ruediPy documentaion

Matthias Brennwald

Version July 12, 2016

Abstract

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.

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[(?)].

Contents

1 Overview **2**

2 Obtaining and installing ruediPy **2**

3 Python classes **3**

3.1 Overview 3

3.2 Python classes reference 4

3.2.1 Class rgams_SRS 4

3.2.2 Class selectorvalve_VICI 17

3.2.3 Class pressuresensor_WIKA 18

3.2.4 Class temperaturesensor_MAXIM 19

3.2.5 Class datafile 21

3.2.6 Class misc 28

4 GNU Octave tools **29**

5 Examples **30**

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(?).

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.

As an example, here's a step-by-step list of terminal commands to install ruediPyon a Linux computer running Ubuntu 16.04. Other Linux distributions will be similar. The user account name in this example is "brennmat", and this user account is enabled for sudo operations (i.e., it has 'admin' rights):

1. Update system software to latest versions and install basic software requirements for ruediPy:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install subversion python-pip
python-serial python-matplotlib python-scipy
```

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

```
sudo pip install pydigitemp
```

2. Download ruediPy:

   ```
   svn co https://github.com/brennmat/ruediPy.git/trunk ~/ruediPy
   ```

3. Permanently add `ruediPy` to the Python searchpath (this requires the user to log out and log back in to become active):

   ```
   echo export PYTHONPATH=~/ruediPy/python >> ~/.profile
   ```

4. Set permission to access the serial ports (this requires the user to log out and log back in to become active):

   ```
   sudo usermod -a -G dialout brennmat
   ```

5. Prepare directory for ruediPy data files and measurement scripts:

   ```
   mkdir ~/ruedi_data
   mkdir ~/ruedi_scripts
   ```

Log out and log back in to make the above changes active. You should also consider setting up the computer to avoid going to 'sleep' mode, because this might interrupt the measurement procedure.

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
- `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)
```

Method `filament_on`
`rgams_SRS.filamenOn()`

```
Turn on filament current at default current value.
```

[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`

```
INPUT:
(none)

OUTPUT:
(none)
```

Method get_RI

```
x = rgams_SRS.get_RI(x)

Get current RI parameter value (peak-position tuning at low mz range)

INPUT:
(none)

OUTPUT:
x:  RI value

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method get_RS

```
x = rgams_SRS.get_RS(x)

Get current RS parameter value (peak-position tuning at high mz range)

INPUT:
(none)

OUTPUT:
x:  RS value

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

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

```
rgams_SRS.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`
`rgams_SRS.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`
`rgams_SRS.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_RI

```
rgams_SRS.set_RI(x)

Set RI parameter value (peak-position tuning at low mz range)

INPUT:
x:  RI voltage

OUTPUT:
(none)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"
```

Method set_RS
rgams_SRS.set_RS(x)

Set RS parameter value (peak-position tuning at high mz range)

INPUT:
x: RS value

OUTPUT:
(none)

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"

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 tune_peak_position
rgams_SRS.tune_peak_position(mz,gate,det,n=1)

Automatically adjust peak positions in mass spectrum to make sure peaks show up at the correct mz values. This is done by scanning peaks at different mz values, and determining their offset in the mz spectrum. The mass spectromter parameters ard then adjusted to minimize the mz offsets (RI and RF). This needs at least two distinct peak mz values at (one at a low and one at a high mz value). The procedure can be repeated several times.

INPUT:
mz: list of mz values where peaks are scanned
gate: list of gate times used in the scans
det: list of detectors to be used in the scans ('F' or 'M')
n: number of repetitions of the tune procedure (optional, default is n = 1)

OUTPUT:
(none)

EXAMPLE:
```
>>> MS = rgams_SRS ( serialport = '/dev/serial/by-id/usb-WuT_USB_Cable_-
2_WT2016234-if00-port0' , label = 'MS_MINIRUEDI_TEST', max_buffer_points
= 1000 )
>>> MS.filament_on()
>>> MS.tune_peak_position([14,18,28,32,40,44,84],[0.2,0.2,0.025,0.1,0.4,0.1,2.4],[
```

NOTE:
See also the SRS RGA manual, chapter 7, section "Peak Tuning Procedure"

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.

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)
unit:  unit of pressure value (string)
```

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 `temperaturesensor_MAXIM`

`ruediPy/python/classes/temperaturesensor_MAXIM.py`

ruediPy class for MAXIM DS1820 type temperature sensors (wrapper class for pydigitemp package).

Method `label`

```
label = temperaturesensor_MAXIM.label()

Return label / name of the TEMPERATURESENSOR object

INPUT:
(none)

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

Method `temperature`

```
temp,unit = temperaturesensor_MAXIM.temperature(f)

Read out current temperaure value.

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

OUTPUT:
temp:  temperature value (float)
unit:  unit of temperature value (string)
```

Method `warning`

```
temperaturesensor_MAXIM.warning(msg)

Issue warning about issues related to operation of pressure sensor.

INPUT:
```

```
msg:  warning message (string)

OUTPUT:
(none)
```

3.2.5 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(,typ='',samplename='',std_species='',std_conc='',std_-
mz='')
```

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

INPUT:
typ (optional): analysis type (string, default: typ = ''). The analysis type is written to the data file, and is appended to the file name. typ can be one of the following analysis types:
typ = 'SAMPLE' (for sample analyses)
typ = 'STANDARD' (for standard / calibration gas analyses)
typ = 'BLANK' (for blank analyses)
typ = 'UNKNOWN' (if analysis type is unknown)
typ = '' (if analysis type is unknown; nothing is added to the file name)
samplename (optional, only used if typ='SAMPLE'): description, name, or ID of sample

OUTPUT:
(none)

Method `warning`

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

Warn about issues related to DATAFILE object

INPUT:
msg: warning message (string)

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.now_UNIX)

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 pressure measurement (see misc.now_UNIX)

OUTPUT:
(none)
```

Method `write_sample_desc`

```
datafile.write_sample_desc(self,desc)

Write line with sample description (e.g., name or ID of sample)

INPUT:
desc:  sample description, name, or ID (string)

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.now_UNIX)

OUTPUT:
(none)
```

Method `write_standard_conc`

```
datafile.write_standard_conc(species,conc,mz)

Write line with standard/calibration gas information to data file:  name,
concentration/mixing ratio, and mz value of gas species.

INPUT:
caller:  type of calling object, i.e.  the "data origin" (string)
species:  name of gas species (string)
conc:  volumetric concentration / mixing ratio (float)
mz:  mz value (integer)

OUTPUT:
(none)
```

Method `write_temperature`

```
datafile.write_temperature(caller,label,value,unit,timestmp)

Write TEMPERATURE 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:  temperature value (float)
unit:  unit of peak intensity value (string)
timestmp:  timestamp of the temperature measurement (see misc.now_UNIX)

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.now_UNIX)

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.now_UNIX)

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.now_UNIX)

OUTPUT:
(none)
```

3.2.6 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]

[3] TO DO: add content

5 Examples

$\star$[4]

[4]TO DO: add content