

PSRSALSA python wrapper manual

P. Weltevrede

September 10, 2026

Contents

1	Introduction	4
1.1	Remarks about how this document is organised	4
1.2	Remarks about how to use the library	4
1.3	Remarks about numpy support	5
2	Module psrsalsa.version	6
2.1	Show and obtain version information	6
2.1.1	version.check_version_mismatch()	6
2.1.2	version.numpy_support()	6
2.1.3	version.show_libversions()	6
2.1.4	version.get_version()	7
3	Module psrsalsa.io	8
3.1	File loading and saving	8
3.1.1	io.psrdata.load()	8
3.1.2	io.psrdata.save()	8
3.1.3	io.printPSRDataFormats()	9
3.2	Data access	9
3.2.1	io.get_data()	9
3.2.2	io.get_data_as_matrix()	10
3.2.3	io.get_pulse_data()	10
3.2.4	io.set_pulse_data()	11
3.3	Archive manipulation	11
3.3.1	io.clone()	11
3.3.2	io.create_empty()	11
3.4	Header information	12
3.4.1	io.printhead()	12
3.4.2	io.get_backend()	12
3.4.3	io.set_backend()	12
3.4.4	io.get_bw()	12
3.4.5	io.set_bw()	12
3.4.6	io.get_centre_freq()	13
3.4.7	io.set_centre_freq()	13
3.4.8	io.get_debase_state()	13
3.4.9	io.set_debase_state()	13
3.4.10	io.get_dec()	13
3.4.11	io.set_dec()	14
3.4.12	io.get_dedisp_state()	14
3.4.13	io.set_dedisp_state()	14
3.4.14	io.get_defarad_state()	14
3.4.15	io.set_defarad_state()	14
3.4.16	io.get_depar_state()	15
3.4.17	io.set_depar_state()	15
3.4.18	io.get_dm()	15
3.4.19	io.set_dm()	15
3.4.20	io.get_duration()	16
3.4.21	io.get_feedtype()	16

3.4.22	io.set_feedtype()	16
3.4.23	io.get_fold_period()	16
3.4.24	io.set_fold_period()	16
3.4.25	io.get_format()	17
3.4.26	io.set_format()	17
3.4.27	io.set_freq_table()	17
3.4.28	io.get_gentype()	17
3.4.29	io.set_gentype()	17
3.4.30	io.get_institute()	18
3.4.31	io.set_institute()	18
3.4.32	io.get_mjd_start()	18
3.4.33	io.set_mjd_start()	18
3.4.34	io.get_nrbins()	19
3.4.35	io.get_nrfreqchans()	19
3.4.36	io.get_nrpols()	19
3.4.37	io.get_nrsbints()	19
3.4.38	io.get_observatory()	19
3.4.39	io.set_observatory()	19
3.4.40	io.get_observer()	20
3.4.41	io.set_observer()	20
3.4.42	io.get_poltype()	20
3.4.43	io.set_poltype()	20
3.4.44	io.get_projectID()	21
3.4.45	io.set_projectID()	21
3.4.46	io.get_psrname()	21
3.4.47	io.set_psrname()	21
3.4.48	io.get_ra()	21
3.4.49	io.set_ra()	21
3.4.50	io.get_reference_freq()	22
3.4.51	io.set_reference_freq()	22
3.4.52	io.get_rm()	22
3.4.53	io.set_rm()	22
3.4.54	io.get_scanID()	23
3.4.55	io.set_scanID()	23
3.4.56	io.get_subint_durations()	23
3.4.57	io.get_subint_freqs()	23
3.4.58	io.get_tsamp()	23
3.4.59	io.set_tsamp()	24
3.4.60	io.set_tsub_table()	24
3.4.61	io.get_xrange()	24
3.4.62	io.get_yrange()	25

4	Module psrsalsa.genops	26
4.1	Operations which generate new archives	26
4.1.1	genops.add_archives()	26
4.1.2	genops.fscr()	26
4.1.3	genops.make_profile()	27
4.1.4	genops.polselect()	27
4.1.5	genops.rebin()	27
4.1.6	genops.shuffle()	28
4.1.7	genops.tscr()	28
4.2	In-place data manipulation	29
4.2.1	genops.debase()	29
4.2.2	genops.dedisperse()	29
4.2.3	genops.deFarad()	29
4.2.4	genops.invariant_interval()	30
4.2.5	genops.norm()	30
4.2.6	genops.rotate()	31

4.2.7	genops.scale()	31
4.2.8	genops.stokes()	31
4.3	Functions that can both create new archives, or act in-place on the input	32
4.3.1	genops.subtract_profile()	32
4.4	Pulse longitude range selection	32
4.4.1	genops.regions_add_binrange()	32
4.4.2	genops.regions_check_bin()	32
4.4.3	genops.regions_count_nrbins()	33
4.4.4	genops.regions_create_empty()	33
4.4.5	genops.regions_get_binrange()	33
4.4.6	genops.regions_get_nr_regions()	33
4.4.7	genops.regions_frac_to_int()	34
4.4.8	genops.regions_int_to_frac()	34
4.4.9	genops.regions_print()	34
4.4.10	genops.regions_select()	34
5	Module psrsalsa.spec	36
5.1	Spectrum generation	36
5.1.1	spec.lrfs()	36
5.1.2	spec.lrfs_mplextent()	37
5.1.3	spec.twodfs()	37
5.1.4	spec.twodfs_xrange()	38
5.1.5	spec.twodfs_mplextent()	38
6	Module psrsalsa.fold	40
6.1	P3 folding	40
6.1.1	fold.foldP3()	40
7	Module psrsalsa.pol	42
7.1	In-place data manipulation	42
7.1.1	pol.ppol_from_IQUV()	42
8	Module psrsalsa.vonMises	43
8.1	File reading and writing	43
8.1.1	vonMises.load_model()	43
8.2	Component manipulation	43
8.2.1	vonMises.add_component()	43
8.2.2	vonMises.change_component()	43
8.2.3	vonMises.clone()	44
8.2.4	vonMises.create_empty()	44
8.2.5	vonMises.get_amp()	44
8.2.6	vonMises.get_concentration()	44
8.2.7	vonMises.get_integral()	45
8.2.8	vonMises.get_nrcomp()	45
8.2.9	vonMises.get_phase()	45
8.3	Model usage	46
8.3.1	vonMises.align_archive()	46
8.3.2	vonMises.print_model()	46
8.3.3	vonMises.create_profile()	46
8.3.4	vonMises.correlate_profile()	47
8.4	Model fitting	47
8.4.1	vonMises.refine_model()	47
9	Module psrsalsa.psrchive	49
9.1	Conversion from psrchive archive	49
9.1.1	psrchive.load()	49
9.1.2	psrchive.convert_from_psrchive()	49
	Index	50

Chapter 1

Introduction

1.1 Remarks about how this document is organised

Each sub-module of the PSRSalsa python module is a separate chapter, and each function is described in a separate subsection. The installation comes with an example python script for each sub-module (so there is a `test_io.py` that explains the use of the `psrsalsa.io` sub-module). The function descriptions in this manual are automatically generated from the source code of the PSRSalsa module and these example scripts.

Many functions have one or more arguments. Each of them is described by a variable name and a data type. The data type describes that of the variable used internally within the python module, which is C code. So these are C variable types. In most cases it should be clear what type of argument is expected. The least clear is the data type “PyObject”, which means it is a type of variable which is not directly supported in C. This can be, for example, a python list. Many functions require variables which are non-standard Python/C data types. An example of this is a variable containing a pulsar data archive. In some places in the manual this is referred to as a variable of type “PyObject”, while in other places it is referred to as having the data type “archive object”. This data type is an example of an object the python module can internally deal with, but these variables are not directly usable in Python. Access to the content of those objects go via function calls to the PSRSalsa module (such as to obtain a header parameter for example). A PyObject variable can therefore refer to different types variables, and the function descriptions should clarify what each variable is.

Optional keywords appear in the manual as `variable=value`. An often encountered example would be `verbose=True` to make the function to print extra information compared to the default `verbose=False`.

1.2 Remarks about how to use the library

Assuming you have a compiled version of the PSRSalsa module, your environment needs to be set up properly before the `psrsalsa` module works. For python to be able to import the PSRSalsa module, it will need to know where it is. Assuming the PSRSalsa module is installed in a location python doesn’t automatically know about, the `PYTHONPATH` environment variable can be used. For example, if PSRSalsa module is installed in `/home/jocelyn/psrsalsa/pymodule/build/lib.linux-x86_64-3.10/` then add this to `PYTHONPATH`. **Important:** Don’t add the directory `/home/jocelyn/psrsalsa/pymodule/build/lib.linux-x86_64-3.10/psrsalsa/`. So the path you add to `PYTHONPATH` should have a subdirectory `psrsalsa`, in which you can find files like `io.cpython-310-x86_64-linux-gnu.so`. Failure to do so means that an `import io` imports the PSRSalsa `io` sub-module rather than the standard python `io` module, which is likely to result in all sorts of weird python errors. The `PYTHONPATH` environment variable can be set (assuming you’re using a bash shell) with:

```
export PYTHONPATH=/home/jocelyn/psrsalsa/pymodule/build/lib.linux-x86_64-3.10/psrsalsa/:$PYTHONPATH
```

Here the path to the PSRSalsa module is put before anything else that was in `PYTHONPATH`. This can be useful if you want to make sure your version of the PSRSalsa module is used before it finds a potentially older version of the module in one of the other directories defined in `PYTHONPATH`.

The PSRSalsa python module depends on the PSRSalsa library. So in order to use the module, the operating system needs to know where to find it. If installed in a non-standard location,

the environment variable `LD_LIBRARY_PATH` can be used. For example, if PSRSalsa library is installed in `/home/jocelyn/psrsalsa/lib/` then add this to `LD_LIBRARY_PATH`. The path you add to `LD_LIBRARY_PATH` should have files like `libpsrsalsa.a` and/or `libpsrsalsa.so`. The `LD_LIBRARY_PATH` environment variable can be set similarly as above with:

```
export LD_LIBRARY_PATH=/home/jocelyn/psrsalsa/lib/:$LD_LIBRARY_PATH
```

1.3 Remarks about numpy support

At the time of writing numpy support is being introduced to the PSRSalsa module, and is therefore only implemented for certain functions. During compilation of the PSRSalsa module, support of numpy can be enabled/disabled as controlled by the `have_numpy` boolean variable in `setup.py`. If disabled, numpy support will not be included in the compilation. Enabling this therefore requires re-compilation of the PSRSalsa module. One can test if numpy support is enabled in the PSRSalsa module by using `psrsalsa.version.numpy_support()`.

If numpy support is enabled, PSRSalsa functions that take a python list as an argument can be provided also as a numpy array. Likewise, functions that return a python list will return a numpy array.

Enabling numpy support adds a layer of complexity to the PSRSalsa module, and an extra dependency. So enabling it adds scope for things to go wrong, such as

- Problems with finding the (correct/desired) numpy installation during compilation of the PSRSalsa module.
- Mismatch between the numpy version used to compile the PSRSalsa module and that used when using the PSRSalsa module.

These potential issues can have severe consequences, including segfaults when using the PSRSalsa module.

If the PSRSalsa module is not compiled with numpy support (or you want to avoid the PSRSalsa module to interpret the numpy array), you can convert a numpy array variable (for example called `data`) to a standard python list using `data.tolist()`.

Chapter 2

Module psrsalsa.version

2.1 Show and obtain version information

2.1.1 version.check_version_mismatch()

[boolean] version.check_version_mismatch()

Description: Check if the version of the psrsalsa library (libpsrsalsa as picked up during runtime) matches that what is expected by the python wrapper module that is being loaded (as might be defined in your PYTHONPATH). If a mismatch is identified, information about this will be printed. Note that this check is automatically performed when calling most psrsalsa functions, although if a mismatch is identified, error messages are generated without raising an exception and terminating execution of python.

Optional keywords: None

Return value:

True: No mismatch is identified

False: Mismatch is identified

2.1.2 version.numpy_support()

[boolean] version.numpy_support()

Description: Check if the psrsalsa python interface was compiled with numpy support (see Sec. 1.3).

Optional keywords: None

Return value: Boolean which is True if numpy support was enabled during compilation of the psrsalsa python interface.

2.1.3 version.show_libversions()

(void) version.show_libversions()

Description: Print the version information of the psrsalsa library which is used, as well as that of other dependencies.

Optional keywords: None

Return value: None

2.1.4 `version.get_version()`

[string] `version.get_version()`

Description: Obtain the pdrsalsa wrapper version as a string.

Optional keywords: None

Return value: String with the pdrsalsa wrapper version that is loaded.

Chapter 3

Module psrsalsa.io

3.1 File loading and saving

3.1.1 io.psrdata_load()

[archive object] = io.psrdata_load(filename [string], format=[string], nowarnings=[int], headeronly=[bool], verbose=[bool], debug=[bool])

Description: Open an (existing) archive file with name filename, and load contents in memory. See also psrchive.load() for an alternative way to load archives, which comes with advantages and disadvantages.

Optional keywords:

format: string to indicate the data format to be assumed. Without this keyword specified, psrsalsa will try to figure out the data format automatically. When this keyword is used, see io.printPSRDataFormats() for supported strings. For example, the string could be "psrfits" if the data is in psrFITS format (although that should be recognized automatically, so format is not necessary as a keyword). As an alternative to "psrfits", also "7" (as a string) should work.

nowarnings: int to indicate if some warnings should be suppressed. 0: no suppression of warnings (this is the default).

1: some warning related to checking of incomplete header information will be suppressed. Could be useful to de-clutter output when you don't rely on the header information.

2: more warnings are suppressed.

headeronly: bool (default is False). If True, the archive is opened, the header information is read in, but not the data. This can be quicker if only header information is needed to be accessed. If the data is needed at a later stage, a new call to psrdata_load() is required, which will open the file and read the header again. WARNING: This functionality has been added as an afterthought. As a result, it is likely that in many places in the python wrapper of psrsalsa no checks are performed to see if the data exists. So calls to functions that attempt to access the data in the archive might well cause python to crash!

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: archive containing header information and data.

3.1.2 io.psrdata_save()

(void) io.psrdata_save(archive [PyObject], filename [string], format=[string], argv=[PyObject], verbose=[bool], debug=[bool])

Description: Save a given archive to a file with given filename.

Optional keywords:

format: string to indicate the data format to be used. The default is psrfits. See io.printPSRDataFormats() for supported strings.

argv: Expected to be set to python variable sys.argv (which is a python list). If so, this list of command-line options is used to append to the history of the output file (if supported by the output format).

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

3.1.3 io.printPSRDataFormats()

(void) io.printPSRDataFormats()

Description: Print supported data formats.

Optional keywords: None

Return value: None

3.2 Data access

3.2.1 io.get_data()

[Python list] = io.get_data(archive [PyObject], no_numpy_out=[bool])

Description: Extract all the data of the given archive as a python list (or numpy array) of floats. It is a single list of numbers, organised such that subint is the outer loop, followed by the frequency channels, polarization channels and bin numbers. See get_pulse_data() and get_data_as_matrix() for ways to extract part of a dataset. To convert this 1D list in a 4D numpy array one can use:

```
array4D = list1D.reshape((nrsubints,nrfreqchans,nrpols,nrbins))
```

or alternatively if the list is not returned as a numpy array:

```
array4D = np.array(list1D).reshape((nrsubints,nrfreqchans,nrpols,nrbins))
```

Then the data can be accessed as array4D[subint][freq][pol][bin]

Optional keywords:

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If no_numpy_out is set to True, a python list is returned instead.

Return value: Python list (or a numpy array) with data values.

3.2.2 io.get_data_as_matrix()

[Python 2D list] = io.get_data_as_matrix(archive [PyObject], order [string], freqchan=[long], polchan=[long], subint=[long], phasebin=[long], no_numpy_out=[bool], verbose=[bool], debug=[bool])

Description: Extract the data as a 2D Python list of floats (or a numpy array, see Sec. 1.3 for some notes about numpy support). The order parameter does define what data dimension is in the columns and rows. The order parameter is a string containing 2 characters, corresponding to: S(ubints), (longitude) B(ins), F(requency channels), P(olarisation channels). Supported order values are "SB", "BS", "FB", "BF", "SF" and "FS". This is mainly a convenience function to avoid having to do a numpy reshape after calling io.get_data(archive).

Taking "SB" as an example in this description: the output matrix is organised as matrix[subint][bin]. If there are multiple frequency channels, polarization channels or phase bins that make the data 4D or 3D rather than 2D: the desired frequency/polarization/phase bin to extract can be set with the optional keywords freqchan, polchan and phasebin.

Optional keywords:

freqchan: Integer value defining the requested frequency channel to be extracted. The default is 0.

polchan: Integer value defining the requested polarization channel to be extracted. The default is 0.

subint: Integer value defining the requested subint to be extracted. The default is 0.

phasebin: Integer value defining the requested phase bin to be extracted. The default is 0.

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If no_numpy_out is set to True, a python list is returned instead.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Python matrix with the data

3.2.3 io.get_pulse_data()

[Python list] = io.get_pulse_data(archive [PyObject], subint [long], polnr [int], freqnr [int], no_numpy_out=[bool], verbose=[bool], debug=[bool])

Description: Extract a single pulse from an archive as a python list of floats. So a single subint number, polarization and frequency channel is selected. What is returned is a copy of the data, so modifying it will not change the archive.

Optional keywords:

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If no_numpy_out is set to True, a python list is returned instead.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Python list (or a numpy array) with data values.

3.2.4 io.set_pulse_data()

(void) io.set_pulse_data(archive [PyObject], pulsedata [PyObject], subint [long], polnr [int], freqnr [int], verbose=[bool], debug=[bool])

Description: Replace a single pulse in the specified archive with a python list of floats (or a numpy array could be provided as well, if numpy suport was enabled during the compilation of the psrsalsa python interface). So the selected single subint number, polarization and frequency channel is replaced. The data is to be provided as a python list (see Sec. 1.3 for some notes about numpy support).

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None.

3.3 Archive manipulation

3.3.1 io.clone()

[archive object] = io.clone(archive [PyObject], verbose=[bool], debug=[bool])

Description: Make a clone of the provided achive: a new copy of the header parameters and data. This is an independent copy, so modifying the original archive will not change the clone and vice versa.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: archive containing the clone.

3.3.2 io.create_empty()

[archive object] = io.create_empty(nsubints [long], nrbins [long], nrfreqchans=[long], nrpolns=[long], header=[PyObject], verbose=[bool], debug=[bool])

Description: Create an archive which is completely uninitialised (but see header option), except that enough memory is allocated to hold data with the requested dimensions. These dimensions, and an empty data block, are stored in the header information of the returned archive. The number of subints and the number of bins (in that order) are the two compulsory arguments of the function.

Optional keywords:

nrfreqchans: Integer value defining the requested number of frequency channels to be created. The default is 1.

nrpolns: Integer value defining the requested polarization channel to be created. The default is 1.

header: Specify an psrsalsa data archive to be used to initialise the header. The data block will be empty. Any header information that depends on the dimensions of the data, which are not necessarily the same in the archive provided with the option header, and the output archive, are ignored. For example, the length of each subint or the frequency of each non-uniformly spaced channel.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: empty archive

3.4 Header information

3.4.1 io.printhead()

(void) io.printhead(archive [PyObject])

Description: Print the header information of the provided archive.

Optional keywords: None.

Return value: None.

3.4.2 io.get_backend()

[string] io.get_backend(archive [PyObject])

Description: Get the name of the backend/instrument used to record the data.

Optional keywords: None

Return value: A string, which will be empty if the backend isn't set.

3.4.3 io.set_backend()

(void) io.set_backend(archive [PyObject], backend [string])

Description: Set the name of the backend/instrument used to record the data to the provided string.

Optional keywords: None

Return value: None

3.4.4 io.get_bw()

[float] = io.get_bw(archive [PyObject])

Description: Get the bandwidth in MHz.

Optional keywords: None

Return value: A floating point, which is the bandwidth.

3.4.5 io.set_bw()

(void) io.set_bw(archive [PyObject], bw [double])

Description: Set the bandwidth in MHz.

Optional keywords: None

Return value: None

3.4.6 io.get_centre_freq()

[float] = io.get_centre_freq(archive [PyObject])

Description: Get the centre frequency in MHz.

Optional keywords: None

Return value: A floating point, which is the centre frequency.

3.4.7 io.set_centre_freq()

(void) io.set_centre_freq(archive [PyObject], freq [double])

Description: Set the centre frequency in MHz.

Optional keywords: None

Return value: None

3.4.8 io.get_debase_state()

[long int] = io.get_debase_state(archive [PyObject])

Description: Set the baseline subtraction state of the data.

Optional keywords: None

Return value: An integer value, which can be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

3.4.9 io.set_debase_state()

(void) io.set_debase_state(archive [PyObject], state [int])

Description: Set the baseline subtraction state of the data. The state should be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

Optional keywords: None

Return value: None

3.4.10 io.get_dec()

[float] = io.get_dec(archive [PyObject])

Description: Get the declination in radians.

Optional keywords: None

Return value: The declination in radians.

3.4.11 io.set_dec()

(void) io.set_dec(archive [PyObject], dec [double])

Description: Set the declination to the value provided in radians.

Optional keywords: None

Return value: None

3.4.12 io.get_dedisp_state()

[long int] = io.get_dedisp_state(archive [PyObject])

Description: Get the dedispersion state of the data.

Optional keywords: None

Return value: An integer value, which can be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

3.4.13 io.set_dedisp_state()

(void) io.set_dedisp_state(archive [PyObject], state [int])

Description: Set the dedispersion state of the data. The state should be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

Optional keywords: None

Return value: None

3.4.14 io.get_defarad_state()

[long int] = io.get_defarad_state(archive [PyObject])

Description: Get the de-Faraday rotation state of the data.

Optional keywords: None

Return value: An integer value, which can be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

3.4.15 io.set_defarad_state()

(void) io.set_defarad_state(archive [PyObject], state [int])

Description: Set the de-Faraday rotation state of the data. The state should be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

Optional keywords: None

Return value: None

3.4.16 io.get_depar_state()

[long int] = io.get_depar_state(archive [PyObject])

Description: Get the de-parallactic angle removal state of the data.

Optional keywords: None

Return value: An integer value, which can be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

3.4.17 io.set_depar_state()

(void) io.set_depar_state(archive [PyObject], state [int])

Description: Set the de-parallactic angle removal state of the data. The state should be one of the following:

-1: The state of the data is unknown 0: The effect is uncorrected 1: The effect is removed from the data

Optional keywords: None

Return value: None

3.4.18 io.get_dm()

[float] = io.get_dm(archive [PyObject])

Description: Get the header DM.

Optional keywords: None

Return value: A floating point, which is the header DM.

3.4.19 io.set_dm()

(void) io.set_dm(archive [PyObject], dm [double])

Description: Set the header DM. This will just change the header value, without attempting changing anything in the data itself. So you can make the data and header inconsistent.

Optional keywords: None

Return value: None

3.4.20 io.get_duration()

[float] = io.get_duration(archive [PyObject])

Description: Get the duration of the observation in seconds from the header.

Optional keywords: None

Return value: Duration in seconds

3.4.21 io.get_feedtype()

[string] io.get_feedtype(archive [PyObject])

Description: Get the feed type.

Optional keywords: None

Return value: A string, which can be "Unknown", "Linear" or "Circular".

3.4.22 io.set_feedtype()

(void) io.set_feedtype(archive [PyObject], feedtype [string])

Description: Set the feed type of the provided archive with the provided string. The string should be one of following: "Unknown", "Linear" or "Circular". If something else, it defaults to "Unknown".

Optional keywords: None

Return value: None.

3.4.23 io.get_fold_period()

[float] = io.get_fold_period(archive [PyObject])

Description: Returns the fold period in the header in second. This refers to the the period of the first subint, although at the time of writing it is assumed to be constant throughout observation. If not folded or if an error was encountered, the returned period will be zero.

Optional keywords: None

Return value: Period in seconds

3.4.24 io.set_fold_period()

(void) io.set_fold_period(archive [PyObject], period [double])

Description: Sets the fold period in the header in second. At the time of writing it is assumed to be constant throughout observation. This will also sets a flag in the header that the data is folded.

Optional keywords: None

Return value: None

3.4.25 io.get_format()

[long int] = io.get_format(archive [PyObject])

Description: Get the "format" of the data. See io.set_format() for additional information.

Optional keywords: None

Return value: Format of the data (a number)

3.4.26 io.set_format()

(void) io.set_format(archive [PyObject], format [int])

Description: Probably there is no reason to use this function, apart from internal use within the python wrapper itself. It sets the "format" of the data. This is a number corresponding to the format type of data. The desired format is passed on in this function as an number. See pmod -formatlist for a list of supported numbers. But probably always the data format will be set to 99, which means that the data is loaded in memory, and therefore is no longer of a particular format.

Optional keywords: None

Return value: None

3.4.27 io.set_freq_table()

(void) io.set_freq_table(archive [PyObject], freqtable [PyObject], verbose=[bool], debug=[bool])

Description: Set/update the table with weighted centre frequencies for each channel in each subint. This table is provided as a 2D Python list (see Sec. 1.3 for some notes about numpy support) of frequencies in MHz, the first index being the subint number and the second index the frequency channel number.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

3.4.28 io.get_gentype()

[long int] = io.get_gentype(archive [PyObject])

Description: Get the "gentype" of the data. See io.set_gentype() for additional information.

Optional keywords: None

Return value: Gentype of the data (a number)

3.4.29 io.set_gentype()

(void) io.set_gentype(archive [PyObject], gentype [int])

Description: Set the "gentype" of the data. This is a number corresponding to the type of data in

the archive. This allows sanity checks related to if the right type of data is used in various functions, and could help software to identify what suitable axes labels should be in a plot. The desired gentype is passed on in this function as an number. See `pmod -gentypelist` for a list of supported numbers.

Optional keywords: None

Return value: None

3.4.30 `io.get_institute()`

[string] `io.get_institute(archive [PyObject])`

Description: Get the name of the institute this data is associated with (maybe that the observer is associated with)..

Optional keywords: None

Return value: A string, which will be empty if not set.

3.4.31 `io.set_institute()`

(void) `io.set_institute(archive [PyObject], institute [string])`

Description: Get the name of the institute this data is associated with (maybe that the observer is associated with).

Optional keywords: None

Return value: None

3.4.32 `io.get_mjd_start()`

[float] = `io.get_mjd_start(archive [PyObject])`

Description: Get the start mjd out of the header. It is supposed to correspond to the MJD of the first sample (left edge) in the specified archive, or 0 if undefined. Note that the conversion to a python float results in a loss of precision (unless a long double in C is the same as a double, which is true on some platforms).

Optional keywords: None

Return value: Start MJD, or 0 if undefined.

3.4.33 `io.set_mjd_start()`

(void) `io.set_mjd_start(archive [PyObject], mjd [double])`

Description: Change or set the start MJD of the provided archive to the specified values.

Optional keywords: None

Return value: None

3.4.34 `io.get_nrbins()`

[long int] = `io.get_nrbins`(archive [PyObject])

Description: Obtain the number of pulse-phase bins in the provided archive.

Optional keywords: None

Return value: Number of bins.

3.4.35 `io.get_nrfreqchans()`

[long int] = `io.get_nrfreqchans`(archive [PyObject])

Description: Obtain the number of frequency channels in the provided archive.

Optional keywords: None

Return value: Number of frequency channels.

3.4.36 `io.get_nrpols()`

[long int] = `io.get_nrpols`(archive [PyObject])

Description: Obtain the number of polarization channels in the provided archive.

Optional keywords: None

Return value: Number of polarization channels.

3.4.37 `io.get_nrsubints()`

[long int] = `io.get_nrsubints`(archive [PyObject])

Description: Obtain the number of subints in the provided archive.

Optional keywords: None

Return value: Number of subints.

3.4.38 `io.get_observatory()`

[string] `io.get_observatory`(archive [PyObject])

Description: Get the name of the observatory used to record the data.

Optional keywords: None

Return value: A string, which will be empty if the observatory isn't set.

3.4.39 `io.set_observatory()`

(void) `io.set_observatory`(archive [PyObject], observatory [string])

Description: Set the name of the observatory used to record the data to the provided string.

Optional keywords: None

Return value: None

3.4.40 io.get_observer()

[string] io.get_observer(archive [PyObject])

Description: Get the name of the observer (name of a person)

Optional keywords: None

Return value: A string, which will be empty if not set.

3.4.41 io.set_observer()

(void) io.set_observer(archive [PyObject], observer [string])

Description: Set the name of the observer (name of a person).

Optional keywords: None

Return value: None

3.4.42 io.get_poltype()

[string] io.get_poltype(archive [PyObject])

Description: Get the type of polarization products.

Optional keywords: None

Return value: A string, which can be one of the following: "Unknown" "Stokes" (so I, Q, U, V) "Coherency" (auto and cross-correlation products), "ILVPAAdPA" (Stokes I, linear polarization L, Stokes V, Position Angle and error on the PA). "PAAdPA" (just PA and error) "ILVPAAdPAEl" (as ILVPAAdPA, but also the total polarization P, the ellipticity and its error).

3.4.43 io.set_poltype()

(void) io.set_poltype(archive [PyObject], poltype [string])

Description: Set the polarization type of the provided archive with the provided strings (see get_poltype() for a description of possible strings). If the string is set to something which is not recognized, it defaults to "Unknown".

Optional keywords: None

Return value: None.

3.4.44 `io.get_projectID()`

[string] `io.get_projectID(archive [PyObject])`

Description: Get the ID of the project (maybe proposal code).

Optional keywords: None

Return value: A string, which will be empty if not set.

3.4.45 `io.set_projectID()`

(void) `io.set_projectID(archive [PyObject], projectID [string])`

Description: Set the ID of the project (maybe proposal code).

Optional keywords: None

Return value: None

3.4.46 `io.get_psrname()`

[string] `io.get_psrname(archive [PyObject])`

Description: Get the name of the pulsar.

Optional keywords: None

Return value: A string, which will be empty if the pulsar name isn't set.

3.4.47 `io.set_psrname()`

(void) `io.set_psrname(archive [PyObject], psrname [string])`

Description: Set the name of the pulsar.

Optional keywords: None

Return value: None

3.4.48 `io.get_ra()`

[float] `io.get_ra(archive [PyObject])`

Description: Get the right ascension in radians.

Optional keywords: None

Return value: The right ascension in radians.

3.4.49 `io.set_ra()`

(void) `io.set_ra(archive [PyObject], ra [double])`

Description: Set the right ascension to the value provided in radians.

Optional keywords: None

Return value: None

3.4.50 `io.get_reference_freq()`

[float] = `io.get_reference_freq`(archive [PyObject])

Description: Get the header reference frequency for dedispersion etc.

Optional keywords: None

Return value: A floating point, which is the reference frequency in MHz. Within `psrsalsa`, a value of `1e10` is treated the same as infinite frequency. A value of `-2` means the reference frequency is undefined.

3.4.51 `io.set_reference_freq()`

(void) `io.set_reference_freq`(archive [PyObject], reference_freq [double])

Description: Set the header reference frequency in MHz for dedispersion etc. This will just change the header value, without attempting changing anything in the data itself. So you can make the data and header inconsistent. Within `psrsalsa`, a value of `1e10` is treated the same as infinite frequency. A value of `-2` means the reference frequency is undefined.

Optional keywords: None

Return value: None

3.4.52 `io.get_rm()`

[float] = `io.get_rm`(archive [PyObject])

Description: Get the header RM.

Optional keywords: None

Return value: A floating point, which is the header RM.

3.4.53 `io.set_rm()`

(void) `io.set_rm`(archive [PyObject], rm [double])

Description: Set the header RM. This will just change the header value, without attempting changing anything in the data itself. So you can make the data and header inconsistent.

Optional keywords: None

Return value: None

3.4.54 `io.get_scanID()`

[string] `io.get_scanID(archive [PyObject])`

Description: Get the ID of the observation. This is an string which can be used for identifying an observation, in addition to the name of the pulsar.

Optional keywords: None

Return value: A string, which will be empty if not set.

3.4.55 `io.set_scanID()`

(void) `io.set_scanID(archive [PyObject], scanID [string])`

Description: Set the ID of the observation. This is an string which can be used for identifying an observation, in addition to the name of the pulsar.

Optional keywords: None

Return value: None

3.4.56 `io.get_subint_durations()`

[Python list] = `io.get_subint_durations(archive [PyObject], no_numpy_out=[bool])`

Description: Get the duration of the subints in seconds from the header.

Optional keywords:

`no_numpy_out`: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If `no_numpy_out` is set to True, a python list is returned instead.

Return value: List (or a numpy array) of durations in seconds

3.4.57 `io.get_subint_freqs()`

[Python list] = `io.get_subint_freqs(archive [PyObject], subint [long], no_numpy_out=[bool])`

Description: Get the weighted frequencies in MHz for all frequency channels in a given subint from the header.

Optional keywords:

`no_numpy_out`: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If `no_numpy_out` is set to True, a python list is returned instead.

Return value: List (or a numpy array) of channel frequencies in MHz

3.4.58 `io.get_tsamp()`

[float] = `io.get_tsamp(archive [PyObject])`

Description: Returns the sampling time in the header in second. This refers to the sampling time of the first subint, although at the time of writing it is assumed to be constant throughout observation. Note this refers to actual durations of the samples stored in the file, not necessarily the sampling time before the data got folded for example. In general, the sampling time is not necessarily the period divided by the number of bins, as not necessarily the whole period is stored.

Optional keywords: None

Return value: Sampling time in seconds

3.4.59 io.set_tsamp()

(void) io.set_tsamp(archive [PyObject], tsamp [double])

Description: Sets the sampling time in the header in second. At the time of writing it is assumed to be constant throughout observation. Note this refers to actual durations of the samples stored in the file, not necessarily the sampling time before the data got folded for example. In general, the sampling time is not necessarily the period divided by the number of bins, as not necessarily the whole period is stored.

Optional keywords: None

Return value: None

3.4.60 io.set_tsub_table()

(void) io.set_tsub_table(archive [PyObject], tsubtable [PyObject], verbose=[bool], debug=[bool])

Description: Set/update the table with integration times for each each subint. This table is provided as a Python list of integration times in seconds (or a numpy array could be provided as well, if numpy suport was enabled during the compilation of the psrsalsa python interface, see Sec. 1.3 for some notes about numpy support).

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

3.4.61 io.get_xrange()

[Python list] = io.get_xrange(archive [PyObject])

Description: Get the xrange header parameter as stored by psrsalsa for some types of data (e.g. a 2dfs). This is returned as a list of two numbers (if defined), or an empty python list. The numbers correspond to the bin centres of first and last bin.

Optional keywords: None

Return value: Python list containing the start and end value of the range. The list is never a numpy array, regardless if numpy support was enabled when the PSRSalsa python interface was compiled (see Sec. 1.3 for some notes about numpy support).

3.4.62 `io.get_yrange()`

[Python list] = `io.get_yrange`(archive [PyObject])

Description: Same, as `get_xrange()`, but for the yrange.

Optional keywords: None

Return value: Python list containing the start and end value of the range.

Chapter 4

Module psrsalsa.genops

4.1 Operations which generate new archives

4.1.1 genops.add_archives()

[archive object] = genops.add_archives(archive_list [PyObject], nsub=[long], stokesI=[bool], poladd=[bool], freqadd=[bool], verbose=[bool], debug=[bool])

Description: This generates a new archive, containing the input archives concatenated together. This is a simple concatenation without further alignment. The original archives will still be available, so this is not done "in place". The archives are provided as a python (not numpy) list of archives.

Optional keywords:

nsub [int]: This number of input subints input archive profiles are summed before written out as an output subint profile. Default is 1, meaning each input subint will be written out without summing.

stokesI=True - Generates an output archive only containing the first polarization channel (Stokes I, assuming these are Stokes parameters). Default is False.

poladd=True - The single polarization channel input archives are treated as being different polarization channels. The output is a single archive with multiple polarization channels. Default is False.

freqadd=True - The single frequency channel input archives are treated as being different frequency channels. The output is a single archive with multiple frequency channels. Default is False.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the generated archive.

4.1.2 genops.fscr()

[archive object] = genops.fscr(archive [PyObject], nrfreqs=[long], complete_only=[bool], verbose=[bool], debug=[bool])

Description: Create a new archive, where frequency channels are summed together (not averaged, as arguably makes more sense). The data should already be dedispersed and de-Faraday rotated. At least at the time of writing, PSRSalsa doesn't propagate weights when frequency channels are zapped. So when there is zapping applied, there differences in the labeling of the new frequency channels can be expected compared to what would be obtained by PSRCHIVE. The original archive will still be available after calling fscr(), so this operation is not done "in place".

Optional keywords:

nrfreqs: Default is the the number of input frequency channels, which means there will be a single output frequency channel. If nrfreqs is negative, each frequency channel appears -nrfreqs times in the output.

complete_only: Default is False. If True, all output channels contain nrfreqs channels. If False, the last frequency channel could be the sum of a smaller number of input channels.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the generated profile.

4.1.3 genops.make_profile()

[archive object] = genops.make_profile(archive [PyObject], stokesI=[bool], median=[bool], verbose=[bool], debug=[bool])

Description: This generates a new archive, containing the pulse profile after scrunching in the frequency and subint direction. The original archive will still be available, so this is not done "in place".

Optional keywords:

median=True - Generate a median profile rather than an average profile (is ignored in the frequency domain at the time of writing, but this might change).

stokesI=True - Generates Stokes I profile by ignoring the other Stokes parameters. Polarizations will be converted from coherency parameters to Stokes parameters first if necessary.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the generated profile.

4.1.4 genops.polselect()

[archive object] = genops.polselect(archive [PyObject], polnr [long], verbose=[bool], debug=[bool])

Description: Extract a given polarization channel number from the data. The original archive will still be available, so this is not done "in place".

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the data with only one polarization channel.

4.1.5 genops.rebin()

[archive object] = genops.rebin(archive [PyObject], nrbins [long], verbose=[bool], debug=[bool])

Description: This generates a new archive, containing the rebinned data with nrbins bins. The original archive will still be available, so this is not done "in place".

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the rebinned data.

4.1.6 genops.shuffle()

[archive object] = genops.shuffle(archive [PyObject], fixseed=[bool], verbose=[bool], debug=[bool])

Description: Create a new archive, where subints are shuffled in a random order. The order will be different each time the function is called, unless fixseed is used. The original archive will still be available, so this is not done "in place".

Optional keywords:

fixseed: bool (default is False) to indicate if the random number generator should be initialised with a fixed seed rather than a random seed. Note that this option is ignored, except the first time this function is used.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the shuffled archive.

4.1.7 genops.tscr()

[archive object] = genops.tscr(archive [PyObject], nrsubints=[long], complete_only=[bool], verbose=[bool], debug=[bool])

Description: Create a new archive, where subints are summed together (not averaged, as arguably makes more sense). At least at the time of writing, PSRSalsa doesn't propagate weights when zapping has been applied. The frequency labelling of the output data will be the same as the input. So when there is zapping applied, differences in the labeling of the new frequency channels can be expected compared to what would be obtained by PSRCHIVE. The original archive will still be available after calling tscr(), so this operation is not done "in place".

Optional keywords:

nrsubints: Default is the the number of input subints, which means there will be a single output subint.

complete_only: Default is False. If True, all output subints contain nrsubints subints. If False, the last subint could be the sum of a smaller number of input subints.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Archive containing the generated profile.

4.2 In-place data manipulation

4.2.1 `genops.debase()`

(void) `genops.debase`(archive [PyObject], `onpulse`=[PyObject], `slope`=[bool], `verbose`=[bool], `debug`=[bool])

Description: Subtracts baseline (flux-density offset) from data. This is an in-place operation.

Optional keywords:

`onpulse`: if defined, the onpulse region will be excluded from the calculation of the average baseline value. This is a region definition. See also section 4.4 of the manual.

`slope`: bool that defines if a simple offset is subtracted (False is the default). If True, a linear gradient is removed from the offpulse region.

`verbose`: bool to indicate if verbose mode should be switched on

`debug`: bool to indicate if debug mode should be switched on

Return value: None

4.2.2 `genops.dedisperse()`

[long int] = `genops.dedisperse`(archive [PyObject], `undo`=[bool], `old_reference_freq`=[double], `verbose`=[bool], `debug`=[bool])

Description: Rotates each profile independent of each other in Fourier space based on the expected dispersive delay w.r.t. the reference frequency specified with `io.set_reference_freq()`. Unless `undo` is set, nothing is done if the data is already dedispersed.

Optional keywords:

`undo`: Bool, default is False. If set to True, a dispersion delay is re-introduced into the data if the header parameters indicate the data was de-dispersed previously. Otherwise nothing will be done (unless `old_reference_freq` is set).

`old_reference_freq`: Floating point. If set, dedispersion will be applied again to change the reference frequency from the specified frequency in MHz (-1 or 1e10 means infinite frequency) to the new reference frequency specified in the header as set by `io.set_reference_freq()`.

`verbose`: bool to indicate if verbose mode should be switched on

`debug`: bool to indicate if debug mode should be switched on

Return value: Integer value.

- 1: Function call was ignored because of warnings (such as data already being dedispersed etc.)
- 2: Dedispersion state was changed.

4.2.3 `genops.deFarad()`

[long int] = `genops.deFarad`(archive [PyObject], `undo`=[bool], `old_reference_freq`=[double], `verbose`=[bool], `debug`=[bool])

Description: Compensate for the effect of Faraday rotation w.r.t. the reference frequency specified with `io.set_reference_freq()`. Unless `undo` is set, nothing is done if the data is already de-Faraday rotated.

Optional keywords:

undo: Bool, default is False. If set to True, Faraday rotation is re-introduced into the data if the header parameters indicate the data was de-Faraday rotated previously. Otherwise nothing will be done (unless old_reference_freq is set).

old_reference_freq: Floating point. If set, de-Faraday rotation will be applied again to change the reference frequency from the specified frequency in MHz (-1 or 1e10 means infinite frequency) to the new reference frequency specified in the header as set by io.set_reference_freq().

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Integer value.

- 1: Function call was ignored because of warnings (such as data already being de-Faraday rotated etc.)
- 2: The de-Faraday rotation state was changed.

4.2.4 genops.invariant_interval()

(void) genops.invariant_interval(archive [PyObject], verbose=[bool], debug=[bool])

Description: Convert Stokes (or coherency parameters) into a single $\sqrt{I^2 - Q^2 - U^2 - V^2}$. The original is modified in-place. Note that the 4 input polarization channels get replaced by a single polarization channel.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

4.2.5 genops.norm()

(void) genops.norm(archive [PyObject], normvalue [float], globalnorm=[bool], verbose=[bool], debug=[bool])

Description: This function normalises each subint/frequency channel independent of each other (based on first polarization channel), such that the peak amplitude in the first polarization channel corresponds to the provided normvalue.

Optional keywords:

globalnorm: bool to indicate if each frequency channel / subint should be normalised independently (False is the default). If True, all frequency channels / subints are scaled by the same normalisation constant. This normalisation constant is based on the largest peak amplitude found in the first polarization channel of all subints and channels. So the amplitude of the pulses in the first polarization channel is at most normvalue.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

4.2.6 genops.rotate()

(void) genops.rotate(archive [PyObject], phaseshift [double], subint=[long], verbose=[bool], debug=[bool])

Description: Rotates each subint in archive independent of each other using an fft algorithm. phaseshift sets a constant phase offset to be applied to each subint.

Optional keywords:

subint: If unset, all subints will be rotated by the same shift. If set (to a non-negative value), only this subint number (counting from zero) will be rotated.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

4.2.7 genops.scale()

(void) genops.scale(archive [PyObject], scale [double], offset=[double], polnr=[int], verbose=[bool], debug=[bool])

Description: Scale each subint / frequency channel / polarization channel (but see polnr) in the archive such that $\text{output} = \text{scale} \times (\text{input} + \text{offset})$.

Optional keywords:

offset: Default is 0.0.

polnr: By default, all polarization channels are scaled. If polnr is set, only this polarization channel (counting from zero) is scaled.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

4.2.8 genops.stokes()

(void) genops.stokes(archive [PyObject], verbose=[bool], debug=[bool])

Description: The polarization information can be converted from coherence parameters to Stokes parameter with the following in-place operation. If the data is already Stokes parameters, nothing will be done.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None

4.3 Functions that can both create new archives, or act in-place on the input

4.3.1 `genops.subtract_profile()`

[archive object] = `genops.subtract_profile`(archive [PyObject], profile [PyObject], `mode=[int]`, `inplace=[bool]`, `verbose=[bool]`, `debug=[bool]`)

Description: Subtract the provided profile (single subint) from each subint in archive.

Optional keywords:

`mode`: The default is 1, and this should be one of the following: `mode=1`: a straightforward subtraction is performed `mode=2`: the data in a given subint is scaled first to match the provided profile `mode=3`: the profile is scaled first to match the data

The scaling includes an offset, and is based on the first polarization channel.

`inplace`: Default is False. If True, the input archive gets modified and nothing is returned by this function. Otherwise the input archive is not modified, and the result is returned as a new archive by this function.

`verbose`: bool to indicate if verbose mode should be switched on

`debug`: bool to indicate if debug mode should be switched on

Return value: If `inplace == False`, the archive containing the result.

4.4 Pulse longitude range selection

4.4.1 `genops.regions_add_binrange()`

(void) `genops.regions_add_binrange`(regions [PyObject], left [int], right [int], `debug=[bool]`)

Description: You can add regions manually to a region selection object with this function. The selection of the left/right edge are numbers provided in bin numbers.

Optional keywords:

`debug`: bool to indicate if debug mode should be switched on

Return value: None.

4.4.2 `genops.regions_check_bin()`

[long int] = `genops.regions_check_bin`(regions [PyObject], bin [long], `regionnr=[long]`, `verbose=[bool]`, `debug=[bool]`)

Description: Check if bin number bin (counting from zero) is part of the region selection object. The default is to check if the bin number is in any of the regions contained in the provided regions (unless `regionnr` is specified, see below).

Optional keywords:

`regionnr`: Check if bin is in the this region number (counting from 1). The default is to check if it is in any of the regions.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Integer value which is either

0: bin is outside any regions (or the region specified with regionnr).

> 0: the region number (counting from 1) in which bin falls

4.4.3 `genops.regions_count_nrbins()`

[long int] = `genops.regions_count_nrbins(regions [PyObject], debug=[bool])`

Description: This function counts the unique bins specified by regions. It does not double count bins which appear in multiple selections.

Optional keywords:

debug: bool to indicate if debug mode should be switched on

Return value: The number of bins [int].

4.4.4 `genops.regions_create_empty()`

[region selection object] = `genops.regions_create_empty(verbose=[bool], debug=[bool])`

Description: You can create an empty regions selection as well, to which selections can be added later with `regions_add_binrange()`.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Region selection object

4.4.5 `genops.regions_get_binrange()`

[Python list] = `genops.regions_get_binrange(regions [PyObject], regionnr [int])`

Description: Obtain the bin selection for region selection regionnr (counting from 0) in the provided region selection object. An empty list is returned if regionnr is not defined, or is not defined in bins in the region selection object.

Optional keywords: None.

Return value: Python list (not a numpy array, even when the PSRSalsa python interface is compiled with numpy support, see Sec. 1.3) with two integers corresponding to the left/right edge. Or an empty list if the requested range doesn't exist.

4.4.6 `genops.regions_get_nr_regions()`

[long int] = `genops.regions_get_nr_regions(regions [PyObject])`

Description: Obtain the number of region selections for the given region selection object.

Optional keywords: None.

Return value: Integer corresponding to the number of defined regions

4.4.7 `genops.regions_frac_to_int()`

(void) `genops.regions_frac_to_int(regions [PyObject], scale [float], offset [float])`

Description: The inverse operation of `region_int_to_frac()` is `region_frac_to_int()`. This calculates the selected region in bin numbers from the defined selected regions in phase. The provided scaling is such that $\text{bin} = \text{phase} \times \text{scale} + \text{offset}$.

Optional keywords: None.

Return value: None.

4.4.8 `genops.regions_int_to_frac()`

(void) `genops.regions_int_to_frac(regions [PyObject], scale [float], offset [float])`

Description: The selected regions as defined with for instance `selectRegions()` are defined in bin numbers. But it can be stored as phases as well. This is typically useful when you want to apply the same selection after rebinning. The phases are calculated as $\text{phase} = \text{bin} \times \text{scale} + \text{offset}$.

Optional keywords: None.

Return value: None.

4.4.9 `genops.regions_print()`

(void) `genops.regions_print(regions [PyObject])`

Description: Print the defined regions in the provided region selection object.

Optional keywords: None.

Return value: None.

4.4.10 `genops.regions_select()`

[region selection object] = `genops.regions_select(profileI [PyObject], onlyOne=[bool], powerTwo=[bool], even=[bool], verbose=[bool], debug=[bool])`

Description: Ask the user to click in a pgplot window to make a selection. The input `profileI` is a python list (see Sec. 1.3 for some notes about numpy support) with numbers which represents the profile to be plotted (so this is not an archive).

Optional keywords:

`onlyOne`: bool to indicate at most one region can be selected. If you attempt to select another region, it overwrites the first. Default is False.

`powerTwo`: bool to indicate the width should be a power of two wide. Default is False.

even: bool to indicate the width should be an even number of bins wide. Default is False.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Region selection object

Chapter 5

Module psrsalsa.spec

5.1 Spetrum generation

5.1.1 spec.lrf()

[Python tuple] = spec.lrf(archive [PyObject], fft_size [unsigned long], modindex=[bool], stddev=[bool], phaseTrack=[bool], subpulseAmp=[bool], phaseTrackOffsets=[bool], onpulse=[PyObject], fmin=[float], fmax=[float], no_numpy_out=[bool], verbose=[bool], debug=[bool])

Description: Different things can be computed by the lrf() function, and they are returned in a Python tuple. The output contains at least the LRFS of the provided archive containing a pulse stack. This is returned as a 2D list (or numpy array, see below) with a length of $\text{nrbins} \times (\text{fft_size}/2 + 1)$ floats. Here nrbins is the number of pulse longitude bins in the input pulse stack and fft_size is expected to be even. The DC channel gets subtracted (set to zero) in the LRFS.

Optional keywords:

modindex: bool to indicate a Python list of modulation indices for each pulse longitude bin is to be obtained. No error-bar is returned. The analytic errorbar is not to be trusted, so bootstrapping is needed to get reliable errorbars. Default is False.

stddev: bool to indicate a list of standard deviations for each pulse longitude bin is to be obtained. No error-bar is returned. The analytic errorbar is not to be trusted, so bootstrapping is needed to get reliable errorbars. Default is False.

phaseTrack: bool to indicate if the subpulse phase track should be returned. This is a Python list with a length equal to the number of pulse longitude bins. You should use the fmin, fmax and onpulse keywords. Do not trust the results of this and subpulseAmp unless a single frequency bin is used in the calculation (use verbose mode to see if this is true). Default is False.

subpulseAmp: bool to indicate if the subpulse amplitude profile should be returned. This is a Python list with a length equal to the number of pulse longitude bins. As is the case for phaseTrack, use fmin and fmax and onpulse. Default is False.

phaseTrackOffsets: bool to indicate if the subpulse phase offsets as applied to the data to align the subpulse phase tracks as computed for the subsequent blocks of input data. This is because the data gets split into blocks of fft_size length. Because the subpulse modulation periodicity is not 100 per-cent constant over time, and because the fft_size will in general not be an exact multiple of the P3 periodicity, the subpulse phase tracks of individual blocks have different phase offsets.

fmin: float to indicate the minimum frequency in cpp for the calculation of subpulse phase/amplitude

fmax: float to indicate the maximum frequency in cpp for the calculation of subpulse phase/amplitude

onpulse: The onpulse region selection object (see section 4.4 of the manual) to be used to align the subpulse phases obtained from the individual blocks of `fft_size` pulses.

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as numpy arrays. If no_numpy_out is set to True, python lists are returned instead.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: A Python tuple with in it (in this order, but only the requested items):

- lrf (2D list / numpy array, always there)
- subpulse phase track (a list/numpy array of nrbins values)
- subpulse amplitude profile (a list/numpy array of nrbins values)
- phase track offsets (a list/numpy array with length equal to the number of subints divided by `fft_size`)
- modulation index (a list/numpy array of nrbins values)
- standard deviations (a list/numpy array of nrbins values)

5.1.2 spec.lrf_mplextent()

[Python list] = spec.lrf_mplextent(`fft_size` [unsigned long])

Description: This function returns a python list with two floats that can be used for the extent parameter of `imshow()` of `matplotlib`. This corresponds to the vertical extent of the lrf or 2dfs given the used `fft_size`. Note that `matplotlib` wants the bin edges as the extent, rather than the bin centres.

Optional keywords:

Return value: The python list with two floats that can be used for the extent parameter of `imshow()` of `matplotlib`.

5.1.3 spec.twodfs()

[Python list] = spec.twodfs(`archive` [PyObject], `fft_size` [unsigned long], `onpulse` [PyObject], `regionnr`=[int], `flip`=[bool], `noisesub`=[int], `subtractDC`=[bool], `no_numpy_out`=[bool], `verbose`=[bool], `debug`=[bool])

Description: The output is Python 2D list (or numpy array, see below) with a length of $\text{nrbins} \times (\text{fft_size}/2 + 1)$ floats corresponding to the 2DFS of the provided archive containing a pulse stack. Here `nrbins` is the number of bins in the selected onpulse region and `fft_size` the requested length of the DFTs. The DC channel gets subtracted (set to zero).

onpulse: The onpulse region selection (see section 4.4 of the manual) to be used to align the phases obtained from the individual blocks of `fft_size` pulses.

Optional keywords:

regionnr: int to indicate which onpulse region is used. Default is 0 (first selection).

flip: A bool which is True by default. If False, the P/P2 frequencies are returned as mathematically defined such that positive drift (towards later phases) comes out with a negative P/P2 frequency. If True (the default), positive drift corresponds to a positive P/P2 frequency.

noisesub: int to indicate how to deal with noise subtraction. Default is 1. The meaning of this parameter is as follows:

0: No subtraction will actually be done.

1: Subtract the offpulse spectra as is, therefore this leads to a $\sqrt{2}$ increase in white noise in the final

spectrum.

2: Average noise spectra in the P2 direction before subtraction. This will therefore not suppress vertical noise structure in the P2 direction, but it does in the P3 direction (slow timescale variability).

3: The noise spectra are averaged in both the P2 and P3 direction (apart from the DC channel at P/P3=0).

subtractDC: bool (default is True) to indicate if the DC channel should be removed.

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If no_numpy_out is set to True, a python list is returned instead.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Python 2D list (or numpy array) containing the 2DFS.

5.1.4 spec.twodfs_xrange()

[Python list] = spec.twodfs_xrange(archive [PyObject], onpulse [PyObject], regionnr=[int], flip=[bool])

Description: Function to obtain the value of the P/P2 frequency, expressed in cycles per period, of the centre of the first and last bin as calculated by twodfs(). The same archive, onpulse region selection object and region number as was used for the call to twodfs() should be provided.

Optional keywords:

regionnr: int to indicate which onpulse region is used. Default is 0 (first selection).

flip: A bool which is True by default. If False, the P/P2 frequencies are returned as mathematically defined such that positive drift (towards later phases) comes out with a negative P/P2 frequency. If True (the default), positive drift corresponds to a positive P/P2 frequency.

Return value: Python list (no numpy array) containing with two values, corresponding to the horizontal range in the 2DFS.

5.1.5 spec.twodfs_mplextent()

[Python list] = spec.twodfs_mplextent(archive [PyObject], fft_size [unsigned long], onpulse [PyObject], regionnr=[int], flip=[bool])

Description: The arguments to this function are the same as the function twodfs(). What is returned is a python list with four floats that can be used for the extent parameter of imshow() of matplotlib. Note that matplotlib wants the bin edges as the extent, rather than the bin centres.

Optional keywords:

regionnr: int to indicate which onpulse region is used. Default is 0 (first selection).

flip: A bool which is True by default. If False, the P/P2 frequencies are returned as mathematically defined such that positive drift (towards later phases) comes out with a negative P/P2 frequency. If True (the default), positive drift corresponds to a positive P/P2 frequency.

Return value: The python list with four floats that can be used for the extent parameter of `imshow()` of `matplotlib`.

Chapter 6

Module psrsalsa.fold

6.1 P3 folding

6.1.1 fold.foldP3()

[Python 2D list] = fold.foldP3(archive [PyObject], foldp3 [float], nrp3bins [int], onpulse=[PyObject], offset=[float], refine=[int], cyclesperblock=[int], noSmooth=[bool], smoothWidth=[float], slope=[float], no_numpy_out=[bool], verbose=[bool], debug=[bool])

Description: Folds the data from the specified archive on a given foldp3 value (expressed in pulse periods). The resulting map is a 2D list which has nrx (the number as pulse longitude bins as the input archive) by nrp3bins samples. The map is returned as a 2D Python list (or numpy array, see below) of float values.

Optional keywords:

onpulse: An onpulse pulse longitude selection (see section 4.4 of the manual). If it is set, the algorithm will only use the onpulse region to align the blocks.

offset: Float (in degrees) to be added to the subpulse phases. Default is 0.0.

refine: int value with the default being 1. If refine = 0 (corresponding to -p3fold_norefine in pfold), no attempt is made to align subsequent blocks (i.e. fixed period folding). If refine = 1, the routine will align the blocks first by doing a cross-correlation in order to try to correct for P3 changes etc. If refine is larger than 1, it will first produce a template, which then will be used to calculate more accurate offsets etc. The larger the value of refine, the better the result should be.

cyclesperblock: int value with the default being 1. This corresponds to -p3fold_cpb in pfold. It is the number of P3 cycles that are folded first before attempting to align the blocks (number of pulses folded first is foldp3×cyclesperblock rounded down).

noSmooth: bool with default value False. True would correspond to using -p3fold_nosmooth in pfold. If set, all power in a given input pulse will end up in a single P3 fold bin.

smoothWidth: This is a float corresponding to the -p3fold_nosmooth option of pfold. By default it is unset, which means each pulse is split over two p3 phase bins (unless noSmooth is set to True), depending how accurate it folds on the integer number of p3 phase bins. However, if smoothWidth is set, then each pulse is given a weight for each p3 phase defined by a gaussian with this width (in p3 phase bins).

slope: Float corresponding to -slope in pfold, given in degrees subpulse phase per degree pulse longitude. The default is 0.0. If nonzero, a subpulse phase slope is subtracted in the P3 fold map.

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with numpy support (see Sec. 1.3), the data is returned as a numpy array. If no_numpy_out is set to True, a

python list is returned instead.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: A 2D python list (or numpy array) of float values corresponding to the P3 fold.

Chapter 7

Module psrsalsa.pol

7.1 In-place data manipulation

7.1.1 `pol.ppol_from_IQUV()`

(void) `pol.ppol_from_IQUV`(archive [PyObject], onpulse [PyObject], `sigma`=[float], `norm`=[bool], `extended`=[bool], `long_offset`=[float], `pa_offset`=[float], `verbose`=[bool], `debug`=[bool])

Description: Generate ppol-style polarization products from a given archive, given an onpulse region selection (see section 4.4 of the manual). This is an in-place operation. By default, the output has 5 polarization channels, which are (in order): I, L, V, PA, PA errorbar. The PA are values in degrees.

Optional keywords:

`sigma`: float to set the sigma limit to accept a PA measurement etc. The default is 3.0.

`norm`: bool to indicate if the output profiles should be normalised. Default is True.

`extended`: If True, the output will be 8 polarization channels rather than 5. The output is now the normal 5 polarizations (I, L, V, PA, PA errorbar) followed by the total polarization P, ellipticity, and its error. The default is False.

`long_offset`: float used to add a longitude offset to the longitudes stored as part of the archive (not used within the python wrapper at the time of writing). Default is 0.0.

`pa_offset`: float used to add a offset to the PA values generated. Default is 0.0.

`verbose`: bool to indicate if verbose mode should be switched on

`debug`: bool to indicate if debug mode should be switched on

Return value: None.

Chapter 8

Module psrsalsa.vonMises

8.1 File reading and writing

8.1.1 vonMises.load_model()

[vonMises model object] = vonMises.load_model(filename [string], verbose=[bool], debug=[bool])

Description: Read a von Mises component selection file (a simple ascii file as generated by paas for example).

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: A von Mises component collection object containing the model.

8.2 Component manipulation

8.2.1 vonMises.add_component()

(void) vonMises.add_component(model [PyObject], centre [double], concentration [double], height [double], verbose=[bool], debug=[bool])

Description: Adds a component to a von Mises collection object, possibly created with create_empty(). The component is defined by the three floating point values.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None.

8.2.2 vonMises.change_component()

(void) vonMises.change_component(model [PyObject], compnr [int], centre [double], concentration [double], height [double], verbose=[bool], debug=[bool])

Description: Very similar to add_component(), but now change the given component number compnr

(counting from zero) in the given von Mises collection object.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None.

8.2.3 vonMises.clone()

[vonMises model object] = vonMises.clone(model [PyObject], verbose=[bool], debug=[bool])

Description: Make a clone of the a von Mises component collection model.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Copy of the input von Mises component collection

8.2.4 vonMises.create_empty()

[vonMises model object] = vonMises.create_empty()

Description: You can create empty von Mises component collection object, to which components can be added later with add_component().

Optional keywords: None.

Return value: An empty von Mises component collection object

8.2.5 vonMises.get_amp()

[float] = vonMises.get_amp(model [PyObject], compnr [int], verbose=[bool], debug=[bool])

Description: Given a von Mises component collection model, return the phase of component compnr (counting from zero).

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: float value with the requested value

8.2.6 vonMises.get_concentration()

[float] = vonMises.get_concentration(model [PyObject], compnr [int], verbose=[bool], debug=[bool])

Description: Given a von Mises component collection model, return the phase of component compnr (counting from zero).

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: float value with the requested value

8.2.7 vonMises.get_integral()

[float] = vonMises.get_integral(model [PyObject], compnr=[int], verbose=[bool], debug=[bool])

Description: Given a von Mises component collection model, return the integral of the model profile (all components combined, but see below) over its entire domain. The units are intensity*radians, where intensity is the unit of the amplitude of the von Mises model. Multiply with 180/pi to get units intensity*deg, or nr_bins/(2pi) to get units intensity*bins.

Optional keywords:

compnr: Integer value, which if set to a non-negative number indicates the component number (counting from zero) to use, rather than combining all components together (default behaviour).

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: float value with the requested value

8.2.8 vonMises.get_nrcomp()

[long int] = vonMises.get_nrcomp(model [PyObject], verbose=[bool], debug=[bool])

Description: Given a von Mises component collection model, return the number of components that are defined.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: float value with the requested value

8.2.9 vonMises.get_phase()

[float] = vonMises.get_phase(model [PyObject], compnr [int], verbose=[bool], debug=[bool])

Description: Given a von Mises component collection model, return the phase of component compnr (counting from zero).

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: float value with the requested value

8.3 Model usage

8.3.1 vonMises.align_archive()

(void) vonMises.align_archive(archive [PyObject], model [PyObject], locked=[bool], verbose=[bool], debug=[bool])

Description: Perform a cross-correlation between data (archive) and a von Mises component collection (model) to align the data with model. The phase offset is determined with a 1 bin precision and is applied to the archive.

Optional keywords:

locked: bool, True by default. If True, all subints are rotated by the same amount to align data. If False, subints are allowed to rotate independently.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: Relative shift as a float.

8.3.2 vonMises.print_model()

(void) vonMises.print_model(model [PyObject], verbose=[bool], debug=[bool])

Description: Print out the component parameters of the given von Mises collection object.

Optional keywords:

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: None.

8.3.3 vonMises.create_profile()

[Python list] = vonMises.create_profile(model [PyObject], nrbins [int], shift=[double], norm=[bool], no_numpy_out=[bool], verbose=[bool], debug=[bool])

Description: Create a profile made out of nrbins bins, defined by the specified von Mises collection object. This is returned as a Python list (or numpy array).

Optional keywords:

shift: double - apply a rotation in phase. Default is 0.0.

norm: bool - indicate if the profile should be normalised by peak amplitude (default is False, i.e. not normalised)

no_numpy_out: bool, default is False. This means that if the PSRSalsa python interface is compiled with

numpy support (see Sec. 1.3), the data is returned as a numpy array. If `no_numpy_out` is set to `True`, a python list is returned instead.

`verbose`: bool to indicate if verbose mode should be switched on

`debug`: bool to indicate if debug mode should be switched on

Return value: Python list (or numpy array) containing the requested profile.

8.3.4 `vonMises.correlate_profile()`

```
[float] = vonMises.correlate_profile(profile [PyObject], model [PyObject], verbose=[bool], debug=[bool])
```

Description: Perform a cross-correlation to find the offset between a profile (provided as a Python list of floats or numpy array if the PSRSalsa python interface is compiled with numpy support, see Sec. 1.3) and a von Mises component collection. The phase offset is the amount the von Mises model should be rotated to later phases to match the profile. The phase offset is determined with a 1 bin precision.

Optional keywords:

`verbose`: bool to indicate if verbose mode should be switched on

`debug`: bool to indicate if debug mode should be switched on

Return value: Relative shift as a float.

8.4 Model fitting

8.4.1 `vonMises.refine_model()`

```
[long int] = vonMises.refine_model(archive [PyObject], model [PyObject], no_neg=[bool], fixamp=[bool],  
fixwidth=[bool], fixphase=[bool], fixrelamp=[bool], fixrelphase=[bool], fixrelamp_2Dlist=[PyObject], pre_align=[bool],  
fit_baseline=[bool], suppress_error=[bool], verbose=[bool], debug=[bool])
```

Description: Fit a given von Mises component collection model to the pulse profile provided as archive. This is done by refining the parameters of model. Without any additional optional keywords, all parameters are refined. But the keywords allow more constrained fits. If there is only one fit parameter (for instance when fitting for an overall scaling only, without fitting for a baseline), a dummy fit parameter is introduced to avoid the downhill-simplex method to fail. Not elegant, but it works.

Optional keywords:

`no_neg`: bool, default `False`. If `True`, fitting of components with negative amplitude are avoided.

`fixamp`: bool, default `False`. If `True`, the amplitudes of all components are held fixed.

`fixwidth`: bool, default `False`. If `True`, the widths of all components are held fixed.

`fixphase`: bool, default `False`. If `True`, the phases of all components are held fixed.

`fixrelamp`: bool, default `False`. If `True`, the components cannot change their relative amplitude to each other. But an overall scaling is fitted for.

`fixrelphase`: bool, default `False`. If `True`, the components cannot shift relative to each other. But an overall shift is fitted for.

fixrelamp_2Dlist: a 2D list (no numpy array). For example `[[0, 1], [2, 3, 4]]` means that the relative amplitude of the first two components are held fixed, as well as the relative amplitude of the last three components. Although their relative amplitudes are fixed, a single scaling factor that will be fitted for. So, the first two, and last three, can be scaled up or down in two groups such that components 0 and 1 go up/down by the same factor. And components 2, 3 and 4 go up/down by a different factor.

pre_align: bool, default True. If False (or fixphase is True), no cross correlation between the initial model and data is performed to get a rough alignment of the model with the data before fitting starts.

fit_baseline: bool, default False. If True, it is not assumed that the baseline is removed. The level of the baseline is treated as a fit parameter.

suppress_error: bool, default False. If True, a failure of the fitting process does not result in python to terminate with an error. Instead, a warning is generated, and the return value can be checked to see if fitting was successful.

verbose: bool to indicate if verbose mode should be switched on

debug: bool to indicate if debug mode should be switched on

Return value: integer value which can be:

0 = No errors encountered

1 = Maximum nr of iterations exceeded, so no convergence expected

2 = Other error occurred

Chapter 9

Module `psrsalsa.psrchive`

9.1 Conversion from `psrchive` archive

9.1.1 `psrchive.load()`

[PyObject] = `psrchive.load(filename [string])`

Description: Somewhat experimental function which loads an archive using the `psrchive` python interface, and then use the `psrsalsa` python interface to construct a `psrsalsa` archive object that can be used in the same way as if loaded using `io.psrdata.load()`. The argument of this function is a string, which is the name of the file to be opened.

Advantage: It uses the latest and greatest `psrchive` data format handling. This includes reading in archives in TIMER format, which `io.psrdata.load()` doesn't handle.

Disadvantage: `io.psrdata.load()` handles header information better (although also not fully). Not all header information is passed on from `psrchive` to the `psrsalsa` archive object. This includes all extra parameters that are stored in fits files written out by `psrsalsa` that are not part of the official `psrfits` definition. The "gentype" or `xrange/yrange` parameters are examples of this.

Optional keywords: None

Return value: PSRSALSA archive object containing the pulsar data

9.1.2 `psrchive.convert_from_psrchive()`

[PyObject] = `psrchive.convert_from_psrchive(achive [PSRCHIVE archive])`

Description: This function is similar to the function `load()` above, except that the input is a `psrchive` archive object, rather than a file name. The same warnings and notes about advantages/disadvantages as `load()` apply here. A potentially powerful use is to load the archive with `psrchive`, do some `psrchive` operation, and then convert the `psrchive` archive object in a `psrsalsa` archive object. This makes it possible to string `psrchive` and `psrsalsa` functionality together in a single script, which could for example avoid the need of writing out intermediate files.

Optional keywords: None

Return value: PSRSALSA archive object containing the pulsar data

Index

- add_archives (in genops), 26
- add_component (in vonMises), 43
- align_archive (in vonMises), 46
- change_component (in vonMises), 43
- check_version_mismatch (in version), 6
- clone (in io), 11
- clone (in vonMises), 44
- convert_from_psrchive (in psrchive), 49
- correlate_profile (in vonMises), 47
- create_empty (in io), 11
- create_empty (in vonMises), 44
- create_profile (in vonMises), 46
- debase (in genops), 29
- dedisperse (in genops), 29
- deFarad (in genops), 29
- fold.
 - foldP3, 40
- foldP3 (in fold), 40
- fscr (in genops), 26
- genops.
 - add_archives, 26
 - debase, 29
 - dedisperse, 29
 - deFarad, 29
 - fscr, 26
 - invariant_interval, 30
 - make_profile, 27
 - norm, 30
 - polselect, 27
 - rebin, 27
 - regions_add_binrange, 32
 - regions_check_bin, 32
 - regions_count_nrbins, 33
 - regions_create_empty, 33
 - regions_frac_to_int, 34
 - regions_get_binrange, 33
 - regions_get_nr_regions, 33
 - regions_int_to_frac, 34
 - regions_print, 34
 - regions_select, 34
 - rotate, 31
 - scale, 31
 - shuffle, 28
 - stokes, 31
 - subtract_profile, 32
 - tscr, 28
 - get_amp (in vonMises), 44
 - get_backend (in io), 12
 - get_bw (in io), 12
 - get_centre_freq (in io), 13
 - get_concentration (in vonMises), 44
 - get_data (in io), 9
 - get_data_as_matrix (in io), 10
 - get_debase_state (in io), 13
 - get_dec (in io), 13
 - get_dedisp_state (in io), 14
 - get_defarad_state (in io), 14
 - get_depar_state (in io), 15
 - get_dm (in io), 15
 - get_duration (in io), 16
 - get_feedtype (in io), 16
 - get_fold_period (in io), 16
 - get_format (in io), 17
 - get_gentype (in io), 17
 - get_institute (in io), 18
 - get_integral (in vonMises), 45
 - get_mjd_start (in io), 18
 - get_nrbins (in io), 19
 - get_nrcomp (in vonMises), 45
 - get_nrfreqchans (in io), 19
 - get_nrpols (in io), 19
 - get_nrsubints (in io), 19
 - get_observatory (in io), 19
 - get_observer (in io), 20
 - get_phase (in vonMises), 45
 - get_poltype (in io), 20
 - get_projectID (in io), 21
 - get_psrname (in io), 21
 - get_pulse_data (in io), 10
 - get_ra (in io), 21
 - get_reference_freq (in io), 22
 - get_rm (in io), 22
 - get_scanID (in io), 23
 - get_subint_durations (in io), 23
 - get_subint_freqs (in io), 23
 - get_tsamp (in io), 23
 - get_version (in version), 7
 - get_xrange (in io), 24
 - get_yrange (in io), 25
- invariant_interval (in genops), 30
- io.
 - clone, 11
 - create_empty, 11
 - get_backend, 12
 - get_bw, 12

- get_centre_freq, 13
- get_data, 9
- get_data_as_matrix, 10
- get_debase_state, 13
- get_dec, 13
- get_dedisp_state, 14
- get_defarad_state, 14
- get_depar_state, 15
- get_dm, 15
- get_duration, 16
- get_feedtype, 16
- get_fold_period, 16
- get_format, 17
- get_gentype, 17
- get_institute, 18
- get_mjd_start, 18
- get_nrbins, 19
- get_nrfreqchans, 19
- get_nrpols, 19
- get_nrsubints, 19
- get_observatory, 19
- get_observer, 20
- get_poltype, 20
- get_projectID, 21
- get_psname, 21
- get_pulse_data, 10
- get_ra, 21
- get_reference_freq, 22
- get_rm, 22
- get_scanID, 23
- get_subint_durations, 23
- get_subint_freqs, 23
- get_tsamp, 23
- get_xrange, 24
- get_yrange, 25
- printhead, 12
- printPSRDataFormats, 9
- psrdata.load, 8
- psrdata.save, 8
- set_backend, 12
- set_bw, 12
- set_centre_freq, 13
- set_debase_state, 13
- set_dec, 14
- set_dedisp_state, 14
- set_defarad_state, 14
- set_depar_state, 15
- set_dm, 15
- set_feedtype, 16
- set_fold_period, 16
- set_format, 17
- set_freq_table, 17
- set_gentype, 17
- set_institute, 18
- set_mjd_start, 18
- set_observatory, 19
- set_observer, 20
- set_poltype, 20
- set_projectID, 21
- set_psname, 21
- set_pulse_data, 11
- set_ra, 21
- set_reference_freq, 22
- set_rm, 22
- set_scanID, 23
- set_tsamp, 24
- set_tsub_table, 24
- load (in psrchive), 49
- load_model (in vonMises), 43
- lfs (in spec), 36
- lfs_mplextent (in spec), 37
- make_profile (in genops), 27
- Module psrsalsa.fold, 40
- Module psrsalsa.genops, 26
- Module psrsalsa.io, 8
- Module psrsalsa.pol, 42
- Module psrsalsa.psrchive, 49
- Module psrsalsa.spec, 36
- Module psrsalsa.version, 6
- Module psrsalsa.vonMises, 43
- norm (in genops), 30
- numpy_support (in version), 6
- pol.
 - ppol_from_IQUV, 42
- polselect (in genops), 27
- ppol_from_IQUV (in pol), 42
- printhead (in io), 12
- printPSRDataFormats (in io), 9
- print_model (in vonMises), 46
- psrchive.
 - convert_from_psrchive, 49
 - load, 49
- psrdata.load (in io), 8
- psrdata.save (in io), 8
- rebin (in genops), 27
- refine_model (in vonMises), 47
- regions_add_binrange (in genops), 32
- regions_check_bin (in genops), 32
- regions_count_nrbins (in genops), 33
- regions_create_empty (in genops), 33
- regions_frac_to_int (in genops), 34
- regions_get_binrange (in genops), 33
- regions_get_nr_regions (in genops), 33
- regions_int_to_frac (in genops), 34
- regions_print (in genops), 34
- regions_select (in genops), 34
- rotate (in genops), 31
- scale (in genops), 31
- set_backend (in io), 12
- set_bw (in io), 12
- set_centre_freq (in io), 13
- set_debase_state (in io), 13

- set_dec (in io), 14
- set_dedisp_state (in io), 14
- set_defarad_state (in io), 14
- set_depar_state (in io), 15
- set_dm (in io), 15
- set_feedtype (in io), 16
- set_fold_period (in io), 16
- set_format (in io), 17
- set_freq_table (in io), 17
- set_gentype (in io), 17
- set_institute (in io), 18
- set_mjd_start (in io), 18
- set_observatory (in io), 19
- set_observer (in io), 20
- set_poltype (in io), 20
- set_projectID (in io), 21
- set_psrname (in io), 21
- set_pulse_data (in io), 11
- set_ra (in io), 21
- set_reference_freq (in io), 22
- set_rm (in io), 22
- set_scanID (in io), 23
- set_tsamp (in io), 24
- set_tsub_table (in io), 24
- show_libversions (in version), 6
- shuffle (in genops), 28
- spec.
 - lrf, 36
 - lrf_mplextent, 37
 - twodfs, 37
 - twodfs_mplextent, 38
 - twodfs_xrange, 38
- stokes (in genops), 31
- subtract_profile (in genops), 32
- tscr (in genops), 28
- twodfs (in spec), 37
- twodfs_mplextent (in spec), 38
- twodfs_xrange (in spec), 38
- version.
 - check_version_mismatch, 6
 - get_version, 7
 - numpy_support, 6
 - show_libversions, 6
- vonMises.
 - add_component, 43
 - align_archive, 46
 - change_component, 43
 - clone, 44
 - correlate_profile, 47
 - create_empty, 44
 - create_profile, 46
 - get_amp, 44
 - get_concentration, 44
 - get_integral, 45
 - get_nrcomp, 45
 - get_phase, 45
 - load_model, 43
 - print_model, 46
 - refine_model, 47