ruediPy documentaion

Matthias Brennwald

Version June 6, 2016

Abstract

ruediPy is a collection of Python programs for instrument control and data acquisition using RUEDI instruments[(1)]. ruediPy also includes some GNU Octave (or Matlab) tools to load, process, and manipulate RUEDI data acquired with ruediPy Python classes.

ruediPy is distributed as free software under the GNU General Public License (see LICENSE.txt).

This document describes the ruediPy software only. The RUEDI instrument is described in a separate document[(1)].

Contents

1 Overview

ruediPy is a collection of Python programs for instrument control and data acquisition using RUEDI instruments. ruediPy also includes some GNU Octave (or Matlab) tools to load, process, and manipulate RUEDI data acquired with ruediPy Python classes. The RUEDI instrument itself is described in a separate document(1).

The Python classes for instrument control and data acquisition are designed to reflect the different hardware units of a RUEDI instrument, such as the mass spectrometer, selector valve, or probes for total gas pressure or temperature. These classes, combined with additional helper classes (e.g., for data file handling), allow writing simple Python scripts that perform user-defined procedures for a specific analysis task.

The GNU Octave tools (m-files) are designed to work hand-in-hand with the data files produced by the data acquisition parts of the Python classes. ★[1]

ruediPy is developed on Linux and Mac OS X systems, but should also work on any other system that run Python and GNU Octave.

2 Obtaining and installing `ruediPy`

ruediPy can be downloaded from http://brennmat.github.io/ruediPy either as a compressed archive file, or using Subversion or Git version control systems. ruediPy can be installed to just about any directory on the computer that is used for instrument control – but the user home directory (~/ruediPy) may seem like a sensible choice, and that's what is assumed throughout the examples shown in this manual.

3 Python classes

3.1 Overview

The Python classes are used to control the various hardware units of the RUEDI instruments, to acquire measurement data, and to write these data to well-formatted and structured data files.

Currently, the following classes are implemented:

- rgams_SRS.py: control and data acquisition from the SRS mass spectrometer

[1] TO DO: expand this: load raw data, process / calibrate data, etc.

- `selectorvalve_VICI.py`: control of the VICI inlet valve
- `pressuresensor_WIKA.py`: control and data acquisition from the WIKA pressure sensor
- `datafile.py`: data file handling
- `misc.py`: helper functions

The Python class files are located at `~/ruediPy/python/classes/`. To make sure Python knows where to find the `ruediPy` Python classes, set your `PYTHONPATH` environment variable accordingly.[2]

These classes are continuously expanded and new classes are added to `ruediPy` as required by new needs or developments of the RUEDI instruments. The various methods / functions included are documented in the class files. Due to the ongoing development of the code, it seems futile to keep an up-to-date copy of the methods / functions documentation in this manual. Please refer to the detailed documentation in the class files directly.

3.2 Python classes reference

3.2.1 Class `rgams_SRS`

`ruediPy/python/classes/rgams_SRS.py`

ruediPy class for SRS RGA-MS control.

Method `filament_off`

```
rgams_SRS.filament_off()

Turn off filament current.

INPUT:
(none)

OUTPUT:
(none)
```

[2] A convenient method to achieve this on Linux or similar UNIXy systems is to put the following line to the `.profile` file: `export PYTHONPATH=~/ruediPy/python`

Method `filament_on`

```
rgams_SRS.filamenOn()

Turn on filament current at default current value.

INPUT:
(none)

OUTPUT:
(none)
```

Method `get_detector`

```
det = rgams_SRS.get_detector()

Return current detector (Faraday or electron multiplier)

INPUT:
(none)

OUTPUT:
det:  detecor (string):
det='F' for Faraday
det='M' for electron Multiplier
```

Method `get_electron_energy`

```
val = rgams_SRS.get_electron_energy()

Return electron energy of the ionizer (in eV).

INPUT:
```

```
(none)

OUTPUT:
val:  electron energy in eV
```

Method `get_filament_current`

```
val = rgams_SRS.get_filament_current()

Return filament current (in mA)

INPUT:
(none)

OUTPUT:
val:  filament current in mA
```

Method `get_noise_floor`

```
val = rgams_SRS.get_noise_floor()

Get noise floor (NF) parameter for RGA measurements (noise floor controls
gate time, i.e., noise vs.  measurement speed).

INPUT:
(none)

OUTPUT:
val:  NF noise floor parameter value, 0...7 (integer)
```

Method `has_multiplier`

```
val = rgams_SRS.has_multiplier()
```

Check if MS has electron multiplier installed.

INPUT:
(none)

OUTPUT:
val: result flag, val = 0 --> MS has no multiplier, val <> 0: MS has multiplier

Method `label`
`l = rgams_SRS.label()`

Return label / name of the RGAMS object.

INPUT:
(none)

OUTPUT:
l: label / name (string)

Method `mz_max`
`val = rgams_SRS.mz_max()`

Determine highest mz value supported by the MS.

INPUT:
(none)

OUTPUT:
val: max. supported mz value

Method `param_IO`

```
ans = rgams_SRS.param_IO(cmd,ansreq)

Set / read parameter value of the SRS RGA.

INPUT:
cmd:  command string that is sent to RGA (see RGA manual for commands
and syntax)
ansreq:  flag indicating if answer from RGA is expected:
ansreq = 1:  answer expected, check for answer
ansreq = 0:  no answer expected, don't check for answer

OUTPUT:
ans:  answer / result returned from RGA
```

Method `peak`

```
val,unit = rgams_SRS.peak(mz,gate,f)

Read out detector signal at single mass (m/z value).

INPUT:
mz:  m/z value (integer)
gate:  gate time (seconds)
f:  file object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.

OUTPUT:
val:  signal intensity (float)
unit:  unit (string)

NOTE FROM THE SRS RGA MANUAL:
Single mass measurements are commonly performed in sets
where several different masses are monitored sequencially
and in a merry-go-round fashion.
For best accuracy of results, it is best to perform the consecutive
mass measurements in a set with the same type of detector
and at the same noise floor (NF) setting.
```

Fixed detector settings eliminate settling time problems in the electrometer and in the CDEM's HV power supply.

Method `peakbuffer_add`

```
srsrga.peakbuffer_add(t,mz,intens)

Add data to PEAKS data buffer

INPUT:
t:  epoch time
mz:  mz values (x-axis)
intens:  intensity values (y-axis)
det:  detector (char/string)

OUTPUT:
(none)
```

Method `plot_peakbuffer`

```
srsrga.plot_peakbuffer()

Plot trend (or update plot) of values in PEAKs data buffer (e.g.  after
adding data)
NOTE: plotting may be slow, and it may therefore be a good idea to keep
the update interval low to avoid affecting the duty cycle.

INPUT:
(none)

OUTPUT:
(none)
```

Method `plot_scan`

```
srsrga.plot_scan(mz,intens,unit)

Plot scan data

INPUT:
mz:  mz values (x-axis)
intens:  intensity values (y-axis)
unit:  intensity unit (string)

OUTPUT:
(none)
```

Method `scan`

```
M,Y,unit = rgams_SRS.scan(low,high,step,gate,f,p)

Analog scan

INPUT:
low:  low m/z value
high:  high m/z value
step:  scan resolution (number of mass increment steps per amu)
step = integer number --> use given number (high number equals small
mass increments between steps)
step = '*' use default value (step = 10)
gate:  gate time (seconds)
f:  file object or 'nofile':
if f is a DATAFILE object, the scan data is written to the current data
file
if f = 'nofile' (string), the scan data is not written to a datafile

OUTPUT:
M: mass values (mz, in amu)
Y: signal intensity values (float)
unit:  unit of Y (string)
```

Method `set_detector`

```
rgams_SRS.set_detector()

Set current detetector used by the MS (direct the ion beam to the Faraday
or electron multiplier detector).

INPUT:
det:  detecor (string):
det='F' for Faraday
det='M' for electron multiplier

OUTPUT:
(none)
```

Method `set_electron_energy`

```
rgams_SRS.set_electron_energy(val)

Set electron energy of the ionizer.

INPUT:
val:  electron energy in eV

OUTPUT:
(none)
```

Method `set_filament_current`

```
rgams_SRS.set_filament_current(val)

Set filament current.

INPUT:
val:  current in mA

OUTPUT:
```

(none)

Method `set_gate_time`

```
val = rgams_SRS.set_gate_time()

Set noi floor (NF) parameter for RGA measurements according to desired
gate time (by choosing the best-match NF value).

INPUT:
gate:  gate time in (fractional) seconds

OUTPUT:
(none)

NOTE (1):
FROM THE SRS RGA MANUAL:
Single mass measurements are commonly performed in sets
where several different masses are monitored sequencially
and in a merry-go-round fashion.
For best accuracy of results, it is best to perform the consecutive
mass measurements in a set with the same type of detector
and at the same noise floor (NF) setting.
Fixed detector settings eliminate settling time problems
in the electrometer and in the CDEM HV power supply.

NOTE (2):
Experiment gave the following gate times vs NF parameter values:

NF gate (seconds)
0 2.4
1 1.21
2 0.48
3 0.25
4 0.163
5 0.060
6 0.043
7 0.025
```

Method `set_noise_floor`
`val = rgams_SRS.set_noise_floor()`

Set noise floor (NF) parameter for RGA measurements (noise floor controls gate time, i.e., noise vs. measurement speed).

INPUT:
NF: noise floor parameter value, 0...7 (integer)

OUTPUT:
(none)

Method `warning`
`rgams_SRS.warning(msg)`

Issue warning about issues related to operation of MS.

INPUT:
msg: warning message (string)

OUTPUT:
(none)

Method `zero`
`val,unit = rgams_SRS.zero(mz,mz_offset,gate,f)`

Read out detector signal at single mass with relative offset to given m/z value (this is useful to determine the baseline near a peak at a given m/z value), see rgams_SRS.peak())
The detector signal is read at mz+mz_offset

```
INPUT:
mz:  m/z value (integer)
mz_offset:  offset relative m/z value (integer).
gate:  gate time (seconds)
f:  file object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.

OUTPUT:
val:  signal intensity (float)
unit:  unit (string)

NOTE FROM THE SRS RGA MANUAL:
Single mass measurements are commonly performed in sets
where several different masses are monitored sequencially
and in a merry-go-round fashion.
For best accuracy of results, it is best to perform the consecutive
mass measurements in a set with the same type of detector
and at the same noise floor (NF) setting.
Fixed detector settings eliminate settling time problems
in the electrometer and in the CDEM's HV power supply.
```

3.2.2 Class `selectorvalve_VICI`

`ruediPy/python/classes/selectorvalve_VICI.py`

ruediPy class for VICI valve control.

Method `getpos`

```
pos = selectorvalve_VICI.getpos()

Get valve position

INPUT:
(none)

OUTPUT:
```

```
pos:  valve postion (integer)
```

Method `label`

```
label = selectorvalve_VICI.label()

Return label / name of the SELECTORVALVE object

INPUT:
(none)

OUTPUT:
label:  label / name (string)
```

Method `setpos`

```
selectorvalve_VICI.setpos(val,f)

Set valve position

INPUT:
val:  new valve position (integer)
f:  datafile object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.

OUTPUT:
(none)
```

Method `warning` No method description available.

3.2.3 Class `pressuresensor_WIKA`

`ruediPy/python/classes/pressuresensor_WIKA.py` ruediPy class for WIKA pressure sensor control.

Method `label`

```
label = pressuresensor_WIKA.label()

Return label / name of the PRESSURESENSOR object

INPUT:
(none)

OUTPUT:
label:  label / name (string)
```

Method `pressure`

```
press,unit = pressuresensor_WIKA.pressure(f)

Read out current pressure value (in hPa).

INPUT:
f:  file object for writing data (see datafile.py).  If f = 'nofile',
data is not written to any data file.

OUTPUT:
press:  pressure value in hPa (float)
```

Method `serial_checksum`

```
cs = pressuresensor_WIKA.serial_checksum( cmd )

Return checksum used for serial port communication with WIKA pressure
sensor.

INPUT:
cmd:  serial-port command string without checksum
```

```
OUTPUT:
cs:  checksum byte
```

Method `warning`

```
pressuresensor_WIKA.warning(msg)

Issue warning about issues related to operation of pressure sensor.

INPUT:
msg:  warning message (string)

OUTPUT:
(none)
```

3.2.4 Class `datafile`

`ruediPy/python/classes/datafile.py`

ruediPy class for handling of data files.

Method `basepath`

```
pat = datafile.basepath()

Return the base path where datafiles are stored

INPUT:
(none)

OUTPUT:
pat:  datafile base path (string)
```

Method `close`

```
datafile.close()

Close the currently open data file (if any)

INPUT:
(none)

OUTPUT:
(none)
```

Method `fid`

```
f = datafile.fid()

Return the file ID / object of the current file

INPUT:
(none)

OUTPUT:
f:  datafile object
```

Method `label`

```
lab = datafile.label()

Return label / name of the DATAFILE object

INPUT:
(none)

OUTPUT:
lab:  label / name (string)
```

Method `name`

```
n = datafile.name()
```

Return the name the current file (or empty string if not datafile has been created)

INPUT:
(none)

OUTPUT:
n: ile name (string)

Method `next`

```
datafile.next()
```

Close then current data file (if it's still open) and start a new file.

INPUT:
typ (optional): string that will be appended to the file name. This may be useful to indicate 'type' of mesurement data, e.g. typ = 'SAMPLE', typ = 'S', typ = 'BLANK', typ = 'B', typ = 'CAL', typ = 'C', etc.). The string can be anything. If omitted or typ = '', nothing will be appended to the file name

OUTPUT:
(none)

Method `warning`

```
datafile.warning(msg)
```

Warn about issues related to DATAFILE object

INPUT:
msg: warning message (string)

OUTPUT:
(none)

Method `write_analysis_type`

```
datafile.write_analysis_type( caller , typ , timestmp )

Write ANALYSIS TYPE info line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
typ:  analysis type (string / char)
timestmp:  timestamp of the peak measurement (see misc.nowUNIX)

OUTPUT:
(none)
```

Method `write_comment`

```
datafile.write_comment(caller,cmt)

Write COMMENT line to the data file.

INPUT:
caller:  label / name of the calling object (string)
cmt:  comment string

OUTPUT:
(none)
```

Method `write_peak`

```
datafile.write_peak(caller,mz,intensity,unit,det,gate,timestmp)
```

```
Write PEAK data line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
mz:  mz value (integer)
intensity:  peak intensity value (float)
unit:  unit of peak intensity value (string)
det:  detector (string), e.g., det='F' for Faraday or det='M' for multiplier
gate:  gate time (float)
timestmp:  timestamp of the peak measurement (see misc.nowUNIX)

OUTPUT:
(none)
```

Method `write_pressure`
`datafile.write_pressure(caller,label,value,unit,timestmp)`

```
Write PRESSURE data line to the data file.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
label:  name/label of the calling object (string)
value:  pressure value (float)
unit:  unit of peak intensity value (string)
timestmp:  timestamp of the peak measurement (see misc.nowUNIX)

OUTPUT:
(none)
```

Method `write_scan`
`datafile.write_scan(caller,mz,intensity,unit,det,gate,timestmp)`

Write PEAK data line to the data file.

INPUT:
caller: type of calling object, i.e. the "data origin" (string)
label: name/label of the calling object (string)
mz: mz values (floats)
intensity: intensity values (floats)
unit: unit of intensity values (string)
det: detector (string), e.g., det='F' for Faraday or det='M' for multiplier
gate: gate time (float)
timestmp: timestamp of the peak measurement (see misc.nowUNIX)

OUTPUT:
(none)

Method `write_valve_pos`
`datafile.write_valve_pos(caller,position,timestmp)`

Write multi-port valve position data line to the data file.

INPUT:
caller: type of calling object, i.e. the "data origin" (string)
label: name/label of the calling object (string)
position: valve position (integer)
timestmp: timestamp of the peak measurement (see misc.nowUNIX)

OUTPUT:
(none)

Method `write_zero`
`datafile.write_zero(caller,mz,mz_offset,intensity,unit,det,gate,timestmp)`

Write ZERO data line to the data file.

INPUT:
caller: type of calling object, i.e. the "data origin" (string)
label: name/label of the calling object (string)
mz: mz value (integer)
mz_offset: mz offset value (integer, positive offset corresponds to higher mz value)
intensity: zero intensity value (float)
unit: unit of peak intensity value (string)
det: detector (string), e.g., det='F' for Faraday or det='M' for multiplier
gate: gate time (float)
timestmp: timestamp of the peak measurement (see misc.nowUNIX)

OUTPUT:
(none)

Method `writeln`
datafile.writeln(caller,identifier,data,timestmp)

Write a text line to the data file (format: TIMESTAMP CALLER[LABEL] IDENTIFIER: DATA). CALLER, LABEL, and IDENTIFIER should not contain spaces or similar white space (will be removed before writing to file). If LABEL == '' or LABEL == CALLER, the [LABEL] part is omitted.

INPUT:
caller: type of calling object, i.e. the "data origin" (string)
label: name/label of the calling object (string)
identifier: data type identifier (string)
data: data / info string
timestmp: timestamp of the data in unix time (see misc.nowUNIX)

OUTPUT:
(none)

3.2.5 Class `misc`

`ruediPy/python/classes/misc.py`

ruediPy class with helper functions.

Method `now_UNIX`

```
dt = misc.now_UNIX()

Return date/time as UNIX time / epoch (seconds after Jan 01 1970 UTC)

INPUT:
(none)

OUTPUT:
dt:  date-time (UNIX / epoch time)
```

Method `now_string`

```
dt = misc.now_string()

Return string with current date and time

INPUT:
(none)

OUTPUT:
dt:  date-time (string) in YYYY-MM-DD hh:mm:ss format
```

Method `warnmessage`

```
misc.warnmessage(caller,msg)

Print a warning message

INPUT:
```

```
caller:  caller label / name of the calling object (string)
msg:  warning message

OUTPUT:
(none)
```

4 GNU Octave tools

$\star$[3]

5 Examples

$\star$[4]

References

[1] M. S. Brennwald, M. Schmidt, J. Oser, and R. Kipfer. A portable mass spectrometric system for on-site environmental gas analysis. *Env. Sci. Technol.*, in prep.

[3] TO DO: add content
[4] TO DO: add content