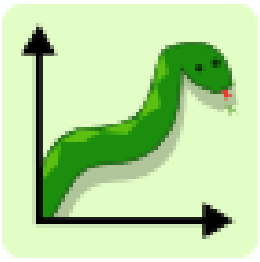
PyMeasure Documentation

Release 0.3

PyMeasure Developers

June 02, 2016



PyMeasure

PyMeasure makes scientific measurements easy to set up and run. The package contains a repository of instrument classes and a system for running experiment procedures, which provides graphical interfaces for graphing live data and managing queues of experiments. Both parts of the package are independent, and when combined provide all the necessary requirements for advanced measurements with only limited coding.

PyMeasure is currently under active development, so please report any issues you experience to our Issues page. The main documentation for the site is organized into a couple sections:

- *Learning PyMeasure*
- *API References*
- *About PyMeasure*

Information about development is also available:

- *Getting involved*

CHAPTER 1

Introduction

PyMeasure uses an object oriented approach for communicating with scientific instruments, which provides an intuitive interface where the low-level SCPI and GPIB commands are hidden from normal use. Users can focus on solving the measurement problems at hand, instead of re-inventing how to communicate with instruments.

Instruments with VISA (GPIB, Serial, etc) are supported through the PyVISA package under the hood. Prologix GPIB adapters are also supported. Communication protocols can be swapped, so that instrument classes can be used with all supported protocols interchangeably.

Before using PyMeasure, you should be acquainted with basic Python programming for the sciences and understand the concept of objects.

1.1 Instrument ready

The package includes a number of instruments already defined. Their definitions are organized based on the manufacturer name of the instrument. For example the class that defines the Keithley 2400 SourceMeter can be imported by calling:

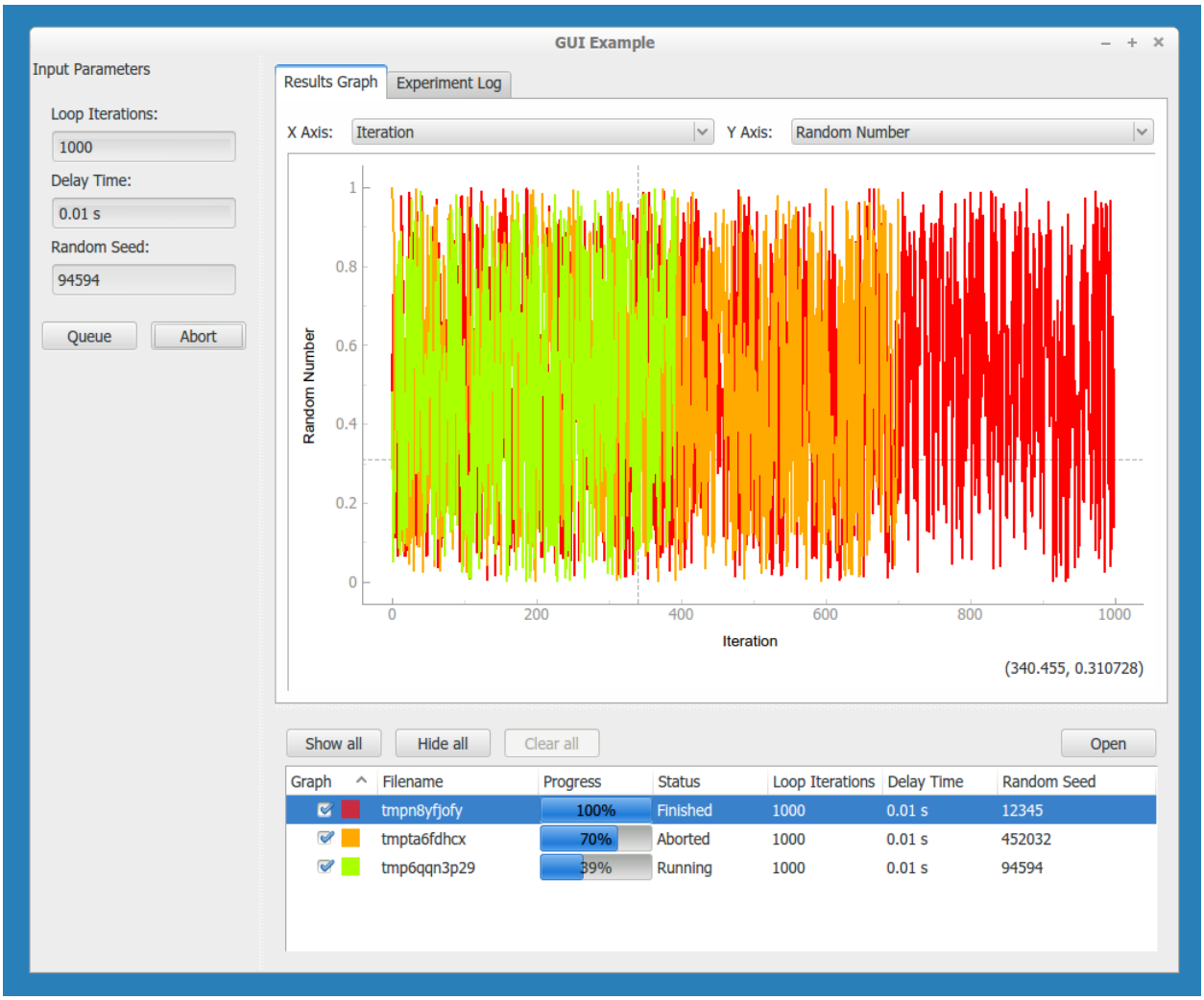
```
from pymeasure.instruments.keithley import Keithley2400
```

The Getting Started section will go into more detail on connecting to an instrument. If you don't find the instrument you are looking for, but are interested in contributing, see the documentation on adding an instrument.

1.2 Graphical displays

Graphical user interfaces (GUIs) can be easily generated to manage execution of measurement procedures with PyMeasure. This includes live plotting for data, and a queue system for managing large numbers of experiments.

These features are explored in the Using a graphical interface tutorial.


GUI Example
Input Parameters
Loop Iterations:
1000
Delay Time:
0.01 s
Random Seed:
94594
Queue
Abort
Results Graph
Experiment Log
X Axis:
Iteration
Y Axis:
Random Number
Random Number
Iteration
(340.455, 0.310728)
Show all
Hide all
Clear all
Open
Graph
Filename
Progress
Status
Loop Iterations
Delay Time
Random Seed
tmpn8yfjofy
100%
Finished
1000
0.01 s
12345
tmpta6fdhcx
70%
Aborted
1000
0.01 s
452032
tmp6qqn3p29
39%
Running
1000
0.01 s
94594

CHAPTER 2

Getting started

This section provides instructions for installing PyMeasure.

2.1 Dependencies

PyMeasure is a Python 3+ library, and does not support Python 2. This is a deliberate move to switch code over to the new conventions, and remove the extra work of back-porting functionality.

2.1.1 Core dependencies

PyMeasure builds on the success of two key Python packages.

- Numpy - Numerical Python, which handles large data sets efficiently
- Pandas - An extension of Numpy that simplifies data management

2.1.2 Optional dependencies

There are a number of other packages that are required for specific functionality.

For communicating with VISA instruments, the PyVISA package is required. PySerial is used for basic serial communication.

- PyVISA - VISA instrument communication library
- PySerial - Serial communication library

The live-plotting and user-interfaces require either PyQt4 or PySide, in combination with PyQtGraph.

- PyQt4 - Cross-platform Qt library for graphical user interfaces
- PySide - Alternative to PyQt4, licensed appropriately for commercial use
- PyQtGraph - Efficient live-plotting library

For listening in on the experimental procedure execution through TCP messaging, the PyZMQ and MsgPack-Numpy libraries are required. This is not necessary for general use.

- PyZMQ - Message communication library
- MsgPack Numpy - Compresses messages and handles Numpy arrays

2.2 Installing

Get the latest release from GitHub or install via the Python `pip` installer:

```
pip install pymeasure
```

If you plan to use any of the addition dependencies, install them seperately.

Now that you have PyMeasure installed, the next step is to connect to an instrument.

CHAPTER 3

Tutorials

The following sections provide instructions for getting started with PyMeasure.

3.1 Connecting to an instrument

After following the Getting Started section, you now have a working installation of PyMeasure. This section describes connecting to an instrument, using a Keithley 2400 SourceMeter as an example. To follow the tutorial, open a command prompt, IPython terminal, or Jupyter notebook.

First import the instrument of interest.

```
from pymeasure.instruments.keithley import Keithley2400
```

Then construct an object by passing the GPIB address. For this example we connect to the instrument over GPIB (using VISA) with an address of 4. See the *adapters* section below for more details.

```
sourcemeter = Keithley2400("GPIB::4")
```

For instruments with standard SCPI commands, an `id` property will return the results of a `*IDN?` SCPI command, identifying the instrument.

```
sourcemeter.id
```

This is equivalent to manually calling the SCPI command.

```
sourcemeter.ask("*IDN?")
```

Here the `ask` method writes the SCPI command, reads the result, and returns that result. This is further equivalent to calling the methods below.

```
sourcemeter.write("*IDN?")
sourcemeter.read()
```

This example illustrates that the top-level methods like `id` are really composed of many lower-level methods. Both can be called depending on the operation that is desired. PyMeasure hides the complexity of these lower-level operations, so you can focus on the bigger picture.

3.1.1 Using adapters

PyMeasure supports a number of adapters, which are responsible for communicating with the underlying hardware. In the example above, we passed the string "GPIB::4" when constructing the instrument. By default this constructs

a VISAAdapter class to connect to the instrument using VISA. Instead of passing a string, we could equally pass an adapter object.

```
from pymeasure.adapters import VISAAdapter

adapter = VISAAdapter("GPIB::4")
sourcemeter = Keithely2400(adapter)
```

To instead use a Prologix GPIB device connected on `/dev/ttyUSB0` (proper permissions are needed in Linux, see PrologixAdapter), the adapter is constructed in a similar way. Unlike the VISA adapter which is specific to each instrument, the Prologix adapter can be shared by many instruments. Therefore, they are addressed separately based on the GPIB address number when passing the adapter into the instrument construction.

```
from pymeasure.adapters import PrologixAdapter

adapter = PrologixAdapter('/dev/ttyUSB0')
sourcemeter = Keithley2400(adapeter.gpib(4))
```

For instruments using serial communication that have particular settings that need to be matched, a custom Adapter sub-class can be made. For example, the LakeShore 425 Gaussmeter connects via USB, but uses particular serial communication settings. Therefore, a LakeShoreUSBAdapter class enables these requirements in the background.

```
from pymeasure.instruments.lakeshore import LakeShore425

gaussmeter = LakeShore425('/dev/lakeshore425')
```

Behind the scenes the `/dev/lakeshore425` port is passed to the LakeShoreUSBAdapter.

The above examples illustrate different methods for communicating with instruments, using adapters to keep instrument code independent from the communication protocols. Next we present the methods for setting up measurements.

3.2 Making a measurement

This tutorial will walk you through using PyMeasure to acquire a current-voltage (IV) characteristic using a Keithley 2400. Even if you don't have access to this instrument, this tutorial will explain the method for making measurements with PyMeasure. First we describe using a simple script to make the measurement. From there, we show how Procedures objects greatly simplify the workflow, which leads to making the measurement with a graphical interface.

3.2.1 Using scripts

Scripts are a quick way to get up and running with a measurement in PyMeasure. For our IV characteristic measurement, we perform the following steps:

1. Import the necessary packages
2. Set the input parameters to define the measurement
3. Connect to the Keithley 2400
4. Set up the instrument for the IV characteristic
5. Allocate arrays to store the resulting measurements
6. Loop through the current points, measure the voltage, and record
7. Save the final data to a CSV file
8. Shutdown the instrument

These steps are expressed in code as follows.

```
# Import necessary packages
from pymeasure.instruments.keithley import Keithley2400
import numpy as np
import pandas as pd
from time import sleep

# Set the input parameters
data_points = 50
averages = 50
max_current = 0.01
min_current = -max_current

# Connect and configure the instrument
sourcemeter = Keithley2400("GPIB::4")
sourcemeter.reset()
sourcemeter.use_front_terminals()
sourcemeter.measure_voltage()
sourcemeter.config_current_source()
sleep(0.1) # wait here to give the instrument time to react
sourcemeter.set_buffer(averages)

# Allocate arrays to store the measurement results
currents = np.linspace(min_current, max_current, num=data_points)
voltages = np.zeros_like(currents)
voltage_stds = np.zeros_like(currents)

# Loop through each current point, measure and record the voltage
for i in range(data_points):
    sourcemeter.current = currents[i]
    sourcemeter.reset_buffer()
    sleep(0.1)
    sourcemeter.start_buffer()
    sourcemeter.wait_for_buffer()

    # Record the average and standard deviation
    voltages[i] = sourcemeter.means
    voltage_stds[i] = sourcemeter.standard_devs

# Save the data columns in a CSV file
data = pd.DataFrame({
    'Current (A)': currents,
    'Voltage (V)': voltages,
    'Voltage Std (V)': voltage_stds,
})
data.to_csv('example.csv')

sourcemeter.shutdown()
```

Running this example script will execute the measurement and save the data to a CSV file. While this may be sufficient for very basic measurements, this example illustrates a number of issues that PyMeasure solves. The issues with the script example include:

- The progress of the measurement is not transparent
- Input parameters are not associated with the data that is saved
- Data is not plotted during the execution (nor at all in this case)

- Data is only saved upon successful completion, which is otherwise lost
- Canceling a running measurement causes the system to end in a undetermined state
- Exceptions also end the system in an undetermined state

The Procedure class allows us to solve all of these issues. The next section introduces the Procedure class and shows how to modify our script example to take advantage of these features.

3.2.2 Using Procedures

The Procedure object bundles the sequence of steps in an experiment with the parameters required for a its successful execution. This simple structure comes with huge benefits, since a number of convenient tools for making the measurement use this common interface.

Let's start with a simple example of a procedure which loops over a certain number of iterations. We make the SimpleProcedure object as a sub-class of Procedure, since SimpleProcedure *is a* Procedure.

```
from time import sleep
from pymeasure.experiment import Procedure
from pymeasure.experiment import IntegerParameter

class SimpleProcedure(Procedure):

    # a Parameter that defines the number of loop iterations
    iterations = IntegerParameter('Loop Iterations')

    # a list defining the order and appearance of columns in our data file
    DATA_COLUMNS = ['Iteration']

    def execute(self):
        """ Loops over each iteration and emits the current iteration,
        before waiting for 0.01 sec, and then checking if the procedure
        should stop
        """
        for i in range(self.iterations):
            self.emit('results', {'Iteration': i})
            sleep(0.01)
            if self.should_stop():
                break
```

At the top of the SimpleProcedure class we define the required Parameters. In this case, `iterations` is a IntegerParameter that defines the number of loops to perform. Inside our Procedure class we reference the value in the iterations Parameter by the class variable where the Parameter is stored (`self.iterations`). PyMeasure swaps out the Parameters with their values behind the scene, which makes accessing the values of parameters very convenient.

We define the data columns that will be recorded in a list stored in `DATA_COLUMNS`. This sets the order by which columns are stored in the file. In this example, we will store the Iteration number for each loop iteration.

The `execute` methods defines the main body of the procedure. Our example method consists of a loop over the number of iterations, in which we emit the data to be recorded (the Iteration number). The data is broadcast to any number of listeners by using the `emit` method, which takes a topic as the first argument. Data with the `'results'` topic and the proper data columns will be recorded to a file. The sleep function in our example provides two very useful features. The first is to delay the execution of the next lines of code by the time argument in units of seconds. The seconds is that during this delay time, the CPU is free to perform other code. Successful measurements often require the intelligent use of sleep to deal with instrument delays and ensure that the CPU is not hogged by a single script. After our delay, we check to see if the Procedure should stop by calling `self.should_stop()`. By checking this flag, the Procedure will react to a user canceling the procedure execution.

This covers the basic requirements of a Procedure object. Now let's construct our SimpleProcedure object with 100 iterations.

```
procedure = SimpleProcedure()
procedure.iterations = 100
```

Next we will show how to run the procedure.

Running Procedures

A Procedure is run by a Worker object. The Worker executes the Procedure in a separate process, which has a speed advantage on computers with multiple processors and allows other scripts to execute asynchronously with the procedure (e.g. a graphical user interface). In addition to performing the measurement, the Worker spawns a Recorder object, which listens for the `'results'` topic in data emitted by the Procedure, and writes those lines to a data file. The Results object provides a convenient abstraction to keep track of where the data should be stored, the data in an accessible form, and the Procedure that pertains to those results.

We first construct a Results object for our Procedure.

```
from pymeasure.experiment import Results

data_filename = 'example.csv'
results = Results(procedure, data_filename)
```

Constructing the Results object for our Procedure creates the file using the `data_filename`, and stores the Parameters for the Procedure. This allows the Procedure and Results objects to be reconstructed later simply by loading the file using `Results.load(data_filename)`. The Parameters in the file are easily readable.

We now construct a Worker with the Results object, since it contains our Procedure.

```
from pymeasure.experiment import Worker

worker = Worker(results)
```

The Worker publishes data and other run-time information through specific queues, but can also publish this information over the local network on a specific TCP port (using the optional `port` argument. Using TCP communication allows great flexibility for sharing information with Listener objects. Queues are used as the standard communication method because they preserve the data order, which is of critical importance to storing data accurately and reacting to the measurement status in order.

Now we are ready to start the worker.

```
worker.start()
```

The Worker process will be launched in a separate process, which allows us to perform other tasks while it is running. When writing a script that should block (wait for the Worker to finish), we need to join the Worker back into the main process.

```
worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
```

Let's put all the pieces together. Our SimpleProcedure can be run in a script by the following.

```
from time import sleep
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter

class SimpleProcedure(Procedure):

    # a Parameter that defines the number of loop iterations
```

```
    iterations = IntegerParameter('Loop Iterations')

    # a list defining the order and appearance of columns in our data file
    DATA_COLUMNS = ['Iteration']

    def execute(self):
        """ Loops over each iteration and emits the current iteration,
        before waiting for 0.01 sec, and then checking if the procedure
        should stop
        """
        for i in range(self.iterations):
            self.emit('results', {'Iteration': i})
            sleep(0.01)
            if self.should_stop():
                break

if __name__ == "__main__":
    procedure = SimpleProcedure()
    procedure.iterations = 100

    data_filename = 'example.csv'
    results = Results(procedure, data_filename)

    worker = Worker(results)
    worker.start()

    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
```

Here we have included an if statement to only run the script if the __name__ is __main__. This precaution allows us to import the SimpleProcedure object without running the execution.

Using Logs

Logs keep track of important details in the execution of a procedure. We describe the use of the Python logging module with PyMeasure, which makes it easy to document the execution of a procedure and provides useful insight when diagnosing issues or bugs.

Let's extend our SimpleProcedure with logging.

```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

from time import sleep
from pymeasure.log import console_log
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter

class SimpleProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')

    DATA_COLUMNS = ['Iteration']

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
```

```
            data = {'Iteration': i}
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            sleep(0.01)
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break

if __name__ == "__main__":
    console_log(log)

    log.info("Constructing a SimpleProcedure")
    procedure = SimpleProcedure()
    procedure.iterations = 100

    data_filename = 'example.csv'
    log.info("Constructing the Results with a data file: %s" % data_filename)
    results = Results(procedure, data_filename)

    log.info("Constructing the Worker")
    worker = Worker(results)
    worker.start()
    log.info("Started the Worker")

    log.info("Joining with the worker in at most 1 hr")
    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
    log.info("Finished the measurement")
```

First, we have imported the Python logging module and grabbed the logger using the `__name__` argument. This gives us logging information specific to the current file. Conversely, we could use the `''` argument to get all logs, including those of pymeasure. We use the `console_log` function to conveniently output the log to the console. Further details on how to use the logger are addressed in the Python logging documentation.

Modifying our script

Now that you have a background on how to use the different features of the Procedure class, and how they are run, we will revisit our IV characteristic measurement using Procedures. Below we present the modified version of our example script, now as a IVProcedure class.

```
# Import necessary packages
from pymeasure.instruments.keithley import Keithley2400
from pymeasure.experiment import Procedure
from pymeasure.experiment import IntegerParameter, FloatParameter
from time import sleep

class IVProcedure(Procedure):

    data_points = IntegerParameter('Data points', default=50)
    averages = IntegerParameter('Averages', default=50)
    max_current = FloatParameter('Maximum Current', unit='A', default=0.01)
    min_current = FloatParameter('Minimum Current', unit='A', default=-0.01)

    DATA_COLUMNS = ['Current (A)', 'Voltage (V)', 'Voltage Std (V)']

    def startup(self):
        log.info("Connecting and configuring the instrument")
        self.sourcemeter = Keithley2400("GPIB::4")
```

```
        self.sourcemeter.reset()
        self.sourcemeter.use_front_terminals()
        self.sourcemeter.measure_voltage()
        self.sourcemeter.config_current_source()
        sleep(0.1) # wait here to give the instrument time to react
        self.sourcemeter.set_buffer(averages)

    def execute(self):
        currents = np.linspace(
            self.min_current,
            self.max_current,
            num=self.data_points
        )

        # Loop through each current point, measure and record the voltage
        for current in currents:
            log.info("Setting the current to %g A" % current)
            self.sourcemeter.current = current
            self.sourcemeter.reset_buffer()
            sleep(0.1)
            self.sourcemeter.start_buffer()
            log.info("Waiting for the buffer to fill with measurements")
            self.sourcemeter.wait_for_buffer()

            self.emit('results', {
                'Current (A)': current,
                'Voltage (V)': self.sourcemeter.means,
                'Voltage Std (V)': self.sourcemeter.standard_devs
            })
            sleep(0.01)
            if self.should_stop():
                log.info("User aborted the procedure")
                break

    def shutdown(self):
        self.sourcemeter.shutdown()
        log.info("Finished measuring")

if __name__ == "__main__":
    console_log(log)

    log.info("Constructing an IVProcedure")
    procedure = IVProcedure()
    procedure.data_points = 100
    procedure.averages = 50
    procedure.max_current = -0.01
    procedure.min_current = 0.01

    data_filename = 'example.csv'
    log.info("Constructing the Results with a data file: %s" % data_filename)
    results = Results(procedure, data_filename)

    log.info("Constructing the Worker")
    worker = Worker(results)
    worker.start()
    log.info("Started the Worker")

    log.info("Joining with the worker in at most 1 hr")
```

```
    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
    log.info("Finished the measurement")
```

At this point, you are familiar with how to construct a Procedure sub-class. The next section shows how to put these procedures to work in a graphical environment, where will have live-plotting of the data and the ability to easily queue up a number of experiments in sequence. All of these features come from using the Procedure object.

3.3 Using a graphical interface

In the previous tutorial we measured the IV characteristic of a sample to show how we can set up a simple experiment in PyMeasure. The real power of PyMeasure comes when we also use the graphical tools that are included to turn our simple example into a full-flegged user interface.

3.3.1 Using the Plotter

While it lacks the nice features of the ManagedWindow, the Plotter object is the simplest way of getting live-plotting. The Plotter takes a Results object and plots the data at a regular interval, grabbing the latest data each time from the file.

Let's extend our SimpleProcedure with a RandomProcedure, which generates random numbers during our loop. This example does not include instruments to provide a simpler example.

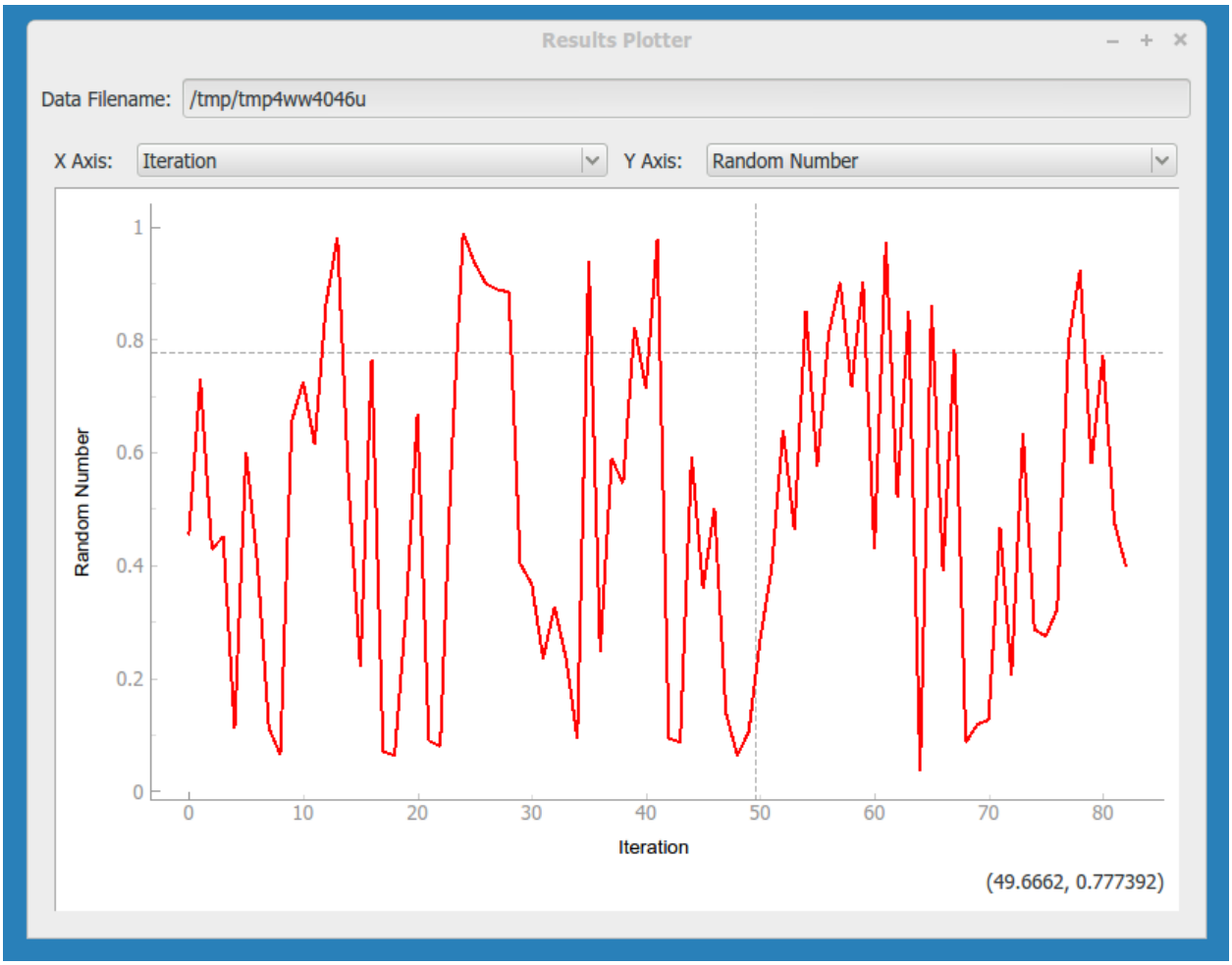
```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import random
from time import sleep
from pymeasure.log import console_log
from pymeasure.display import Plotter
from pymeasure.experiment import Procedure, Results, Worker
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')
    delay = FloatParameter('Delay Time', units='s', default=0.2)
    seed = Parameter('Random Seed', default='12345')

    DATA_COLUMNS = ['Iteration', 'Random Number']

    def startup(self):
        log.info("Setting the seed of the random number generator")
        random.seed(self.seed)

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number': random.random()
            }
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            sleep(self.delay)
```

```
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break


if __name__ == "__main__":
    console_log(log)

    log.info("Constructing a SimpleProcedure")
    procedure = SimpleProcedure()
    procedure.iterations = 100

    data_filename = 'random.csv'
    log.info("Constructing the Results with a data file: %s" % data_filename)
    results = Results(procedure, data_filename)

    log.info("Constructing the Plotter")
    plotter = Plotter(results)
    plotter.start()
    log.info("Started the Plotter")

    log.info("Constructing the Worker")
    worker = Worker(results)
    worker.start()
    log.info("Started the Worker")

    log.info("Joining with the worker in at most 1 hr")
    worker.join(timeout=3600) # wait at most 1 hr (3600 sec)
    log.info("Finished the measurement")
```

The important addition is the construction of the Plotter from the Results object.

```
plotter = Plotter(results)
plotter.start()
```

Just like the Worker, the Plotter is started in a different process so that it can be run on a separate CPU for higher performance. The Plotter launches a Qt graphical interface using pyqtgraph which allows the Results data to be viewed based on the columns in the data.



3.3.2 Using the ManagedWindow

The ManagedWindow is the most convenient tool for running measurements with your Procedure. This has the major advantage of accepting the input parameters graphically. From the parameters, a graphical form is automatically generated that allows the inputs to be typed in. With this feature, measurements can be started dynamically, instead of defined in a script.

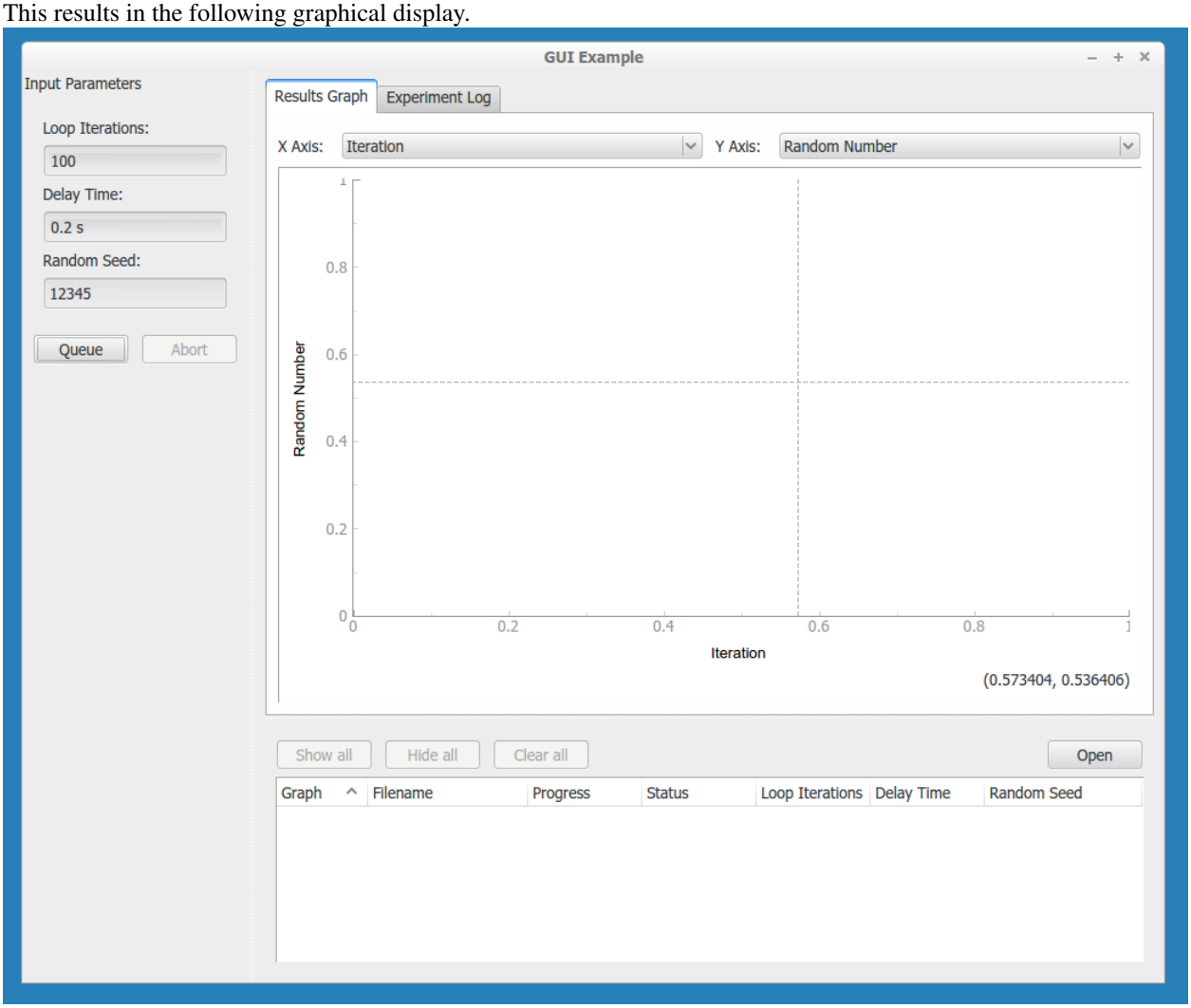
Another major feature of the ManagedWindow is its support for running measurements in a sequential queue. This allows you to set up a number of measurements with different input parameters, and watch them unfold on the live-plot. This is especially useful for long running measurements. The ManagedWindow achieves this through the Manager object, which coordinates which Procedure the Worker should run and keeps track of its status as the Worker progresses.

Below we adapt our previous example to use a ManagedWindow.

```
import logging
log = logging.getLogger(__name__)
log.addHandler(logging.NullHandler())

import random
from time import sleep
from pymeasure.log import console_log
from pymeasure.display.Qt import QtGui
from pymeasure.display.windows import ManagedWindow
```

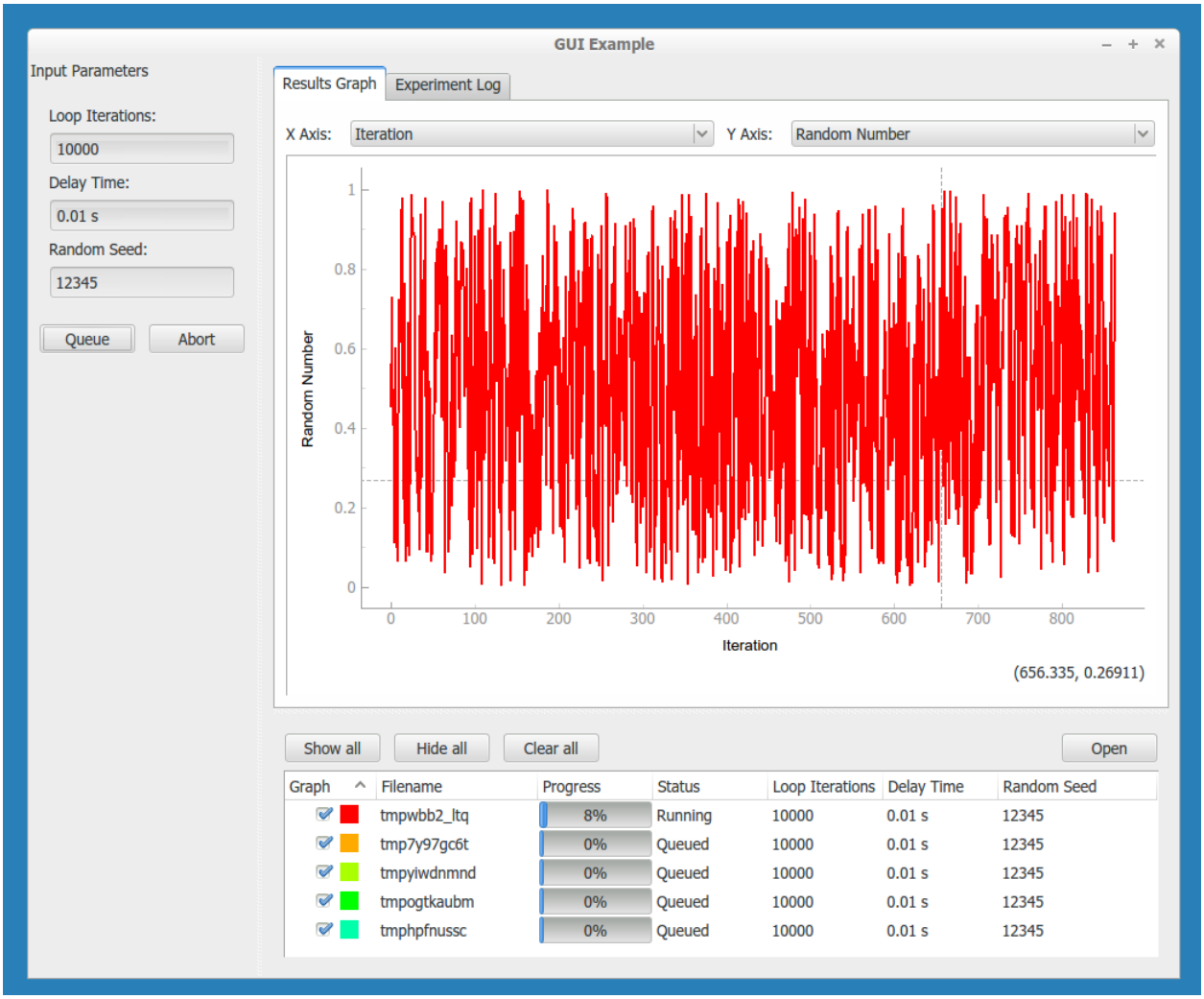
```
from pymeasure.experiment import Procedure, Results
from pymeasure.experiment import IntegerParameter, FloatParameter, Parameter

class RandomProcedure(Procedure):

    iterations = IntegerParameter('Loop Iterations')
    delay = FloatParameter('Delay Time', units='s', default=0.2)
    seed = Parameter('Random Seed', default='12345')

    DATA_COLUMNS = ['Iteration', 'Random Number']

    def startup(self):
        log.info("Setting the seed of the random number generator")
        random.seed(self.seed)

    def execute(self):
        log.info("Starting the loop of %d iterations" % self.iterations)
        for i in range(self.iterations):
            data = {
                'Iteration': i,
                'Random Number': random.random()
            }
            self.emit('results', data)
            log.debug("Emitting results: %s" % data)
            sleep(self.delay)
            if self.should_stop():
                log.warning("Caught the stop flag in the procedure")
                break


class MainWindow(ManagedWindow):

    def __init__(self):
        super(MainWindow, self).__init__(
            procedure_class=RandomProcedure,
            inputs=['iterations', 'delay', 'seed'],
            displays=['iterations', 'delay', 'seed'],
            x_axis='Iteration',
            y_axis='Random Number'
        )
        self.setWindowTitle('GUI Example')

    def queue(self):
        filename = tempfile.mktemp()

        procedure = self.make_procedure()
        results = Results(procedure, filename)
        experiment = self.new_experiment(results)

        self.manager.queue(experiment)


if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

This results in the following graphical display.

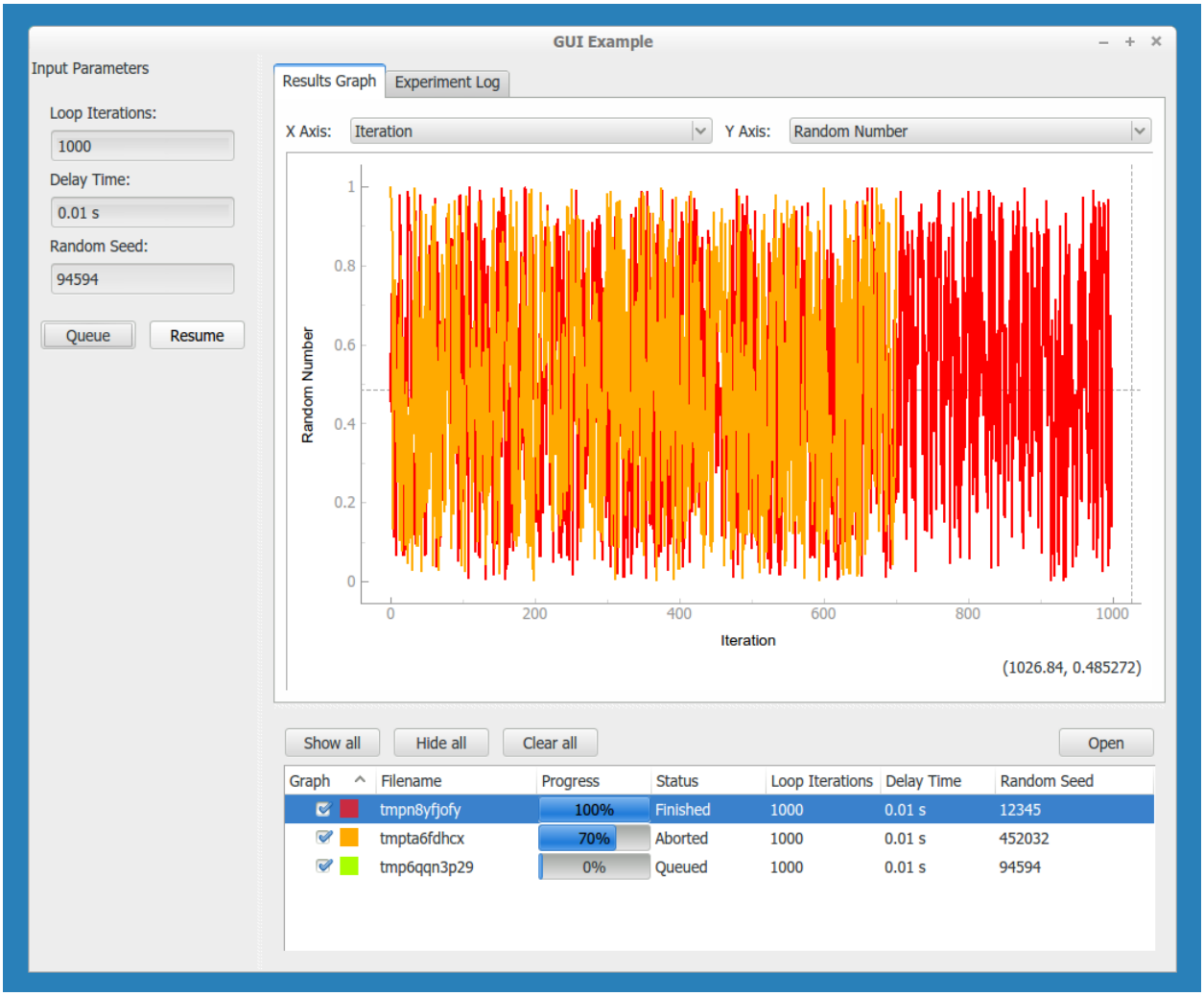


In the code, the MainWindow class is a sub-class of the ManagedWindow class. We overwrite the constructor to provide information about the procedure class and its options. The `inputs` are a list of Parameters class-variable names, which the display will generate graphical fields for. The `displays` is a similar list, which instead defines the parameters to display in the browser window. This browser keeps track of the experiments being run in the sequential queue.

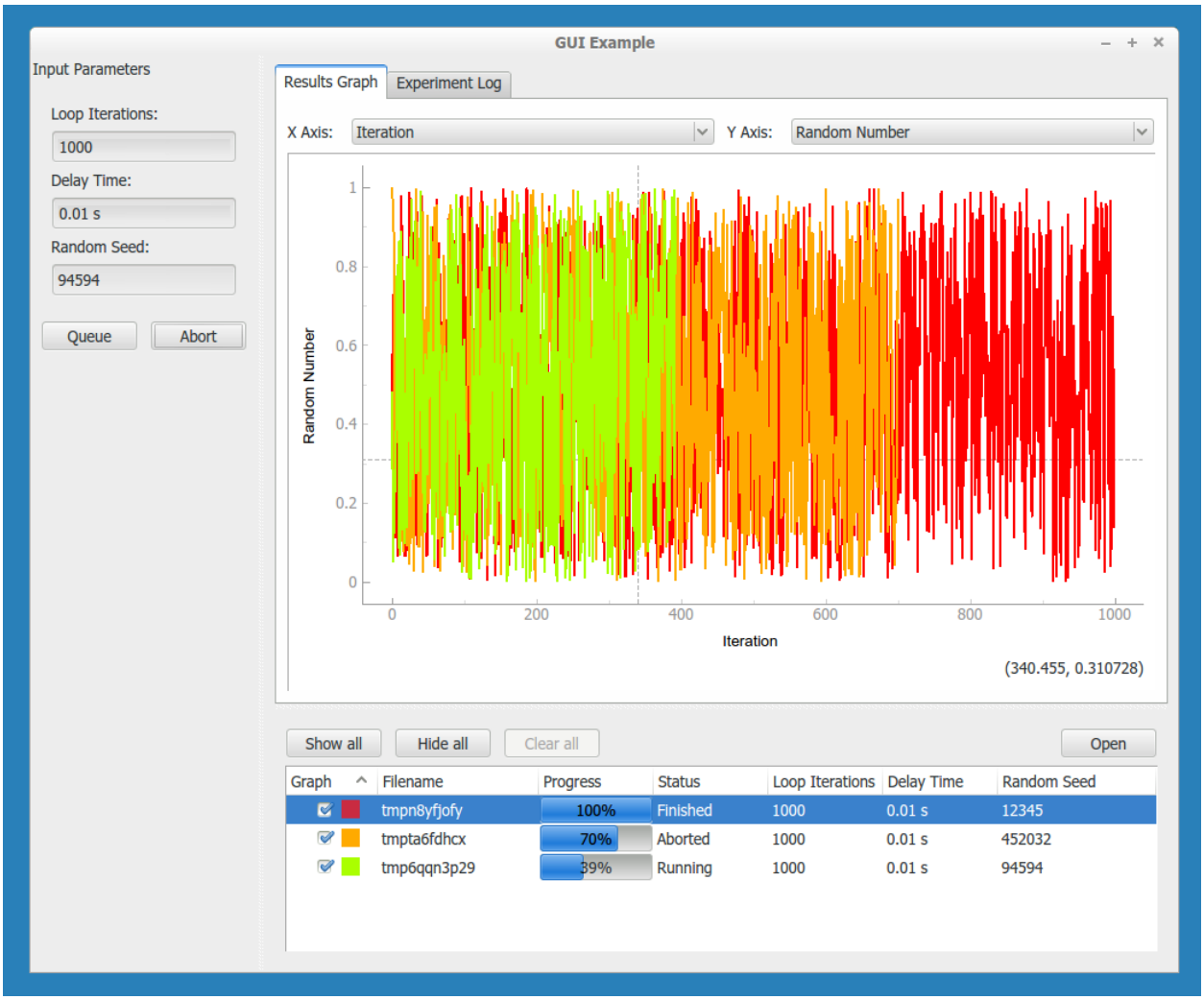
The `queue` method establishes how the Procedure object is constructed. We use the `self.make_procedure` method to create a Procedure based on the graphical input fields. Here we are free to modify the procedure before putting it on the queue. In this context, the Manager uses an Experiment object to keep track of the Procedure, Results, and its associated graphical representations in the browser and live-graph. This is then given to the Manager to queue the experiment.



By default the Manager starts a measurement when its procedure is queued. The abort button can be pressed to stop an experiment. In the Procedure, the `self.should_stop` call will catch the abort event and halt the measurement. It is important to check this value, or the Procedure will not be responsive to the abort event.



If you abort a measurement, the resume button must be pressed to continue the next measurement. This allows you to adjust anything, which is presumably why the abort was needed.



Now that you have learned about the ManagedWindow, you have all of the basics to get up and running quickly with a measurement and produce an easy to use graphical interface with PyMeasure.

CHAPTER 4

pymeasure.adapters

The adapter classes allow the instruments to be independent of the communication method used. The classes can be directly imported from `pymeasure.adapters` for convenience.

Adapters for specific instruments should be grouped in a `adapters.py` file in the corresponding manufacturer's folder of `pymeasure.instruments`.

4.1 Adapter base class

class `pymeasure.adapters.adapter.`**Adapter**

Base class for Adapter child classes, which adapt between the Instrument object and the connection, to allow flexible use of different connection techniques.

This class should only be inhereted from.

ask(*command*)

Writes the command to the instrument and returns the resulting ASCII response

Parameters **command** – SCPI command string to be sent to the instrument

Returns String ASCII response of the instrument

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)

Returns a numpy array from a query for binary data

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

read()

Reads until the buffer is empty and returns the resulting ASCII respone

Returns String ASCII response of the instrument.

values(*command*)

Writes a command to the instrument and returns a list of formatted values from the result

Parameters **command** – SCPI command to be sent to the instrument

Returns String ASCII response of the instrument

write(*command*)
Writes a command to the instrument

Parameters **command** – SCPI command string to be sent to the instrument

4.2 Fake adapter

class pymeasure.adapters.adapter.**FakeAdapter**
The Fake adapter class is provided for debugging purposes, which returns valid data for each Adapter method

4.3 Serial adapter

class pymeasure.adapters.serial.**SerialAdapter**(*port*, ***kwargs*)
Adapter class for using the Python Serial package to allow serial communication to instrument

Parameters

- **port** – Serial port
- **kwargs** – Any valid key-word argument for serial.Serial

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)
Returns a numpy array from a query for binary data

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

read()
Reads until the buffer is empty and returns the resulting ASCII respone

Returns String ASCII response of the instrument.

values(*command*)
Writes a command to the instrument and returns a list of formatted values from the result

Parameters **command** – SCPI command to be sent to the instrument

Returns String ASCII response of the instrument

write(*command*)
Writes a command to the instrument

Parameters **command** – SCPI command string to be sent to the instrument

4.4 Prologix adapter

class pymeasure.adapters.prologix.**PrologixAdapter**(*port*, *address=None*, ***kwargs*)
Encapsulates the additional commands necessary to communicate over a Prologix GPIB-USB Adapter, using the SerialAdapter.

Each PrologixAdapter is constructed based on a serial port or connection and the GPIB address to be communicated to. Serial connection sharing is achieved by using the *gpib()* method to spawn new PrologixAdapters for different GPIB addresses.

Parameters

- **port** – The Serial port name or a serial.Serial object
- **address** – Integer GPIB address of the desired instrument
- **kwargs** – Key-word arguments if constructing a new serial object

Variables **address** – Integer GPIB address of the desired instrument

To allow user access to the Prologix adapter in Linux, create the file: `/etc/udev/rules.d/51-prologix.rules`, with contents:

```
SUBSYSTEMS=="usb",ATTRS{idVendor}=="0403",ATTRS{idProduct}=="6001",MODE="0666"
```

Then reload the udev rules with:

```
sudo udevadm control --reload-rules
sudo udevadm trigger
```

gpib(*address*)
Returns and PrologixAdapter object that references the GPIB address specified, while sharing the Serial connection with other calls of this function

Parameters **address** – Integer GPIB address of the desired instrument

Returns PrologixAdapter for specific GPIB address

read()
Reads the response of the instrument until timeout

Returns String ASCII response of the instrument

set_defaults()
Sets up the default behavior of the Prologix-GPIB adapter

wait_for_srq(*timeout=25*, *delay=0.1*)
Blocks until a SRQ, and leaves the bit high

Parameters

- **timeout** – Timeout duration in seconds
- **delay** – Time delay between checking SRQ in seconds

write(*command*)
Writes the command to the GPIB address stored in the `address`

Parameters **command** – SCPI command string to be sent to the instrument

4.5 VISA adapter

class `pymeasure.adapters.visa.`**VISAAdapter**(*resourceName*, ***kwargs*)
Adapter class for the VISA library using PyVISA to communicate to instruments. Inherit from either class VISAAdapter14 or VISAAdapter15.

Parameters

- **resource** – VISA resource name that identifies the address

- **kwargs** – Any valid key-word arguments for constructing a PyVISA instrument

binary_values(*command*, *header_bytes=0*, *dtype=<class 'numpy.float32'>*)
Returns a numpy array from a query for binary data

Parameters

- **command** – SCPI command to be sent to the instrument
- **header_bytes** – Integer number of bytes to ignore in header
- **dtype** – The NumPy data type to format the values with

Returns NumPy array of values

read()
Reads until the buffer is empty and returns the resulting ASCII respone

Returns String ASCII response of the instrument.

values(*command*, *separator=','*)
Writes a command to the instrument and returns a list of numerical values from the result.

Parameters **command** – SCPI command to be sent to the instrument.

Returns A list of numerical values.

version
The string of the PyVISA version in use

write(*command*)
Writes a command to the instrument

Parameters **command** – SCPI command string to be sent to the instrument

CHAPTER 5

pymeasure.experiment

This section contains specific documentation on the classes and methods of the package.

5.1 Experiment class

The Experiment class is intended for use in the Jupyter notebook environment.

class pymeasure.experiment.experiment.**Experiment**(*title*, *procedure*, *analyse=<function Experiment.<lambda>>*)

Class which starts logging and creates/runs the results and worker processes.

```
    procedure = Procedure()
    experiment = Experiment(title, procedure)
    experiment.start()
    experiment.plot_live('x', 'y', style='.-')

    for a multi-subplot graph:

    import pylab as pl
    ax1 = pl.subplot(121)
    experiment.plot('x','y',ax=ax1)
    ax2 = pl.subplot(122)
    experiment.plot('x','z',ax=ax2)
    experiment.plot_live()
```

Variables **value** – The value of the parameter

Parameters

- **title** – The experiment title
- **procedure** – The procedure object
- **analyse** – Post-analysis function, which takes a pandas dataframe as input and returns it with added (analysed) columns. The analysed results are accessible via experiment.data, as opposed to experiment.results.data for the 'raw' data.
- **_data_timeout** – Time limit for how long live plotting should wait for datapoints.

clear_plot()

Clear the figures and plot lists.

data
Data property which returns analysed data, if an analyse function is defined, otherwise returns the raw data.

pcolor(*xname*, *yname*, *zname*, **args*, ***kwargs*)
Plot the results from the experiment.data pandas dataframe in a pcolor graph. Store the plots in a plots list attribute.

plot(**args*, ***kwargs*)
Plot the results from the experiment.data pandas dataframe. Store the plots in a plots list attribute.

plot_live(**args*, ***kwargs*)
Live plotting loop for jupyter notebook, which automatically updates (an) in-line matplotlib graph(s). Will create a new plot as specified by input arguments, or will update (an) existing plot(s).

start()
Start the worker

update_line(*ax*, *hl*, *xname*, *yname*)
Update a line in a matplotlib graph with new data.

update_pcolor(*ax*, *xname*, *yname*, *zname*)
Update a pcolor graph with new data.

update_plot()
Update the plots in the plots list with new data from the experiment.data pandas dataframe.

wait_for_data()
Wait for the data attribute to fill with datapoints.

pymeasure.experiment.experiment.**create_filename**(*title*)
Create a new filename according to the style defined in the config file. If no config is specified, create a temporary file.

pymeasure.experiment.experiment.**get_array**(*start*, *stop*, *step*)
Returns a numpy array from start to stop

pymeasure.experiment.experiment.**get_array_steps**(*start*, *stop*, *numsteps*)
Returns a numpy array from start to stop in numsteps

pymeasure.experiment.experiment.**get_array_zero**(*maxval*, *step*)
Returns a numpy array from 0 to maxval to -maxval to 0

5.2 Listener class

class pymeasure.experiment.listeners.**Listener**(*port*, *topic=''*, *timeout=0.01*)
Base class for Threads that need to listen for messages on a ZMQ TCP port and can be stopped by a thread-safe method call

class pymeasure.experiment.listeners.**Recorder**(*results*, *queue*)
Recorder loads the initial Results for a filepath and appends data by listening for it over a queue. The queue ensures that no data is lost between the Recorder and Worker.

5.3 Procedure class

class pymeasure.experiment.procedure.**Procedure**(***kwargs*)
Provides the base class of a procedure to organize the experiment execution. Procedures should be run by

Workers to ensure that asynchronous execution is properly managed.

```
procedure = Procedure()
results = Results(procedure, data_filename)
worker = Worker(results, port)
worker.start()
```

Inheriting classes should define the startup, execute, and shutdown methods as needed. The shutdown method is called even with a software exception or abort event during the execute method.

If keyword arguments are provided, they are added to the object as attributes.

check_parameters()
Raises an exception if any parameter is missing before calling the associated function. Ensures that each value can be set and got, which should cast it into the right format. Used as a decorator @check_parameters on the startup method

execute()
Preforms the commands needed for the measurement itself. During execution the shutdown method will always be run following this method. This includes when Exceptions are raised.

gen_measurement()
Create MEASURE and DATA_COLUMNS variables for get_datapoint method.

parameter_objects()
Returns a dictionary of all the Parameter objects and grabs any current values that are not in the default definitions

parameter_values()
Returns a dictionary of all the Parameter values and grabs any current values that are not in the default definitions

parameters_are_set()
Returns True if all parameters are set

refresh_parameters()
Enforces that all the parameters are re-cast and updated in the meta dictionary

set_parameters(*parameters*, *except_missing=True*)
Sets a dictionary of parameters and raises an exception if additional parameters are present if except_missing is True

shutdown()
Executes the commands necessary to shut down the instruments and leave them in a safe state. This method is always run at the end.

startup()
Executes the commands needed at the start-up of the measurement

class pymeasure.experiment.procedure.**UnknownProcedure**(*parameters*)
Handles the case when a *Procedure* object can not be imported during loading in the *Results* class

5.4 Parameter classes

The parameter classes are used to define input variables for a *Procedure*. They each inherit from the *Parameter* base class.

class pymeasure.experiment.parameters.**BooleanParameter**(*name*, *default=None*, *ui_class=None*)
Parameter sub-class that uses the boolean type to store the value.

Variables **value** – The boolean value of the parameter

Parameters

- **name** – The parameter name
- **default** – The default boolean value
- **ui_class** – A Qt class to use for the UI of this parameter

class pymeasure.experiment.parameters.**FloatParameter**(*name*, *units=None*, *minimum=-1000000000.0*, *maximum=1000000000.0*, ***kwargs*)

Parameter sub-class that uses the floating point type to store the value.

Variables **value** – The floating point value of the parameter

Parameters

- **name** – The parameter name
- **units** – The units of measure for the parameter
- **minimum** – The minimum allowed value (default: -1e9)
- **maximum** – The maximum allowed value (default: 1e9)
- **default** – The default floating point value
- **ui_class** – A Qt class to use for the UI of this parameter

class pymeasure.experiment.parameters.**IntegerParameter**(*name*, *units=None*, *minimum=-1000000000.0*, *maximum=1000000000.0*, ***kwargs*)

Parameter sub-class that uses the integer type to store the value.

Variables **value** – The integer value of the parameter

Parameters

- **name** – The parameter name
- **units** – The units of measure for the parameter
- **minimum** – The minimum allowed value (default: -1e9)
- **maximum** – The maximum allowed value (default: 1e9)
- **default** – The default integer value
- **ui_class** – A Qt class to use for the UI of this parameter

class pymeasure.experiment.parameters.**ListParameter**(*name*, *choices=None*, *units=None*)

Parameter sub-class that stores the value as a list.

Parameters

- **name** – The parameter name
- **choices** – An explicit list of choices, which is disregarded if None
- **units** – The units of measure for the parameter
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter

class pymeasure.experiment.parameters.**Measurable**(*name*, *fget=None*, *units=None*, *measure=True*, *default=None*, ***kwargs*)

Encapsulates the information for a measurable experiment parameter with information about the name, fget function and units if supplied. The value property is called when the procedure retrieves a datapoint and calls the fget function. If no fget function is specified, the value property will return the latest set value of the parameter (or default if never set).

Variables **value** – The value of the parameter

Parameters

- **name** – The parameter name
- **fget** – The parameter fget function (e.g. an instrument parameter)
- **default** – The default value

class pymeasure.experiment.parameters.**Parameter**(*name*, *default=None*, *ui_class=None*)

Encapsulates the information for an experiment parameter with information about the name, and units if supplied.

Variables **value** – The value of the parameter

Parameters

- **name** – The parameter name
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter

is_set()

Returns True if the Parameter value is set

class pymeasure.experiment.parameters.**VectorParameter**(*name*, *length=3*, *units=None*, ***kwargs*)

Parameter sub-class that stores the value in a vector format.

Variables **value** – The value of the parameter as a list of floating point numbers

Parameters

- **name** – The parameter name
- **length** – The integer dimensions of the vector
- **units** – The units of measure for the parameter
- **default** – The default value
- **ui_class** – A Qt class to use for the UI of this parameter

5.5 Worker class

class pymeasure.experiment.workers.**Worker**(*results*, *log_queue=None*, *log_level=20*, *port=None*)

Worker runs the procedure and emits information about the procedure and its status over a ZMQ TCP port. In a child thread, a Recorder is run to write the results to

emit(*topic*, *data*)

Emits data of some topic over TCP

5.6 Results class

class pymeasure.experiment.results.**Results**(*procedure*, *data_filename*)

The Results class provides a convenient interface to reading and writing data in connection with a *Procedure* object.

Variables

- **COMMENT** – The character used to identify a comment (default: #)
- **DELIMITER** – The character used to delimit the data (default: ,)
- **LINE_BREAK** – The character used for line breaks (default n)
- **CHUNK_SIZE** – The length of the data chuck that is read

Parameters

- **procedure** – Procedure object
- **data_filename** – The data filename where the data is or should be stored

format(*data*)

Returns a formatted string containing the data to be written to a file

header()

Returns a text header to accompany a datafile so that the procedure can be reconstructed

labels()

Returns the columns labels as a string to be written to the file

static load(*data_filename*, *procedure_class=None*)

Returns a Results object with the associated Procedure object and data

parse(*line*)

Returns a dictionary containing the data from the line

static parse_header(*header*, *procedure_class=None*)

Returns a Procedure object with the parameters as defined in the header text.

reload()

Preforms a full reloading of the file data, neglecting any changes in the comments

pymeasure.experiment.results.**unique_filename**(*directory*, *prefix='DATA'*, *suffix=''*, *ext='csv'*, *dated_folder=False*, *index=True*, *datetimeformat='%Y%m%d'*)

Returns a unique filename based on the directory and prefix

CHAPTER 6

pymeasure.display

This section contains specific documentation on the classes and methods of the package.

6.1 Browser classes

class pymeasure.display.browser.**Browser**(*procedure_class*, *display_parameters*, *measured_quantities*, *sort_by_filename=False*, *parent=None*)

Graphical list view of *Experiment* objects allowing the user to view the status of queued Experiments as well as loading and displaying data from previous runs.

In order that different Experiments be displayed within the same Browser, they must have entries in *DATA_COLUMNS* corresponding to the *measured_quantities* of the Browser.

add(*experiment*)

Add a *Experiment* object to the Browser. This function checks to make sure that the Experiment measures the appropriate quantities to warrant its inclusion, and then adds a BrowserItem to the Browser, filling all relevant columns with Parameter data.

6.2 Curves classes

class pymeasure.display.curves.**BufferCurve**(*errors=False*, ***kwargs*)

Creates a curve based on a predefined buffer size and allows data to be added dynamically, in additon to supporting error bars

append(*x*, *y*, *xError=None*, *yError=None*)

Appends data to the curve with optional errors

prepare(*size*, *dtype=<class 'numpy.float32'>*)

Prepares the buffer based on its size, data type

class pymeasure.display.curves.**Crosshairs**(*plot*, *pen=None*)

Attaches crosshairs to the a plot and provides a signal with the x and y graph coordinates

mouseMoved(*event=None*)

Updates the mouse position upon mouse movement

update()

Updates the mouse position based on the data in the plot. For dynamic plots, this is called each time the data changes to ensure the x and y values correspond to those on the display.

class pymeasure.display.curves.**ResultsCurve**(*results*, *x*, *y*, *xerr=None*, *yerr=None*, *force_reload=False*, ***kwargs*)

Creates a curve loaded dynamically from a file through the Results object and supports error bars. The data can be forced to fully reload on each update, useful for cases when the data is changing across the full file instead of just appending.

update()

Updates the data by polling the results

6.3 Inputs classes

class pymeasure.display.inputs.**Input**(*parameter*)

Takes a Parameter object in the constructor and has a parameter method

update_parameter()

Mutates the self._parameter variable to update its value

6.4 Listeners classes

class pymeasure.display.listeners.**Monitor**(*queue*)

Monitor listens for status and progress messages from a Worker through a queue to ensure no messages are lost

class pymeasure.display.listeners.**QListener**(*port*, *topic=''*, *timeout=0.01*)

Base class for QThreads that need to listen for messages on a ZMQ TCP port and can be stopped by a thread- and process-safe method call

6.5 Log classes

6.6 Manager classes

class pymeasure.display.manager.**Experiment**(*results*, *curve*, *browser_item*, *parent=None*)

The Experiment class helps group the *Procedure*, *Results*, and their display functionality. Its function is only a convenient container.

Parameters

- **procedure** – *Procedure* object
- **results** – *Results* object
- **curve** – *ResultsCurve* object
- **browser_item** – BrowserItem object

class pymeasure.display.manager.**ExperimentQueue**

Represents a Queue of Experiments and allows queries to be easily preformed

has_next()

Returns True if another item is on the queue

next()

Returns the next experiment on the queue

class pymeasure.display.manager.**Manager**(*plot*, *browser*, *port=5888*, *log_level=20*, *parent=None*)

Controls the execution of *Experiment* classes by implementing a queue system in which Experiments are added, removed, executed, or aborted. When instantiated, the Manager is linked to a *Browser* and a PyQtGraph *PlotItem* within the user interface, which are updated in accordance with the execution status of the Experiments.

abort()
Aborts the currently running Experiment, but raises an exception if there is no running experiment

clear()
Remove all Experiments

is_running()
Returns True if a procedure is currently running

load(*experiment*)
Load a previously executed Experiment

next()
Initiates the start of the next experiment in the queue as long as no other experiments are currently running and there is a procedure in the queue.

queue(*experiment*)
Adds an experiment to the queue.

remove(*experiment*)
Removes an Experiment

resume()
Resume processing of the queue.

6.7 Plotter class

class pymeasure.display.plotter.**Plotter**(*results*, *refresh_time=0.1*)
Plotter dynamically plots data from a file through the Results object and supports error bars.

6.8 Qt classes

All Qt imports should reference pymeasure.display.Qt, for consistant importing from either PySide or PyQt4.

Qt.**fromUi**(**args*, ***kwargs*)
Returns a Qt object constructed using loadUiType based on its arguments. All QWidget objects in the form class are set in the returned object for easy accessiblity.

6.9 Thread classes

class pymeasure.display.thread.**StoppableQThread**(*parent=None*)
Base class for QThreads which require the ability to be stopped by a thread-safe method call

join(*timeout=0*)
Joins the current thread and forces it to stop after the timeout if necessary

Parameters **timeout** – Timeout duration in seconds

6.10 Widget classes

class pymeasure.display.widgets.**PlotFrame**(*x_axis=None*, *y_axis=None*, *refresh_time=0.2*, *check_status=True*, *parent=None*)

Combines a PyQtGraph Plot with Crosshairs. Refreshes the plot based on the refresh_time, and allows the axes to be changed on the fly, which updates the plotted data

parse_axis(*axis*)

Returns the units of an axis by searching the string

class pymeasure.display.widgets.**PlotWidget**(*columns*, *x_axis=None*, *y_axis=None*, *refresh_time=0.2*, *check_status=True*, *parent=None*)

Extends the PlotFrame to allow different columns of the data to be dynamically choosen

6.11 Windows classes

class pymeasure.display.windows.**ManagedWindow**(*procedure_class*, *inputs=[]*, *displays=[]*, *x_axis=None*, *y_axis=None*, *log_channel=''*, *log_level=20*, *parent=None*)

The ManagedWindow uses a Manager to control Workers in a Queue, and provides a simple interface. The queue method must be overwritten by a child class which is required to pass an Experiment containing the Results and Procedure to self.manager.queue.

queue()

This method should be overwritten by the child class. The self.manager.queue method should be passed an Experiment object which contains the Results and Procedure to be run.

set_parameters(*parameters*)

This method should be overwritten by the child class. The parameters argument is a dictionary of Parameter objects. The Parameters should overwrite the GUI values so that a user can click "Queue" to capture the same parameters.

CHAPTER 7

pymeasure.instruments

This section contains specific documentation on the classes and methods of the package.

7.1 Keithley instruments

This section contains specific documentation on the classes and methods of the package.

7.1.1 Keithley 2000 Multimeter

class pymeasure.instruments.keithley.keithley2000.**Keithley2000**(*resourceName*, ***kwargs*)

average
Obtain the filter setting.

Returns (number of counts, status ON/OFF, control MOVing/REPeat)

bandwidth
Obtain the bandwidth.

beep(*freq*, *dur*)
Make a beep sound

Parameters

- **freq** – Frequency, Hz
- **dur** – Duration, seconds

check_errors()
Read all errors from the instrument.

config
Return the current configuration.

get_average()
Obtain the filter setting.

Returns (number of counts, status ON/OFF, control MOVing/REPeat)

get_bandwidth()
Obtain the bandwidth.

get_config()
Return the current configuration.

get_nplc()
Return the current NPLC (number of power line cycles).

get_range()
Get the maximum limit of current configuration.

Returns (Maximum limit, Auto Range status)

get_reference()
Obtain the reference setting.

Returns (Relative value, status ON/OFF)

nplc
Return the current NPLC (number of power line cycles).

range
Get the maximum limit of current configuration.

Returns (Maximum limit, Auto Range status)

reference
Obtain the reference setting.

Returns (Relative value, status ON/OFF)

reset()
Reset instrument.

set_average(*count*, *method='REPeat'*)
Make multiple readings and output the average

Parameters

- **count** – number of repeats, 1 - 100

  if count = 1, average is OFF

- **method** – either "REPeat" (default) or "MOVing"

set_bandwidth(*bandwidth*)
Set bandwidth for AC measurement.

set_config(*config*, *range=0*, *nplc=2*, *bandwidth=1000*)
Set configuration.

Parameters

- **config** – String describing the function, such as 'VAC', 'R4W', etc.
- **Range** – Maximum limit of reading, default = 0 (auto range).
- **nplc** – Number of power line cycles, default = 2.
- **bandwidth** – Bandwidth for AC measurement, default = 1000.

set_range(*maxvalue*)
Set range to accommodate maxvalue.

auto range ON if maxvalue = 0

set_reference(*RefValue*)
Set reference value for output. No reference if RefValue is 0

7.1.2 Keithley 2400 SourceMeter

class pymeasure.instruments.keithley.keithley2400.**Keithley2400**(*resourceName*, ***kwargs*)

This is the class for the Keithley 2000-series instruments

config_current_source(*source_current=0.0*, *complicance_voltage=0.1*, *current_range=0.001*, *auto_range=True*)
Set up to source current

config_voltage_source(*source_voltage=0.0*, *compliance_current=0.1*, *current_range=2.0*, *voltage_range=2.0*, *auto_range=True*)
Set up to source voltage

disable_buffer()
Disables the connection between measurements and the buffer, but does not abort the measurement process

disable_output_trigger()
Disables the output trigger for the Trigger layer

is_buffer_full()
Returns True if the buffer is full of measurements

max_current
Returns the maximum current from the buffer

max_resistance
Returns the maximum resistance from the buffer

max_voltage
Returns the maximum voltage from the buffer

maximums
Returns the calculated maximums for voltage, current, and resistance from the buffer data as a list

mean_current
Returns the mean current from the buffer

mean_resistance
Returns the mean resistance from the buffer

mean_voltage
Returns the mean voltage from the buffer

means
Returns the calculated means (averages) for voltage, current, and resistance from the buffer data as a list

measure_resistance(*nplc=1*, *resistance=1000.0*, *auto_range=True*)
Sets up to measure resistance

measure_voltage(*nplc=1*, *voltage=1000.0*, *auto_range=True*)
Sets up to measure voltage

min_current
Returns the minimum current from the buffer

min_resistance
Returns the minimum resistance from the buffer

min_voltage
Returns the minimum voltage from the buffer

minimums
Returns the calculated minimums for voltage, current, and resistance from the buffer data as a list

set_continous()
Sets the Keithley to continously read samples and turns off any buffer or output triggering

set_external_trigger(*line=1*)
Sets up the measurments to be taken on the specified line of an external trigger

set_immediate_trigger()
Sets up the measurement to be taken with the internal trigger at the maximum sampling rate

set_output_trigger(*line=1*, *after='DEL'*)
Sets up an output trigger on the specified trigger link line number, with the option of supplyiny the part of the measurement after which the trigger should be generated (default to Delay, which is right before the measurement)

set_timed_arm(*interval*)
Sets up the measurement to be taken with the internal trigger at a variable sampling rate defined by the interval in seconds between sampling points

set_trigger_counts(*arm*, *trigger*)
Sets the number of counts for both the sweeps (arm) and the points in those sweeps (trigger), where the total number of points can not exceed 2500

standard_devs
Returns the calculated standard deviations for voltage, current, and resistance from the buffer data as a list

std_current
Returns the current standard deviation from the buffer

std_resistance
Returns the resistance standard deviation from the buffer

std_voltage
Returns the voltage standard deviation from the buffer

stop_buffer()
Aborts the arming and triggering sequence and uses a Selected Device Clear (SDC) if possible

use_front_terminals()
Uses the front terminals instead of the rear

use_rear_terminals()
Uses the rear terminals instead of the front

wait_for_buffer(*has_aborted=<function Keithley2400.<lambda>>*, *time_out=60*, *time_step=0.01*)
Blocks waiting for a full buffer or an abort event with timing set in units of seconds

CHAPTER 8

Contributing

Contributions to the instrument repository and the main code base are encouraged. Since the code is hosted on GitHub, contributions should be added by forking the repository and submitting a pull request. Do not make your updates on the master branch. Instead make a new branch and work on that branch. To ensure consistency, follow the coding standards for PyMeasure.

Unit testing is an important part of keeping the package running. When adding a feature that can be readily tested, include a unit test compatible with py.test so that our continuous integration services can ensure that your features are retained and do not conflict with existing behavior.

CHAPTER 9

Reporting an error

Please report all errors to the Issues section of the PyMeasure GitHub repository. Use the search function to determine if there is an existing or resolved issued before posting.

CHAPTER 10

Adding Instruments

CHAPTER 11

Coding Standards

In order to maintain consistency across the different instruments in the PyMeasure repository, we enforce the following standards.

11.1 Python style guides

Python 3 is used in PyMeasure. The PEP8 style guide and PEP257 docstring conventions should be followed.

Function and variable names should be lower case with underscores as needed to seperate words. Camel case should not be used, unless working with Qt, where it is common.

11.2 Standard naming

Since many instruments have similar functions, a few naming conventions have been adopted to make the interface more consistent.

11.3 Usage of getter and setter

Many settings (such as range, enabled status, etc) are provided by the instrument with a pair of actions: one is to read the current setting value, the other is to assign a value to the setting. One can write two methods, get_setting() and set_setting() for instance, to handle these two actions; or altenatively use getter and setter decorators. In most cases, the two ways are equivalent. In order to incorporate different programming styles, and for the convenience of users, our convention is as follow: - Write two functions get_setting() and set_setting(). The latter one should have only one non-keyword argument (but can have many keyword arguments). - Define a property setting = property(get_setting, set_setting).

Using a buffer

```
set_buffer
wait_for_buffer
get_buffer
```

CHAPTER 12

Authors

PyMeasure was started in 2013 by Colin Jermain and Graham Rowlands at Cornell University, when it became appearent that both were working on similar Python packages for scientific measurements. PyMeasure combined these efforts and continues to gain valuable contributions from other scientists who are interested in advancing measurement software.

The following developers have contributed to the PyMeasure package:

Colin Jermain
Graham Rowlands
Minh-Hai Nguyen
Guen Prawiro-Atmodjo

CHAPTER 13

License

p

H

I

J

K

L

M

N

P

Q

R

S

U

V

W