scanning-squid Documentation

Logan Bishop-Van Horn

Aug 18, 2019

DOCUMENTATION:

1 Getting Started **3**
- 1.1 Installation . . . 3
  - 1.1.1 Windows . . . 3
  - 1.1.2 Mac . . . 3

2 Microscope **5**
- 2.1 Microscope Components . . . 6
  - 2.1.1 Attocubes . . . 6
  - 2.1.2 Scanner . . . 7
  - 2.1.3 SQUID . . . 9
  - 2.1.4 DAQ . . . 9
  - 2.1.5 Others Instruments . . . 11

3 Configuration Files **17**
- 3.1 Microscope Configuration . . . 17
- 3.2 Measurement Configuration . . . 19

4 Physical Units **23**

5 Measurements **25**
- 5.1 Capacitive touchdown . . . 25
- 5.2 Approaching the Sample . . . 27
- 5.3 Acquiring a Surface . . . 27
- 5.4 Scanning . . . 28

6 Plots **29**
- 6.1 ScanPlot . . . 29
- 6.2 TDCPlot . . . 30

7 Utility Functions & Classes **33**

8 DataSet Example **39**
- 8.1 Plotting . . . 39
- 8.2 Explore DataSet metadata . . . 40
- 8.3 Convert DataSet to arrays with real units . . . 42
- 8.4 Export data to a MAT file: . . . 43

9 Typical Workflow **45**
- 9.1 Preliminary Steps . . . 45
- 9.2 Initialize the Microscope . . . 45
- 9.3 Load the Measurement Configuration . . . 45

9.4 Approach the Sample 45
9.5 Acquire a Surface 46
9.6 Scan Over the Plane 46
9.7 Move Around the Sample 46

Python Module Index **47**

Index **49**

scanning-squid is an instrument control and data acquisition package for scanning SQUID (Superconducting QUantum Interference Device) microscopy developed in the Moler Group at Stanford University. It is based on the QCoDeS data acquisition framework.

To install scanning-squid, see *Getting Started*.

Contact: Logan Bishop-Van Horn (logan.bvh [at] gmail [dot] com).

CHAPTER

ONE

GETTING STARTED

scanning-squid is an instrument control and data acquisition package for scanning SQUID (Superconducting QUantum Interference Device) microscopy. It is based on the QCoDeS framework, and is designed to run in either a standalone Jupyter Notebook, or in Jupyter Lab.

Contact: Logan Bishop-Van Horn (logan.bvh [at] gmail [dot] com).

1.1 Installation

We recommend that you set up a `conda env` in which to run scanning-squid by following the steps below. This will install all of the packages on which scanning-squid depends.

1.1.1 Windows

- Download and install Anaconda (the latest Python 3 version).
- Download environment.yml from the scanning-squid repository
- Launch an Anaconda Prompt (start typing anaconda in the start menu and click on Anaconda Prompt)
- In the Anaconda Prompt, navigate to the directory containing environment.yml (`cd <path-to-directory-containing-environment-file>`)
- Run the following two commands in the Anaconda Prompt:
  - `conda env create -f environment.yml`
  - `activate scanning-squid`

1.1.2 Mac

- Download and install Anaconda (the latest Python 3 version).
- Download environment.yml from the scanning-squid repository
- Launch a Terminal.
- In the Terminal, navigate to the directory containing environment.yml (`cd <path-to-directory-containing-environment-file>`)
- Run the following two commands in the Terminal:
  - `conda env create -f environment.yml`
  - `source activate scanning-squid`

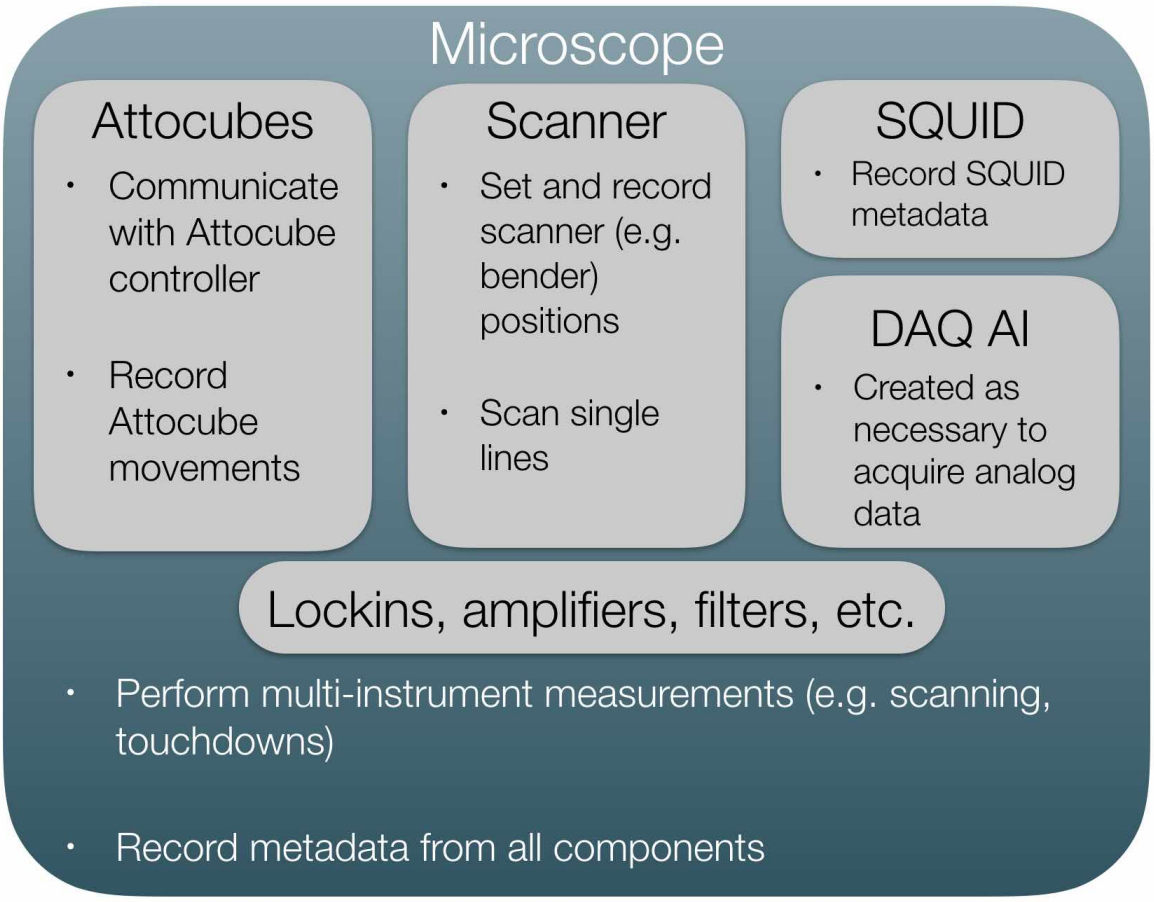
After cloning the scanning-squid repository, to run scanning-squid from a Windows (Mac) machine, open the Anaconda Prompt (Terminal) and run `activate scanning-squid` (`source activate scanning-squid`), and launch a `jupyter notebook` or `jupyter lab`. Alternatively, on a Windows machine, you can launch a Jupyter Notebook from the start menu (just make sure the Jupyter Notebook icon says "(scanning-squid)" next to it).

CHAPTER

TWO

MICROSCOPE

A physical scanning SQUID microscope is represented by an instance of the *microscope.microscope.Microscope* class (or liklely one of its subclasses, like *microscope.susceptometer.SusceptometerMicroscope*). A `Microscope` is a `qcodes.station.Station`, to which we can attach components (instances of `qcodes.Instrument` or its subclasses) whose metadata we would like to save during a measurement.



During a typical measurment (scan or capacitive touchdown), all settings/parameters of all instruments attached to the microscope are automatically queried and recorded, forming a "snapshot" of the microscope at the time of the measurement. This snapshot is saved along with a raw data file and a MATLAB .mat file containing data converted

to real units. See /examples/ScanSurfaceExample.ipynb for a demonstration of scanning a sample surface with a `microscope.susceptometer.SusceptometerMicroscope`.

2.1 Microscope Components

A `Microscope` is made up of several `qcodes.Instrument` objects used to control and acquire data from physical instruments.

2.1.1 Attocubes

The `instruments.atto.AttocubeController` interfaces via GPIB with the Attocube hardware (e.g. an ANC300 controller). It enforces stepping voltage limits based on the current temperature mode (either `'LT'` or `'RT'`).

class instruments.atto.**AttocubeController**(*atto_config: Dict, temp: str, ureg: Any, timestamp_fmt: str, **kwargs*)

Base class for Attocube controller instrument.

Parameters

- **atto_config** – Configuration dict loaded from microscope configuration json file.
- **temp** – 'LT' or 'RT', depending on whether the attocubes are at room temp. or low temp.
- **ureg** – pint UnitRegistry instance, used for managing physical units.
- **timestamp_fmt** – Timestamp format to be used for logging.
- ****kwargs** – Keyword arguments to be passed to VisaInstrument constructor.

ask_raw(*cmd: str*) → str

Query instrument with cmd and return response.

Parameters **cmd** – Command to write to controller.

Returns

response Response of Attocube controller to the query *cmd*.

Return type str

check_response(*response: str*) → None

Raise an exception if controller responds with 'ERROR'.

Parameters **response** – Response from controller.

clear_instances()

Remove instrument instance.

step(*axis: Union[int, str], steps: int*) → None

Performs a given number of Attocube steps. steps > 0 corresponds to stepu (up), steps < 0 corresponds to stepd (down).

Parameters

- **axis** – Either axis label (str, e.g. 'y') or index (int, e.g. 2)
- **steps** – Number of steps to perform (>0 for 'u', <0 for 'd')

stop(*axis: Union[int, str]*) → None

Stops all motion along axis and then grounds the output.

Parameters **axis** – Either axis label (str, e.g. 'y') or index (int, e.g. 2)

write_raw(*cmd: str*) → str

Write cmd and don't wait for response.

Parameters **cmd** – Command to write to controller.

class instruments.atto.**ANC300**(*atto_config: Dict, temp: str, ureg: Any, timestamp_format: str, **kwargs*)

ANC300 Attocube controller instrument.

initialize() → None

Initialize instrument with parameters from self.metadata.

class instruments.atto.**ANC150**(*atto_config: Dict, temp: str, ureg: Any, timestamp_format: str, **kwargs*)

ANC150 Attocube controller instrument.

initialize() → None

Initialize instrument with parameters from self.metadata.

2.1.2 Scanner

The `scanner.Scanner` represents the x, y, z scanner that controls the relative motion between the sample and the SQUID. It enforces voltage limits based on the current temperature mode (either `'LT'` or `'RT'`). A `scanner.Scanner` instance creates and closes nidaqmx DAQ analog ouput and input tasks as needed to drive the scanner and sense its current position.

class scanner.**Scanner**(*scanner_config: Dict[str, Any], daq_config: Dict[str, Any], temp: str, ureg: Any, **kwargs*)

Controls DAQ AOs to drive the scanner.

Parameters

- **scanner_config** – Scanner configuration dictionary as defined in microscope configuration JSON file.
- **daq_config** – DAQ configuration dictionary as defined in microscope configuration JSON file.
- **temp** – 'LT' or 'RT' - sets the scanner voltage limit for each axis based on temperature mode.
- **ureg** – pint UnitRegistry, manages units.

check_for_td(*tdc_plot: Any, data_set: Any, counter: Any*) → None

Check whether touchdown has occurred during a capacitive touchdown.

Parameters

- **tdc_plot** – plots.TDCPlot instance, which contains current data and parameters of the touchdown Loop.
- **data_set** – DataSet containing capacitance data generated by Loop.
- **counter** – utils.Counter intance to keep track of which point in the Loop we're at.

clear_instances()

Clear scanner instances.

control_ao_task(*cmd: str*) → None

Write commands to the DAQ AO Task. Used during qc.Loops.

Parameters **cmd** – What you want the Task to do. For example, self.control_ao_task('stop') is equivalent to self.ao_task.stop()

get_pos() → numpy.ndarray

Get current scanner [x, y, z] position.

Returns

pos Array of current [x, y, z] scanner voltage.

Return type numpy.ndarray

get_td_height (*tdc_plot: Any*, *task: bool = True*) → None

If a touchdown has occurred, finds the z voltage at which it occurred.

Parameters

- **tdc_plot** – plots.TDCPlot instance containing data from touchdown.
- **task** – True if get_td_height is being called as a qcodes Task (no return value allowed). Default True.

goto (*new_pos: List[float], retract_first: Optional[bool] = False, speed: Optional[str] = None, quiet: Optional[bool] = False*) → None

Move scanner to given position. By default moves all three axes simultaneously, if necessary.

Parameters

- **new_pos** – List of [x, y, z] scanner voltage to go to.
- **retract_first** – If True, scanner retracts to value determined by self.temp, then moves in the x,y plane, then moves in z to new_pos. Default: False.
- **speed** – Speed at which to move the scanner (e.g. '2 V/s') in DAQ voltage units. Default set in microscope configuration JSON file.
- **quiet** – If True, only logs changes in logging.DEBUG mode. (goto is called many times during, e.g., a scan.) Default: False.

goto_start_of_next_line (*scan_grids: Dict[str, numpy.ndarray], counter: Any*) → None

Moves scanner to the start of the next line to scan.

Parameters

- **scan_grids** – Dict of {axis_name: axis_meshgrid} from utils.make_scan_grids().
- **counter** – utils.Counter instance, determines current line of the grid.

load_surface (*fname: str*, *function: Optional[str] = 'multiquadric'*, *smooth: Optional[float] = 0*) → None

Loads a previously acquired sample surface; updates self.metadata['plane'], self.metadata['td_grid'], and self.surface_interp.

Parameters

- **fname** – Full file path for .mat file containing measured surface.
- **function** – String defining the radial basis function for scipy.interpolate.Rbf (e.g. 'cubic' or 'linear'). Default: 'multiquadric', the scipy default value.
- **smooth** – Smoothing factor for scipy.interpolate.Rbf. smooth=0 means exact interpolation. Only uses smoothing if function='linear'. Default: 0.

make_ramp (*pos0: List, pos1: List, speed: Union[int, float]*) → numpy.ndarray

Generates a ramp in x,y,z scanner voltage from point pos0 to point pos1 at given speed.

Parameters

- **pos0** – List of initial [x, y, z] scanner voltages.
- **pos1** – List of final [x, y, z] scanner votlages.
- **speed** – Speed at which to go to pos0 to pos1, in DAQ voltage/second.

Returns

ramp Array of x, y, z values to write to DAQ AOs to move scanner

from pos0 to pos1.

Return type numpy.ndarray

retract(*speed: Optional[str] = None, quiet: Optional[bool] = False*) → None

Retracts z-bender fully based on whether temp is LT or RT.

Parameters **speed** – Speed at which to move the scanner (e.g. '2 V/s') in DAQ voltage units. Default set in microscope configuration JSON file.

scan_line(*scan_grids: Dict[str, numpy.ndarray], ao_channels: Dict[str, int], daq_rate: Union[int, float], counter: Any, reverse=False*) → None

Scan a single line of a plane.

Parameters

- **scan_grids** – Dict of {axis_name: axis_meshgrid} from utils.make_scan_grids().
- **ao_channels** – Dict of {axis_name: ao_index} for the scanner ao channels.
- **daq_rate** – DAQ sampling rate in Hz.
- **counter** – utils.Counter instance, determines current line of the grid.
- **reverse** – Determines scan direction (i.e. forward or backward).

2.1.3 SQUID

The `squids.SQUID` and subclasses like `squids.Susceptometer` record SQUID parameters and metadata.

class squids.**SQUID**(*squid_config: Dict[str, Any], **kwargs*)

SQUID sensor base class. Simply records sensor metadata. No gettable or settable parameters.

Parameters

- **squid_config** – SQUID configuration dict. Simply added to instrument metadata.
- ***kwargs** – Keyword arguments passed to Instrument constructor.

class squids.**Susceptometer**(*squid_config: Dict[str, Any], **kwargs*)

Records SQUID susceptometer metadata.

Parameters

- **squid_config** – SQUID configuration dict. Simply added to instrument metadata.
- ***kwargs** – Keyword arguments passed to Instrument constructor.

2.1.4 DAQ

Instances of the `instruments.daq.DAQAnalogInputs` instrument are created only as needed for a measurement, and removed once the measurement is completed. This ensures that the DAQ hardware resources are available when needed. A `instruments.daq.DAQAnalogInputs` instrument has a single gettable parameter, `instruments.daq.DAQAnalogInputVoltages`, which aqcuires a given number of samples from the requested DAQ analog input channels.

class instruments.daq.**DAQAnalogInputVoltages**(*name: str, task: Any, samples_to_read: int, shape: Sequence[int], timeout: Union[float, int], **kwargs*)

Acquires data from one or several DAQ analog inputs.

Parameters

- **name** – Name of parameter (usually 'voltage').
- **task** – nidaqmx.Task with appropriate analog inputs channels.
- **samples_to_read** – Number of samples to read. Will be averaged based on shape.
- **shape** – Desired shape of averaged array, i.e. (nchannels, target_points).
- **timeout** – Acquisition timeout in seconds.
- ****kwargs** – Keyword arguments to be passed to ArrayParameter constructor.

get_raw()

Averages data to get *self.target_points* points per channel. If *self.target_points* == *self.samples_to_read*, no averaging is done.

class instruments.daq.**DAQAnalogInputs**(*name: str, dev_name: str, rate: Union[int, float], channels: Dict[str, int], task: Any, min_val: Optional[float] = -5, max_val: Optional[float] = 5, clock_src: Optional[str] = None, samples_to_read: Optional[int] = 2, target_points: Optional[int] = None, timeout: Union[int, float, None] = 60, **kwargs*)

Instrument to acquire DAQ analog input data in a qcodes Loop or measurement.

Parameters

- **name** – Name of instrument (usually 'daq_ai').
- **dev_name** – NI DAQ device name (e.g. 'Dev1').
- **rate** – Desired DAQ sampling rate per channel in Hz.
- **channels** – Dict of analog input channel configuration.
- **task** – fresh nidaqmx.Task to be populated with ai_channels.
- **min_val** – minimum of input voltage range (-0.1, -0.2, -0.5, -1, -2, -5 [default], or -10)
- **max_val** – maximum of input voltage range (0.1, 0.2, 0.5, 1, 2, 5 [default], or 10)
- **clock_src** – Sample clock source for analog inputs. Default: None

- **samples_to_read** – Number of samples to acquire from the DAQ per channel per measurement/loop iteration. Default: 2 (minimum number of samples DAQ will acquire in this timing mode).
- **target_points** – Number of points per channel we want in our final array. samples_to_read will be averaged down to target_points.
- **timeout** – Acquisition timeout in seconds. Default: 60.
- ****kwargs** – Keyword arguments to be passed to Instrument constructor.

clear_instances()
Clear instances of DAQAnalogInputs Instruments.

class instruments.daq.**DAQAnalogOutputVoltage**(*name: str, dev_name: str, idx: int, **kwargs*)

Writes data to one or several DAQ analog outputs.

Parameters

- **name** – Name of parameter (usually 'voltage').
- **dev_name** – DAQ device name (e.g. 'Dev1').
- **idx** – AO channel inde.
- ****kwargs** – Keyword arguments to be passed to ArrayParameter constructor.

get_raw()
Returns last voltage array written to outputs.

class instruments.daq.**DAQAnalogOutputs**(*name: str, dev_name: str, channels: Dict[str, int], **kwargs*)

Instrument to write DAQ analog output data in a qcodes Loop or measurement.

Parameters

- **name** – Name of instrument (usually 'daq_ao').
- **dev_name** – NI DAQ device name (e.g. 'Dev1').
- **channels** – Dict of analog output channel configuration.
- ****kwargs** – Keyword arguments to be passed to Instrument constructor.

clear_instances()
Clear instances of DAQAnalogOutputs Instruments.

2.1.5 Others Instruments

Lockins

- SR830 driver courtesy of QCoDeS.

class qcodes.instrument_drivers.stanford_research.SR830.**SR830**(*name*, *address*, ***kwargs*)

This is the qcodes driver for the Stanford Research Systems SR830 Lock-in Amplifier

- Driver for a single Zurich Instruments HF2LI "lockin channel".

Temperature Controllers

Lakeshore temperature controllers.

class instruments.lakeshore.**Model_331**(*name*, *address*, ***kwargs*)

Lakeshore Model 331 Temperature Controller Driver Controlled via sockets Adapted from QCoDeS Lakeshore 336 driver

class instruments.lakeshore.**Model_335**(*name*, *address*, ***kwargs*)

Lakeshore Model 335 Temperature Controller Driver Controlled via sockets Adapted from QCoDeS Lakeshore 336 driver

class instruments.lakeshore.**Model_340**(*name*, *address*, *active_channels={'A': 'cernox', 'B': 'diode'}*, ***kwargs*)

Lakeshore Model 340 Temperature Controller Driver Controlled via sockets Adapted from QCoDeS Lakeshore 336 driver

class instruments.lakeshore.**Model_372**(*name*, *address*, *active_channels={'ch1': '50K Plate', 'ch2': '3K Plate'}*, ***kwargs*)

Lakeshore Model 372 Temperature Controller Driver Controlled via sockets Adapted from QCoDeS Lakeshore 336 driver

SourceMeters

Keithley SourceMeters.

class instruments.keithley.**Keithley_2400**(*name*, *address*, ***kwargs*)

QCoDeS driver for the Keithley 2400 voltage source.

reset()

Reset the instrument. When the instrument is reset, it performs the following actions.

Returns the SourceMeter to the GPIB default conditions.

Cancels all pending commands.

Cancels all previously send **OPC* and **OPC?*

Arbitrary Function Generators

Tektronix AFG3000 series.

class instruments.afg3000.**AFG3000**(*name*, *address*, ***kwargs*)

Qcodes driver for Tektronix AFG3000 series arbitrary function generator.

Not all instrument functionality is included here. By default, most parameters are not queried during a snapshot. Logan Bishop-Van Horn (2018)

clear_instances()
Clear instances of AFG3000 Instruments.

Digital Delay Generators

Stanford Research DG645.

class instruments.dg645.**DG645**(*name*, *address*, ***kwargs*)
Qcodes driver for SRS DG645 digital delay generator.

Not all instrument functionality is included here. Logan Bishop-Van Horn (2018)

calibrate() → None
Run auto-calibration routine.

clear_instances()
Clear instances of DG645 Instruments.

local() → None
Go to local.

remote() → None
Go to remote.

reset() → None
Reset instrument.

save_settings(*location: int*) → None
Save instrument settings to given location.
Parameters **location** – Location to which to save the settings (in [1..9]).

self_test() → None
Run self-test routine.

trigger() → None
Initiates a single trigger if instrument is in single shot mode.

wait() → None
Wait for all prior commands to execute before continuing.

Heater Power Supply

AIM & Thurlby Thandar PSU (BlueFors warmup heater).

class instruments.heater.**EL320P**(*name*, *address*, ***kwargs*)
Qcodes driver for AIM & Thurlby Thandar EL320P power supply.

class microscope.microscope.**Microscope**(*config_file: str*, *temp: str*, *ureg: Any = <pint.registry.UnitRegistry object>*, *log_level: Any = 20*, *log_name: str = None*, ***kwargs*)
Base class for scanning SQUID microscope.

Parameters

- **config_file** – Path to microscope configuration JSON file.

- **temp** – 'LT' or 'RT', depending on whether the microscope is cold or not. Sets the voltage limits for the scanner and Attocubes.
- **ureg** – pint UnitRegistry for managing physical units.
- **log_level** – e.g. logging.DEBUG or logging.INFO
- **log_name** – Log file will be saved as logs/{log_name}.log. Default is the name of the microscope configuration file.
- ****kwargs** – Keyword arguments to be passed to Station constructor.

approach(*tdc_params: Dict[str, Any], attosteps: int = 100*) → None

Approach the sample by iteratively stepping z Attocube and performing td_cap().

Parameters

- **tdc_params** – Dict of capacitive touchdown parameters as defined in measurement configuration file.
- **attosteps** – Number of z atto steps to perform per iteration. Default 100.

get_surface(*x_vec: numpy.ndarray, y_vec: numpy.ndarray, tdc_params: Dict[str, Any]*) → None

Performs touchdowns on a grid and fits a plane to the resulting surface.

Parameters

- **x_vec** – 1D array of x positions (must be same length as y_vec).
- **y_vec** – 1D array of y positions (must be same length as x_vec).
- **tdc_params** – Dict of capacitive touchdown parameters as defined in measurement configuration file.

remove_component(*name: str*) → None

Remove a component (instrument) from the microscope.

Parameters **name** – Name of component to remove.

set_lockins(*measurement: Dict[str, Any]*) → None

Initialize lockins for given measurement.

Parameters **measurement** – Dict of measurement parameters as defined in measurement configuration file.

td_cap(*tdc_params: Dict[str, Any], update_snap: bool = True*) → Tuple[Any]

Performs a capacitive touchdown.

Parameters

- **tdc_params** – Dict of capacitive touchdown parameters as defined in measurement configuration file.
- **update_snap** – Whether to update the microscope snapshot. Default True. (You may want this to be False when getting a plane or approaching.)

Returns

data, tdc_plot DataSet and plot generated by the touchdown Loop.

Return type Tuple[qcodes.DataSet, *plots.TDCPlot*]

class microscope.susceptometer.**SusceptometerMicroscope**(*config_file: str, temp: str, ureg: Any = <pint.registry.UnitRegistry object>, log_level: Any = 20, log_name: str = None, **kwargs*)

Scanning SQUID susceptometer microscope class.

Parameters

- **config_file** – Path to microscope configuration JSON file.
- **temp** – 'LT' or 'RT', depending on whether the microscope is cold or not. Sets the voltage limits for the scanner and Attocubes.
- **ureg** – pint UnitRegistry for managing physical units.
- **log_level** – e.g. logging.DEBUG or logging.INFO
- **log_name** – Log file will be saved as logs/{log_name}.log. Default is the name of the microscope configuration file.
- ****kwargs** – Keyword arguments to be passed to Station constructor.

get_prefactors(*measurement: Dict[str, Any], update: bool = True*) → Dict[str, Any]

For each channel, calculate prefactors to convert DAQ voltage into real units.

Parameters

- **measurement** – Dict of measurement parameters as defined in measurement configuration file.
- **update** – Whether to query instrument parameters or simply trust the latest values (should this even be an option)?

Returns

prefactors Dict of {channel_name: prefactor} where prefactor is a pint Quantity.

Return type Dict[str, pint.Quantity]

scan_surface(*scan_params: Dict[str, Any]*) → None

Scan the current surface while acquiring data in the channels defined in measurement configuration file (e.g. MAG, SUSCX, SUSCY, CAP).

Parameters **scan_params** – Dict of scan parameters as defined in measuremnt configuration file.

Returns None

class microscope.sampler.**SamplerMicroscope**(*config_file: str, temp: str, ureg: Any = <pint.registry.UnitRegistry object>, log_level: Any = 20, log_name: str = None, **kwargs*)

Scanning SQUID sampler microscope class.

iv_mod_tek(*ivm_params: Dict[str, Any]*) → Tuple[Dict[str, Any]]

Measures IV characteristic at different mod coil voltages.

Parameters **ivm_params** – Dict of measurement parameters as definted in config_measurements json file.

Returns

data_dict, metadict Dictionaries containing data arrays and instrument metadata.

Return type Tuple[Dict]

iv_tek_mod_daq(*ivm_params: Dict[str, Any]*) → None

Performs digital feedback on mod coil to measure flux vs. delay.

AFG ch1 is used for pulse generator bias. AFG ch2 is used for comparator bias. DG ch1 is used for pulse generator trigger.

Parameters **ivm_params** – Dict of measurement parameters as definted in config_measurements json file.

Returns

data_dict, metadict Dictionaries containing data arrays and instrument metadata.

Return type Tuple[Dict]

CHAPTER

THREE

CONFIGURATION FILES

Parameters of both the microscope itself and of measurements it will perform are defined in JSON files. These parameters are loaded into memory as an OrderedDict using `utils.load_json_ordered()` so that they are accessible to the `Microscope`.

3.1 Microscope Configuration

Example configuration file for a `microscope.susceptometer.SusceptometerMicroscope`.

```
{
    "instruments": {
        "daq": {
            "model": "NI USB-6363",
            "name": "Dev1",
            "channels": {
                "analog_inputs": {
                    "MAG": 0,
                    "SUSCX": 1,
                    "SUSCY": 2,
                    "CAP": 3,
                    "x": 4,
                    "y": 5,
                    "z": 6},
                "analog_outputs": {"x": 0,"y": 1,"z": 2}
            },
            "rate": "1 MHz",
            "max_rate": {
                "analog_inputs": {
                    "1 channel": "2 MHz",
                    "multichannel": "1 MHz",
                    "comment": "Multichannel value is aggregate for all ai␣
↪channels."},
                "analog_outputs": {
                    "1 channels": "2.86 MHz",
                    "2 channels": "2.00 MHz",
                    "3 channels": "1.54 MHz",
                    "4 channels": "1.25 MHz",
                    "comment": "Max ao rates are per channel."
                }
            }
        },
        "lockins": {
            "SUSC": {
```

(continues on next page)

(continued from previous page)

```
                "model": "SR830",
                "address": "GPIB0::12::7::INSTR"},
            "CAP": {
                "model": "SR830",
                "address": "GPIB0::28::7::INSTR"
            }
        },
        "atto": {
            "name": "atto",
            "model": "ANC300",
            "address": "ASRL1::INSTR",
            "timeout": 5,
            "terminator": "\r\n",
            "stopbits": 1,
            "baud_rate" : 38400,
            "axes": {"x": 1,"y": 2,"z": 3},
            "voltage_limits": {
                "RT":{"x": "25 V","y": "25 V","z": "40 V"},
                "LT":{"x": "60 V","y": "60 V","z": "60 V"}
            },
            "default_frequency": {"x": "100 Hz", "y": "100 Hz", "z": "100 Hz
↪"},
            "constants":{
                "x": "-0.66 um",
                "y": "-0.55 um",
                "z": "0.2 um",
                "comment": "Approximate um/step at 3 K, 60 V. Sign is␣
↪relative to scanner sign. 2018/02/22"},
            "history": {}
        },
        "ls372":{
            "name": "ls372",
            "address": "GPIB0::13::7::INSTR"
        },
        "ls331":{
            "name": "ls331",
            "address": "GPIB0::30::7::INSTR"
        },
        "scanner": {
            "name": "benders",
            "constants": {
                "x": "17 um/V",
                "y": "18 um/V",
                "z": "2 um/V",
                "comment": "um/V_daq (2018/02)"},
            "voltage_limits":{
                "RT":{"x": [-2, 2], "y": [-2, 2], "z": [-2, 2]},
                "LT":{"x": [-10, 10],"y": [-10, 10], "z": [-10, 10]},
                "unit": "V",
                "comment":"V_daq should never be outside of voltage_limits."}
↪,
            "voltage_retract":{"RT": "-2 V","LT": "-10 V"},
            "speed": {
                "value": "2 V/s",
                "comment":"Rate of movement of the scanner (when not␣
↪scanning)."
            },
```

(continues on next page)

(continued from previous page)

```
            "plane": {},
            "cantilever": {
                "calibration": "326 uV/pF",
                "comment":"CAP lockin X reading, freq = 6.821 kHz, amp = 1 V"
            }
        }
    },
    "SQUID": {
        "name": "SQUID",
        "type": "susceptometer",
        "description": "IBM deep-etched 0.3um susceptometer",
        "modulation_width": "0.19 V/Phi0",
        "FC_PU_mutual": "0 Phi0/A",
        "feedback": {
            "type": "Red Pitaya + pyrpl"
        },
        "dimensions": {
            "PU_ri": "0.3 um",
            "PU_ro": "0.5 um",
            "FC_ri": "1.0 um",
            "FC_ro": "1.5 um"
        }
    },
    "info": {
        "timestamp_format":"%Y-%m-%d_%H:%M:%S"
    }
}
```

3.2 Measurement Configuration

Example configuration file for *microscope.susceptometer.SusceptometerMicroscope* measurements.

```
{
    "scan": {
        "fname": "scan",
        "surface_type": "plane",
        "fast_ax": "x",
        "range": {"x": "5 V", "y": "5 V"},
        "center": {"x": "0 V", "y": "0 V"},
        "height": "-0.2 V",
        "scan_rate": "10 pixels/s",
        "scan_size": {"x": 50, "y": 50},
        "channels": {
            "MAG": {
                "label": "Magnetometry",
                "gain": 10,
                "filters": {
                    "lowpass": {"cutoff": "100 kHz", "slope": "12 dB/octave"}
↪,
                    "highpass": {"cutoff": "0 Hz","slope": "0 dB/octave"}
                },
                "unit": "mPhi0",
                "unit_latex": "m$\\Phi_0$"
```

(continues on next page)

(continued from previous page)

```
            },
            "SUSCX": {
                "lockin": {
                    "name": "SUSC",
                    "amplitude": "1 V",
                    "frequency": "131.79 Hz"
                },
                "label": "Susceptibility",
                "gain": 10,
                "r_lead": "1 kOhm",
                "unit": "Phi0/A",
                "unit_latex": "$\\Phi_0$/A"
            },
            "SUSCY": {
                "lockin": {
                    "name": "SUSC"
                },
                "label": "Susceptibility (out of phase)",
                "gain": 10,
                "r_lead": "1 kOhm",
                "unit": "Phi0/A",
                "unit_latex": "$\\Phi_0$/A"
            },
            "CAP": {
                "lockin": {
                    "name": "CAP",
                    "amplitude": "1 V",
                    "frequency": "6.281 kHz"
                },
                "label": "Capacitance",
                "gain": 1,
                "unit": "fF",
                "unit_latex": "fF"
            }
        }
    },
    "td_cap": {
        "fname": "td_cap",
        "dV": "0.1 V",
        "range": ["-9.5 V","9.5 V"],
        "channels": {
            "CAP": {
                "lockin": {
                    "name": "CAP",
                    "amplitude": "1 V",
                    "frequency": "6.281 kHz"
                },
                "label": "Capacitance",
                "gain": 1,
                "unit": "fF",
                "unit_latex": "fF"
            },
            "SUSCX": {
                "lockin": {
                    "name": "SUSC",
                    "amplitude": "1 V",
                    "frequency": "131.79 Hz"
```

(continues on next page)

(continued from previous page)

```
                },
                "label": "Susceptibility",
                "gain": 10,
                "r_lead": "1 kOhm",
                "unit": "Phi0/A",
                "unit_latex": "$\\Phi_0$/A"
            },
            "SUSCY": {
                "lockin": {
                    "name": "SUSC"
                },
                "label": "Susceptibility (out of phase)",
                "gain": 10,
                "r_lead": "1 kOhm",
                "unit": "Phi0/A",
                "unit_latex": "$\\Phi_0$/A"
            }
        },
        "constants": {
            "max_slope": "0.8 fF/V",
            "max_delta_cap": "5 fF",
            "max_slope": "3 fF/V",
            "max_delta_cap": "15 fF",
            "initial_cap":"0 pF",
            "nfitmin":10,
            "nwindow":30,
            "ntest":8,
            "wait_factor":2
        }
    }
}
```

CHAPTER

FOUR

PHYSICAL UNITS

`scanning-squid` knows about physical units thanks to pint, a package designed to operate on and manipulate physical quantities.

`pint` is based around the `UnitRegistry`, an object that knows a set of physical units and the relationships between them.

[1]:
```
from pint import UnitRegistry
ureg = UnitRegistry()
```

We can use the our instance of `UnitRegistry` (here called `ureg`) to convert a dimensionless number like `2` into a dimensionful quantity like 2 μV, or parse a string like `'2 uV'` into a `Quantity` with a magnitude of `2` and unit of `microvolt`.

[2]:
```
print(2 * ureg('uV'))
```

```
2 microvolt
```

[3]:
```
v = ureg.Quantity('2 uV')
print(v)
```

```
2 microvolt
```

`ureg` knows how units are related to one another:

[4]:
```
print(v.to('mV'))
```

```
0.002 millivolt
```

[5]:
```
fJ = ureg.Quantity('483.6 MHz / uV') # a.c. Josephson effect
print(fJ)
print(fJ * v)
print((fJ * v).to('GHz'))
print(fJ * ureg.Quantity('1 nA') * ureg.Quantity('1 ohm'))
print((fJ * ureg.Quantity('1 uA') * ureg.Quantity('1 ohm')).to('MHz'))
```

```
483.6 megahertz / microvolt
967.2 megahertz
0.9672000000000001 gigahertz
483.6 megahertz * nanoampere * ohm / microvolt
483.6 megahertz
```

`ureg` doesn't by default know what a Φ_0 is, but we can teach it:

[6]:
```
with open('squid_units.txt', 'w') as f:
    f.write('Phi0 = 2.067833831e-15 * Wb\n')
ureg.load_definitions('./squid_units.txt')
```

[7]:
```
phi0 = ureg('Phi0')
```

[8]:
```
print(phi0)
print(phi0.to('gauss * um**2'))
print(phi0.to('aT * acre'))
```

```
1 Phi0
20.67833831 gauss * micrometer ** 2
0.5109708237305385 acre * attotesla
```

[ ]:
```

```

CHAPTER

FIVE

MEASUREMENTS

The primary measurements for `microscope.microscope.Microscope` and `microscope.susceptometer.SusceptometerMicroscope` include

- *Capacitive touchdown*
- *Approaching the Sample*
- *Acquiring a Surface*
- *Scanning* at a given height over the surface of a sample

5.1 Capacitive touchdown

See also:

`microscope.microscope.Microscope.td_cap()`, `scanner.Scanner.check_for_td()`, `scanner.Scanner.get_td_height()`, and `plots.TDCPlot`.

In a capacitive touchdown, we sweep the scanner z position (`scanner.Scanner.position_z`) and measure the cantilever capacitance via a capacitance bridge and the `CAP_lockin`. If there is a change in the slope of capacitance as a function of DAQ AO voltage above some prescribed threshold (e.g. 1 fF/V), a touchdown has been detected. The measurement parameters (usually loaded into an OrderedDict called `tdc_params`) for a capacitive touchdown are defined in the *Measurement Configuration* file as follows:

```
{
    "td_cap": {
        "fname": "td_cap",
        "dV": "0.1 V",
        "range": ["-9.5 V","9.5 V"],
        "channels": {
            "CAP": {
                "lockin": {
                    "name": "CAP",
                    "amplitude": "1 V",
                    "frequency": "6.281 kHz"
                },
                "label": "Capacitance",
                "gain": 1,
                "unit": "fF",
                "unit_latex": "fF"
            },
            "SUSCX": {
                "lockin": {
```

(continues on next page)

(continued from previous page)

```
                    "name": "SUSC",
                    "amplitude": "1 V",
                    "frequency": "131.79 Hz"
                },
                "label": "Susceptibility",
                "gain": 10,
                "r_lead": "1 kOhm",
                "unit": "Phi0/A",
                "unit_latex": "$\\Phi_0$/A"
            },
            "SUSCY": {
                "lockin": {
                    "name": "SUSC"
                },
                "label": "Susceptibility (out of phase)",
                "gain": 10,
                "r_lead": "1 kOhm",
                "unit": "Phi0/A",
                "unit_latex": "$\\Phi_0$/A"
            }
        },
        "constants": {
            "max_slope": "0.8 fF/V",
            "max_delta_cap": "5 fF",
            "initial_cap":"0 pF",
            "nfitmin":10,
            "nwindow":30,
            "ntest":8,
            "wait_factor":2
        }
    }
}
```

The algorithm for performing a capacitive touchdown is as follows:

1. Sweep `scanner.position_z` through `tdc_params['range']` with DAQ voltage steps given by `tdc_params['dV']` and use the DAQ to measure the X output of `CAP_lockin`. After each change in DAQ AO voltage, allow the lockin to settle for `max(CAP_lockin.time_constant(), SUSC_lockin.time_constant()) * tdc_params['constants']['wait_factor']`.
2. If at any point the capacitance is greater than `tdc_params['constants']['max_delta_cap']` (i.e. if the capacitance bridge is very unbalanced), or if the pre-touchdown slope is greater than `tdc_params['constants']['max_slope']`, something has gone wrong, so abort the touchdown.
3. Once `tdc_params['constants']['nwindow']` points have been acquired, partition the last `tdc_params['constants']['nwindow']` points into two subsets (with the boundary not lying within `tdc_params['constants']['nfitmin']` of either end of the window). For each allowed partition boundary point, fit a line to each of the two subsets, and select the boundary point that minimizes the RMS of the fit residuals.
4. If the absolute value of the difference in slope between the two best-fit lines exceeds `tdc_params['constants']['max_slope']`, a touchdown has occurred.
5. If a touchdown is detected, repeat the fitting routine in step 4 to find the touchdown point, and exit the loop.
6. If no touchdown is detected over the whole `tdc_params['range']`, exit the loop.

The *`microscope.microscope.Microscope.td_cap()`* will break its `qcodes.Loop` if either `scanner.Scanner.break_loop` or `scanner.Scanner.td_has_occurred` is `True`. The former is set to `True` if:

any of the safety limits are exceeded, the touchdown is interrupted by the user, or a touchdown is detected. The latter is only set to `True` if a touchdown is detected.

Note: Whenever `scanner.Scanner.break_loop` is set to `True`, the scanner will be retracted to the voltage prescribed by the microscope's temperature mode (`'LT'` or `'RT'`).

Note: It is very important to find a low-noise regime for the capacitance measurment in order to avoid false touchdowns or not detecting a real touchdown. It seems the most effective knob to turn in order fix noise problems is `CAP_lockin.frequency`. In the Bluefors 3K system, scatter of < 1 fF is typical and acceptable.

5.2 Approaching the Sample

See also:

/examples/ApproachGetSurfaceExample.ipynb, *microscope.microscope.Microscope.approach()* and *Capacitive touchdown*.

The initial approach of the sample is done by iteratively performing capacitive touchdowns and *instruments.atto.AttocubeController.step()* towards the sample in the z direction until a touchdown is detected. The basic flow of *microscope.microscope.Microscope.approach()* goes as follows:

- Run *microscope.microscope.Microscope.td_cap()* to see if the SQUID is already close to the sample.
- If no touchdown is detected, while the *microscope.microscope.Microscope.td_cap()* loop is not broken:
  - Perform the requested number of z Attocube steps towards the sample
  - Run *microscope.microscope.Microscope.td_cap()*
- If the loop was broken because a touchdown was detected, run *microscope.microscope.Microscope.td_cap()* to confirm that a touchdown occurred.

5.3 Acquiring a Surface

See also:

/examples/ApproachGetSurfaceExample.ipynb, *utils.make_scan_grids()*, *utils.make_xy_grids()*, and *Capacitive touchdown*.

In order to scan, we must know where the sample surface is. To acquire a surface, we perform capacitive touchdowns on a grid of x, y positions and fit a plane to the measured touchdown heights. The resulting fit coefficients are stored in the dictionary `scanner.Scanner.metadata['plane']`, which has keys `'x'`, `'y'`, and `'z'`. The sample plane for given x and y grids is then given by:

```
coeffs = scanner.Scanner.metadata['plane']
sample_plane = x_grid * coeffs['x'] + y_grid * coeffs['y'] + coeffs['z']
```

This means that `coeffs['x']` and `coeffs['y']` are the x and y gradients of the sample plane in DAQ voltage units, and `coeffs['z']` is the touchdown height at the origin `[x_position, y_position] == [0, 0]`. To scan, say, `0.5 V` above the sample surface, the z-axis scan grid is simply `sample_plane - 0.5`.

Note: The sample topography (i.e. touchdown voltage vs. x,y voltage) and plane are saved in a .mat file, and can be loaded into the program using *scanner.Scanner.load_surface()*.

Note: When you perform a touchdown at the origin `[x_position, y_position] == [0, 0]`, `scanner.Scanner.metadata['plane']` is automatically updated with the new touchdown voltage.

Note: This plane is trusted until the Attocubes are moved by `atto.AttocubeController.step()`, at which point `atto.AttocubeController.surface_is_current` is set to `False`, and you will not be able to scan until you've acquired a new plane or manually set `atto.surface_is_current = True`.

For samples that are not flat and therefore not well-approximated by a plane, there is the option to instead scan parallel to a surface formed by interpolating the touchdown points, by setting `"surface_type": "surface"` in the *Measurement Configuration* file. The `scanner.Scanner.surface_interp` object is an instance of `scipy.interpolate.Rbf`, which forms a radial basis function representation of multi-dimensional data (similar to spline interpolation, but more general). To see what the expected touchdown voltage at point `x, y` is, one can simply run `scanner.Scanner.surface_interp(x,y)`.

Warning: Calculation of of the `Rbf` representation of the scan array (array of voltages to be written to the DAQ AOs during a scan) is very memory intensive. If the DAQ sampling rate is too high or the scan is too large or slow, you will get a `MemoryError`.

Warning: It is easy to introduce measurement artifacts when scanning an interpolated surface, particularly for measurements that are very sensitive to SQUID-sample separation (e.g. local susceptibility). You should only use this functionality if you can be reasonably sure you are not introducing artifacts.

5.4 Scanning

See also:

plots.ScanPlot

Note: When measuring susceptibility while scanning, it is very important to choose the susceptibility lockin frequency and scan parameters such that each pixel corresponds to an integer number of lockin periods, so as to avoid beating/aliasing effects.

See /examples/ScanSurfaceExample.ipynb for a demonstration of scanning a plane with a *microscope.susceptometer.SusceptometerMicroscope*.

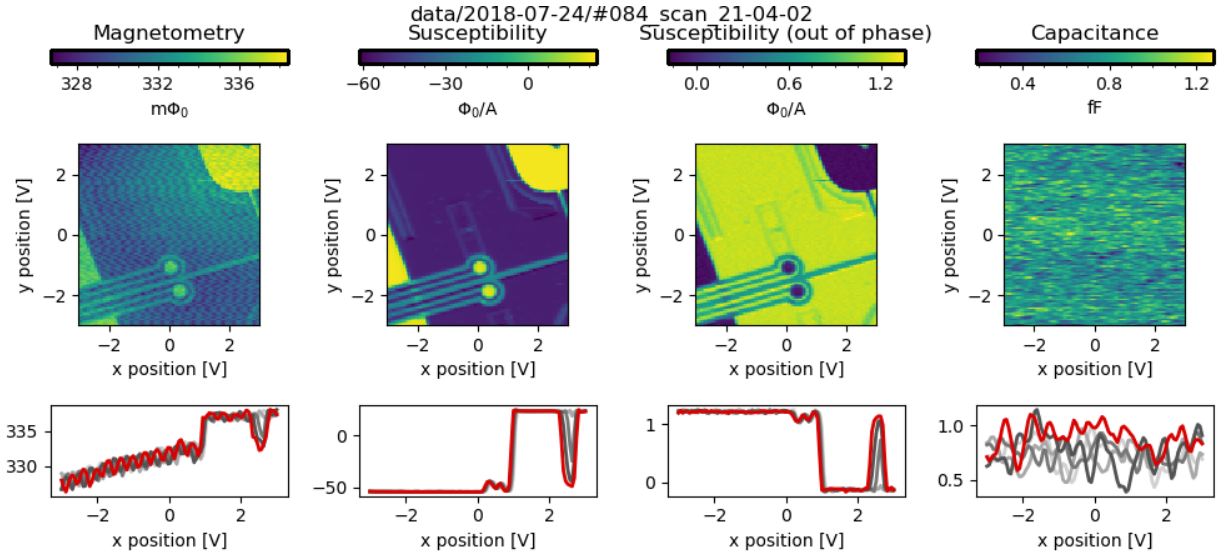
CHAPTER

SIX

PLOTS

6.1 ScanPlot

This is the plot that is displayed during the course of a scan. It shows magnetometry, susceptibility (in and out of phase), and cantilever capacitance data as a function of x,y scanner voltage in the units requested in the *Measurement Configuration* file. The plot is saved as a png file to the DataSet location after each line of the scan. The last five lines of data are displayed below the colorplot, with the most recent line in red.



class `plots.`**`ScanPlot`**(*scan_params: Dict[str, Any], ureg: Any, **kwargs*)

Plot displaying acquired images in all measurement channels, updated live during a scan.

Parameters

- **`scan_params`** – Scan parameters as defined in measurement configuration file.
- **`prefactors`** – Dict of pint quantities defining conversion factor from DAQ voltage to real units for each measurement channel.
- **`ureg`** – pint UnitRegistry, manages units.

`init_empty`()

Initialize the plot with all images empty. They will be filled during the scan.

`save`(*fname=None*)

Save plot to png file.

Parameters **fname** – File to which to save the plot. If fname is None, saves to data location as {scan_params['fname']}.png

update(*data_set: Any, loop_counter: Any, num_lines: Optional[int] = 5, offline: Optional[bool] = False*) → None

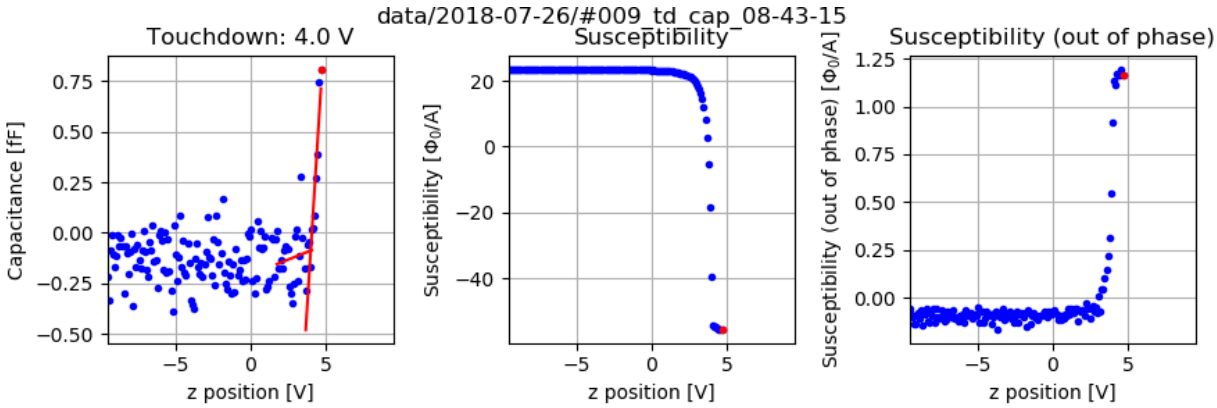
Update the plot with updated DataSet. Called after each line of the scan.

Parameters

- **DataSet** – active data set, with a new line of data added with each loop iteration.
- **loop_counter** – utils.Counter instance, lets us know where we are in the scan.
- **num_lines** – Number of previous linecuts to plot, including the line just scanned. Currently can only handle num_lines <= 5.
- **offline** – False if this is being called during a scan.

6.2 TDCPlot

This is the plot that is displayed during a touchdown. It shows cantilever capacitance and susceptibility (in and out of phase) as a function of z scanner voltage in the units requested in the *Measurement Configuration* file. The plot is saved as a png file to the DataSet location at the end of the measurement.



class plots.**TDCPlot**(*tdc_params: Dict[str, Any], ureg: Any*)

Plot displaying capacitance as a function of z voltage, updated live during a scan.

Parameters

- **tdc_params** – Touchdown parameters as defined in measurement configuration file.
- **ureg** – pint UnitRegistry, manages units.

init_empty()

Initialize the plot with no data.

save(*fname=None*)

Save plot to png file.

Parameters **fname** – File to which to save the plot. If fname is None, saves to data location as {tdc_params['fname']}.png

update(*data_set: Any*) → None

Update plot with data from data_set.

Parameters **data_set** – DataSet generated by Loop in Microscope.td_cap().

CHAPTER

SEVEN

UTILITY FUNCTIONS & CLASSES

utils is a module containing useful classes and functions that come up in the course of a scanning SQUID experiment.

class utils.**Counter**

Simple counter used to keep track of progress in a Loop.

utils.**fit_line**(*x: Union[list, numpy.ndarray], y: Union[list, numpy.ndarray]*) → Tuple[numpy.ndarray, float]

Fits a line to x, y(x) and returns (polynomial_coeffs, rms_residual).

Parameters

- **x** – List or np.ndarry, independent variable.
- **y** – List or np.ndarry, dependent variable.

Returns

p, rms Array of best-fit polynomial coefficients, rms of residuals.

Return type Tuple[np.ndarray, float]

utils.**load_json_ordered**(*filename: str*) → collections.OrderedDict

Loads json file as an ordered dict.

Parameters **filename** – Path to json file to be loaded.

Returns

odict OrderedDict containing data from json file.

Return type OrderedDict

utils.**make_scan_grids**(*scan_vectors: Dict[str, Sequence[float]], slow_ax: str, fast_ax: str, fast_ax_pts: int, plane: Dict[str, float], height: float*) → Dict[str, Any]

Makes meshgrids of scanner positions to write to DAQ analog outputs.

Parameters

- **scan_vectors** – Dict of {axis_name: axis_vector} for x, y axes (from make_scan_vectors).
- **slow_ax** – Name of the scan slow axis ('x' or 'y').
- **fast_ax** – Name of the scan fast axis ('x' or 'y').
- **fast_ax_pts** – Number of points to write to DAQ analog outputs to scan fast axis.

- **plane** – Dict of x, y, z values defining the plane to scan (provided by scanner.get_plane).
- **height** – Height above the sample surface (in DAQ voltage) at which to scan. More negative means further from sample; 0 means 'in contact'.

Returns

scan_grids {axis_name: axis_scan_grid} for x, y, z, axes.

Return type Dict

utils.**make_scan_surface**(*surface_type: str, scan_vectors: Dict[str, Sequence[float]], slow_ax: str, fast_ax: str, fast_ax_pts: int, plane: Dict[str, float], height: float, interpolator: Optional[Callable] = None*)

Makes meshgrids of scanner positions to write to DAQ analog outputs.

Parameters

- **surface_type** – Either 'plane' or 'surface'.
- **scan_vectors** – Dict of {axis_name: axis_vector} for x, y axes (from make_scan_vectors).
- **slow_ax** – Name of the scan slow axis ('x' or 'y').
- **fast_ax** – Name of the scan fast axis ('x' or 'y').
- **fast_ax_pts** – Number of points to write to DAQ analog outputs to scan fast axis.
- **plane** – Dict of x, y, z values defining the plane to scan (provided by scanner.get_plane).
- **height** – Height above the sample surface (in DAQ voltage) at which to scan. More negative means further from sample; 0 means 'in contact'.
- **interpolator** – Instance of scipy.interpolate.Rbf used to interpolate touchdown points. Only required if surface_type == 'surface'. Default: None.

Returns

scan_grids {axis_name: axis_scan_grid} for x, y, z, axes.

Return type Dict

utils.**make_scan_vectors**(*scan_params: Dict[str, Any], ureg: Any*) → Dict[str, Sequence[float]]

Creates x and y vectors for given scan parameters.

Parameters

- **scan_params** – Scan parameter dict
- **ureg** – pint UnitRegistry, manages units.

Returns

scan_vectors {axis_name: axis_vector} for x, y axes.

Return type Dict

utils.**make_xy_grids**(*scan_vectors: Dict[str, Sequence[float]], slow_ax: str, fast_ax: str*) → Dict[str, Any]

Makes meshgrids from x, y scan_vectors (used for plotting, etc.).

Parameters

- **scan_vectors** – Dict of {axis_name: axis_vector} for x, y axes (from make_scan_vectors).
- **slow_ax** – Name of scan slow axis ('x' or 'y').
- **fast_ax** – Name of scan fast axis ('x' or 'y').

Returns

xy_grids {axis_name: axis_grid} for x, y axes.

Return type Dict

utils.**moving_avg**(*x: Union[List, numpy.ndarray], y: Union[List, numpy.ndarray], window_width: int*) → Tuple[numpy.ndarray]

Given 1D arrays x and y, calculates the moving average of y.

Parameters

- **x** – x data (1D array).
- **y** – y data to be averaged (1D array).
- **window_width** – Width of window over which to average.

Returns

x, ymvg_avg x data with ends trimmed according to width_width, moving average of y data

Return type Tuple[np.ndarray]

utils.**next_file_name**(*fpath: str, extension: str*) → str

Appends an integer to fpath to create a unique file name: fpath + {next unused integer} + '.' + extension

Parameters

- **fpath** – Path to file you want to create (no extension).
- **extension** – Extension of file you want to create.

Returns

next_file_name Unique file name starting with fpath and ending with extension.

Return type str

utils.**scan_to_arrays**(*scan_data: Any, ureg: Optional[Any] = None, real_units: Optional[bool] = True, xy_unit: Optional[str] = None*) → Dict[str, Any]

Extracts scan data from DataSet and converts to requested units.

Parameters

- **scan_data** – qcodes DataSet created by Microscope.scan_plane
- **ureg** – pint UnitRegistry, manages physical units.
- **real_units** – If True, converts z-axis data from DAQ voltage into units specified in measurement configuration file.
- **xy_unit** – String describing quantity with dimensions of length. If xy_unit is not None, scanner x, y DAQ ao voltage will be converted to xy_unit according to scanner constants defined in microscope configuration file.

Returns

arrays Dict of x, y vectors and grids, and measured data in requested units.

Return type Dict

`utils.scan_to_mat_file`(*scan_data: Any*, *real_units: Optional[bool] = True*, *xy_unit: Optional[bool] = None*, *fname: Optional[str] = None*, *interpolator: Optional[Callable] = None*) → None

Export DataSet created by microscope.scan_surface to .mat file for analysis.

Parameters

- **scan_data** – qcodes DataSet created by Microscope.scan_plane
- **real_units** – If True, converts z-axis data from DAQ voltage into units specified in measurement configuration file.
- **xy_unit** – String describing quantity with dimensions of length. If xy_unit is not None, scanner x, y DAQ ao voltage will be converted to xy_unit according to scanner constants defined in microscope configuration file.
- **fname** – File name (without extension) for resulting .mat file. If None, uses the file name defined in measurement configuration file.
- **interpolator** – Instance of scipy.interpolate.Rbf, used to interpolate touchdown points. Default: None.

`utils.td_to_arrays`(*td_data: Any*, *ureg: Optional[Any] = None*, *real_units: Optional[bool] = True*) → Dict[str, Any]

Extracts scan data from DataSet and converts to requested units.

Parameters

- **td_data** – qcodes DataSet created by Microscope.td_cap
- **ureg** – pint UnitRegistry, manages physical units.
- **real_units** – If True, converts data from DAQ voltage into units specified in measurement configuration file.

Returns

arrays Dict of measured data in requested units.

Return type Dict

`utils.td_to_mat_file`(*td_data: Any*, *real_units: Optional[bool] = True*, *fname: Optional[str] = None*) → None

Export DataSet created by microscope.td_cap to .mat file for analysis.

Parameters

- **td_data** – qcodes DataSet created by Microscope.td_cap
- **real_units** – If True, converts data from DAQ voltage into units specified in measurement configuration file.
- **fname** – File name (without extension) for resulting .mat file. If None, uses the file name defined in measurement configuration file.

`utils.to_real_units`(*data_set: Any*, *ureg: Any = None*) → Any

Converts DataSet arrays from DAQ voltage to real units using recorded metadata. Preserves shape of DataSet arrays.

Parameters

- **data_set** – qcodes DataSet created by Microscope.scan_plane
- **ureg** – Pint UnitRegistry. Default None.

Returns

data ndarray like the DataSet array, but in real units as prescribed by factors in DataSet metadata.

Return type np.ndarray

`utils.validate_scan_params`(*scanner_config: Dict[str, Any], scan_params: Dict[str, Any], scan_grids: Dict[str, Any], pix_per_line: int, pts_per_line: int, temp: str, ureg: Any, logger: Any*) → None

Checks whether requested scan parameters are consistent with microscope limits.

Parameters

- **scanner_config** – Scanner configuration dict as defined in microscope configuration file.
- **scan_params** – Scan parameter dict as defined in measurements configuration file.
- **scan_grids** – Dict of x, y, z scan grids (from make_scan_grids).
- **pix_per_line** – Number of pixels per line of the scan.
- **pts_per_line** – Number of points per line sampled by the DAQ (to be averaged down to pix_per_line)
- **temp** – Temperature mode of the microscope ('LT' or 'RT').
- **ureg** – pint UnitRegistry, manages physical units.
- **logger** – Used to log the fact that the scan was validated.

Returns None

CHAPTER

EIGHT

DATASET EXAMPLE

[2]:
```
import qcodes as qc
import pprint as pp
import utils
from plots import ScanPlotFromDataSet
%matplotlib notebook
from IPython.display import Image
```

[3]:
```
data = qc.load_data('data/2018-06-06/#002_scan_09-29-05')
```

[4]:
```
data
```

[4]:
```
DataSet:
   location = 'data/2018-06-06/#002_scan_09-29-05'
   <Type>   | <array_id>             | <array.name> | <array.shape>
   Setpoint | benders_position_x_set | position_x   | (50,)
   Setpoint | index0_set             | index0       | (50, 4)
   Setpoint | index1_set             | index1       | (50, 4, 50)
   Measured | daq_ai_voltage         | voltage      | (50, 4, 50)
```

8.1 Plotting

Generate interactive plot like the one created during the scan:

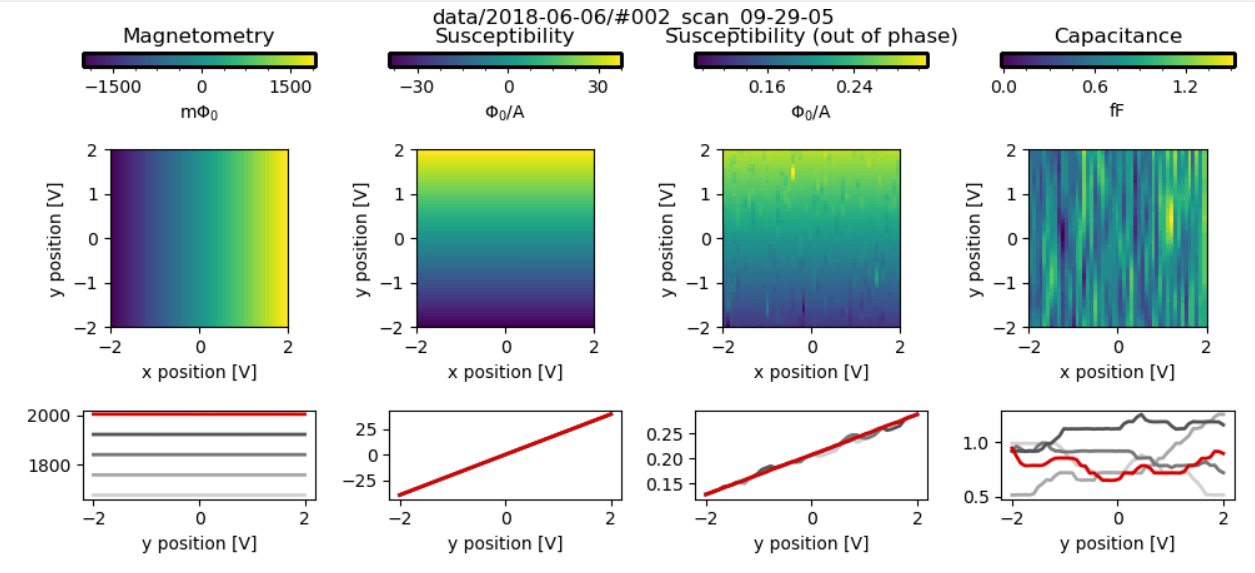
[5]:
```
scan_plot = ScanPlotFromDataSet(data)
```

```
FigureCanvasNbAgg()
```

Display plot as image (not interactive):

[6]:
```
Image(filename=data.location + '/' + data.metadata['loop']['metadata']['fname'] + '.
↪png')
```

[6]:



8.2 Explore DataSet metadata

[7]:
```
print(list(data.metadata.keys()))
print(list(data.metadata['station'].keys()))
print(list(data.metadata['loop']['metadata'].keys()))
```

```
['station', 'loop', '__class__', 'location', 'arrays', 'formatter', 'io']
['instruments', 'parameters', 'components', 'default_measurement']
['fname', 'dir', 'fast_ax', 'range', 'center', 'height', 'scan_rate', 'scan_size',␣
↪'channels', 'prefactors']
```

Measurement loop metadata

[8]:
```
pp.pprint(data.metadata['loop']['metadata']['channels'])
```

```
{'CAP': {'ai': 3,
         'gain': 1,
         'label': 'Capacitance',
         'lockin': {'amplitude': '1 V',
                    'frequency': '18.437 kHz',
                    'name': 'CAP'},
         'unit': 'fF',
         'unit_latex': 'fF'},
 'MAG': {'ai': 0,
         'filters': {'highpass': {'cutoff': '0 Hz', 'slope': '0 dB/octave'},
                     'lowpass': {'cutoff': '30 kHz', 'slope': '12 dB/octave'}},
         'gain': 1,
         'label': 'Magnetometry',
         'unit': 'mPhi0',
         'unit_latex': 'm$\\Phi_0$'},
 'SUSCX': {'ai': 1,
           'gain': 1,
           'label': 'Susceptibility',
           'lockin': {'amplitude': '1 V',
                      'frequency': '131.79 Hz',
```

(continues on next page)

(continued from previous page)

```
                        'name': 'SUSC'},
            'r_lead': '1 kOhm',
            'unit': 'Phi0/A',
            'unit_latex': '$\\Phi_0$/A'},
 'SUSCY': {'ai': 2,
            'gain': 1,
            'label': 'Susceptibility (out of phase)',
            'lockin': {'name': 'SUSC'},
            'r_lead': '1 kOhm',
            'unit': 'Phi0/A',
            'unit_latex': '$\\Phi_0$/A'}}
```

[9]:
```
data.metadata['loop']['metadata']['prefactors']
```

[9]:
```
{'MAG': '1.0 Phi0 / volt',
 'SUSCX': '0.02 Phi0 * kiloOhm / volt ** 2',
 'SUSCY': '0.02 Phi0 * kiloOhm / volt ** 2',
 'CAP': '1.5337423312883435e-07 picofarad / microvolt'}
```

Instrument snapshots

[10]:
```
SUSC_snap = data.metadata['station']['instruments']['SUSC_lockin']
for name, param in SUSC_snap['parameters'].items():
    if 'value' in param.keys():
        print(name, param['value'], param['unit'])
```

```
IDN {'vendor': 'Stanford_Research_Systems', 'model': 'SR830', 'serial': 's/n53956',␣
↪'firmware': 'ver1.07'}
timeout 5.0 s
phase -144.83 deg
reference_source internal
frequency 131.79 Hz
ext_trigger TTL rising
harmonic 1
amplitude 1.0 V
input_config a
input_shield float
input_coupling AC
notch_filter off
sensitivity 0.2 V
reserve normal
time_constant 0.01 s
filter_slope 24 dB/oct
sync_filter off
X_offset [0.0, 0]
Y_offset [0.0, 0]
R_offset [0.0, 0]
aux_in1 -0.000333333 V
aux_out1 -0.423 V
aux_in2 0.004 V
aux_out2 0.107 V
aux_in3 0.005 V
aux_out3 0.0 V
aux_in4 0.0126667 V
aux_out4 0.0 V
output_interface GPIB
ch1_ratio none
```

(continues on next page)

(continued from previous page)

```
ch1_display X
ch2_ratio none
ch2_display Y
X 0.0 V
Y -7.62945e-06 V
R 0.0 V
P 0.0 deg
buffer_SR 1 Hz
buffer_acq_mode single shot
buffer_trig_mode OFF
buffer_npts 0
```

8.3 Convert DataSet to arrays with real units

Leave everything in DAQ voltage units

[11]:
```
arrays = utils.scan_to_arrays(data, real_units=False)
```

[12]:
```
for name, array in arrays.items():
    print((name, array.units))
```

```
('X', <Unit('volt')>)
('Y', <Unit('volt')>)
('x', <Unit('volt')>)
('y', <Unit('volt')>)
('MAG', <Unit('volt')>)
('SUSCX', <Unit('volt')>)
('SUSCY', <Unit('volt')>)
('CAP', <Unit('volt')>)
```

Convert z-data to real units, but leave x and y as voltages

[13]:
```
arrays = utils.scan_to_arrays(data, real_units=True)
```

[14]:
```
for name, array in arrays.items():
    print((name, array.units))
```

```
('X', <Unit('volt')>)
('Y', <Unit('volt')>)
('x', <Unit('volt')>)
('y', <Unit('volt')>)
('MAG', <Unit('milliPhi0')>)
('SUSCX', <Unit('Phi0 / ampere')>)
('SUSCY', <Unit('Phi0 / ampere')>)
('CAP', <Unit('femtofarad')>)
```

Convert z-data to real units and x, y to μm:

[15]:
```
arrays = utils.scan_to_arrays(data, real_units=True, xy_unit='um')
```

[16]:
```
for name, array in arrays.items():
    print((name, array.units))
```

```
('X', <Unit('micrometer')>)
('Y', <Unit('micrometer')>)
('x', <Unit('micrometer')>)
('y', <Unit('micrometer')>)
('MAG', <Unit('milliPhi0')>)
('SUSCX', <Unit('Phi0 / ampere')>)
('SUSCY', <Unit('Phi0 / ampere')>)
('CAP', <Unit('femtofarad')>)
```

[17]:
```
print((arrays['x'].magnitude[0], arrays['x'].units))
```

```
(-34.0, <Unit('micrometer')>)
```

Convert z-data to real units and x, y to nm:

[18]:
```
arrays = utils.scan_to_arrays(data, real_units=True, xy_unit='nm')
```

[19]:
```
for name, array in arrays.items():
    print((name, array.units))
```

```
('X', <Unit('nanometer')>)
('Y', <Unit('nanometer')>)
('x', <Unit('nanometer')>)
('y', <Unit('nanometer')>)
('MAG', <Unit('milliPhi0')>)
('SUSCX', <Unit('Phi0 / ampere')>)
('SUSCY', <Unit('Phi0 / ampere')>)
('CAP', <Unit('femtofarad')>)
```

[20]:
```
print((arrays['x'].magnitude[0], arrays['x'].units))
```

```
(-33999.999999999993, <Unit('nanometer')>)
```

8.4 Export data to a MAT file:

[21]:
```
utils.scan_to_mat_file(data, real_units=True, xy_unit='um')
```

[22]:
```
utils.scan_to_mat_file(data, real_units=True, xy_unit=None)
```

[23]:
```
utils.scan_to_mat_file(data, real_units=False)
```

[ ]:
```

```

CHAPTER

NINE

TYPICAL WORKFLOW

9.1 Preliminary Steps

- Align SQUID at room temperature.
- Move the SQUID vertically far from the sample using the Attocubes.
- Cool down your fridge.
- Test and tune the SQUID once it is cold.
- Measure and record the scanner and Attocube capacitances.
- Check the cantilever capacitance.
- Create a directory on the data acquisition computer to hold all of the data and documentation for the cooldown.
- In this directory, create *Microscope Configuration* and *Measurement Configuration* JSON files, then launch a Jupyter Notebook.
- In the Notebook, import any modules you'll need during the cooldown and add the scanning-squid repository to your path.

9.2 Initialize the Microscope

- Initialize the microscope from the *Microscope Configuration* file.
- If something goes wrong, you can always restart the Jupyter Notebook kernel and/or re-initialize the microscope.

9.3 Load the Measurement Configuration

- Load the *Measurement Configuration* file using `utils.load_json_ordered()`.
- When you make changes to this file, be sure to re-load it.

9.4 Approach the Sample

- See: *Approaching the Sample*.

Note: If the initial touchdown occurs at negative z scanner voltage, consider using the Attocubes to move such that touchdown occurs at a positive voltage. This way, if something goes wrong and the DAQ analog outputs go to 0 V, the SQUID will not be slammed into the sample.

Warning: This is the most dangerous/uncertain part of most measurements. If the capacitance is very noisy or the cantilever is not well-constructed, you risk not detecting the touchdown and crashing the SQUID into the sample using the Attocubes.

9.5 Acquire a Surface

- See: *Acquiring a Surface*

9.6 Scan Over the Plane

- Define your scan parameters in the *Measurement Configuration* then reload the file using `utils.load_json_ordered()`.
- Start *Scanning*, sit back, and enjoy the *ScanPlot*!

9.7 Move Around the Sample

- Use the `instruments.atto.AttocubeController` to move around the sample, keeping in mind the angle between SQUID and sample so as not to accidentally crash.
- Unless the sample is very flat, it will be necessary to acquire a new plane after moving the Attocubes.
- If the sample is very flat and you still trust the old plane after moving the Attocubes, you can perform a single *Capacitive touchdown* at the origin and manually set `atto.surface_is_current = True` to update the plane.

PYTHON MODULE INDEX

i

instruments.afg3000, 12
instruments.atto, 6
instruments.daq, 10
instruments.dg645, 13
instruments.heater, 13
instruments.keithley, 12
instruments.lakeshore, 12

m

microscope.microscope, 13
microscope.sampler, 15
microscope.susceptometer, 14

p

plots, 30

q

qcodes.instrument_drivers.stanford_research.SR830, 12

s

scanner, 7
squids, 9

u

utils, 33

INDEX

A

AFG3000 (*class in instruments.afg3000*), 12
ANC150 (*class in instruments.atto*), 7
ANC300 (*class in instruments.atto*), 6
approach() (*microscope.microscope.Microscope method*), 14
ask_raw() (*instruments.atto.AttocubeController method*), 6
AttocubeController (*class in instruments.atto*), 6

C

calibrate() (*instruments.dg645.DG645 method*), 13
check_for_td() (*scanner.Scanner method*), 7
check_response() (*instruments.atto.AttocubeController method*), 6
clear_instances() (*instruments.afg3000.AFG3000 method*), 13
clear_instances() (*instruments.atto.AttocubeController method*), 6
clear_instances() (*instruments.daq.DAQAnalogInputs method*), 11
clear_instances() (*instruments.daq.DAQAnalogOutputs method*), 11
clear_instances() (*instruments.dg645.DG645 method*), 13
clear_instances() (*scanner.Scanner method*), 7
control_ao_task() (*scanner.Scanner method*), 7
Counter (*class in utils*), 33

D

DAQAnalogInputs (*class in instruments.daq*), 10
DAQAnalogInputVoltages (*class in instruments.daq*), 10
DAQAnalogOutputs (*class in instruments.daq*), 11
DAQAnalogOutputVoltage (*class in instruments.daq*), 11
DG645 (*class in instruments.dg645*), 13

E

EL320P (*class in instruments.heater*), 13

F

fit_line() (*in module utils*), 33

G

get_pos() (*scanner.Scanner method*), 7
get_prefactors() (*microscope.susceptometer.SusceptometerMicroscope method*), 15
get_raw() (*instruments.daq.DAQAnalogInputVoltages method*), 10
get_raw() (*instruments.daq.DAQAnalogOutputVoltage method*), 11
get_surface() (*microscope.microscope.Microscope method*), 14
get_td_height() (*scanner.Scanner method*), 8
goto() (*scanner.Scanner method*), 8
goto_start_of_next_line() (*scanner.Scanner method*), 8

I

init_empty() (*plots.ScanPlot method*), 29
init_empty() (*plots.TDCPlot method*), 30
initialize() (*instruments.atto.ANC150 method*), 7
initialize() (*instruments.atto.ANC300 method*), 7
instruments.afg3000 (*module*), 12
instruments.atto (*module*), 6
instruments.daq (*module*), 10
instruments.dg645 (*module*), 13
instruments.heater (*module*), 13
instruments.keithley (*module*), 12
instruments.lakeshore (*module*), 12
iv_mod_tek() (*microscope.sampler.SamplerMicroscope method*), 15
iv_tek_mod_daq() (*microscope.sampler.SamplerMicroscope method*), 16

K

Keithley_2400 (*class in instruments.keithley*), 12

L

load_json_ordered() (*in module utils*), 33
load_surface() (*scanner.Scanner method*), 8
local() (*instruments.dg645.DG645 method*), 13

M

make_ramp() (*scanner.Scanner method*), 8
make_scan_grids() (*in module utils*), 33
make_scan_surface() (*in module utils*), 34
make_scan_vectors() (*in module utils*), 34
make_xy_grids() (*in module utils*), 34
Microscope (*class in microscope.microscope*), 13
microscope.microscope (*module*), 13
microscope.sampler (*module*), 15
microscope.susceptometer (*module*), 14
Model_331 (*class in instruments.lakeshore*), 12
Model_335 (*class in instruments.lakeshore*), 12
Model_340 (*class in instruments.lakeshore*), 12
Model_372 (*class in instruments.lakeshore*), 12
moving_avg() (*in module utils*), 35

N

next_file_name() (*in module utils*), 35

P

plots (*module*), 29, 30

Q

qcodes.instrument_drivers.stanford_research.SR830 (*module*), 12

R

remote() (*instruments.dg645.DG645 method*), 13
remove_component() (*microscope.microscope.Microscope method*), 14
reset() (*instruments.dg645.DG645 method*), 13
reset() (*instruments.keithley.Keithley_2400 method*), 12
retract() (*scanner.Scanner method*), 9

S

SamplerMicroscope (*class in microscope.sampler*), 15
save() (*plots.ScanPlot method*), 29
save() (*plots.TDCPlot method*), 30
save_settings() (*instruments.dg645.DG645 method*), 13
scan_line() (*scanner.Scanner method*), 9
scan_surface() (*microscope.susceptometer.SusceptometerMicroscope method*), 15
scan_to_arrays() (*in module utils*), 35
scan_to_mat_file() (*in module utils*), 36
Scanner (*class in scanner*), 7
scanner (*module*), 7
ScanPlot (*class in plots*), 29
self_test() (*instruments.dg645.DG645 method*), 13
set_lockins() (*microscope.microscope.Microscope method*), 14
SQUID (*class in squids*), 9
squids (*module*), 9
SR830 (*class in qcodes.instrument_drivers.stanford_research.SR830*), 12
step() (*instruments.atto.AttocubeController method*), 6
stop() (*instruments.atto.AttocubeController method*), 6
Susceptometer (*class in squids*), 9
SusceptometerMicroscope (*class in microscope.susceptometer*), 14

T

td_cap() (*microscope.microscope.Microscope method*), 14
td_to_arrays() (*in module utils*), 36
td_to_mat_file() (*in module utils*), 36
TDCPlot (*class in plots*), 30
to_real_units() (*in module utils*), 36
trigger() (*instruments.dg645.DG645 method*), 13

U

update() (*plots.ScanPlot method*), 30
update() (*plots.TDCPlot method*), 30
utils (*module*), 33

V

validate_scan_params() (*in module utils*), 37

W

wait() (*instruments.dg645.DG645 method*), 13
write_raw() (*instruments.atto.AttocubeController method*), 6