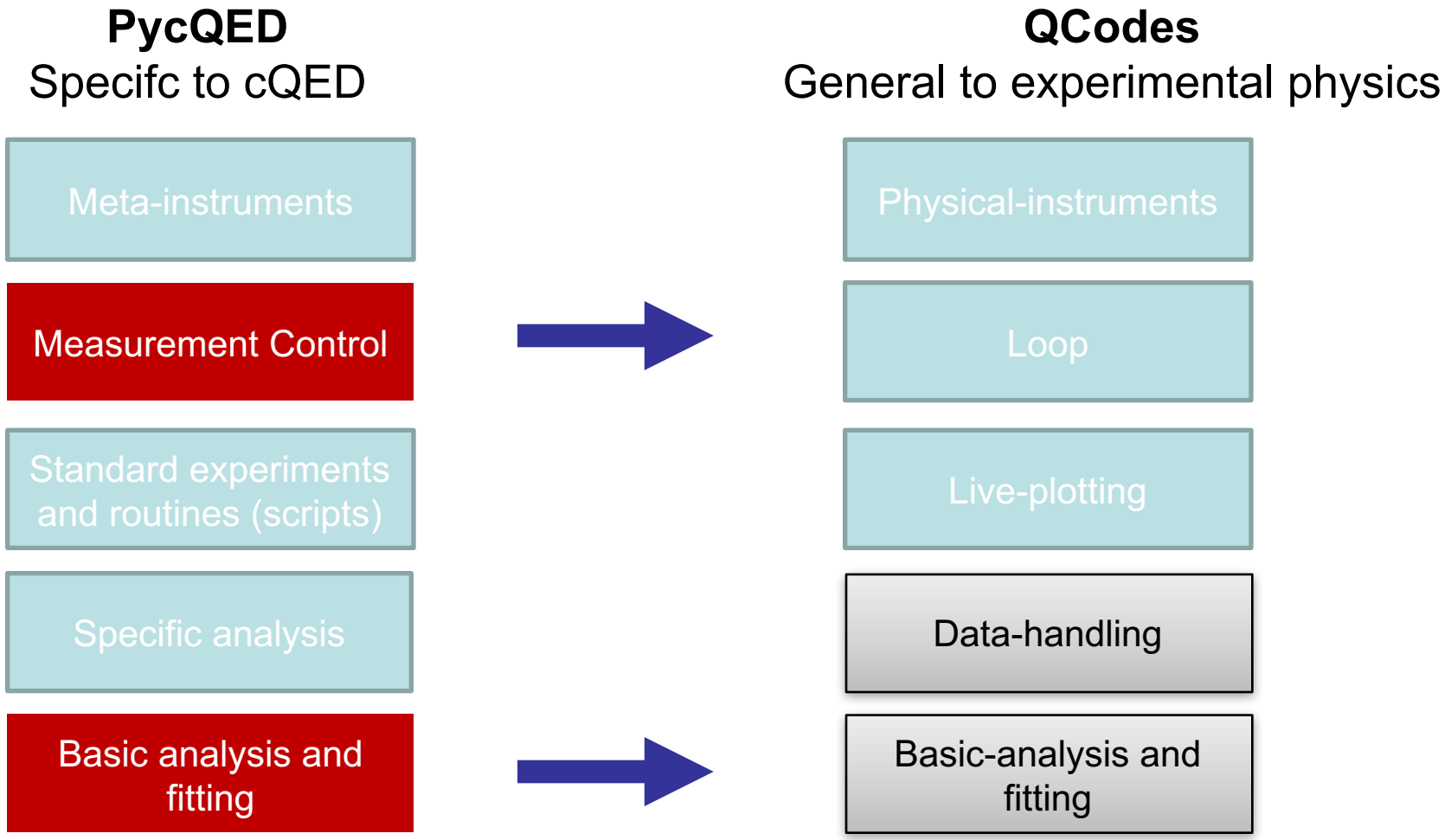
- Includes fix-point correction
- Automated sideband modulation
- Allows for pulse definitions including markers
- In-prinicple extensible to arbitrary nr of channels

```
else:
    cl_seq = rb.randomized_benchmarking_sequence(n_cl)
    pulse_keys = rb.decompose_clifford_seq(cl_seq)
    pulse_list = [pulses[x] for x in pulse_keys]
    pulse_list += [RO_pars]
    el = multi_pulse_elt(i, station, pulse_list)
el_list.append(el)
seq.append_element(el, trigger_wait=True)
```

QUTech

Surf

We should collaborate on those parts of our system that are common



QCoDeS in DiCarlo-Lab

July 14 2016 – Adriaan Rol