
PyFstat

Release 2.0.2

Gregory Ashton, David Keitel, Reinhard Prix, Rodrigo Tenorio

Oct 12, 2023

CONTENTS:

1	PyFstat	3
1.1	Installation	4
1.1.1	pip install from PyPI	4
1.1.2	conda installation	4
1.1.3	Docker container	4
1.1.4	pip install from github	5
1.1.5	install PyFstat from source (Zenodo or git clone)	5
1.1.6	Dependencies	5
1.1.7	Optional dependencies	6
1.1.8	Using LALSuite built from source	7
1.2	Contributing to PyFstat	7
1.3	Contributors	7
1.4	Citing this work	8
2	PyFstat module documentation	9
2.1	pyfstat package	9
2.1.1	Subpackages	9
2.1.1.1	pyfstat.utils package	9
2.1.2	Submodules	20
2.1.3	pyfstat.core module	20
2.1.4	pyfstat.grid_based_searches module	34
2.1.5	pyfstat.gridcorner module	41
2.1.6	pyfstat.injection_parameters module	42
2.1.7	pyfstat.logging module	45
2.1.8	pyfstat.make_sfts module	46
2.1.9	pyfstat.mcmc_based_searches module	55
2.1.9.1	Defining the prior	55
2.1.10	pyfstat.optimal_setup_functions module	66
2.1.11	pyfstat.snr module	67
2.1.12	pyfstat.tcw_fstat_map_funcs module	70
2.1.13	Module contents	75
3	Examples	77
3.1	Grid searches for isolated CW	77
3.1.1	Directed grid search: Linear spindown	77
3.1.2	Directed grid search: Monochromatic source	80
3.1.3	Targeted grid search with line-robust BSGL statistic	82
3.1.4	Directed grid search: Quadratic spindown	84
3.2	MCMC searches for isolated CW signals	87
3.2.1	MCMC search: Semicoherent F-statistic with initialisation	87

3.2.2	MCMC search: Semicohherent F-statistic	89
3.2.3	MCMC search: Fully coherent F-statistic	91
3.2.4	MCMC search with fully coherent BSGL statistic	93
3.3	Comparison between MCMC and Grid searches	96
3.3.1	MCMC search v.s. grid search	96
3.4	Multi-stage MCMC follow up	103
3.4.1	Follow up example	103
3.5	Binary-modulated CW searches	106
3.5.1	Binary CW example: Semicohherent MCMC search	106
3.5.2	Binary CW example: Comparison between MCMC and grid search	109
3.6	Glitch robust CW searches	115
3.6.1	MCMC search on data presenting a glitch	115
3.6.2	Glitch examples: Make data	116
3.6.3	Glitch robust grid search	118
3.6.4	Glitch robust MCMC search	120
3.7	Transient CW searches	122
3.7.1	Long transient search examples: Make data	122
3.7.2	Short transient search examples: Make data	124
3.7.3	Short transient grid search	125
3.7.4	Long transient MCMC search	127
3.7.5	Short transient MCMC search	129
3.8	Other examples	131
3.8.1	Compute a spectrogram	131
3.8.2	Cumulative coherent 2F	132
3.8.3	Software injection into pre-existing data files	134
3.8.4	Randomly sampling parameter space points	137
4	Indices and tables	141
	Bibliography	143
	Python Module Index	145
	Index	147

This is a python package providing an interface to perform F-statistic based searches for continuous gravitational waves (CWs), built on top of the [LALSuite](#) library.

The source repository and issue tracker for PyFstat can be found at github.com/PyFstat/PyFstat.

This page contains basic information about the PyFstat package, including installation instructions, a contributing guide and the proper way to cite the package and the underlying scientific literature. This is equivalent to the package's [README.md](#) file .

See [here](#) for the full API documentation.

PYFSTAT

This is a python package providing an interface to perform F-statistic based continuous gravitational wave (CW) searches, built on top of the [LALSuite library](#).

Getting started:

- This README provides information on *installing*, *contributing* to and *citing* PyFstat.
- PyFstat usage and its API are documented at pyfstat.readthedocs.io.
- We also have a number of [tutorials](#) and [examples](#), demonstrating different use cases. You can run them locally, or online as jupyter notebooks with [binder](#).
- The [project wiki](#) is mainly used for developer information.
- A [changelog](#) is also available.
- PyFstat $\geq 2.0.0$ requires a recent LALSuite (≥ 7.13) / lalpulsar (≥ 6.0) including the [SFTv3 specification](#). If you need to work with older versions, the last PyFstat release supporting those was 1.19.1.

1.1 Installation

PyFstat releases can be installed in a variety of ways, including `pip install` from PyPI [<#pip-install-from-PyPi>](#), *conda*, *Docker/Singularity images*, and *from source releases on Zenodo*. Latest development versions can *also be installed with pip or from a local git clone*.

If you don't have a matching python installation (currently 3.8 to 3.11) on your system, then Docker or conda are the easiest paths.

In either case, be sure to also check out the notes on *dependencies* and *citing this work*.

If you run into problems with ephemerides files, check the wiki page on [ephemerides installation](#).

1.1.1 pip install from PyPI

PyPI releases are available from <https://pypi.org/project/PyFstat/>.

A simple

```
pip install pyfstat
```

should give you the latest release version with all dependencies; recent releases now also include (or pull in, via dependency) a sufficient minimal set of ephemerides files.

If you are not installing into a *venv* or *conda environment* (you really should!), on many systems you may need to use the `--user` flag.

Note that the PyFstat installation will fail at the LALSuite dependency stage if your `pip` is too old (e.g. 18.1); to fix this, do

```
pip install --upgrade pip setuptools
```

1.1.2 conda installation

See [this wiki page](#) for further instructions on installing conda itself, installing PyFstat into an existing environment, or for `.yaml` recipes to set up a PyFstat-specific environment both for normal users and for developers.

If getting PyFstat from conda-forge, it already includes the required ephemerides files.

1.1.3 Docker container

Ready-to-use PyFstat containers are available at the [Packages](#) page. A GitHub account together with a personal access token is required. [Go to the wiki page](#) to learn how to pull them from the GitHub registry using Docker or Singularity.

1.1.4 pip install from github

Development versions of PyFstat can also be easily installed by pointing pip directly to this git repository, which will give you the latest version of the master branch:

```
pip install git+https://github.com/PyFstat/PyFstat
```

or, if you have an ssh key installed in github:

```
pip install git+ssh://git@github.com/PyFstat/PyFstat
```

This should pull in all dependencies in the same way as installing from PyPI.

1.1.5 install PyFstat from source (Zenodo or git clone)

You can download a source release tarball from [Zenodo](#) and extract to an arbitrary temporary directory. Alternatively, clone this repository:

```
git clone https://github.com/PyFstat/PyFstat.git
```

The module and associated scripts can be installed system wide (or to the currently active venv), assuming you are in the (extracted or cloned) source directory, via

```
python setup.py install
```

As a developer, alternatively

```
python setup.py develop
```

or

```
pip install -e /path/to/PyFstat
```

can be useful so you can directly see any changes you make in action. Alternatively (not recommended!), add the source directory directly to your python path.

To check that the installation was successful, run

```
python -c 'import pyfstat'
```

if no error message is output, then you have installed pyfstat. Note that the module will be installed to whichever python executable you call it from.

This should pull in all dependencies in the same way as installing from PyPI.

1.1.6 Dependencies

PyFstat uses the following external python modules, which should all be pulled in automatically if you use pip:

- [corner](#)
- [dill](#)
- [lalsuite](#)
- [matplotlib](#)

- `numpy`
- `pathos`
- `ptemcee`
- `scipy`
- `solar-system-ephemerides` (from `lalsuite`>7.15 on)
- `tqdm`
- `versioneer`

For a general introduction to installing modules, see [here](#).

NOTE: currently we have pinned to `numpy`<1.24.0 (due to an incompatibility with `ptemcee`).

NOTE: We require a recent LALSuite (>=7.13) / `lalpulsar` (>=6.0). If you need to work with older versions, the last PyFstat release supporting those was 1.19.1.

NOTE: Through `pip`, we install the dependency `lalsuite[lalpulsar]`. The optional flag `[lalpulsar]` is ignored for `lalsuite` <= 7.15, as those releases carry their own ephemerides. For `lalsuite` > 7.15, `[lalpulsar]` installs `solar-system-ephemerides` and uses this auxiliary package to access the ephemerides.

1.1.7 Optional dependencies

PyFstat manages optional dependencies through `setuptools`'s `extras_require`.

Available sets of optional dependencies are:

- **chainconsumer** ([Samreay/Chainconsumer](#)): Required to run some optional plotting methods and some of the [example scripts](#).
- **dev**: Collects docs, style, test and wheel.
- **docs**: Required dependencies to build the documentation.
- **pycuda** ([PyPI](#)): Required for the `tCWFstatMapView=pycuda` option of the `TransientGridSearch` class. (Note: Installing `pycuda` requires a working `nvcc` compiler in your path.)
- **style**: Includes the `flake8` linter (`[flake8.pycqa](https://flake8.pycqa.org/en/latest)`), `black` style checker (`[black.readthedocs]`), and `isort` for import ordering (`[pycqa.github.io]`). These checks are required to pass by the online integration pipeline.
- **test**: For running the test suite locally using `[pytest](https://docs.pytest.org)` and some of its addons (`python -m pytest tests/`).
- **wheel**: Includes `wheel` and `check-wheel-contents`.

Installation can be done by adding one or more of the aforementioned tags to the installation command.

For example, installing PyFstat including `chainconsumer`, `pycuda` and `style` dependencies would look like (mind the lack of whitespaces!)

```
pip install pyfstat[chainconsumer,pycuda,style]
```

This command accepts the “development mode” tag `-e`.

Note that LALSuite is a default requirement, not an optional one, but its installation from PyPI can be disabled by setting the `NO_LALSUITE_FROM_PYPI` environment variable, e.g. for a development install from a local git clone:

```
NO_LALSUITE_FROM_PYPI=1 pip install -e .
```

This can be useful to avoid duplication when in a conda environment or installing LALSuite from source.

1.1.8 Using LALSuite built from source

Instructions to use a custom local LALSuite installation can be found in [here on the wiki](#).

1.2 Contributing to PyFstat

This project is open to development, please feel free to contact us for advice or just jump in and submit an [issue](#) or [pull request](#).

Here's what you need to know:

- As a developer, you should install directly from a git clone, with either `pip install -e .[dev]` into some environment or creating a development-enabled conda environment directly from the `pyfstat-dev.yml` file as explained on [this wiki page](#). Please also run, just once after installing:

```
pre-commit install
```

This sets up everything for automated code quality tests (see below) to be checked for you at every commit.

- The github automated tests currently run on python [3.8,3.9,3.10,3.11] and new PRs need to pass all these.
- You can also run the full test suite locally via `pytest tests/`, or run individual tests as explained [on this page](#).
- The automated test on github also runs the [black](#) style checker, the [flake8](#) linter, and the [isort](#) import ordering helper.
- If you have installed the dev dependencies correctly via pip or conda, and ran `pre-commit install` once, then you're ready to let the `pre-commit` tool do all of this automatically for you every time you do `git commit`. For anything that would fail on the github integration tests, it will then either automatically reformat your code to match our style or print warnings for things to fix. The first time it will take a while for setup, later it should be faster.
- If for some reason you can't use `pre-commit`, you can still manually run these tools before pushing changes / submitting PRs: `isort .` to sort package imports, `flake8 --count --statistics .` to find common coding errors and then fix them manually, `black --check --diff .` to show the required style changes, or `black .` to automatically apply them.

1.3 Contributors

Maintainers:

- Greg Ashton
- David Keitel

Active contributors:

- Reinhard Prix
- Rodrigo Tenorio

Other contributors:

- Karl Wette

- Sylvia Zhu
- Dan Foreman-Mackey (pyfstat.gridcorner is based on DFM's [corner.py](#))

1.4 Citing this work

If you use PyFstat in a publication we would appreciate if you cite both a release DOI for the software itself (see below) and one or more of the following scientific papers:

- The recent JOSS (Journal of Open Source Software) paper summarising the package: Keitel, Tenorio, Ashton & Prix 2021 ([inspire:1842895](#) / [ADS:2021arXiv210110915K](#)).
- The original paper introducing the package and the MCMC functionality: Ashton&Prix 2018 ([inspire:1655200](#) / [ADS:2018PhRvD..97j3020A](#)).
- The methods paper introducing a Bayes factor to evaluate the multi-stage follow-up: Tenorio, Keitel, Sintes 2021 ([inspire:1865975](#) / [ADS:2021PhRvD.104h4012T](#))
- For transient searches: Keitel&Ashton 2018 ([inspire:1673205](#) / [ADS:2018CQGra..35t5003K](#)).
- For glitch-robust searches: Ashton, Prix & Jones 2018 ([inspire:1672396](#) / [ADS:2018PhRvD..98f3011A](#))

If you'd additionally like to cite the PyFstat package in general, please refer to the [version-independent Zenodo listing](#) or use directly the following BibTeX entry:

```
@misc{pyfstat,
  author      = {Ashton, Gregory and
                 Keitel, David and
                 Prix, Reinhard
                 and Tenorio, Rodrigo},
  title       = {{PyFstat}},
  month       = jul,
  year        = 2020,
  publisher   = {Zenodo},
  doi         = {10.5281/zenodo.3967045},
  url         = {https://doi.org/10.5281/zenodo.3967045},
  note        = {\url{https://doi.org/10.5281/zenodo.3967045}}
}
```

You can also obtain DOIs for individual versioned releases (from 1.5.x upward) from the right sidebar at [Zenodo](#).

Alternatively, if you've used PyFstat up to version 1.4.x in your works, the DOIs for those versions can be found from the sidebar at [this older Zenodo record](#) and please amend the BibTeX entry accordingly.

PyFstat uses the ``ptemcee`` sampler <https://github.com/willvouden/ptemcee>, which can be cited as Vousden, Far & Mandel 2015 ([ADS:2016MNRAS.455.1919V](#)) and Foreman-Mackey, Hogg, Lang, and Goodman 2012 ([2013PASP..125..306F](#)).

PyFstat also makes generous use of functionality from the LALSuite library and it will usually be appropriate to also cite that project (see [this recommended bibtex entry](#)) and also Wette 2020 ([inspire:1837108](#) / [ADS:2020SoftX..1200634W](#)) for the C-to-python SWIG bindings.

PYFSTAT MODULE DOCUMENTATION

These pages document the full API and set of classes provided by PyFstat.

See [here](#) for installation instructions and other general information.

2.1 pyfstat package

These pages document the full API and set of classes provided by PyFstat.

See [here](#) for installation instructions and other general information.

2.1.1 Subpackages

2.1.1.1 pyfstat.utils package

A collection of helpful functions to facilitate ease-of-use of PyFstat.

Most of these are used internally by other parts of the package and are of interest mostly only for developers, but others can also be helpful for end users.

Functions in these modules can be directly accessed via `pyfstat.utils` without explicitly mentioning the specific module in where they reside. (E.g. just call `pyfstat.utils.some_function`, not `pyfstat.utils.some_topic.some_function`.)

Submodules

`pyfstat.utils.atoms` module

`pyfstat.utils.atoms.extract_singleIF0multiFatoms_from_multiAtoms(multiAtoms, X)`

Extract a length-1 MultiFstatAtomVector from a larger MultiFstatAtomVector.

The result is needed as input to `lalpulsar.ComputeTransientFstatMap` in some places.

The new object is freshly allocated, and we do a deep copy of the actual per-timestamp atoms.

Parameters

- **multiAtoms** (MultiFstatAtomVector) – Fully allocated multi-detector struct of *length* > *X*.
- **X** (int) – The detector index for which to extract atoms.

Returns

singleIFOMultiFatoms – Length-1 MultiFstatAtomVector with only the data for detector X.

Return type

lalpulsar.MultiFstatAtomVector

`pyfstat.utils.atoms.copy_FstatAtomVector(dest, src)`

Deep-copy an FstatAtomVector with all its per-SFT FstatAtoms.

The two vectors must have the same length, and the destination vector must already be allocated.

Parameters

- **dest** (FstatAtomVector) – The destination vector to copy to. Must already be allocated. Will be modified in-place.
- **src** (FstatAtomVector) – The source vector to copy from.

pyfstat.utils.cli module

`pyfstat.utils.cli.run_commandline(cl, raise_error=True, return_output=False)`

Run a string command as a subprocess.

Parameters

- **cl** (str) – Command to run
- **raise_error** (bool) – If True, raise an error if the subprocess fails. If False, just log the error, continue, and return None. True
- **return_output** (bool) – If True, return the `subprocess.CompletedProcess` object. If False, return None. False

Returns

out – The `subprocess.CompletedProcess` of the subprocess if `return_output=True`. None if `return_output=False` or on failed execution if `raise_error=False`.

Return type

`subprocess.CompletedProcess` or None

`pyfstat.utils.cli.match_commandlines(cl1, cl2, be_strict_about_full_executable_path=False)`

Check if two commandline strings match element-by-element, regardless of order.

Parameters

- **cl1** (str) – Commandline strings of *executable -key1=val1 -key2=val2* etc format.
- **cl2** (str) – Commandline strings of *executable -key1=val1 -key2=val2* etc format.
- **be_strict_about_full_executable_path** (bool, optional) – If False (default), only checks the basename of the executable. If True, requires its full path to match. False

Returns

match – Whether the executable and all *key=val* pairs of the two strings matched.

Return type

bool

pyfstat.utils.converting module

`pyfstat.utils.converting.get_dictionary_from_lines(lines, comments, raise_error)`

Return a dictionary of key=val pairs for each line in a list.

Parameters

- **lines** (*list of strings*) – The list of lines to parse.
- **comments** (*str or list of strings*) – Characters denoting that a row is a comment.
- **raise_error** (*bool*) – If True, raise an error for lines which are not comments, but cannot be read. Note that CFSv2 “loudest” files contain complex numbers which fill raise an error unless this is set to False.

Returns

d – The *key=val* pairs as a dictionary.

Return type

dict

`pyfstat.utils.converting.parse_list_of_numbers(val)`

Convert a number, list of numbers or comma-separated str into a list of numbers.

This is useful e.g. for *sqrSX* inputs where the user can be expected to try either type of input.

Parameters

val (*float, list or str*) – The input to be parsed.

Returns

out – The parsed list.

Return type

list

`pyfstat.utils.converting.gps_to_datestr_utc(gps)`

Convert an integer count of GPS seconds to a UTC date-time string.

This uses the locale’s default string formatting as per *datetime.strftime()*. It is intended just for informing the user and may not be as reliable in all situations as *lal[apps]_tconvert*. If you want to do any postprocessing of the date-time string, for safety you should probably call that commandline tool.

Parameters

gps (*int*) – Integer seconds since GPS seconds.

Returns

dtstr – A string representation of date-time in UTC and locale format.

Return type

str

`pyfstat.utils.converting.convert_h0_cosi_to_aPlus_aCross(h0, cosi)`

Converts amplitude parameters from a pair of (*h0,cosi*) to a pair of (*aPlus,aCross*).

See e.g. Eq. (30) of <https://dcc.ligo.org/T0900149-v6/public> .

If both inputs are single numbers, both outputs will be as well. If at least one input is a list or *np.array*, both outputs will be *np.arrays*.

Parameters

- **h0** (*float, list or np.array*) – Nominal GW amplitude.
- **cosi** (*float, list or np.array*) – Cosine of the source inclination w.r.t. line of sight.

Returns

- **aPlus** (*float or np.array*) – Plus polarization amplitude.
- **aCross** (*float or np.array*) – Cross polarization amplitude.

`pyfstat.utils.converting.convert_aPlus_aCross_to_h0_cosi(aPlus, aCross)`

Converts amplitude parameters from a pair of $(aPlus, aCross)$ to a pair of $(h0, cosi)$.

Inverse to `convert_h0_cosi_to_aPlus_aCross()`.

Conversion in this direction is only well-defined if $aPlus \geq \text{abs}(aCross) \geq 0$, as expected for GWs from neutron stars at twice the spin frequency, but not necessarily in all other CW emission scenarios. See e.g. Eq. (32) of <https://dcc.ligo.org/T0900149-v6/public>.

If both inputs are single numbers, both outputs will be as well. If at least one input is a list or `np.array`, both outputs will be `np.arrays`.

Parameters

- **aPlus** (*float, list or np.array*) – Plus polarization amplitude (must be $\geq \text{abs}(aCross)$ and ≥ 0).
- **aCross** (*float, list or np.array*) – Cross polarization amplitude.

Returns

- **h0** (*float or np.array*) – Nominal GW amplitude.
- **cosi** (*float or np.array*) – Cosine of the source inclination w.r.t. line of sight.

pyfstat.utils.ephemeris module

`pyfstat.utils.ephemeris.get_ephemeris_files()`

Set the ephemeris files to use for the Earth and Sun.

This looks first for a configuration file `~/pyfstat.conf` giving individual earth/sun file paths like this:

```
` earth_ephem = '/my/path/earth00-40-DE405.dat.gz' sun_ephem = '/my/path/
sun00-40-DE405.dat.gz' `
```

If such a file is not found or does not conform to that format, then we rely on `lal`'s recently improved ability to find proper default fallback paths for the `[earth/sun]00-40-DE405` ephemerides with both `pip`- and `conda`-installed packages,

Alternatively, ephemeris options can be set manually on each class instantiation.

NOTE that the `$LALPULSAR_DATADIR` environment variable is no longer supported!

Returns

earth_ephem, sun_ephem – Paths of the two files containing positions of Earth and Sun.

Return type

`str`

pyfstat.utils.formatting module`pyfstat.utils.formatting.round_to_n(x, n)`

Simple rounding function for getting a fixed number of digits.

Parameters

- **x** (*float*) – The number to round.
- **n** (*int*) – The number of digits to round to (before plus after the decimal separator).

Returns

rounded – The rounded number.

Return type

float

`pyfstat.utils.formatting.texify_float(x, d=2)`

Format float numbers nicely for LaTeX output, including rounding.

Numbers with absolute values between 0.01 and 100 will be returned in plain float format, while smaller or larger numbers will be returned in powers-of-ten notation.

Parameters

- **x** (*float*) – The number to round and format.
- **n** (*int*) – The number of digits to round to (before plus after the decimal separator).

Returns

formatted – The formatted string.

Return type

str

`pyfstat.utils.formatting.get_doppler_params_output_format(keys)`

Set a canonical output precision for frequency evolution parameters.

This uses the same format (*%16g*) as the *write_FstatCandidate_to_fp()* function of the *ComputeFstatistic_v2* executable.

This assigns that format to each parameter name in *keys* which matches a hardcoded list of known standard ‘Doppler’ parameters, and ignores any others.

Parameters

keys (*dict*) – The parameter keys for which to select formats.

Returns

fmt – A dictionary assigning the default format to each parameter key from the hardcoded list of standard ‘Doppler’ parameters.

Return type

dict

pyfstat.utils.gsl module

`pyfstat.utils.gsl.convert_array_to_gsl_matrix(array)`

Convert a numpy array to a LAL-wrapped GSL matrix.

Parameters

array (*np.ndarray*) – The array to convert. *array.shape* must have 2 dimensions.

Returns

gsl_matrix – The LAL-wrapped GSL matrix object.

Return type

`lal.gsl_matrix`

pyfstat.utils.importing module

`pyfstat.utils.importing.initializer(func)`

Decorator to automatically assign the parameters of a class instantiation to self.

`pyfstat.utils.importing.safe_X_less_plt()`

pyfstat.utils.io module

`pyfstat.utils.io.read_par(filename=None, label=None, outdir=None, suffix='par', comments=['%', '#'], raise_error=False)`

Read in a .par or .loudest file, returns a dictionary of the key=val pairs.

Notes

This can also be used to read in .loudest files produced by the *ComputeFstatistic_v2* executable, or any file which has rows of *key=val* data (in which the val can be understood using *eval(val)*).

Parameters

- **filename** (*str*, *optional*) – Filename (path) containing rows of *key=val* data to read in. *None*
- **label** (*str*, *optional*) – If filename is *None*, form the file to read as *outdir/label.suffix*. *None*
- **outdir** (*str*, *optional*) – If filename is *None*, form the file to read as *outdir/label.suffix*. *None*
- **suffix** (*str*, *optional*) – If filename is *None*, form the file to read as *outdir/label.suffix*. 'par'
- **comments** (*str* or *list of strings*, *optional*) – Characters denoting that a row is a comment. ['%', '#']
- **raise_error** (*bool*, *optional*) – If True, raise an error for lines which are not comments, but cannot be read. False

Returns

d – The *key=val* pairs as a dictionary.

Return type

`dict`

`pyfstat.utils.io.read_txt_file_with_header(f, names=True, comments='#')`

Wrapper to `np.genfromtxt` with smarter handling of variable-length commented headers.

The header is identified as an uninterrupted block of lines from the beginning of the file, each starting with the given *comments* character.

After identifying a header of length *Nhead*, this function then tells `np.genfromtxt()` to skip *Nhead-1* lines (to allow for reading field names from the last commented line before the actual data starts).

Parameters

- **f** (*str*) – Name of the file to read.
- **names** (*bool*, *optional*) – Passed on to `np.genfromtxt()`: If True, the field names are read from the last header line. True
- **comments** (*str*, *optional*) – The character used to indicate the start of a comment. Also passed on to `np.genfromtxt()`. '#'

Returns

data – The data array read from the file after skipping the header.

Return type

`np.ndarray`

`pyfstat.utils.io.read_parameters_dict_lines_from_file_header(outfile, comments='#', strip_spaces=True)`

Load a list of pretty-printed parameters dictionary lines from a commented file header.

Returns a list of lines from a commented file header that match the pretty-printed parameters dictionary format as generated by `BaseSearchClass.get_output_file_header()`. The opening/closing bracket lines (`{,}``) are not included. Newline characters at the end of each line are stripped.

Parameters

- **outfile** (*str*) – Name of a PyFstat-produced output file.
- **comments** (*str*, *optional*) – Comment character used to start header lines. '#'
- **strip_spaces** (*bool*, *optional*) – Whether to strip leading/trailing spaces. True

Returns

dict_lines – A list of unparsed pprinted dictionary entries.

Return type

`list`

`pyfstat.utils.io.get_parameters_dict_from_file_header(outfile, comments='#', eval_values=False)`

Load a parameters dict from a commented file header.

Returns a parameters dictionary, as generated by `BaseSearchClass.get_output_file_header()`, from an output file header. Always returns a proper python dictionary, but the values will be unparsed strings if not requested otherwise.

Parameters

- **outfile** (*str*) – Name of a PyFstat-produced output file.
- **comments** (*str*, *optional*) – Comment character used to start header lines. '#'
- **eval_values** (*bool*, *optional*) – If False, return dictionary values as unparsed strings. If True, evaluate each of them. DANGER! Only do this if you trust the source of the file! False

Returns

params_dict – A dictionary of parameters (with values either as unparsed strings, or evaluated).

Return type

dictionary

pyfstat.utils.predict module

```
pyfstat.utils.predict.predict_fstat(h0=None, cosi=None, psi=None, Alpha=None, Delta=None,
                                       F0=None, sftfilepattern=None, timestampsFiles=None,
                                       minStartTime=None, duration=None, IFOs=None,
                                       assumeSqrtSX=None, temporary_filename='fs.tmp',
                                       earth_ephem=None, sun_ephem=None,
                                       transientWindowType='none', transientStartTime=None,
                                       transientTau=None)
```

Wrapper to PredictFstat executable for predicting expected F-stat values.

Parameters

- **h0** (*float, optional*) – Signal parameters, see *lalpulsar_PredictFstat --help* for more info. None
- **cosi** (*float, optional*) – Signal parameters, see *lalpulsar_PredictFstat --help* for more info. None
- **psi** (*float, optional*) – Signal parameters, see *lalpulsar_PredictFstat --help* for more info. None
- **Alpha** (*float, optional*) – Signal parameters, see *lalpulsar_PredictFstat --help* for more info. None
- **Delta** (*float, optional*) – Signal parameters, see *lalpulsar_PredictFstat --help* for more info. None
- **F0** (*float or None, optional*) – Signal frequency. Only needed for noise floor estimation when given *sftfilepattern* but *assumeSqrtSX=None*. The actual F-stat prediction is frequency-independent. None
- **sftfilepattern** (*str or None, optional*) – Pattern matching the SFT files to use for inferring detectors, timestamps and/or estimating the noise floor. None
- **timestampsFiles** (*str or None, optional*) – Comma-separated list of per-detector files containing timestamps to use. Only used if *sftfilepattern=None*. None
- **minStartTime** (*int or None, optional*) – If *sftfilepattern* given: used as optional constraints. If *timestampsFiles* given: ignored. If neither given: used as the interval for prediction. None
- **duration** (*int or None, optional*) – If *sftfilepattern* given: used as optional constraints. If *timestampsFiles* given: ignored. If neither given: used as the interval for prediction. None
- **IFOs** (*str or None, optional*) – Comma-separated list of detectors. Required if *sftfilepattern=None*, ignored otherwise. None
- **assumeSqrtSX** (*float or str, optional*) – Assume stationary per-detector noise-floor instead of estimating from SFTs. Single float or str value: use same for all IFOs. Comma-separated string: must match *len(IFOs)* and/or the data in *sftfilepattern*. Detectors will be paired to list elements following alphabetical order. Required if *sftfilepattern=None*, optional otherwise.. None

- **temporary_filename** (*str, optional*) – Temporary file used for *PredictFstat* output, will be deleted at the end. 'fs.tmp'
- **earth_ephem** (*str or None, optional*) – Ephemerides files, defaults will be used if *None*. *None*
- **sun_ephem** (*str or None, optional*) – Ephemerides files, defaults will be used if *None*. *None*
- **transientWindowType** (*str, optional*) – Optional parameter for transient signals, see *lalpulsar_PredictFstat -help*. Default of *none* means a classical Continuous Wave signal. 'none'
- **transientStartTime** (*int or None, optional*) – Optional parameters for transient signals, see *lalpulsar_PredictFstat -help*. *None*
- **transientTau** (*int or None, optional*) – Optional parameters for transient signals, see *lalpulsar_PredictFstat -help*. *None*

Returns

twoF_expected, twoF_sigma – The expectation and standard deviation of $2F$.

Return type

float

`pyfstat.utils.predict.get_predict_fstat_parameters_from_dict(signal_parameters, transientWindowType=None)`

Extract a subset of parameters as needed for predicting F-stats. Given a dictionary with arbitrary signal parameters, this extracts only those ones required by *helper_functions.predict_fstat()*: Freq, Alpha, Delta, h0, cosi, psi. Also preserves transient parameters, if included in the input dict.

Parameters

- **signal_parameters** (*dict*) – Dictionary containing at least those signal parameters required by *helper_functions.predict_fstat*. This dictionary's keys must follow the PyFstat convention (e.g. F0 instead of Freq).
- **transientWindowType** (*str, optional*) – Transient window type to store in the output dict. Currently required because the typical input dicts produced by various PyFstat functions tend not to store this property. If there is a key with this name already, its value will be overwritten. *None*

Returns

predict_fstat_params – The dictionary of selected parameters.

Return type

dict

pyfstat.utils.runlalsuite module

`pyfstat.utils.runlalsuite.get_covering_band(tref, tstart, tend, F0, F1, F2, F0band=0.0, F1band=0.0, F2band=0.0, maxOrbitAsini=0.0, minOrbitPeriod=0.0, maxOrbitEcc=0.0)`

Get the covering band for CW signals for given time and parameter ranges.

This uses the *lalpulsar* function *XLALCWSignalCoveringBand()*, accounting for the spin evolution of the signals within the given [F0,F1,F2] ranges, the maximum possible Doppler modulation due to detector motion (i.e. for the worst-case sky locations), and for worst-case binary orbital motion.

Parameters

- **tref** (*int*) – Reference time (in GPS seconds) for the signal parameters.
- **tstart** (*int*) – Start time (in GPS seconds) for the signal evolution to consider.
- **tend** (*int*) – End time (in GPS seconds) for the signal evolution to consider.
- **F0** (*float*) – Minimum frequency and spin-down of signals to be covered.
- **F1** (*float*) – Minimum frequency and spin-down of signals to be covered.
- **F1** – Minimum frequency and spin-down of signals to be covered.
- **F0band** (*float*, *optional*) – Ranges of frequency and spin-down of signals to be covered. 0.0
- **F1band** (*float*) – Ranges of frequency and spin-down of signals to be covered. 0.0
- **F1band** – Ranges of frequency and spin-down of signals to be covered.
- **maxOrbitAsini** (*float*, *optional*) – Largest orbital projected semi-major axis to be covered. 0.0
- **minOrbitPeriod** (*float*, *optional*) – Shortest orbital period to be covered. 0.0
- **maxOrbitEcc** (*float*, *optional*) – Highest orbital eccentricity to be covered. 0.0

Returns

minCoverFreq, **maxCoverFreq** – Estimates of the minimum and maximum frequencies of the signals from the given parameter ranges over the *[tstart,tend]* duration.

Return type

float

```
pyfstat.utils.runlalsuite.generate_loudest_file(max_params, tref, outdir, label, sftfilepattern,
                                              minStartTime=None, maxStartTime=None,
                                              transientWindowType=None, earth_ephem=None,
                                              sun_ephem=None)
```

Use ComputeFStatistic_v2 executable to produce a .loudest file.

Parameters

- **max_params** (*dict*) – Dictionary of a single parameter-space point. This needs to already have been translated to lal conventions and must NOT include detection statistic entries!
- **tref** (*int*) – Reference time of the max_params.
- **outdir** (*str*) – Directory to place the .loudest file in.
- **label** (*str*) – Search name bit to be used in the output filename.
- **sftfilepattern** (*str*) – Pattern to match SFTs using wildcards (*?) and ranges [0-9]; mutiple patterns can be given separated by colons.
- **minStartTime** (*int or None, optional*) – GPS seconds of the start time and end time; default: use al available data. None
- **maxStartTime** (*int or None, optional*) – GPS seconds of the start time and end time; default: use al available data. None
- **transientWindowType** (*str or None, optional*) – optional: transient window type, needs to go with t0 and tau parameters inside max_params. None
- **earth_ephem** (*str or None, optional*) – optional: user-set Earth ephemeris file None
- **sun_ephem** (*str or None, optional*) – optional: user-set Sun ephemeris file None

Returns

loudest_file – The filename of the CFSv2 output file.

Return type

str

pyfstat.utils.sft module

`pyfstat.utils.sft.get_sft_as_arrays(sftfilepattern, fMin=None, fMax=None, constraints=None)`

Read binary SFT files into NumPy arrays.

Parameters

- **sftfilepattern** (str) – Pattern to match SFTs using wildcards (*) and ranges [0-9]; multiple patterns can be given separated by colons.
- **fMin** (Optional[float]) – Restrict frequency range to $[fMin, fMax]$. If None, retrieve the full frequency range. None
- **fMax** (Optional[float]) – Restrict frequency range to $[fMin, fMax]$. If None, retrieve the full frequency range. None
- **constraints** (Optional[SFTConstraints]) – Constrains to be fed into XLALSFT-dataFind to specify detector, GPS time range or timestamps to be retrieved. None

Return type

Tuple[ndarray, Dict, Dict]

Returns

- **freqs** (*np.ndarray*) – The frequency bins in each SFT. These will be the same for each SFT, so only a single 1D array is returned.
- **times** (*Dict*) – The SFT start times as a dictionary of 1D arrays, one for each detector. Keys correspond to the official detector names as returned by `lalpulsar.ListIFOsInCatalog`.
- **data** (*Dict*) – A dictionary of 2D arrays of the complex Fourier amplitudes of the SFT data for each detector in each frequency bin at each timestamp. Keys correspond to the official detector names as returned by `lalpulsar.ListIFOsInCatalog`.

`pyfstat.utils.sft.get_commandline_from_SFTDescriptor(descriptor)`

Extract a commandline from the ‘comment’ entry of a SFT descriptor.

Most LALSuite SFT creation tools save their commandline into that entry, so we can extract it and reuse it to reproduce that data.

Since lalapps 9.0.0 / lalpulsar 5.0.0 the relevant executables have been moved to lalpulsar, but we allow for lalapps backwards compatibility here,

Parameters

descriptor (*SFTDescriptor*) – Element of a *lalpulsar.SFTCatalog* structure.

Returns

cmd – A lalapps/lalpulsar commandline string, or an empty string if no match in comment.

Return type

str

`pyfstat.utils.sft.get_official_sft_filename(IFO, numSFTs, Tsft, tstart, duration, label=None, window_type=None, window_param=None)`

Wrapper to predict the canonical lalpulsar names for SFT files.

Parameters

- **IFO** (*str*) – Two-char detector name, e.g. *H1*.
- **numSFTs** (*int*) – numSFTs number of SFTs in SFT-file
- **Tsft** (*int*) – time-baseline in (integer) seconds
- **tstart** (*int*) – GPS seconds of first SFT start time
- **duration** (*int*) – total time-spanned by all SFTs in seconds
- **label** (*str or None, optional*) – optional ‘Misc’ entry in the SFT ‘D’ field *None*
- **window_type** (*str or None, optional*) – window function applied to SFTs *None*
- **window_param** (*float or None, optional*) – additional parameter for some window functions *None*

Returns

filename – The canonical SFT file name for the input parameters.

Return type

str

Module contents

A collection of helpful functions to facilitate ease-of-use of PyFstat.

Most of these are used internally by other parts of the package and are of interest mostly only for developers, but others can also be helpful for end users.

2.1.2 Submodules

2.1.3 pyfstat.core module

The core tools used in pyfstat

class `pyfstat.core.BaseSearchClass(*args, **kwargs)`

Bases: `object`

The base class providing parent methods to other PyFstat classes.

This does not actually have any ‘search’ functionality, which needs to be added by child classes along with full initialization and any other custom methods.

set_ephemeris_files(*earth_ephem=None, sun_ephem=None*)

Set the ephemeris files to use for the Earth and Sun.

NOTE: If not given explicit arguments, default values from `utils.get_ephemeris_files()` are used.

Parameters

- **earth_ephem** (*str, optional*) – Paths of the two files containing positions of Earth and Sun, respectively at evenly spaced times, as passed to `CreateFstatInput` *None*
- **sun_ephem** (*str, optional*) – Paths of the two files containing positions of Earth and Sun, respectively at evenly spaced times, as passed to `CreateFstatInput` *None*

pprint_init_params_dict()

Pretty-print a parameters dictionary for output file headers.

Returns

pretty_init_parameters – A list of lines to be printed, including opening/closing “{” and “}”, consistent indentation, as well as end-of-line commas, but no comment markers at start of lines.

Return type

list

get_output_file_header()

Constructs a meta-information header for text output files.

This will include PyFstat and LALSuite versioning, information about when/where/how the code was run, and input parameters of the instantiated class.

Returns

header – A list of formatted header lines.

Return type

list

read_par(filename=None, label=None, outdir=None, suffix='par', raise_error=True)

Read a *key=val* file and return a dictionary.

Parameters

- **filename** (*str or None, optional*) – Filename (path) containing rows of *key=val* data to read in. *None*
- **label** (*str or None, optional*) – If filename is *None*, form the file to read as *outdir/label.suffix*. *None*
- **outdir** (*str or None, optional*) – If filename is *None*, form the file to read as *outdir/label.suffix*. *None*
- **suffix** (*str or None, optional*) – If filename is *None*, form the file to read as *outdir/label.suffix*. 'par'
- **raise_error** (*bool, optional*) – If *True*, raise an error for lines which are not comments, but cannot be read. *True*

Returns

params_dict – A dictionary of the parsed *key=val* pairs.

Return type

dict

static translate_keys_to_lal(dictionary)

Convert input keys into lalpulsar convention.

In PyFstat’s convention, input keys (search parameter names) are F0, F1, F2, ..., while lalpulsar functions prefer to use Freq, f1dot, f2dot, ...

Since lalpulsar keys are only used internally to call lalpulsar routines, this function is provided so the keys can be translated on the fly.

Parameters

dictionary (*dict*) – Dictionary to translate. A copy will be made (and returned) before translation takes place.

Returns

translated_dict – Copy of “dictionary” with new keys according to lalpulsar convention.

Return type

dict

class pyfstat.core.**ComputeFstat**(*args, **kwargs)

Bases: [BaseSearchClass](#)

Base search class providing an interface to *lalpulsar.ComputeFstat*.

In most cases, users should be using one of the higher-level search classes from the `grid_based_searches` or `mcmc_based_searches` modules instead.

See the lalpulsar documentation at https://lscsoft.docs.ligo.org/lalsuite/lalpulsar/group___compute_fstat__h.html and R. Prix, The F-statistic and its implementation in ComputeFStatistic_v2 (<https://dcc.ligo.org/T0900149/public>) for details of the lalpulsar module and the meaning of various technical concepts as embodied by some of the class’s parameters.

Normally this will read in existing data through the *sftfilepattern* argument, but if that option is *None* and the necessary alternative arguments are used, it can also generate simulated data (including noise and/or signals) on the fly.

NOTE that the detection statistics that can be computed from an instance of this class depend on the *BSGL*, *BtSG* and *transientWindowType* arguments given at initialisation. See *get_fullycoherent_detstat()* and *get_transient_detstats()* for details. To change what you want to compute, you may need to initialise a new instance with different options.

NOTE for GPU users (*tCWFstatMapVersion*=“*pycuda*”): This class tries to conveniently deal with GPU context management behind the scenes. A known problematic case is if you try to instantiate it twice from the same session/script. If you then get some messages like *RuntimeError: make_default_context()* and *invalid device context*, that is because the GPU is still blocked from the first instance when you try to initiate the second. To avoid this problem, use context management:

```
with pyfstat.ComputeFstat(
    [...],
    tCWFstatMapVersion="pycuda",
) as search:
    search.get_fullycoherent_detstat([...])
```

or manually call the *search.finalizer_()* method where needed.

Parameters

- **tref** (*int*) – GPS seconds of the reference time.
- **sftfilepattern** (*str*, *optional*) – Pattern to match SFTs using wildcards (*) and ranges [0-9]; mutiple patterns can be given separated by colons. *None*
- **minStartTime** (*int*, *optional*) – Only use SFTs with timestamps starting from within this range, following the XLALCWGPSinRange convention: half-open intervals [minStartTime,maxStartTime]. *None*
- **maxStartTime** (*int*, *optional*) – Only use SFTs with timestamps starting from within this range, following the XLALCWGPSinRange convention: half-open intervals [minStartTime,maxStartTime]. *None*
- **Tsft** (*int*, *optional*) – SFT duration in seconds. Only required if *sftfilepattern=None* and hence simulated data is generated on the fly. **1800**
- **binary** (*bool*, *optional*) – If true, search over binary parameters. *False*

- **singleFstats** (*bool*, *optional*) – If true, also compute the single-detector twoF values. *False*
- **BSGL** (*bool*, *optional*) – If true, compute the log10BSGL statistic rather than the twoF value. For details, see Keitel et al (PRD 89, 064023, 2014): <https://arxiv.org/abs/1311.5738> Note this automatically sets *singleFstats=True* as well. Tuning parameters are currently hardcoded: *False*
 - *Fstar0=15* for coherent searches.
 - A p-value of 1e-6 and correspondingly recalculated *Fstar0* for semicoherent searches.
 - Uniform per-detector prior line-vs-Gaussian odds.
- **BtSG** (*bool*, *optional*) – If true and *transientWindowType* is not *None*, compute the transient $\ln \mathcal{B}_{\text{tS/G}}$ statistic from Prix, Giampanis & Messenger (PRD 84, 023007, 2011) (tCWF-statMap marginalised over uniform *t0*, *tau* priors). rather than the maxTwoF value. *False*
- **transientWindowType** (*str*, *optional*) – If *rect* or *exp*, allow for the Fstat to be computed over a transient range. (*none* instead of *None* explicitly calls the transient-window function, but with the full range, for debugging.) (If not *None*, will also force atoms regardless of computeAtoms option.) *None*
- **t0Band** (*int*, *optional*) – Search ranges for transient start-time *t0* and duration *tau*. If >0, search *t0* in (*minStartTime*, *minStartTime*+*t0Band*) and *tau* in (*tauMin*, 2**Tsft*+*tauBand*). If =0, only compute the continuous-wave Fstat with *t0*=*minStartTime*, *tau*=*maxStartTime*-*minStartTime*. *None*
- **tauBand** (*int*, *optional*) – Search ranges for transient start-time *t0* and duration *tau*. If >0, search *t0* in (*minStartTime*, *minStartTime*+*t0Band*) and *tau* in (*tauMin*, 2**Tsft*+*tauBand*). If =0, only compute the continuous-wave Fstat with *t0*=*minStartTime*, *tau*=*maxStartTime*-*minStartTime*. *None*
- **tauMin** (*int*, *optional*) – Minimum transient duration to cover, defaults to 2**Tsft*. *None*
- **dt0** (*int*, *optional*) – Grid resolution in transient start-time, defaults to *Tsft*. *None*
- **dtau** (*int*, *optional*) – Grid resolution in transient duration, defaults to *Tsft*. *None*
- **detectors** (*str*, *optional*) – Two-character references to the detectors for which to use data. Specify *None* for no constraint. For multiple detectors, separate by commas. *None*
- **minCoverFreq** (*float*, *optional*) – The min and max cover frequency passed to *lalpulsar.CreateFstatInput*. For negative values, these will be used as offsets from the min/max frequency contained in the *sftfilepattern*. If either is *None*, the *search_ranges* argument is used to estimate them. If the automatic estimation fails and you do not have a good idea what to set these two options to, setting both to -0.5 will reproduce the default behaviour of PyFstat <=1.4 and may be a reasonably safe fallback in many cases. *None*
- **maxCoverFreq** (*float*, *optional*) – The min and max cover frequency passed to *lalpulsar.CreateFstatInput*. For negative values, these will be used as offsets from the min/max frequency contained in the *sftfilepattern*. If either is *None*, the *search_ranges* argument is used to estimate them. If the automatic estimation fails and you do not have a good idea what to set these two options to, setting both to -0.5 will reproduce the default behaviour of PyFstat <=1.4 and may be a reasonably safe fallback in many cases. *None*
- **search_ranges** (*dict*, *optional*) – Dictionary of ranges in all search parameters, only used to estimate frequency band passed to *lalpulsar.CreateFstatInput*, if *minCoverFreq*, *maxCoverFreq* are not specified (==`None`). For actually running searches, grids/points will have to be passed separately to the *.run()* method. The entry for each parameter must be a list of length 1, 2 or 3: [*single_value*], [*min*,*max*] or [*min*,*max*,*step*]. *None*

- **injectSources**(*dict or str, optional*) – Either a dictionary of the signal parameters to inject, or a string pointing to a .cff file defining a signal. *None*
- **injectSqrtSX**(*float or list or str, optional*) – Single-sided PSD values for generating fake Gaussian noise on the fly. Single float or str value: use same for all IFOs. List or comma-separated string: must match len(detectors) and/or the data in sftfilepattern. Detectors will be paired to list elements following alphabetical order. *None*
- **randSeed**(*int or None, optional*) – random seed for on-the-fly noise generation using *injectSqrtSX*. Setting this to 0 or *None* is equivalent; both will randomise the seed, following the behaviour of *XLALAddGaussianNoise()*, while any number not equal to 0 will produce a reproducible noise realisation. *None*
- **assumeSqrtSX**(*float or list or str, optional*) – Don't estimate noise-floors but assume this (stationary) single-sided PSD. Single float or str value: use same for all IFOs. List or comma-separated string: must match len(detectors) and/or the data in sftfilepattern. Detectors will be paired to list elements following alphabetical order. If working with signal-only data, please set *assumeSqrtSX=1*. *None*
- **SSBprec**(*int, optional*) – Flag to set the Solar System Barycentring (SSB) calculation in *lalpulsar*: 0=Newtonian, 1=relativistic, 2=relativistic optimised, 3=DMoff, 4=NO_SPIN *None*
- **RngMedWindow**(*int, optional*) – Running-Median window size for F-statistic noise normalization (number of SFT bins). *None*
- **tCWFstatMapVersion**(*str, optional*) – Choose between implementations of the transient F-statistic functionality: standard *lal* implementation, *pycuda* for GPU version, and some others only for devel/debug. *'lal'*
- **cudaDeviceName**(*str, optional*) – GPU name to be matched against *drv.Device* output, only for *tCWFstatMapVersion=pycuda*. *None*
- **computeAtoms**(*bool, optional*) – Request calculation of 'F-statistic atoms' regardless of *transientWindowType*. *False*
- **earth_ephem**(*str, optional*) – Earth ephemeris file path. If *None*, will check standard sources as per *utils.get_ephemeris_files()*. *None*
- **sun_ephem**(*str, optional*) – Sun ephemeris file path. If *None*, will check standard sources as per *utils.get_ephemeris_files()*. *None*
- **allowedMismatchFromSFTLength**(*float, optional*) – Maximum allowed mismatch from SFTs being too long [Default: what's hardcoded in *XLALFstatMaximumSFTLength*] *None*

init_computefstatistic()

Initialization step for the F-stastic computation internals.

This sets up the special input and output structures the *lalpulsar* module needs, the ephemerides, optional on-the-fly signal injections, and extra options for multi-detector consistency checks and transient searches.

All inputs are taken from the pre-initialized object, so this function does not have additional arguments of its own.

estimate_min_max_CoverFreq()

Extract spanned spin-range at reference -time from the template bank.

To use this method, *self.search_ranges* must be a dictionary of lists per search parameter which can be either [single_value], [min,max] or [min,max,step].

get_fullycoherent_detstat(*F0, F1, F2, Alpha, Delta, asini=None, period=None, ecc=None, tp=None, argp=None, tstart=None, tend=None*)

Computes the detection statistic(s) fully-coherently at a single point.

Currently supported statistics:

- twoF (CW)
- log10BSGL (CW or transient)
- maxTwoF (transient)
- lnBtSG (transient)

All computed statistics are stored as attributes, but only one statistic is returned.

As the basic statistic of this class, *twoF* is always computed and stored as *self.twoF* as well, and it is the default return value.

If *self.singleFstats*, additionally the single-detector 2F-stat values are stored in *self.twoFX*.

If *self.BSGL*, the *log10BSGL* statistic for CWs is additionally stored, and it is returned instead of *twoF*.

If transient parameters are enabled (*self.transientWindowType* is set), *maxTwoF* will always be computed and stored, and returned by default. Depending on the *self.BSGL* and *self.BtSG* options, either *log10BSGL* (a transient version of it, superseding the CW version) or *lnBtSG* will also be computed, stored, and returned instead of *maxTwoF*. The full transient-F-stat map is also computed here, but stored in *self.FstatMap*, not returned.

Parameters

- **F0** (*float*) – Parameters at which to compute the statistic.
- **F1** (*float*) – Parameters at which to compute the statistic.
- **F2** (*float*) – Parameters at which to compute the statistic.
- **Alpha** (*float*) – Parameters at which to compute the statistic.
- **Delta** (*float*) – Parameters at which to compute the statistic.
- **asini** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **period** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **ecc** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **tp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **argp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **tstart** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. This is only passed on to *self.get_transient_detstats()*, i.e. only used if *self.transientWindowType* is set. *None*
- **tend** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. This is only passed on to *self.get_transient_detstats()*, i.e. only used if *self.transientWindowType* is set. *None*

Returns

stat – A single value of the main detection statistic at the input parameter values.

Return type

float

get_fullycoherent_twoF(*F0, F1, F2, Alpha, Delta, asini=None, period=None, ecc=None, tp=None, argp=None*)

Computes the fully-coherent 2F statistic at a single point.

NOTE: This always uses the full data set as defined when initialising the search object. If you want to restrict the range of data used for a single 2F computation, you need to set a *self.transientWindowType* and then call *self.get_fullycoherent_detstat()* with *tstart* and *tend* options instead of this function.

Parameters

- **F0** (*float*) – Parameters at which to compute the statistic.
- **F1** (*float*) – Parameters at which to compute the statistic.
- **F2** (*float*) – Parameters at which to compute the statistic.
- **Alpha** (*float*) – Parameters at which to compute the statistic.
- **Delta** (*float*) – Parameters at which to compute the statistic.
- **asini** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. None
- **period** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. None
- **ecc** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. None
- **tp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. None
- **argp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. None

Returns

twoF – A single value of the fully-coherent 2F statistic at the input parameter values. Also stored as *self.twoF*.

Return type

float

get_fullycoherent_single_IFO_twoFs()

Computes single-detector F-stats at a single point.

This requires *self.get_fullycoherent_twoF()* to be run first.

Returns

twoFX – A list of the single-detector detection statistics twoF. Also stored as *self.twoFX*.

Return type

list

get_fullycoherent_log10BSGL()

Computes the line-robust statistic log10BSGL at a single point.

This requires *self.get_fullycoherent_twoF()* and *self.get_fullycoherent_single_IFO_twoFs()* to be run first.

Returns

log10BSGL – A single value of the detection statistic log10BSGL at the input parameter values. Also stored as *self.log10BSGL*.

Return type

float

get_transient_detstats(*tstart=None, tend=None*)

Computes one or more transient detection statistics at a single point.

This requires *self.get_fullycoherent_twoF()* to be run first.

All computed statistics will be stored as attributes of *self*, but only one (*twoF*, *log10BSGL* or *lnBtSG*) will be the return value.

The full transient-F-stat map will also be computed here, but stored in *self.FstatMap*, not returned.

Parameters

- **tstart** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. *None*
- **tend** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. *None*

Returns

detstat – A single value of the main chosen detection statistic (*maxTwoF*, *log10BSGL* or *lnBtSG*) at the input parameter values.

Return type

float

get_transient_maxTwoFstat(*tstart=None, tend=None*)

Computes the transient *maxTwoF* statistic at a single point.

This requires *self.get_fullycoherent_twoF()* to be run first, and is itself now only a backwards compatibility / convenience wrapper around the more general *get_transient_detstats()*.

The full transient-F-stat map will also be computed here, but stored in *self.FstatMap*, not returned.

Parameters

- **tstart** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. *None*
- **tend** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. *None*

Returns

maxTwoF – A single value of the detection statistic at the input parameter values. Also stored as *self.maxTwoF*.

Return type

float

get_transient_log10BSGL()

Computes a transient detection statistic *log10BSGL* at a single point.

This should normally be called through *get_transient_detstats()*, but if called stand-alone, it requires *self.get_transient_maxTwoFstat()* to be run first.

The single-detector 2F-stat values used for that computation (at the index of *maxTwoF*) are saved in *self.twoFXatMaxTwoF*, not returned.

Returns

log10BSGL – A single value of the detection statistic *log10BSGL* at the input parameter values. Also stored as *self.log10BSGL*.

Return type

float

calculate_twoF_cumulative(*F0, F1, F2, Alpha, Delta, asini=None, period=None, ecc=None, tp=None, argp=None, tstart=None, tend=None, transient_tstart=None, transient_duration=None, num_segments=1000*)

Calculate the cumulative twoF over subsets of the observation span.

This means that we consider sub-“segments” of the [tstart,tend] interval, each starting at the overall tstart and with increasing durations, and compute the 2F for each of these, which for a true CW signal should increase roughly with duration towards the full value.

Parameters

- **F0** (*float*) – Parameters at which to compute the cumulative twoF.
- **F1** (*float*) – Parameters at which to compute the cumulative twoF.
- **F2** (*float*) – Parameters at which to compute the cumulative twoF.
- **Alpha** (*float*) – Parameters at which to compute the cumulative twoF.
- **Delta** (*float*) – Parameters at which to compute the cumulative twoF.
- **asini** (*float, optional*) – Optional: Binary parameters at which to compute the cumulative 2F. None
- **period** (*float, optional*) – Optional: Binary parameters at which to compute the cumulative 2F. None
- **ecc** (*float, optional*) – Optional: Binary parameters at which to compute the cumulative 2F. None
- **tp** (*float, optional*) – Optional: Binary parameters at which to compute the cumulative 2F. None
- **argp** (*float, optional*) – Optional: Binary parameters at which to compute the cumulative 2F. None
- **tstart** (*int or None, optional*) – GPS times to restrict the range of data used. If None: falls back to self.minStartTime and self.maxStartTime;. If outside those: auto-truncated. None
- **tend** (*int or None, optional*) – GPS times to restrict the range of data used. If None: falls back to self.minStartTime and self.maxStartTime;. If outside those: auto-truncated. None
- **num_segments** (*int, optional*) – Number of segments to split [tstart,tend] into. 1000
- **transient_tstart** (*float or None, optional*) – These are not actually used by this function, but just included so a parameters dict can be safely passed. None
- **transient_duration** (*float or None, optional*) – These are not actually used by this function, but just included so a parameters dict can be safely passed. None

Returns

- **cumulative_durations** (*ndarray of shape (num_segments,)*) – Offsets of each segment’s tend from the overall tstart.
- **twoFs** (*ndarray of shape (num_segments,)*) – Values of twoF computed over [[tstart,tstart+duration] for duration in cumulative_durations].

predict_twoF_cumulative(*F0, Alpha, Delta, h0, cosi, psi, tstart=None, tend=None, num_segments=10, **predict_fstat_kwargs*)

Calculate expected 2F, with uncertainty, over subsets of the observation span.

This yields the expected behaviour that `calculate_twoF_cumulative()` can be compared against: 2F for CW signals increases with duration as we take longer and longer subsets of the total observation span.

Parameters

- **F0** (*float*) – Parameters at which to compute the cumulative predicted twoF.
- **Alpha** (*float*) – Parameters at which to compute the cumulative predicted twoF.
- **Delta** (*float*) – Parameters at which to compute the cumulative predicted twoF.
- **h0** (*float*) – Parameters at which to compute the cumulative predicted twoF.
- **cosi** (*float*) – Parameters at which to compute the cumulative predicted twoF.
- **psi** (*float*) – Parameters at which to compute the cumulative predicted twoF.
- **tstart** (*int or None, optional*) – GPS times to restrict the range of data used. If None: falls back to `self.minStartTime` and `self.maxStartTime`. If outside those: auto-truncated. None
- **tend** (*int or None, optional*) – GPS times to restrict the range of data used. If None: falls back to `self.minStartTime` and `self.maxStartTime`. If outside those: auto-truncated. None
- **num_segments** (*int, optional*) – Number of segments to split [tstart,tend] into. 10
- **predict_fstat_kwargs** – Other kwargs to be passed to `utils.predict_fstat()`.

Returns

- **tstart** (*int*) – GPS start time of the observation span.
- **cumulative_durations** (*ndarray of shape (num_segments,)*) – Offsets of each segment's tend from the overall tstart.
- **pfs** (*ndarray of size (num_segments,)*) – Predicted 2F for each segment.
- **pfs_sigma** (*ndarray of size (num_segments,)*) – Standard deviations of predicted 2F.

plot_twoF_cumulative(*CFS_input, PFS_input=None, tstart=None, tend=None, num_segments_CFS=1000, num_segments_PFS=10, custom_ax_kwargs=None, savefig=False, label=None, outdir=None, **PFS_kwargs*)

Plot how 2F accumulates over time.

This compares the accumulation on the actual data set ('CFS', from `self.calculate_twoF_cumulative()`) against (optionally) the average expectation ('PFS', from `self.predict_twoF_cumulative()`).

Parameters

- **CFS_input** (*dict*) – Input arguments for `self.calculate_twoF_cumulative()` (besides [tstart, tend, num_segments]).
- **PFS_input** (*dict, optional*) – Input arguments for `self.predict_twoF_cumulative()` (besides [tstart, tend, num_segments]). If None: do not calculate predicted 2F. None
- **tstart** (*int or None, optional*) – GPS times to restrict the range of data used. If None: falls back to `self.minStartTime` and `self.maxStartTime`. If outside those: auto-truncated. None

- **tend** (*int or None, optional*) – GPS times to restrict the range of data used. If *None*: falls back to *self.minStartTime* and *self.maxStartTime*. If outside those: auto-truncated. *None*
- **num_segments_(CFS|PFS)** (*int*) – Number of time segments to (compute|predict) twoF.
- **custom_ax_kwargs** (*dict, optional*) – Optional axis formatting options. *None*
- **savefig** (*bool, optional*) – If true, save the figure in *outdir*. If false, return an axis object without saving to disk. *False*
- **label** (*str, optional*) – Output filename will be constructed by appending *_twoFcumulative.png* to this label. (Ignored unless *savefig=true*.) *None*
- **outdir** (*str, optional*) – Output folder (ignored unless *savefig=true*). *None*
- **PFS_kwargs** (*dict*) – Other kwargs to be passed to *self.predict_twoF_cumulative()*.

Returns

ax – The axes object containing the plot.

Return type

`matplotlib.axes._subplots_AxesSubplot`, optional

write_atoms_to_file(*fnamebase=""*, *comments='%%'*)

Save F-statistic atoms (time-dependent quantities) for a given parameter-space point.

Parameters

- **fnamebase** (*str, optional*) – Basis for output filename, full name will be *{fnamebase}_Fstatatoms_{dopplerName}.dat* where *dopplerName* is a canonical lalpulsar formatting of the ‘Doppler’ parameter space point (frequency-evolution parameters). *''*
- **comments** (*str, optional*) – Comments marker character(s) to be prepended to header lines. Note that the column headers line (last line of the header before the atoms data) is printed by lalpulsar, with *%%* as comments marker, so (different from most other PyFstat functions) the default here is *%%* too. *'%%'*

class `pyfstat.core.SemiCoherentSearch`(*args, **kwargs)

Bases: `ComputeFstat`

A simple semi-coherent search class.

This will split the data set into multiple segments, run a coherent F-stat search over each, and produce a final semi-coherent detection statistic as the sum over segments.

This does not include any concept of refinement between the two steps, as some grid-based semi-coherent search algorithms do; both the per-segment coherent F-statistics and the incoherent sum are done at the same parameter space point.

The implementation is based on a simple trick using the transient F-stat map functionality: basic F-stat atoms are computed only once over the full data set, then the transient code with rectangular ‘windows’ is used to compute the per-segment F-stats, and these are summed to get the semi-coherent result.

Only parameters with a special meaning for `SemiCoherentSearch` itself are explicitly documented here. For all other parameters inherited from `pyfstat.ComputeFStat` see the documentation of that class.

Parameters

- **label** (*str*) – A label and directory to read/write data from/to.
- **outdir** (*str*) – A label and directory to read/write data from/to.
- **tref** (*int*) – GPS seconds of the reference time.

- **nsecs** (*int, optional*) – The (fixed) number of segments to split the data set into. *None*
- **sftfilepattern** (*str, optional*) – Pattern to match SFTs using wildcards (*) and ranges [0-9]; multiple patterns can be given separated by colons. *None*
- **minStartTime** (*int, optional*) – Only use SFTs with timestamps starting from this range, following the XLALCWGPSinRange convention: half-open intervals [minStartTime,maxStartTime]. Also used to set up segment boundaries, i.e. $\text{maxStartTime} - \text{minStartTime}$ will be divided by *nsecs* to obtain the per-segment coherence time *Tcoh*. *None*
- **maxStartTime** (*int, optional*) – Only use SFTs with timestamps starting from this range, following the XLALCWGPSinRange convention: half-open intervals [minStartTime,maxStartTime]. Also used to set up segment boundaries, i.e. $\text{maxStartTime} - \text{minStartTime}$ will be divided by *nsecs* to obtain the per-segment coherence time *Tcoh*. *None*

init_semicohherent_parameters()

Set up a list of equal-length segments and the corresponding transient windows.

For a requested number of segments *self.nsecs*, *self.tboudaries* will have *self.nsecs+1* entries covering [*self.minStartTime*,*self.maxStartTime*] and *self.Tcoh* will be the total duration divided by *self.nsecs*.

Each segment is required to be at least two SFTs long.

get_semicohherent_det_stat(F0, F1, F2, Alpha, Delta, asini=None, period=None, ecc=None, tp=None, argp=None, record_segments=False)

Computes the detection statistic (twoF or log10BSGL) semi-coherently at a single point.

As the basic statistic of this class, *self.twoF* is always computed. If *self.singleFstats*, additionally the single-detector 2F-stat values are saved in *self.twoFX* and (optionally) *self.twoFX_per_segment*.

Parameters

- **F0** (*float*) – Parameters at which to compute the statistic.
- **F1** (*float*) – Parameters at which to compute the statistic.
- **F2** (*float*) – Parameters at which to compute the statistic.
- **Alpha** (*float*) – Parameters at which to compute the statistic.
- **Delta** (*float*) – Parameters at which to compute the statistic.
- **asini** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **period** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **ecc** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **tp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **argp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **record_segments** (*boolean, optional*) – If True, store the per-segment F-stat values as *self.twoF_per_segment* and (if *self.singleFstats*) the per-detector per-segment F-stats as *self.twoFX_per_segment*. *False*

Returns

stat – A single value of the detection statistic (semi-coherent twoF or log10BSGL) at the input parameter values. Also stored as *self.twoF* or *self.log10BSGL*.

Return type

float

get_semicohherent_twoF(*F0, F1, F2, Alpha, Delta, asini=None, period=None, ecc=None, tp=None, argp=None, record_segments=False*)

Computes the semi-coherent twoF statistic at a single point.

Parameters

- **F0** (*float*) – Parameters at which to compute the statistic.
- **F1** (*float*) – Parameters at which to compute the statistic.
- **F2** (*float*) – Parameters at which to compute the statistic.
- **Alpha** (*float*) – Parameters at which to compute the statistic.
- **Delta** (*float*) – Parameters at which to compute the statistic.
- **asini** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **period** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **ecc** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **tp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **argp** (*float, optional*) – Optional: Binary parameters at which to compute the statistic. *None*
- **record_segments** (*boolean, optional*) – If True, store the per-segment F-stat values as *self.twoF_per_segment*. *False*

Returns**twoF** – A single value of the semi-coherent twoF statistic at the input parameter values. Also stored as *self.twoF*.**Return type**

float

get_semicohherent_single_IFO_twoFs(*record_segments=False*)

Computes the semi-coherent single-detector F-statss at a single point.

This requires *self.get_semicohherent_twoF()* to be run first.**Parameters****record_segments** (*boolean, optional*) – If True, store the per-detector per-segment F-stat values as *self.twoFX_per_segment*. *False***Returns****twoFX** – A list of the single-detector detection statistics twoF. Also stored as *self.twoFX*.**Return type**

list

get_semicohherent_log10BSGL()

Computes the semi-coherent log10BSGL statistic at a single point.

This requires *self.get_semicohherent_twoF()* and *self.get_semicohherent_single_IFO_twoFs()* to be run first.

Returns

log10BSGL – A single value of the semi-coherent log10BSGL statistic at the input parameter values. Also stored as *self.log10BSGL*.

Return type

float

class pyfstat.core.**SearchForSignalWithJumps**(*args, **kwargs)

Bases: [BaseSearchClass](#)

Internal helper class with some useful methods for glitches or timing noise.

Users should never need to interact with this class, just with the derived search classes.

class pyfstat.core.**SemiCoherentGlitchSearch**(*args, **kwargs)

Bases: [SearchForSignalWithJumps](#), [ComputeFstat](#)

A semi-coherent search for CW signals from sources with timing glitches.

This implements a basic semi-coherent F-stat search in which the data is divided into segments either side of the proposed glitch epochs and the fully-coherent F-stat in each segment is summed to give the semi-coherent F-stat.

Only parameters with a special meaning for `SemiCoherentGlitchSearch` itself are explicitly documented here. For all other parameters inherited from `pyfstat.ComputeFStat` see the documentation of that class.

Parameters

- **label** (*str*) – A label and directory to read/write data from/to.
- **outdir** (*str*) – A label and directory to read/write data from/to.
- **tref** (*int*) – GPS seconds of the reference time, and start and end of the data.
- **minStartTime** (*int*) – GPS seconds of the reference time, and start and end of the data.
- **maxStartTime** (*int*) – GPS seconds of the reference time, and start and end of the data.
- **nglitch** (*int*, *optional*) – The (fixed) number of glitches. This is also allowed to be zero, but occasionally this causes issues, in which case please use the basic `ComputeFstat` class instead. 1
- **sftfilepattern** (*str*, *optional*) – Pattern to match SFTs using wildcards (*) and ranges [0-9]; multiple patterns can be given separated by colons. `None`
- **theta0_idx** (*int*, *optional*) – Index (zero-based) of which segment the theta (searched parameters) refer to. This is useful if providing a tight prior on theta to allow the signal to jump to theta (and not just from). 0

get_semicoherent_nglitch_twoF(*F0*, *F1*, *F2*, *Alpha*, *Delta*, *args)

Returns the semi-coherent glitch summed twoF.

Parameters

- **F0** (*float*) – Parameters at which to compute the statistic.
- **F1** (*float*) – Parameters at which to compute the statistic.
- **F2** (*float*) – Parameters at which to compute the statistic.
- **Alpha** (*float*) – Parameters at which to compute the statistic.
- **Delta** (*float*) – Parameters at which to compute the statistic.
- **args** (*dict*) – Additional arguments for the glitch parameters; see the source code for full details.

Returns

twoFSum – A single value of the semi-coherent summed detection statistic at the input parameter values.

Return type

float

compute_glitch_fstat_single(*F0, F1, F2, Alpha, Delta, delta_F0, delta_F1, tglitch*)

Returns the semi-coherent glitch summed twoF for *nglitch*=1.

NOTE: OBSOLETE, used only for testing.

class `pyfstat.core.DeprecatedClass(*args, **kwargs)`

Bases: `object`

Outdated classes are marked for future removal by inheriting from this.

class `pyfstat.core.DefunctClass(*args, **kwargs)`

Bases: `object`

Removed classes are retained for a while but marked by inheriting from this.

last_supported_version = `None`

pr_welcome = `True`

2.1.4 pyfstat.grid_based_searches module

PyFstat search classes using grid-based methods.

class `pyfstat.grid_based_searches.GridSearch(*args, **kwargs)`

Bases: `BaseSearchClass`

A search evaluating the F-statistic over a regular grid in parameter space.

This implements a simple ‘square box’ grid with fixed spacing and ranges in each dimension, i.e. for each parameter there’s a simple 1D list of grid points and the total grid is just the Cartesian product of these.

For *N* parameter space dimensions and a total of *M* points in the product grid, the basic output is a (*N*+1,*M*)-dimensional array with the detection statistic (twoF or log10BSGL) appended.

NOTE: if a large number of grid points are used, checks against cached data may be slow as the array is loaded into memory. To avoid this, run with the *clean* option which uses a generator instead.

Most parameters are the same as for the *core.ComputeFstat* class, only the additional ones are documented here:

Parameters

- **label** (*str*) – Output filenames will be constructed using this label.
- **outdir** (*str*) – Output directory.
- **F0s** (*tuple*) – A length 3 tuple describing the grid for each parameter, e.g [F0min, F0max, dF0]. Alternatively, for a fixed value simply give [F0]. Unless *input_arrays=True*, then these are the exact arrays to search over.
- **F1s** (*tuple*) – A length 3 tuple describing the grid for each parameter, e.g [F0min, F0max, dF0]. Alternatively, for a fixed value simply give [F0]. Unless *input_arrays=True*, then these are the exact arrays to search over.

- **F2s** (*tuple*) – A length 3 tuple describing the grid for each parameter, e.g [F0min, F0max, dF0]. Alternatively, for a fixed value simply give [F0]. Unless *input_arrays=True*, then these are the exact arrays to search over.
- **Alphas** (*tuple*) – A length 3 tuple describing the grid for each parameter, e.g [F0min, F0max, dF0]. Alternatively, for a fixed value simply give [F0]. Unless *input_arrays=True*, then these are the exact arrays to search over.
- **Deltas** (*tuple*) – A length 3 tuple describing the grid for each parameter, e.g [F0min, F0max, dF0]. Alternatively, for a fixed value simply give [F0]. Unless *input_arrays=True*, then these are the exact arrays to search over.
- **nsegs** (*int, optional*) – Number of segments to split the data set into. If *nsegs=1*, the basic ComputeFstat class is used. If *nsegs>1*, the SemiCoherentSearch class is used. 1
- **input_arrays** (*bool, optional*) – If true, use the F0s, F1s, etc as arrays just as they are given (do not interpret as 3-tuples of [min,max,step]). False
- **clean** (*bool, optional*) – If true, ignore existing data and overwrite. Otherwise, re-use existing data if no inconsistencies are found. False

```
tex_labels = {'Alpha': '$\alpha$', 'Delta': '$\delta$', 'F0': '$f$', 'F1':
'$\dot{f}$', 'F2': '$\ddot{f}$', 'lnBtSG': '$\ln\mathcal{B}_{\mathrm{tS/G}}$',
'log10BSGL': '$\log_{10}\mathcal{B}_{\mathrm{S/GL}}$', 'maxTwoF':
'$\max\widetilde{2\mathcal{F}}$', 'twoF': '$\widetilde{2\mathcal{F}}$'}
```

Formatted labels used for plot annotations.

```
tex_labels0 = {'Alpha': '$-\alpha_0$', 'Delta': '$-\delta_0$', 'F0': '$-f_0$',
'F1': '$-\dot{f}_0$', 'F2': '$-\ddot{f}_0$'}
```

Formatted labels used for annotating central values in plots.

```
fmt_detstat = '%.9g'
```

Standard output precision for detection statistics.

```
check_old_data_is_okay_to_use()
```

Check if an existing output file matches this search and reuse the results.

Results will be loaded from old output file, and no new search run, if all of the following checks pass:

1. Output file with matching name found in *outdir*.
2. Output file is not older than SFT files matching *sftfilepattern*.
3. Parameters string in file header matches current search setup.
4. Data in old file can be loaded successfully, its input parts (i.e. minus the detection statistic columns) matches in dimension with current grid, and the values in those input columns match with the current grid.

Through *utils.read_txt_file_with_header()*, the existing file is read in with *np.genfromtxt()*.

```
run(return_data=False)
```

Execute the actual search over the full grid.

This iterates over all points in the multi-dimensional product grid and the end result is either returned as a numpy array or saved to disk.

Parameters

return_data (*boolean, optional*) – If true, the final inputs+outputs data set is returned as a numpy array. If false, it is saved to disk and nothing is returned. False

Returns

data – The final inputs+outputs data set. Only if *return_data=true*.

Return type

np.ndarray

save_array_to_disk()

Save the results array to a txt file.

This includes a header with version and parameters information.

It should be flexible enough to be reused by child classes, as long as the *_get_savetxt_fmt_dict()* method is suitably overridden to account for any additional parameters.

plot_1D(*xkey*, *ax=None*, *x0=None*, *xrescale=1*, *savefig=True*, *xlabel=None*, *ylabel=None*, *agg_chunksize=None*)

Make a plot of the detection statistic over a single grid dimension.

Parameters

- **xkey** (*str*) – The name of the search parameter to plot against.
- **ax** (*matplotlib.axes._subplots_AxesSubplot or None, optional*) – An optional pre-existing axes set to draw into. *None*
- **x0** (*float or None, optional*) – Plot x values relative to this central value. *None*
- **xrescale** (*float, optional*) – Rescale all x values by this factor. 1
- **savefig** (*bool, optional*) – If true, save the figure in *self.outdir*. If false, return an axis object without saving to disk. *True*
- **xlabel** (*str or None, optional*) – Override default text label for the x-axis. *None*
- **ylabel** (*str or None, optional*) – Override default text label for the y-axis. *None*
- **agg_chunksize** (*int or None, optional*) – Set this to some high value to work around matplotlib ‘Exceeded cell block limit’ errors. *None*

Returns

ax – The axes object containing the plot, only if *savefig=false*.

Return type

matplotlib.axes._subplots_AxesSubplot, optional

plot_2D(*xkey*, *ykey*, *ax=None*, *savefig=True*, *vmin=None*, *vmax=None*, *add_mismatch=None*, *xN=None*, *yN=None*, *flat_keys=[]*, *rel_flat_idxs=[]*, *flatten_method=<function amax>*, *title=None*, *predicted_twoF=None*, *cm=None*, *cbarkwargs={}*, *x0=None*, *y0=None*, *colorbar=False*, *xrescale=1*, *yrescale=1*, *xlabel=None*, *ylabel=None*, *zlabel=None*)

Plots the detection statistic over a 2D grid.

FIXME: this will currently fail if the search went over >2 dimensions.

Parameters

- **xkey** (*str*) – The name of the first search parameter to plot against.
- **ykey** (*str*) – The name of the second search parameter to plot against.
- **ax** (*matplotlib.axes._subplots_AxesSubplot or None, optional*) – An optional pre-existing axes set to draw into. *None*
- **savefig** (*bool, optional*) – If true, save the figure in *self.outdir*. If false, return an axis object without saving to disk. *True*

- **vmin** (*float or None, optional*) – Cutoffs for rescaling the colormap. None
- **vmax** (*float or None, optional*) – Cutoffs for rescaling the colormap. None
- **add_mismatch** (*tuple or None, optional*) – If given a tuple (*xhat, yhat, Tseg*), add a secondary axis with the metric mismatch from the point (*xhat, yhat*) with duration *Tseg*. None
- **xN** (*int or None, optional*) – Number of tick label intervals. None
- **yN** (*int or None, optional*) – Number of tick label intervals. None
- **flat_keys** (*list, optional*) – Keys to be used in flattening higher-dimensional arrays. []
- **rel_flat_idxs** (*list, optional*) – Indices to be used in flattening higher-dimensional arrays. []
- **flatten_method** (*numpy function, optional*) – Function to use in flattening the *flat_keys*, default: *np.max*. <function *amax*>
- **title** (*str or None, optional*) – Optional plot title text. None
- **predicted_twoF** (*float or None, optional*) – Expected/predicted value of twoF, used to rescale the z-axis. None
- **cm** (*matplotlib.colors.ListedColormap or None, optional*) – Override standard (viridis) colormap. None
- **cbarkwargs** (*dict, optional*) – Additional arguments for colorbar formatting. {}
- **x0** (*float, optional*) – Plot x values relative to this central value. None
- **y0** (*float, optional*) – Plot y values relative to this central value. None
- **xrescale** (*float, optional*) – Rescale all x values by this factor. 1
- **yrescale** (*float, optional*) – Rescale all y values by this factor. 1
- **xlabel** (*str, optional*) – Override default text label for the x-axis. None
- **ylabel** (*str, optional*) – Override default text label for the y-axis. None
- **zlabel** (*str, optional*) – Override default text label for the z-axis. None

Returns

ax – The axes object containing the plot, only if *savefig=false*.

Return type

matplotlib.axes._subplots_AxesSubplot, optional

get_max_det_stat()

Get the maximum detection statistic over the grid.

This requires the *run()* method to have been called before.

Returns

d – Dictionary containing parameters and detection statistic at the maximum.

Return type

dict

get_max_twoF()

Get the maximum twoF over the grid.

This requires the *run()* method to have been called before.

Returns

d – Dictionary containing parameters and twoF value at the maximum.

Return type

dict

print_max_twoF()

Get and print the maximum twoF point over the grid.

This prints out the full dictionary from `get_max_twoF()`, i.e. the maximum value and its corresponding parameters.

generate_loudest()

Use `ComputeFstatistic_v2` executable to produce a .loudest file

set_out_file(*extra_label=None*)

Set (or reset) the name of the main output file.

File will always be stored in `self.outdir` and the base of the name be determined from `self.label` and other parts of the search setup, but this method allows to attach an `extra_label` bit if desired.

Parameters

extra_label (*str*, *optional*) – Additional text bit to be attached at the end of the file-name (but before the extension). `None`

class pyfstat.grid_based_searches.TransientGridSearch(*args, **kwargs)

Bases: `GridSearch`

A search for transient CW-like signals using the F-statistic.

This is based on the transient signal model and transient-F-stat algorithm from Prix, Giampanis & Messenger (PRD 84, 023007, 2011): <https://arxiv.org/abs/1104.1704>

The frequency evolution parameters are searched over in a grid just like in the normal `GridSearch`, then at each point the time-dependent ‘atoms’ are used to evaluate partial sums of the F-statistic over a 2D array in transient start times *t0* and duration parameters *tau*.

The signal templates are modulated by a ‘transient window function’ which can be

1. *none* (standard, persistent CW signal)
2. *rect* (rectangular: constant amplitude within $[t0, t0+tau]$, zero outside)
3. *exp* (exponential decay over $[t0, t0+3*tau]$, zero outside)

This class currently only supports fully-coherent searches (*nsegs=1* is hardcoded).

Also see Keitel & Ashton (CQG 35, 205003, 2018): <https://arxiv.org/abs/1805.05652> for a detailed discussion of the GPU implementation.

NOTE for GPU users (*tCWFstatMapVersion="pycuda"*): The underlying `ComputeFstat` class tries to conveniently deal with GPU context management behind the scenes. A known problematic case is if you try to instantiate it twice from the same session/script. If you then get some messages like `RuntimeError: make_default_context()` and `invalid device context`, that is because the GPU is still blocked from the first instance when you try to initiate the second. To avoid this problem, use context management:

```
with pyfstat.TransientGridSearch(
    [...],
    tCWFstatMapVersion="pycuda",
) as search:
    search.search.run()
```

or manually call the `search.search.finalizer_()` method where needed.

Most parameters are the same as for `GridSearch` and the `core.ComputeFstat` class, only the additional ones are documented here:

Parameters

- **transientWindowType** (*str*, *optional*) – If *rect* or *exp*, allow for the Fstat to be computed over a transient range. (*none* instead of *None* explicitly calls the transient-window function, but with the full range, for debugging.) *None*
- **t0Band** (*int*, *optional*) – Search ranges for transient start-time *t0* and duration *tau*. If >0 , search *t0* in $(minStartTime, minStartTime+t0Band)$ and *tau* in $(tauMin, 2*Tsft+tauBand)$. If $=0$, only compute the continuous-wave F-stat with $t0=minStartTime$, $tau=maxStartTime-minStartTime$. *None*
- **tauBand** (*int*, *optional*) – Search ranges for transient start-time *t0* and duration *tau*. If >0 , search *t0* in $(minStartTime, minStartTime+t0Band)$ and *tau* in $(tauMin, 2*Tsft+tauBand)$. If $=0$, only compute the continuous-wave F-stat with $t0=minStartTime$, $tau=maxStartTime-minStartTime$. *None*
- **tauMin** (*int*, *optional*) – Minimum transient duration to cover, defaults to $2*Tsft$. *None*
- **dt0** (*int*, *optional*) – Grid resolution in transient start-time, defaults to *Tsft*. *None*
- **dtau** (*int*, *optional*) – Grid resolution in transient duration, defaults to *Tsft*. *None*
- **outputTransientFstatMap** (*bool*, *optional*) – If true, write additional output files for (*t0, tau*) F-stat maps. (One file for each grid point!) *False*
- **outputAtoms** (*bool*, *optional*) – If true, write additional output files for the F-stat *atoms*. (One file for each grid point!) *False*
- **tCWFstatMapVersion** (*str*, *optional*) – Choose between implementations of the transient F-statistic functionality: *standard* *lal* implementation, *pycuda* for GPU version, and some others only for devel/debug. *'lal'*
- **cudaDeviceName** (*str*, *optional*) – GPU name to be matched against *drv.Device* output, only for *tCWFstatMapVersion=pycuda*. *None*

run(*return_data=False*)

Execute the actual search over the full grid.

This iterates over all points in the multi-dimensional product grid and the end result is either returned as a numpy array or saved to disk.

If the *outputTransientFstatMap* or *outputAtoms* options have been set when initiating the search, additional files are written for each frequency-evolution parameter-space point ('Doppler' point).

Parameters

return_data (*boolean*, *optional*) – If true, the final inputs+outputs data set is returned as a numpy array. If false, it is saved to disk and nothing is returned. *False*

Returns

data – The final inputs+outputs data set. Only if *return_data=True*.

Return type

np.ndarray

get_transient_fstat_map_filename(*param_point*)

Filename convention for given grid point: `freq_alpha_delta_f1dot_f2dot`

Parameters

param_point (*tuple, dict, list, np.void or np.ndarray*) – A multi-dimensional parameter point. If not a type with named fields (e.g. a plain tuple or list), the order must match that of *self.output_keys*.

Returns

f – The constructed filename.

Return type

str

```
class pyfstat.grid_based_searches.SliceGridSearch(*args, **kwargs)
```

Bases: [*DefunctClass*](#)

```
last_supported_version = '1.9.0'
```

```
class pyfstat.grid_based_searches.GridUniformPriorSearch(*args, **kwargs)
```

Bases: [*DefunctClass*](#)

```
last_supported_version = '1.9.0'
```

```
class pyfstat.grid_based_searches.GridGlitchSearch(*args, **kwargs)
```

Bases: [*GridSearch*](#)

A grid search using the *SemiCoherentGlitchSearch* class.

This implements a basic semi-coherent F-stat search in which the data is divided into segments either side of the proposed glitch epochs and the fully-coherent F-stat in each segment is summed to give the semi-coherent F-stat.

This class currently only works for a single glitch in the observing time.

Most parameters are the same as for *GridSearch* and the *core.SemiCoherentGlitchSearch* class, only the additional ones are documented here:

Parameters

- **delta_F0s** (*tuple, optional*) – A length 3 tuple describing the grid of frequency jumps, or just [*delta_F0*] for a fixed value. [0]
- **delta_F1s** (*tuple, optional*) – A length 3 tuple describing the grid of spindown parameter jumps, or just [*delta_F1*] for a fixed value. [0]
- **tglitchs** (*tuple, optional*) – A length 3 tuple describing the grid of glitch epochs, or just [*tglitch*] for a fixed value. These are relative time offsets, referenced to zero at *minStartTime*. None

```
class pyfstat.grid_based_searches.SlidingWindow(*args, **kwargs)
```

Bases: [*DefunctClass*](#)

```
last_supported_version = '1.9.0'
```

```
class pyfstat.grid_based_searches.FrequencySlidingWindow(*args, **kwargs)
```

Bases: [*DefunctClass*](#)

```
last_supported_version = '1.9.0'
```

```
class pyfstat.grid_based_searches.EarthTest(*args, **kwargs)
```

Bases: [*DefunctClass*](#)

```
last_supported_version = '1.9.0'
```

```
class pyfstat.grid_based_searches.DMoff_NO_SPIN(*args, **kwargs)
```

Bases: *DefunctClass*

```
last_supported_version = '1.9.0'
```

2.1.5 pyfstat.gridcorner module

A corner plotting tool for an array (grid) of dependent values.

Given an N-dimensional set of data (i.e. some function evaluated over a grid of coordinates), plot all possible 1D and 2D projections in the style of a ‘corner’ plot.

This code has been copied from Gregory Ashton’s repository <https://gitlab.aei.uni-hannover.de/GregAshton/gridcorner> and it uses both the central idea and some specific code from Daniel Foreman-Mackey’s <https://github.com/dfm/corner.py> re-used under the following licence requirements:

Copyright (c) 2013-2020 Daniel Foreman-Mackey

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project.

```
pyfstat.gridcorner.log_mean(loga, axis)
```

Calculate the $\log(\langle a \rangle)$ mean

Given N logged value log , calculate the $\log_mean \log(\langle loga \rangle) = \log(\text{sum}(\text{np.exp}(loga))) - \log(N)$. Useful for marginalizing over logged likelihoods for example.

Parameters

- **loga** (*array_like*) – Input_array.
- **axes** (*None or int or type of ints, optional*) – Axis or axes over which the sum is taken. By default axis is None, and all elements are summed.

Returns

log_mean – The logged average value ($\text{shape } loga.\text{shape}$)

Return type

ndarray

```
pyfstat.gridcorner.max_slice(D, axis)
```

Return the slice along the given axis

```
pyfstat.gridcorner.idx_array_slice(D, axis, slice_idx)
```

Return the slice along the given axis

```
pyfstat.gridcorner.gridcorner(D, xyz, labels=None, projection='max_slice', max_n_ticks=4, factor=2,
                               whitespace=0.05, showDvals=True, lines=None, label_offset=0.4, **kwargs)
```

Generate a grid corner plot

Parameters

- **D** (*array_like*) – N-dimensional data to plot, *D.shape* should be $(n1, n2, \dots, nN)$, where *N*, is the number of grid points along dimension *i*.
- **xyz** (*list*) – List of 1-dimensional arrays of coordinates. *xyz[i]* should have length *N* (see help for *D*).
- **labels** (*list, optional*) – N+1 length list of labels; the first N correspond to the coordinates labels, the final label is for the dependent (D) variable. *None*
- **projection** (*str or func, optional*) – If a string, one of {"log_mean", "max_slice"} to use *inbuilt functions to calculate either the logged mean or maximum slice projection. Else a function to use for projection, must take an 'axis' argument*. Default is *gridcorner.max_slice()*, to project out a slice along the maximum. 'max_slice'
- **max_n_ticks** (*int, optional*) – Number of ticks for x and y axis of the *pcolormesh* plots. 4
- **factor** (*float, optional*) – Controls the size of one window. 2
- **showDvals** (*bool, optional*) – If true (default) show the D values on the right-hand-side of the 1D plots and add a label. True
- **lines** (*array_like, optional*) – N-dimensional list of values to delineate. *None*

Returns

The figure and NxN set of axes

Return type

fig, axes

```
pyfstat.gridcorner.projection_2D(ax, x, y, D, xidx, yidx, projection, lines=None, **kwargs)
```

```
pyfstat.gridcorner.projection_1D(ax, x, D, xidx, projection, showDvals=True, lines=None, **kwargs)
```

2.1.6 pyfstat.injection_parameters module

Generate injection parameters drawn from different prior populations

```
pyfstat.injection_parameters.custom_prior(prior_function)
```

Intended to be used as a decorator to add custom functions to the list of available priors for *InjectionParametersGenerator*.

For example:

```
@pyfstat.custom_prior
def negative_log_uniform(generator, size):
    return -10**(generator.uniform(size=size))
```

will add the key `negative_log_uniform` to `_pyfstat_custom_priors` with said function as the corresponding value.

A function decorated with `custom_prior` *must* take `generator` and `size` as keyword arguments; otherwise, a `TypeError` will be raised. Additional arguments can be provided as needed.

See docstring of `InjectionParametersGenerator()` for an example on how to draw samples from a custom prior.

Parameters

prior_function (Callable) – Function to be added into `_pyfstat_custom_priors` with a key corresponding *exactly* to the name it was given at definition time.

Returns

prior_function – Same function as the input function.

Return type

Callable

`pyfstat.injection_parameters.uniform_sky_declination(generator, size)`

Return declination values such that, when paired with right ascension values sampled uniformly along $[0, 2\pi]$, the resulting pairs of samples are uniformly distributed on the 2-sphere.

Parameters

- **generator** (Generator) – As required by `InjectionParametersGenerator`.
- **size** (int) – As required by `InjectionParameterGenerator`. Gets passed directly as a kwarg to generator's methods.

Returns

declination – Declination value distributed

Return type

`np.ndarray`

`class pyfstat.injection_parameters.InjectionParametersGenerator(priors, seed=None, generator=None)`

Bases: `object`

Draw injection parameter samples from priors and return in dictionary format.

Parameters

- **priors** (dict) – Each key refers to one of the signal's parameters (following the PyFstat convention).

Each parameter's prior should be given as a dictionary entry as follows: `{"parameter": {"<function>": {**kwargs}}}` where `<function>` may be (exclusively) either a user-defined function decorated with `@custom_prior` or the name of a `scipy.stats` random variable.

- If a user-defined function is used, such a function *must* take | a `generator` kwarg as one of its arguments and use such a generator | (`np.random.Generator` type) | to generate any required random number within the function. | (A deterministic function could then simply ignore this kwarg.) | For example, a negative log-distributed random number could be constructed as

```
import pyfstat

@pyfstat.injection_parameters.custom_prior
```

(continues on next page)

(continued from previous page)

```
def negative_log_uniform(generator, size):
    return -10**(generator.uniform(size=size))

priors = {"my_parameter": {"negative_log_uniform": {}}}
ipg = pyfstat.InjectionParametersGenerator(priors=priors, seed=42)
ipg.draw()
```

- If a `scipy.stats` function is used, it *must* be given as `stats.*` | (i.e. the `stats` namespace should be explicitly included). | For example, a uniform prior between 3 and 5 would be written as `{"parameter": {"stats.uniform": {"loc": 3, "scale": 2}}}`.

```
import pyfstat

priors = {"my_parameter": "stats.uniform": {"loc": 3, "scale": 2}}
ipg = pyfstat.InjectionParametersGenerator(priors=priors, seed=42)
ipg.draw()
```

Delta-priors (i.e. priors for a deterministic output) can also be specified by giving the fixed value to be returned as-is. For example, specifying a fixed value of 1 for the parameter A would be `{"A": 1}`.

- **generator** (Optional[Generator]) – Numpy random number generator (e.g. `np.random.default_rng`) which will be used to draw random numbers from. Conflicts with `seed`. `None`
- **seed** (Optional[int]) – Random seed to create instantiate `np.random.default_rng` from scratch. Conflicts with `generator`. If neither `seed` nor `generator` are given, a random number generator will be instantiated using `seed=None` and a warning will be thrown. `None`

draw()

Draw a single multi-dimensional parameter space point from the given priors.

Returns

parameters – Dictionary of parameter values (one value each). Each key corresponds to that found in `self.priors`.

Return type

Dict

draw_many(size)

Draw size multi-dimensional parameter space points from the given priors.

Parameters

size – Number of samples to return.

Returns

parameters – Dictionary of arrays. Each key corresponds to that found in `self.priors`. Values are numpy arrays of shape `size` as returned by their corresponding method.

Return type

Dict

```
class pyfstat.injection_parameters.AllSkyInjectionParametersGenerator(priors=None,
                                                                    seed=None,
                                                                    generator=None)
```

Bases: *InjectionParametersGenerator*

Draw injection parameter samples from priors and return in dictionary format. This class works in exactly the same way as *InjectionParametersGenerator*, but including by default two extra keys, *Alpha* and *Delta* (sky position's right ascension and declination in radians), which are sample isotropically across the celestial sphere.

Alpha's distribution is *Uniform*(0, 2 pi), and *sin(Delta)*'s distribution is *Uniform*(-1, 1).

2.1.7 pyfstat.logging module

PyFstat's logging implementation.

PyFstat main logger is called *pyfstat* and can be accessed via:

```
import logging
logger = logging.getLogger('pyfstat')
```

Basics of logging: For all our purposes, there are *logger* objects and *handler* objects. Loggers are the ones in charge of logging, hence you call them to emit a logging message with a specific logging level (e.g. `logger.info`); handlers are in charge of redirecting that message to a specific place (e.g. a file or your terminal, which is usually referred to as a *stream*).

The default behaviour upon importing *pyfstat* is to attach a `logging.StreamHandler` to the *pyfstat* logger, printing out to `sys.stdout`. This is only done if the root logger has no handlers attached yet; if it does have at least one handler already, then we inherit those and do not add any extra handlers by default. If, for any reason, `logging` cannot access `sys.stdout` at import time, the exception is reported via `print` and no handlers are attached (i.e. the logger won't print to `sys.stdout`).

The user can modify the logger's behaviour at run-time using `set_up_logger`. This function attaches extra `logging.StreamHandler` and `logging.FileHandler` handlers to the logger, allowing to redirect logging messages to a different stream or a specific output file specified using the `outdir`, `label` variables (with the same format as in the rest of the package).

Finally, logging can be disabled, or the level changed, at run-time by manually configuring the *pyfstat* logger. For example, the following block of code will suppress logging messages below **WARNING**:

```
import logging
logging.getLogger('pyfstat').setLevel(logging.WARNING)
```

```
pyfstat.logging.set_up_logger(outdir=None, label='pyfstat', log_level='INFO',
                             streams=(<_io.TextIOWrapper name='<stdout>' mode='w'
                                     encoding='utf-8'>, ), append=True)
```

Add file and stream handlers to the *pyfstat* logger.

Handler names generated from `streams` and `outdir`, `label` must be unique and no duplicated handler will be attached by this function.

Parameters

- **outdir** (Optional[str]) – Path to outdir directory. If *None*, no file handler will be added. *None*
- **label** (Optional[str]) – Label for the file output handler, i.e. the log file will be called *label.log*. Required, in conjunction with `outdir`, to add a file handler. Ignored otherwise. *'pyfstat'*

- **log_level** (str) – Level of logging. This level is imposed on the logger itself and *every single handler* attached to it. 'INFO'
- **streams** (Optional[Iterable[TextIOWrapper]]) – Stream to which logging messages will be passed using a StreamHandler object. By default, log to sys.stdout. Other common streams include e.g. sys.stderr. (<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>,))
- **append** (bool) – If False, removes all handlers from the *pyfstat* logger before adding new ones. This removal is not propagated to handlers on the *root* logger. True

Returns

logger – Configured instance of the logging.Logger class.

Return type

logging.Logger

2.1.8 pyfstat.make_sfts module

PyFstat tools to generate and manipulate data in the form of SFTs.

class pyfstat.make_sfts.Writer(*args, **kwargs)

Bases: [BaseSearchClass](#)

The main class for generating data in the form of SFTs.

Short Fourier Transforms (SFTs) are a standard data format used in LALSuite, containing the Fourier transform of strain data over a duration Tsft.

SFT data can be generated from scratch, including Gaussian noise and/or simulated CW signals or transient signals. Existing SFTs (real data or previously simulated) can also be reused through the *noiseSFTs* option, allowing to ‘inject’ additional signals into them.

This class currently relies on the *Makefakedata_v5* executable which will be run in a subprocess. See *lalpulsar_Makefakedata_v5 -help* for more detailed help with some of the parameters.

Parameters

- **label** (string, optional) – A human-readable label to be used in naming the output files. NOTE: to agree with the v3 SFT naming specification (<https://dcc.ligo.org/T040164-v2/public>) label can only contain ASCII alphanumeric characters in their “description” field, i.e. no underscores, hyphens etc. Also, internally a “channel”/“frame” name is constructed as *IFO:label*, which may not exceed 64 characters, so a label may only be 60 characters long. 'PyFstat'
- **tstart** (int, optional) – Starting GPS epoch of the data set. If *noiseSFT* are given, this is used as a LALPulsar [SFTConstraint](#). NOTE: mutually exclusive with *timestamps*. None
- **duration** (int, optional) – Duration (in GPS seconds) of the total data set. If *noiseSFT* are given, this is used as a LALPulsar [SFTConstraint](#). NOTE: mutually exclusive with *timestamps*. None
- **tfref** (float or None, optional) – Reference time for simulated signals. Default is None, which sets the reference time to *tstart*. None
- **F0** (float or None, optional) – Frequency of a signal to inject. Also used (if *Band* is not None) as center of frequency band. Also needed when noise-only (*h0=None* or *h0==0*) but no *noiseSFTs* given, in which case it is also used as center of frequency band. None

- **F1** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. 0
- **F2** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. 0
- **Alpha** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. None
- **Delta** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. None
- **h0** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. None
- **cosi** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. None
- **psi** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. 0.0
- **phi** (*float or None, optional*) – Additional frequency evolution and amplitude parameters for a signal. If $h0=None$ or $h0=0$, these are all ignored. If $h0>0$, then at least $[Alpha, Delta, cosi]$ need to be set explicitly. 0
- **Tsft** (*int, optional*) – The SFT duration in seconds. Will be ignored if *noiseSFTs* are given. 1800
- **outdir** (*str, optional*) – The directory where files are written to. Default: current working directory. '.'
- **sqrtSX** (*float or list or str or None, optional*) – Single-sided PSD values for generating fake Gaussian noise. Single float or str value: use same for all detectors. List or comma-separated string: must match $\text{len}(\text{detectors})$. Detectors will be paired to list elements following alphabetical order. None
- **noiseSFTs** (*str or None, optional*) – Existing SFT files on top of which signals will be injected. If not *None*, additional constraints can be applied using the arguments *tstart* and *duration*. NOTE: mutually exclusive with *timestamps*. None
- **SFTWindowType** (*str or None, optional*) – LAL name of the windowing function to apply to the data. None
- **SFTWindowParam** (*float, optional*) – Optional parameter for some windowing functions. None
- **SFTWindowBeta** (*float, optional*) – Defunct alias to *SFTWindowParam*. Will be removed in a future release. None
- **Band** (*float or None, optional*) – If float, and *F0* is also not *None*, then output SFTs cover $[F0-Band/2, F0+Band/2]$. If *None* and *noiseSFTs* given, use their bandwidth. If *None* and no *noiseSFTs* given, a minimal covering band for a perfectly-matched single-template ComputeFstat analysis is estimated. None

- **detectors** (*str or None, optional*) – Comma-separated list of detectors to generate data for. May be required depending on *timestamps*; see its documentation. *None*
- **earth_ephem** (*str or None, optional*) – Paths of the two files containing positions of Earth and Sun. If *None*, will check standard sources as per `utils.get_ephemeris_files()`. *None*
- **sun_ephem** (*str or None, optional*) – Paths of the two files containing positions of Earth and Sun. If *None*, will check standard sources as per `utils.get_ephemeris_files()`. *None*
- **transientWindowType** (*str, optional*) – If *none*, a fully persistent CW signal is simulated. If *rect* or *exp*, a transient signal with the corresponding amplitude evolution is simulated. *'none'*
- **transientStartTime** (*int or None, optional*) – Start time for a transient signal. *None*
- **transientTau** (*int or None, optional*) – Duration (*rect* case) or decay time (*exp* case) of a transient signal. *None*
- **randSeed** (*int or None, optional*) – Optionally fix the random seed of Gaussian noise generation for reproducibility. *None*
- **timestamps** (*str or dict, optional*) – Dictionary of timestamps (each key must refer to a detector), a single list of timestamps (will be replicated for all detectors; *detectors* must be set), or comma-separated list of per-detector timestamps files (simple text files, comments must use %, the first column is interpreted as SFT start times and additional columns are ignored; *detectors* must be set, and the length and order must match). Timestamps must be integers; otherwise, will be implicitly cast by this method and MFDv5. NOTE: mutually exclusive with [*tstart, duration*] and with *noiseSFTs*. *None*

mfd = `'lalpulsar_Makefakedata_v5'`

The executable; can be overridden by child classes.

signal_parameter_labels = `['tref', 'F0', 'F1', 'F2', 'Alpha', 'Delta', 'h0', 'cosi', 'psi', 'phi', 'transientWindowType', 'transientStartTime', 'transientTau']`

Default convention of labels for the various signal parameters.

gps_time_and_string_formats_as_LAL = `{'refTime': ':10.9f', 'transientStartTime': ':10.0f', 'transientTau': ':10.0f', 'transientWindowType': ':s'}`

Dictionary to ensure proper format handling for some special parameters.

GPS times should NOT be parsed using scientific notation. LAL routines would silently parse them wrongly.

required_signal_parameters = `['F0', 'Alpha', 'Delta', 'cosi']`

List of parameters required for a successful execution of `Makefakedata_v5`. The rest of available parameters are not required as they have default values silently given by `Makefakedata_v5`

property tend

Defined as *self.start + self.duration*.

If stored as an attribute, there would be the risk of it going out of sync with the other two values.

calculate_fmin_Band()

Set *fmin* and *Band* for the output SFTs to cover.

Either uses the user-provided *Band* and puts *F0* in the middle; does nothing to later reuse the full bandwidth of *noiseSFTs* (only if using MFDv5); or if *F0!=None*, *noiseSFTs=None* and *Band=None* it estimates a minimal band for just the injected signal: F-stat covering band plus extra bins for demod default parameters. This way a perfectly matched single-template *ComputeFstat* analysis should run through perfectly on the

returned SFTs. For any wider-band or mismatched search, one needs to set *Band* manually. If using MFDv4, at least *F0* is required even if *noiseSFTs!=None*.

make_cff(*verbose=False*)

Generates a .cff file including signal injection parameters.

This will be saved to *self.config_file_name*.

Parameters

verbose (*boolean, optional*) – If true, increase logging verbosity. False

check_cached_data_okay_to_use(*cl_mfd*)

Check if SFT files already exist that can be re-used.

This does not check the actual data contents of the SFTs, but only the following criteria:

- filename
- if injecting a signal, that the .cff file is older than the SFTs (but its contents should have been checked separately)
- that the commandline stored in the (first) SFT header matches

Parameters

cl_mfd (*str*) – The commandline we'd execute if not finding matching files.

make_data(*verbose=False*)

A convenience wrapper to generate a cff file and then SFTs.

run_makefakedata()

Generate the SFT data calling Makefakedata_v5 executable.

This first builds the full commandline, then calls *check_cached_data_okay_to_use()* to see if equivalent data files already exist, and else runs the actual generation code.

predict_fstat(*assumeSqrtSX=None*)

Predict the expected F-statistic value for the injection parameters.

Through *utils.predict_fstat()*, this wraps the PredictFstat executable.

Parameters

assumeSqrtSX (*float, str or None, optional*) – If None, PSD is estimated from *self.sftfilepath*. Else, assume this stationary per-detector noise-floor instead. Single float or str value: use same for all IFOs. Comma-separated string: must match *len(self.detectors)* and the data in *self.sftfilepath*. Detectors will be paired to list elements following alphabetical order. None

class pyfstat.make_sfts.BinaryModulatedWriter(*args, **kwargs)

Bases: *Writer*

Special Writer variant for simulating a CW signal for a source in a binary system.

Most parameters are the same as for the basic *Writer* class, only the additional ones are documented here:

Parameters

- **tp** (*optional*) – binary orbit parameters 0.0
- **argp** (*optional*) – binary orbit parameters 0.0
- **asini** (*optional*) – binary orbit parameters 0.0
- **ecc** (*optional*) – binary orbit parameters 0.0

- **period** (*optional*) – binary orbit parameters 0.0

class pyfstat.make_sfts.LineWriter(*args, **kwargs)

Bases: *Writer*

Inject a simulated line-like detector artifact into SFT data.

A (transient) line is defined as a constant amplitude and constant excess power artifact in the data.

In practice, it corresponds to a CW without Doppler or antenna-pattern-induced amplitude modulation.

NOTE: This functionality is implemented via *Makefakedata_v4*'s *lineFeature* option. This version of MFD only supports one interferometer at a time.

NOTE: All signal parameters except for *h0*, *Freq*, *phi0* and transient parameters will be ignored.

Parameters

- **label** (*string*) – A human-readable label to be used in naming the output files. NOTE: to agree with the v3 SFT naming specification (<https://dcc.ligo.org/T040164-v2/public>) label can only contain ASCII alphanumeric characters in their “description” field, i.e. no under-scores, hyphens etc. Also, internally a “channel”/“frame” name is constructed as *IFO:label*, which may not exceed 64 characters, so a label may only be 60 characters long.
- **tstart** (*int*) – Starting GPS epoch of the data set. If *noiseSFT* are given, this is used as a LALPulsar *SFTConstraint*. NOTE: mutually exclusive with *timestamps*.
- **duration** (*int*) – Duration (in GPS seconds) of the total data set. If *noiseSFT* are given, this is used as a LALPulsar *SFTConstraint*. NOTE: mutually exclusive with *timestamps*.
- **treref** (*float or None*) – Reference time for simulated signals. Default is *None*, which sets the reference time to *tstart*.
- **F0** (*float or None*) – Frequency of a signal to inject. Also used (if *Band* is not *None*) as center of frequency band. Also needed when noise-only (*h0=None* or *h0==0*) but no *noiseSFTs* given, in which case it is also used as center of frequency band.
- **F1** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha*,*Delta*,*cosi*] need to be set explicitly.
- **F2** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha*,*Delta*,*cosi*] need to be set explicitly.
- **Alpha** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha*,*Delta*,*cosi*] need to be set explicitly.
- **Delta** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha*,*Delta*,*cosi*] need to be set explicitly.
- **h0** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha*,*Delta*,*cosi*] need to be set explicitly.
- **cosi** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha*,*Delta*,*cosi*] need to be set explicitly.

- **psi** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha,Delta,cosi*] need to be set explicitly.
- **phi** (*float or None*) – Additional frequency evolution and amplitude parameters for a signal. If *h0=None* or *h0=0*, these are all ignored. If *h0>0*, then at least [*Alpha,Delta,cosi*] need to be set explicitly.
- **Tsft** (*int*) – The SFT duration in seconds. Will be ignored if *noiseSFTs* are given.
- **outdir** (*str*) – The directory where files are written to. Default: current working directory.
- **sqrSX** (*float or list or str or None*) – Single-sided PSD values for generating fake Gaussian noise. Single float or str value: use same for all detectors. List or comma-separated string: must match len(detectors). Detectors will be paired to list elements following alphabetical order.
- **noiseSFTs** (*str or None*) – Existing SFT files on top of which signals will be injected. If not *None*, additional constraints can be applied using the arguments *tstart* and *duration*. NOTE: mutually exclusive with *timestamps*.
- **SFTWindowType** (*str or None*) – LAL name of the windowing function to apply to the data.
- **SFTWindowParam** (*float*) – Optional parameter for some windowing functions.
- **SFTWindowBeta** (*float*) – Defunct alias to *SFTWindowParam*. Will be removed in a future release.
- **Band** (*float or None*) – If float, and *F0* is also not *None*, then output SFTs cover [*F0-Band/2,F0+Band/2*]. If *None* and *noiseSFTs* given, use their bandwidth. If *None* and no *noiseSFTs* given, a minimal covering band for a perfectly-matched single-template ComputeFstat analysis is estimated.
- **detectors** (*str or None*) – Comma-separated list of detectors to generate data for. May be required depending on *timestamps*; see its documentation.
- **earth_ephem** (*str or None*) – Paths of the two files containing positions of Earth and Sun. If *None*, will check standard sources as per `utils.get_ephemeris_files()`.
- **sun_ephem** (*str or None*) – Paths of the two files containing positions of Earth and Sun. If *None*, will check standard sources as per `utils.get_ephemeris_files()`.
- **transientWindowType** (*str*) – If *none*, a fully persistent CW signal is simulated. If *rect* or *exp*, a transient signal with the corresponding amplitude evolution is simulated.
- **transientStartTime** (*int or None*) – Start time for a transient signal.
- **transientTau** (*int or None*) – Duration (*rect* case) or decay time (*exp* case) of a transient signal.
- **randSeed** (*int or None*) – Optionally fix the random seed of Gaussian noise generation for reproducibility.
- **timestamps** (*str or dict*) – Dictionary of timestamps (each key must refer to a detector), a single list of timestamps (will be replicated for all detectors; *detectors* must be set), or comma-separated list of per-detector timestamps files (simple text files, comments must use %, the first column is interpreted as SFT start times and additional columns are ignored; *detectors* must be set, and the length and order must match). Timestamps must be integers; otherwise, will be implicitly cast by this method and MFDv5. NOTE: mutually exclusive with [*tstart, duration*] and with *noiseSFTs*.

mfd = 'lalpulsar_Makefakedata_v4'

The executable (older version that supports the *–lineFeature* option).

required_signal_parameters = ['F0', 'phi', 'h0']

Required parameters for Makefakedata_v4 to success. Any other parameter is silently given a default value by Makefakedata_v4.

signal_parameters_labels = ['F0', 'phi', 'h0', 'transientWindowType', 'transientStartTime', 'transientTau']

Other signal parameters will be removed before passing to Makefakedata_v4.

class pyfstat.make_sfts.GlitchWriter(*args, **kwargs)

Bases: [SearchForSignalWithJumps](#), [Writer](#)

Special Writer variant for simulating a CW signal containing a timing glitch.

Most parameters are the same as for the basic *Writer* class, only the additional ones are documented here:

Parameters

- **dtglitch** (*float or None, optional*) – Time (in GPS seconds) of the glitch after *tstart*. To create data without a glitch, set *dtglitch=None*. *None*
- **delta_phi** (*float, optional*) – Instantaneous glitch magnitudes in rad, Hz, and Hz/s respectively. 0
- **delta_F0** (*float, optional*) – Instantaneous glitch magnitudes in rad, Hz, and Hz/s respectively. 0
- **delta_F1** (*float, optional*) – Instantaneous glitch magnitudes in rad, Hz, and Hz/s respectively. 0

make_cff(*verbose=False*)

Generates a .cff file including signal injection parameters, including a glitch.

This will be saved to *self.config_file_name*.

Parameters

- **verbose** (*boolean, optional*) – If true, increase logging verbosity. *False*

class pyfstat.make_sfts.FrequencyModulatedArtifactWriter(*args, **kwargs)

Bases: [Writer](#)

Specialized Writer variant to generate SFTs containing simulated instrumental artifacts.

Contrary to the main *Writer* class, this calls the older *Makefakedata_v4* executable which supports the special *–lineFeature* option. See *lalpulsar_Makefakedata_v4 –help* for more detailed help with some of the parameters.

Parameters

- **label** (*string*) – A human-readable label to be used in naming the output files.
- **outdir** (*str, optional*) – The directory where files are written to. Default: current working directory. '.'
- **tstart** (*int, optional*) – Starting GPS epoch of the data set. 7000000000
- **duration** (*int, optional*) – Duration (in GPS seconds) of the total data set. 86400
- **F0** (*float, optional*) – Frequency of the artifact. 30
- **F1** (*float, optional*) – Frequency drift of the artifact. 0
- **treref** (*float or None, optional*) – Reference time for simulated signals. Default is *None*, which sets the reference time to *tstart*. *None*

- **h0** (*float, optional*) – Amplitude of the artifact. 10
- **Tsft** (*int, optional*) – The SFT duration in seconds. Will be ignored if *noiseSFTs* are given. 1800
- **sqrtSX** (*float, optional*) – Background detector noise level. 0.0
- **Band** (*float, optional*) – Output SFTs cover $[F0-Band/2, F0+Band/2]$. 4
- **Pmod** (*float, optional*) – Modulation period of the artifact. 86164.09053133354
- **Pmod_phi** (*float, optional*) – Additional parameters for modulation of the artifact. 0
- **Pmod_amp** (*float, optional*) – Additional parameters for modulation of the artifact. 1
- **Alpha** (*float or None, optional*) – If not none: add an orbital modulation to the artifact corresponding to a signal from that sky position, in radians. None
- **Delta** (*float or None, optional*) – If not none: add an orbital modulation to the artifact corresponding to a signal from that sky position, in radians. None
- **detectors** (*str or None, optional*) – Comma-separated list of detectors to generate data for. None
- **earth_ephem** (*str or None, optional*) – Paths of the two files containing positions of Earth and Sun. If None, will check standard sources as per `utils.get_ephemeris_files()`. None
- **sun_ephem** (*str or None, optional*) – Paths of the two files containing positions of Earth and Sun. If None, will check standard sources as per `utils.get_ephemeris_files()`. None
- **randSeed** (*int or None, optional*) – Optionally fix the random seed of Gaussian noise generation for reproducibility. None

get_frequency(*t*)

Evolve the artifact frequency in time.

This includes a drift term and optionally, if *Alpha* and *Delta* are not *None*, a simulated orbital modulation.

Parameters

t (*float*) – Time stamp to evaluate the frequency at.

Returns

f – Frequency at time *t*.

Return type

float

get_h0(*t*)

Evaluate the artifact amplitude at a given time.

NOTE: Here it's actually implemented as a constant!

Parameters

t (*float*) – Time stamp to evaluate at.

Returns

h0 – Amplitude at time *t*.

Return type

float

concatenate_sft_files()

Merges the individual SFT files via `splitSFTs` executable.

pre_compute_evolution()

Precomputes evolution parameters for the artifact.

This computes midtimes, frequencies, phases and amplitudes over the list of SFT timestamps.

make_ith_sft(*i*)

Call MFDv4 to create a single SFT with evolved artifact parameters.

make_data(*num_threads=1*)

Create a full multi-SFT data set.

This loops over SFTs and generate them serially or in parallel, then concatenates the results together at the end.

Parameters

num_processes (*int*) – Number threads to use when running in parallel. Verbatim implementation of the former *args.N*.

run_makefakedata_v4(*mid_time, lineFreq, linePhi, h0, tmp_outdir*)

Generate SFT data using the MFDv4 code with the `-lineFeature` option.

class pyfstat.make_sfts.FrequencyAmplitudeModulatedArtifactWriter(args*, ***kwargs*)**

Bases: [*FrequencyModulatedArtifactWriter*](#)

A variant of *FrequencyModulatedArtifactWriter* with evolving amplitude.

Parameters

- **label** (*string*) – A human-readable label to be used in naming the output files.
- **outdir** (*str, optional*) – The directory where files are written to. Default: current working directory. `'.'`
- **tstart** (*int, optional*) – Starting GPS epoch of the data set. `7000000000`
- **duration** (*int, optional*) – Duration (in GPS seconds) of the total data set. `86400`
- **F0** (*float, optional*) – Frequency of the artifact. `30`
- **F1** (*float, optional*) – Frequency drift of the artifact. `0`
- **tref** (*float or None, optional*) – Reference time for simulated signals. Default is *None*, which sets the reference time to *tstart*. `None`
- **h0** (*float, optional*) – Amplitude of the artifact. `10`
- **Tsft** (*int, optional*) – The SFT duration in seconds. Will be ignored if *noiseSFTs* are given. `1800`
- **sqrtsX** (*float, optional*) – Background detector noise level. `0.0`
- **Band** (*float, optional*) – Output SFTs cover $[F0-Band/2, F0+Band/2]$. `4`
- **Pmod** (*float, optional*) – Modulation period of the artifact. `86164.09053133354`
- **Pmod_phi** (*float, optional*) – Additional parameters for modulation of the artifact. `0`
- **Pmod_amp** (*float, optional*) – Additional parameters for modulation of the artifact. `1`
- **Alpha** (*float or None, optional*) – If not none: add an orbital modulation to the artifact corresponding to a signal from that sky position, in radians. `None`
- **Delta** (*float or None, optional*) – If not none: add an orbital modulation to the artifact corresponding to a signal from that sky position, in radians. `None`

- **detectors** (*str or None, optional*) – Comma-separated list of detectors to generate data for. *None*
- **earth_ephem** (*str or None, optional*) – Paths of the two files containing positions of Earth and Sun. If *None*, will check standard sources as per `utils.get_ephemeris_files()`. *None*
- **sun_ephem** (*str or None, optional*) – Paths of the two files containing positions of Earth and Sun. If *None*, will check standard sources as per `utils.get_ephemeris_files()`. *None*
- **randSeed** (*int or None, optional*) – Optionally fix the random seed of Gaussian noise generation for reproducibility. *None*

get_h0(*t*)

Evaluate the artifact amplitude at a given time.

NOTE: Here it's actually changing over time!

Parameters

t (*float*) – Time stamp to evaluate at.

Returns

h0 – Amplitude at time *t*.

Return type

float

2.1.9 pyfstat.mcmc_based_searches module

PyFstat search & follow-up classes using MCMC-based methods

The general approach is described in Ashton & Prix (PRD 97, 103020, 2018): <https://arxiv.org/abs/1802.05450> and we use the *ptemcee* sampler described in Vousden et al. (MNRAS 455, 1919-1937, 2016): <https://arxiv.org/abs/1501.05823> and based on Foreman-Mackey et al. (PASP 125, 306, 2013): <https://arxiv.org/abs/1202.3665>

2.1.9.1 Defining the prior

The MCMC based searches (i.e. *pyfstat.MCMC**) require a prior specification for each model parameter, implemented via a *python dictionary*. This is best explained through a simple example, here is the prior for a *directed* search with a *uniform* prior on the frequency and a *normal* prior on the frequency derivative:

```
theta_prior = {'F0': {'type': 'unif',
                      'lower': 29.9,
                      'upper': 30.1},
               'F1': {'type': 'norm',
                      'loc': 0,
                      'scale': 1e-10},
               'F2': 0,
               'Alpha': 2.3,
               'Delta': 1.8
               }
```

For the sky positions Alpha and Delta, we give the fixed values (i.e. they are considered *known* by the MCMC simulation), the same is true for F2, the second derivative of the frequency which we fix at 0. Meanwhile, for the frequency F0 and first frequency derivative F1 we give a dictionary specifying their prior distribution. This dictionary must contain three arguments: the *type* (in this case either *unif* or *norm*) which specifies the type of distribution, then two shape arguments. The shape parameters will depend on the *type* of distribution, but here we use *lower* and *upper*, required for the *unif* prior while *loc* and *scale* are required for the *norm* prior.

Currently, two other types of prior are implemented: `halfnorm`, `neghalfnorm` (both of which require `loc` and `scale` shape parameters). Further priors can be added by modifying `pyfstat.MCMCSearch._generic_inprior`.

class `pyfstat.mcmc_based_searches.MCMCSearch(*args, **kwargs)`

Bases: [`BaseSearchClass`](#)

MCMC search using `ComputeFstat`.

Evaluates the coherent F-statistic across a parameter space region corresponding to an isolated/binary-modulated CW signal.

Parameters

- **theta_prior** (*dict*) – Dictionary of priors and fixed values for the search parameters. For each parameters (key of the dict), if it is to be held fixed the value should be the constant float, if it is to be searched, the value should be a dictionary of the prior.
- **tref** (*int*) – GPS seconds of the reference time, start time and end time. While `tref` is required, `minStartTime` and `maxStartTime` default to `None` in which case all available data is used.
- **minStartTime** (*int, optional*) – GPS seconds of the reference time, start time and end time. While `tref` is required, `minStartTime` and `maxStartTime` default to `None` in which case all available data is used. `None`
- **maxStartTime** (*int, optional*) – GPS seconds of the reference time, start time and end time. While `tref` is required, `minStartTime` and `maxStartTime` default to `None` in which case all available data is used. `None`
- **label** (*str*) – A label and output directory (optional, default is `data`) to name files
- **outdir** (*str, optional*) – A label and output directory (optional, default is `data`) to name files 'data'
- **sftfilepattern** (*str, optional*) – Pattern to match SFTs using wildcards (*) and ranges [0-9]; multiple patterns can be given separated by colons. `None`
- **detectors** (*str, optional*) – Two character reference to the detectors to use, specify `None` for no constraint and comma separated strings for multiple references. `None`
- **nsteps** (*list (2,), optional*) – Number of burn-in and production steps to take, [`nburn`, `nprod`]. See `pyfstat.MCMCSearch.setup_initialisation()` for details on adding initialisation steps. `[100, 100]`
- **nwalkers** (*int, optional*) – The number of walkers and temperates to use in the parallel tempered PTSampler. `100`
- **ntemps** (*int, optional*) – The number of walkers and temperates to use in the parallel tempered PTSampler. `1`
- **log10beta_min** (*float < 0, optional*) – The `log10(beta)` value. If given, the set of betas passed to PTSampler are generated from `np.logspace(0, log10beta_min, ntemps)` (given in descending order to `ptemcee`). `-5`
- **theta_initial** (*dict, array, optional*) – A dictionary of distribution about which to distribute the initial walkers about. `None`
- **rho_hat_max** (*float, optional*) – Upper bound for the SNR scale parameter (required to normalise the Bayes factor) - this needs to be carefully set when using the evidence. `1000`
- **binary** (*bool, optional*) – If true, search over binary orbital parameters. `False`
- **BSGL** (*bool, optional*) – If true, use the BSGL statistic. `False`

- **BtSG** (*bool, optional*) – If true, use the transient lnBtSG statistic. (Only for transient searches.) False
- **SSBPrec** (*int, optional*) – SSBPrec (SSB precision) to use when calling `ComputeFstat`. See `core.ComputeFstat`.
- **RngMedWindow** (*int, optional*) – Running-Median window size (number of bins) for `ComputeFstat`. See `core.ComputeFstat`. None
- **minCoverFreq** (*float, optional*) – Minimum and maximum instantaneous frequency which will be covered over the SFT time span as passed to `CreateFstatInput`. See `core.ComputeFstat`. None
- **maxCoverFreq** (*float, optional*) – Minimum and maximum instantaneous frequency which will be covered over the SFT time span as passed to `CreateFstatInput`. See `core.ComputeFstat`. None
- **injectSources** (*dict, optional*) – If given, inject these properties into the SFT files before running the search. See `core.ComputeFstat`. None
- **assumeSqrtSX** (*float or list or str, optional*) – Don't estimate noise-floors, but assume (stationary) per-IFO $\sqrt{SX}$. See `core.ComputeFstat`. None
- **transientWindowType** (*str, optional*) – If 'rect' or 'exp', compute atoms so that a transient (t_0, τ) map can later be computed. ('none' instead of None explicitly calls the transient-window function, but with the full range, for debugging). See `core.ComputeFstat`. Currently only supported for `nsegs=1`. None
- **tcWFstatMapVersion** (*str, optional*) – Choose between standard 'lal' implementation, 'pycuda' for gpu, and some others for devel/debug. 'lal'
- **allowedMismatchFromSFTLength** (*float, optional*) – Maximum allowed mismatch from SFTs being too long [Default: what's hardcoded in `XLALFstatMaximumSFTLength`]. None
- **clean** (*bool, optional*) – If true, ignore existing data and overwrite. Otherwise, re-use existing data if no inconsistencies are found. False

```
symbol_dictionary = {'Alpha': '$\\alpha$', 'Delta': '$\\delta$', 'F0': '$f$',
                    'F1': '$\\dot{f}$$', 'F2': '$\\ddot{f}$$', 'argp': 'argp', 'asini': 'asini',
                    'ecc': 'ecc', 'period': 'P', 'tp': 'tp'}
```

Key, val pairs of the parameters ($F_0, F_1, \dots$), to LaTeX math symbols for plots

```
unit_dictionary = {'Alpha': 'rad', 'Delta': 'rad', 'F0': 'Hz', 'F1': 'Hz/s',
                  'F2': 'Hz/s^2$', 'argp': '', 'asini': '', 'ecc': '', 'period': 's', 'tp':
                  's'}
```

Key, val pairs of the parameters (i.e. F_0, F_1), and the units (i.e. Hz)

```
transform_dictionary = {}
```

Key, val pairs of the parameters (i.e. F_0, F_1), where the key is itself a dictionary which can item *multiplier*, *subtractor*, or *unit* by which to transform by and update the units.

```
setup_initialisation(nburn0, scatter_val=1e-10)
```

Add an initialisation step to the MCMC run

If called prior to `run()`, adds an initial step in which the MCMC simulation is run for `nburn0` steps. After this, the MCMC simulation continues in the usual manner (i.e. for `nburn` and `nprod` steps), but the walkers are reset scattered around the maximum likelihood position of the initialisation step.

Parameters

- **nburn0** (*int*) – Number of initialisation steps to take.
- **scatter_val** (*float, optional*) – Relative number to scatter walkers around the maximum likelihood position after the initialisation step. If the maximum likelihood point is located at p , the new walkers are randomly drawn from a multivariate gaussian distribution centered at p with standard deviation $\text{diag}(\text{scatter_val} * p)$. $1\text{e-}10$

run(*proposal_scale_factor=2, save_pickle=True, export_samples=True, save_loudest=True, plot_walkers=True, walker_plot_args=None, window=50*)

Run the MCMC simulation

Parameters

- **proposal_scale_factor** (*float, optional*) – The proposal scale factor $a > 1$ used by the sampler. See Goodman & Weare (Comm App Math Comp Sci, Vol 5, No. 1, 2010): 10.2140/camcos.2010.5.65. The bigger the value, the wider the range to draw proposals from. If the acceptance fraction is too low, you can raise it by decreasing the a parameter; and if it is too high, you can reduce it by increasing the a parameter. See Foreman-Mackay et al. (PASP 125 306, 2013): <https://arxiv.org/abs/1202.3665>. 2
- **save_pickle** (*bool, optional*) – If true, save a pickle file of the full sampler state. True
- **export_samples** (*bool, optional*) – If true, save ASCII samples file to disk. See *MCMCSearch.export_samples_to_disk*. True
- **save_loudest** (*bool, optional*) – If true, save a CFSv2 .loudest file to disk. See *MCMCSearch.generate_loudest*. True
- **plot_walkers** (*bool, optional*) – If true, save trace plots of the walkers. True
- **walker_plot_args** (*optional*) – Dictionary passed as kwargs to *_plot_walkers* to control the plotting. Histogram of sampled detection statistic values can be retrieved setting “plot_det_stat” to *True*. Parameters corresponding to an injected signal can be passed through “injection_parameters” as a dictionary containing the parameters of said signal. All parameters being searched for must be present, otherwise this option is ignored. If both “fig” and “axes” entries are set, the plot is not saved to disk directly, but (fig, axes) are returned. None
- **window** (*int, optional*) – The minimum number of autocorrelation times needed to trust the result when estimating the autocorrelation time (see *ptemcee.Sampler.get_autocorr_time* for further details). 50

plot_corner(*figsize=(10, 10), add_prior=False, nstds=None, label_offset=0.4, dpi=300, rc_context={}, tglitch_ratio=False, fig_and_axes=None, save_fig=True, **kwargs*)

Generate a corner plot of the posterior

Using the *corner* package (<https://pypi.python.org/pypi/corner/>), generate estimates of the posterior from the production samples.

Parameters

- **figsize** (*tuple (7, 7), optional*) – Figure size in inches (passed to *plt.subplots*) (10, 10)
- **add_prior** (*bool, str, optional*) – If true, plot the prior as a red line. If ‘full’ then for uniform priors plot the full extent of the prior. False
- **nstds** (*float, optional*) – The number of standard deviations to plot centered on the median. Standard deviation is computed from the samples using *numpy.std*. None

- **label_offset** (*float, optional*) – Offset the labels from the plot: useful to prevent overlapping the tick labels with the axis labels. This option is passed to `ax.[x|y]axis.set_label_coords. 0.4`
- **dpi** (*int, optional*) – Passed to `plt.savefig. 300`
- **rc_context** (*dict, optional*) – Dictionary of rc values to set while generating the figure (see matplotlib rc for more details). `{}`
- **tglitch_ratio** (*bool, optional*) – If true, and `tglitch` is a parameter, plot posteriors as the fractional time at which the glitch occurs instead of the actual time. `False`
- **fig_and_axes** (*tuple, optional*) – (fig, axes) tuple to plot on. The axes must be of the right shape, namely (ndim, ndim) `None`
- **save_fig** (*bool, optional*) – If true, save the figure, else return the fig, axes. `True`
- ****kwargs** – Passed to `corner.corner`. Use “truths” to plot the true parameters of a signal.

Returns

The matplotlib figure and axes, only returned if `save_fig = False`.

Return type

fig, axes

plot_chainconsumer(*save_fig=True, label_offset=0.25, dpi=300, **kwargs*)

Generate a corner plot of the posterior using the *chainconsumer* package.

chainconsumer is an optional dependency of PyFstat. See <https://samreay.github.io/ChainConsumer/>.

Parameters are akin to the ones described in `MCMCSearch.plot_corner`. Only the differing parameters are explicitly described.

Parameters

****kwargs** – Passed to `chainconsumer.plotter.plot`. Use “truths” to plot the true parameters of a signal.

plot_prior_posterior(*normal_stds=2, injection_parameters=None, fig_and_axes=None, save_fig=True*)

Plot the prior and posterior probability distributions in the same figure

Parameters

- **normal_stds** (*int, optional*) – Bounds of priors in terms of their standard deviation. Only used if *norm*, *halfnorm*, *neghalfnorm* or *lognorm* priors are given, otherwise ignored. `2`
- **injection_parameters** (*dict, optional*) – Dictionary containing the parameters of a signal. All parameters being searched must be present as dictionary keys, otherwise this option is ignored. `None`
- **fig_and_axes** (*tuple, optional*) – (fig, axes) tuple to plot on. `None`
- **save_fig** (*bool, optional*) – If true, save the figure, else return the fig, axes. `True`

Returns

(**fig, ax**) – If `save_fig` evaluates to `False`, return figure and axes.

Return type

(matplotlib.pyplot.figure, matplotlib.pyplot.axes)

plot_cumulative_max(***kwargs*)

Plot the cumulative twoF for the maximum posterior estimate.

This method accepts the same arguments as *pyfstat.core.ComputeFstat.plot_twoF_cumulative*, except for *CFS_input*, which is taken from the loudest candidate; and *label* and *outdir*, which are taken from the instance of this class.

For example, one can pass signal arguments to *predic_twoF_cumulative* through *PFS_kwargs*, or set the number of segments using *num_segments_(CFS|PFS)*. The same applies for other options such as *tstart*, *tend* or *savefig*. Every single of these arguments will be passed to *pyfstat.core.ComputeFstat.plot_twoF_cumulative* as they are, using their default argument otherwise.

See *pyfstat.core.ComputeFstat.plot_twoF_cumulative* for a comprehensive list of accepted arguments and their default values.

Unlike the core function, here *savefig=True* is the default, for consistency with other MCMC plotting functions.

get_saved_data_dictionary()

Read the data saved in *self.pickle_path* and return it as a dictionary.

Returns

d – Dictionary containing the data saved in the pickle *self.pickle_path*.

Return type

dict

export_samples_to_disk()

Export MCMC samples into a text file using *numpy.savetxt*.

get_max_twoF()

Get the max. likelihood (loudest) sample and the compute its corresponding detection statistic.

The employed detection statistic depends on *self.BSGL* (i.e. 2F if *self.BSGL* evaluates to *False*, log10BSGL otherwise).

Returns

- **d** (*dict*) – Parameters of the loudest sample.
- **maxtwoF** (*float*) – Detection statistic (2F or log10BSGL) corresponding to the loudest sample.

get_summary_stats()

Returns a dict of point estimates for all production samples.

Point estimates are computed on the MCMC samples using *numpy.mean*, *numpy.std* and *numpy.quantiles* with *q*=[0.005, 0.05, 0.25, 0.5, 0.75, 0.95, 0.995].

Returns

d – Dictionary containing point estimates corresponding to [“mean”, “std”, “lower99”, “lower90”, “lower50”, “median”, “upper50”, “upper90”, “upper99”].

Return type

dict

check_if_samples_are_railing(threshold=0.01)

Returns a boolean estimate of if the samples are railing

Parameters

threshold (*float* [0, 1], *optional*) – Fraction of the uniform prior to test (at upper and lower bound) 0.01

Returns

return_flag – IF true, the samples are railing

Return type

bool

write_par(*method*='median')

Writes a .par of the best-fit params with an estimated std

Parameters**method** (*str*, *optional*) – How to select the *best-fit* params. Available methods: “median”, “mean”, “twoFmax”. 'median'**generate_loudest**()

Use ComputeFstatistic_v2 executable to produce a .loudest file

write_prior_table()

Generate a .tex file of the prior

print_summary()

Prints a summary of the max twoF found to the terminal

get_p_value(*delta_F0*=0, *time_trials*=0)

Gets the p-value for the maximum twoFhat value assuming Gaussian noise

Parameters

- **delta_F0** (*float*, *optional*) – Frequency variation due to a glitch. 0
- **time_trials** (*int*, *optional*) – Number of trials in each glitch + 1. 0

compute_evidence(*make_plots*=False, *write_to_file*=None)

Computes the evidence/marginal likelihood for the model.

Parameters

- **make_plots** (*bool*, *optional*) – Plot the results and save them to os.path.join(self.outdir, self.label + “_beta_lnl.png”) False
- **write_to_file** (*str*, *optional*) – If given, dump evidence and uncertainty estimation to the specified path. None

Returns

- **log10evidence** (*float*) – Estimation of the log10 evidence.
- **log10evidence_err** (*float*) – Log10 uncertainty of the evidence estimation.

static read_evidence_file_to_dict(*evidence_file_name*='Evidences.txt')

Read evidence file and put it into an OrderedDict

An evidence file contains paris (log10evidence, log10evidence_err) for each considered model. These pairs are prepended by the *self.label* variable.**Parameters****evidence_file_name** (*str*, *optional*) – Filename to read. 'Evidences.txt'**Returns****EvidenceDict** – Dictionary with the contents of *evidence_file_name***Return type**

dict

write_evidence_file_from_dict(*EvidenceDict*, *evidence_file_name*)

Write evidence dict to a file

Parameters

- **EvidenceDict** (*dict*) – Dictionary to dump into a file.
- **evidence_file_name** (*str*) – File name to dump dict into.

class pyfstat.mcmc_based_searches.MCMCGlitchSearch(*args, **kwargs)

Bases: [MCMCSearch](#)

MCMC search using the SemiCoherentGlitchSearch

See parent MCMCSearch for a list of all additional parameters, here we list only the additional init parameters of this class.

Parameters

- **nglitch** (*int*, *optional*) – The number of glitches to allow 1
- **dtglitchmin** (*int*, *optional*) – The minimum duration (in seconds) of a segment between two glitches or a glitch and the start/end of the data 86400
- **theta0_idx** (*optional*) – Index (zero-based) of which segment the theta refers to - useful if providing a tight prior on theta to allow the signal to jump too theta (and not just from) 0
- **int** – Index (zero-based) of which segment the theta refers to - useful if providing a tight prior on theta to allow the signal to jump too theta (and not just from)

```
symbol_dictionary = {'Alpha': '$\\alpha$', 'Delta': '$\\delta$', 'F0': '$f$', 'F1': '$\\dot{f}$', 'F2': '$\\ddot{f}$', 'delta_F0': '$\\delta f$', 'delta_F1': '$\\delta \\dot{f}$', 'tglitch': '$t_\\mathrm{glitch}$'}
```

Key, val pairs of the parameters ($F0$, $F1$, ...), to LaTeX math symbols for plots

```
glitch_symbol_dictionary = {'delta_F0': '$\\delta f$', 'delta_F1': '$\\delta \\dot{f}$', 'tglitch': '$t_\\mathrm{glitch}$'}
```

Key, val pairs of glitch parameters ($dF0$, $dF1$, $tglitch$), to LaTeX math symbols for plots. This dictionary included within `self.symbol_dictionary`.

```
unit_dictionary = {'Alpha': 'rad', 'Delta': 'rad', 'F0': 'Hz', 'F1': 'Hz/s', 'F2': 'Hz/s$^2$', 'delta_F0': 'Hz', 'delta_F1': 'Hz/s', 'tglitch': 's'}
```

Key, val pairs of the parameters ($F0$, $F1$, ..., including glitch parameters), and the units (Hz , Hz/s , ...).

```
transform_dictionary = {'tglitch': {'label': '$t^{g}_0$ \\n [d]', 'multiplier': 1.1574074074074073e-05, 'subtractor': 'minStartTime', 'unit': 'day'}}
```

Key, val pairs of the parameters ($F0$, $F1$, ...), where the key is itself a dictionary which can item *multiplier*, *subtractor*, or *unit* by which to transform by and update the units.

plot_cumulative_max(*savefig=True*)

Override MCMCSearch.plot_cumulative_max implementation to deal with the split at glitches.

Parameters

- **savefig** (*boolean*, *optional*) – included for consistency with core plot_twoF_cumulative() function. If true, save the figure in outdir. If false, return an axis object. True

class pyfstat.mcmc_based_searches.MCMCSemiCoherentSearch(*args, **kwargs)

Bases: [MCMCSearch](#)

MCMC search for a signal using the semicoherent ComputeFstat.

Evaluates the semicoherent F-statistic across a parameter space region corresponding to an isolated/binary-modulated CW signal.

See MCMCSearch for a list of additional parameters, here we list only the additional init parameters of this class.

Parameters

nsegs (*int*, *optional*) – The number of segments into which the input datastream will be divided. Coherence time is computed internally as $(\text{maxStartTime} - \text{minStartTime}) / \text{nsegs}$. *None*

class `pyfstat.mcmc_based_searches.MCMCFollowUpSearch(*args, **kwargs)`

Bases: *MCMCSemiCoherentSearch*, *DeprecatedClass*

Hierarchical follow-up procedure

Executes MCMC runs with increasing coherence times in order to follow up a parameter space region. The main idea is to use an MCMC run to identify an interesting parameter space region to then zoom-in said region using a finer “effective resolution” by increasing the coherence time. See Ashton & Prix (PRD 97, 103020, 2018): <https://arxiv.org/abs/1802.05450>

See MCMCSemiCoherentSearch for a list of additional parameters, here we list only the additional init parameters of this class.

Parameters

nsegs (*int*) – The number of segments into which the input datastream will be divided. Coherence time is computed internally as $(\text{maxStartTime} - \text{minStarTime}) / \text{nsegs}$.

run(*run_setup=None*, *proposal_scale_factor=2*, *NstarMax=10*, *Nsegs0=None*, *save_pickle=True*, *export_samples=True*, *save_loudest=True*, *plot_walkers=True*, *walker_plot_args=None*, *log_table=True*, *gen_tex_table=True*, *window=50*)

Run the follow-up with the given run_setup.

See MCMCSearch.run’s docstring for a description of the remainder arguments.

Parameters

- **run_setup** (*optional*) – See *MCMCFollowUpSearch.init_run_setup*. *None*
- **log_table** (*optional*) – See *MCMCFollowUpSearch.init_run_setup*. *True*
- **gen_tex_table** (*optional*) – See *MCMCFollowUpSearch.init_run_setup*. *True*
- **NstarMax** (*optional*) – See *pyfstat.optimal_setup_functions.get_optimal_setup*. *10*
- **Nsegs0** (*optional*) – See *pyfstat.optimal_setup_functions.get_optimal_setup*. *None*

init_run_setup(*run_setup=None*, *NstarMax=1000*, *Nsegs0=None*, *log_table=True*, *gen_tex_table=True*, *setup_only=False*, *no_template_counting=True*)

Initialize the setup of the follow-up run computing the required quantities fro, NstarMax and Nsegs0.

Parameters

- **NstarMax** (*int*, *optional*) – Required parameters to create a new follow-up setup. See *pyfstat.optimal_setup_functions.get_optimal_setup*. *1000*
- **Nsegs0** (*int*, *optional*) – Required parameters to create a new follow-up setup. See *pyfstat.optimal_setup_functions.get_optimal_setup*. *None*
- **run_setup** (*optional*) – If *None*, a new setup will be created from NstarMax and Nsegs0. Use *MCMCFollowUpSearch.read_setup_input_file* to read a previous setup file. *None*
- **log_table** (*bool*, *optional*) – Log follow-up setup using *logger.info* as a table. *True*
- **gen_tex_table** (*bool*, *optional*) – Dump follow-up setup into a text file as a tex table. File is constructed as *os.path.join(self.outdir, self.label + “_run_setup.tex”)*. *True*

Returns

run_setup – List containing the setup of the follow-up run.

Return type

list

read_setup_input_file(*run_setup_input_file*)**class** pyfstat.mcmc_based_searches.**MCMCTransientSearch**(*args, **kwargs)Bases: [*MCMCSearch*](#)

MCMC search for a transient signal using ComputeFstat

Parameters

- **theta_prior** (*dict*) – Dictionary of priors and fixed values for the search parameters. For each parameters (key of the dict), if it is to be held fixed the value should be the constant float, if it is to be searched, the value should be a dictionary of the prior.
- **tref** (*int*) – GPS seconds of the reference time, start time and end time. While tref is requirede, minStartTime and maxStartTime default to None in which case all available data is used.
- **minStartTime** (*int, optional*) – GPS seconds of the reference time, start time and end time. While tref is requirede, minStartTime and maxStartTime default to None in which case all available data is used. **None**
- **maxStartTime** (*int, optional*) – GPS seconds of the reference time, start time and end time. While tref is requirede, minStartTime and maxStartTime default to None in which case all available data is used. **None**
- **label** (*str*) – A label and output directory (optional, default is *data*) to name files
- **outdir** (*str, optional*) – A label and output directory (optional, default is *data*) to name files 'data'
- **sftfilepattern** (*str, optional*) – Pattern to match SFTs using wildcards (*) and ranges [0-9]; mutiple patterns can be given separated by colons. **None**
- **detectors** (*str, optional*) – Two character reference to the detectors to use, specify **None** for no constraint and comma separated strings for multiple references. **None**
- **nsteps** (*list (2,), optional*) – Number of burn-in and production steps to take, [nburn, nprod]. See *pyfstat.MCMCSearch.setup_initialisation()* for details on adding initialisation steps. [100, 100]
- **nwalkers** (*int, optional*) – The number of walkers and temperates to use in the parallel tempered PTSampler. 100
- **ntemps** (*int, optional*) – The number of walkers and temperates to use in the parallel tempered PTSampler. 1
- **log10beta_min** (*float < 0, optional*) – The log₁₀(beta) value. If given, the set of betas passed to PTSampler are generated from *np.logspace(0, log10beta_min, ntemps)* (given in descending order to ptemcee). -5
- **theta_initial** (*dict, array, optional*) – A dictionary of distribution about which to distribute the initial walkers about. **None**
- **rho_hat_max** (*float, optional*) – Upper bound for the SNR scale parameter (required to normalise the Bayes factor) - this needs to be carefully set when using the evidence. 1000
- **binary** (*bool, optional*) – If true, search over binary orbital parameters. **False**
- **BSGL** (*bool, optional*) – If true, use the BSGL statistic. **False**

- **BtSG** (*bool, optional*) – If true, use the transient lnBtSG statistic. (Only for transient searches.) False
- **SSBPrec** (*int, optional*) – SSBPrec (SSB precision) to use when calling `ComputeFstat`. See `core.ComputeFstat`.
- **RngMedWindow** (*int, optional*) – Running-Median window size (number of bins) for `ComputeFstat`. See `core.ComputeFstat`. None
- **minCoverFreq** (*float, optional*) – Minimum and maximum instantaneous frequency which will be covered over the SFT time span as passed to `CreateFstatInput`. See `core.ComputeFstat`. None
- **maxCoverFreq** (*float, optional*) – Minimum and maximum instantaneous frequency which will be covered over the SFT time span as passed to `CreateFstatInput`. See `core.ComputeFstat`. None
- **injectSources** (*dict, optional*) – If given, inject these properties into the SFT files before running the search. See `core.ComputeFstat`. None
- **assumeSqrtSX** (*float or list or str, optional*) – Don't estimate noise-floors, but assume (stationary) per-IFO $\sqrt{SX}$. See `core.ComputeFstat`. None
- **transientWindowType** (*str, optional*) – If 'rect' or 'exp', compute atoms so that a transient (t_0, τ) map can later be computed. ('none' instead of None explicitly calls the transient-window function, but with the full range, for debugging). See `core.ComputeFstat`. Currently only supported for `nsegs=1`. None
- **tCWFstatMapVersion** (*str, optional*) – Choose between standard 'lal' implementation, 'pycuda' for gpu, and some others for devel/debug. 'lal'
- **allowedMismatchFromSFTLength** (*float, optional*) – Maximum allowed mismatch from SFTs being too long [Default: what's hardcoded in `XLALFstatMaximumSFTLength`]. None
- **clean** (*bool, optional*) – If true, ignore existing data and overwrite. Otherwise, re-use existing data if no inconsistencies are found. False

```
symbol_dictionary = {'Alpha': '$\\alpha$', 'Delta': '$\\delta$', 'F0': '$f$',
                    'F1': '$\\dot{f}$$', 'F2': '$\\ddot{f}$$', 'transient_duration': '$\\Delta T$',
                    'transient_tstart': '$t_{\\mathrm{start}}$'}
```

Key, val pairs of the parameters ($F_0, F_1, \dots$), to LaTeX math symbols for plots

```
unit_dictionary = {'Alpha': 'rad', 'Delta': 'rad', 'F0': 'Hz', 'F1': 'Hz/s',
                  'F2': 'Hz/s$^2$', 'transient_duration': 's', 'transient_tstart': 's'}
```

Key, val pairs of the parameters ($F_0, F_1, \dots$, including glitch parameters), and the units ($Hz, Hz/s, \dots$).

```
transform_dictionary = {'transient_duration': {'multiplier':
1.1574074074073e-05, 'symbol': 'Transient duration', 'unit': 'day'},
'transient_tstart': {'label': 'Transient start-time \n days after minStartTime',
'multiplier': 1.1574074074073e-05, 'subtractor': 'minStartTime', 'unit':
'day'}}
```

Key, val pairs of the parameters ($F_0, F_1, \dots$), where the key is itself a dictionary which can item *multiplier*, *subtractor*, or *unit* by which to transform by and update the units.

2.1.10 pyfstat.optimal_setup_functions module

Provides functions to aid in calculating the optimal setup for zoom follow up

`pyfstat.optimal_setup_functions.get_optimal_setup(NstarMax, Nsegs0, tref, minStartTime, maxStartTime, prior, detector_names)`

Using an optimisation step, calculate the optimal setup ladder

The details of the methods are described in Sec Va of arXiv:1802.05450. Here we provide implementation details. All equation numbers refer to arXiv:1802.05450.

Parameters

- **NstarMax** (*float*) – The ratio of the size at the old coherence time to the new coherence time for each step, see Eq. (31). Larger values allow a more rapid “zoom” of the search space at the cost of convergence. Smaller values are more conservative at the cost of additional computing time. The exact choice should be optimized for the problem in hand, but values of 100-1000 are typically okay.
- **Nsegs0** (*int*) – The number of segments for the initial step of the ladder
- **tref** (*int*) – GPS times of the reference, start, and end time.
- **minStartTime** (*int*) – GPS times of the reference, start, and end time.
- **maxStartTime** (*int*) – GPS times of the reference, start, and end time.
- **prior** (*dict*) – Prior dictionary, each item must either be a fixed scalar value, or a uniform prior.
- **detector_names** (*list or str*) – Names of the detectors to use

Returns

nsegs, **Nstar** – Ladder of segment numbers and the corresponding Nstar

Return type

list

`pyfstat.optimal_setup_functions.get_Nstar_estimate(nsegs, tref, minStartTime, maxStartTime, prior, detector_names)`

Returns N* estimated from the super-sky metric

Parameters

- **nsegs** (*int*) – Number of semi-coherent segments
- **tref** (*int*) – Reference time in GPS seconds
- **minStartTime** (*int*) – Minimum and maximum SFT timestamps
- **maxStartTime** (*int*) – Minimum and maximum SFT timestamps
- **prior** (*dict*) – The prior dictionary
- **detector_names** (*array*) – Array of detectors to average over

Returns

Nstar – The estimated approximate number of templates to cover the prior parameter space at a mismatch of unity, assuming the normalised thickness is unity.

Return type

int

2.1.11 pyfstat.snr module

class pyfstat.snr.SignalToNoiseRatio(*, detector_states, noise_weights=None, assumeSqrtSX=None)

Bases: object

Compute the optimal SNR of a CW signal as expected in Gaussian noise.

The definition of SNR (shortcut for “optimal signal-to-noise ratio”) is taken from Eq. (76) of <https://dcc.ligo.org/T0900149-v6/public> and is such that $\langle 2\mathcal{F} \rangle = 4 + \text{SNR}^2$, where $\langle 2\mathcal{F} \rangle$ represents the expected value over noise realizations of twice the F-statistic of a template perfectly matched to an existing signal in the data.

Computing SNR^2 requires two quantities:

- The antenna pattern matrix $\mathcal{M}$, which depends on the sky position $\vec{n}$ and polarization angle ψ and encodes the effect of the detector’s antenna pattern response over the course of the observing run.
- The JKS amplitude parameters ($\mathcal{A}^0, \mathcal{A}^1, \mathcal{A}^2, \mathcal{A}^3$) [JKS1998] which are functions of the CW’s amplitude parameters ($h_0, \cos \iota, \psi, \phi_0$) or, alternatively, ($A_+, A_\times, \psi, \phi_0$).

Parameters

- **detector_states** (*lalpulsar.MultiDetectorStateSeries*) – MultiDetectorStateSeries as produced by DetectorStates. Provides the required information to compute the antenna pattern contribution.
- **noise_weights** (*Union[lalpulsar.MultiNoiseWeights, None]*, *optional*) – Optional, incompatible with *assumeSqrtSX*. Can be computed from SFTs using *SignalToNoiseRatio.from_sfts*. Noise weights to account for a varying noise floor or unequal noise floors in different detectors. *None*
- **assumeSqrtSX** (*float*, *optional*) – Optional, incompatible with *noise_weights*. Single-sided amplitude spectral density (ASD) of the detector noise. This value is used for all detectors, meaning it’s not currently possible to manually specify different noise floors without creating SFT files. (To be improved in the future; developer note: will require SWIG constructor for MultiNoiseWeights.) *None*

Method generated by attrs for class SignalToNoiseRatio.

detector_states: MultiDetectorStateSeries

noise_weights: Optional[MultiNoiseWeights]

assumeSqrtSX: float

classmethod from_sfts(*F0*, *sftfilepath*, *time_offset=None*, *running_median_window=101*, *sft_constraint=None*)

Alternative constructor to retrieve detector states and noise weights from SFT files. This method is based on `DetectorStates.multi_detector_states_from_sfts()`. This is currently the other way in which varying / different noise floors can be used when computing SNRs.

Parameters

- **F0** (*float*) – Central frequency [Hz] to retrieve from the SFT files to compute noise weights.
- **sftfilepath** (*str*) – Path to SFT files in a format compatible with `XLALSFTdataFind`.

- **time_offset** (*float, optional*) – Timestamp offset to retrieve detector states. Defaults to LALSuite’s default of using the central time of an STF (SFT’s timestamp + Tsft/2). *None*
- **running_median_window** (*int, optional*) – Window used to compute the running-median noise floor estimation. Default value is consistent with that used in PredictFstat executable. *101*
- **sft_constraint** (*lalpulsar.SFTConstraint, optional*) – Optional argument to specify further constraints in XLALSFTdataFind. *None*

compute_snr2(*Alpha, Delta, psi, phi, h0=None, cosi=None, aPlus=None, aCross=None*)

Compute the SNR^2 of a CW signal using `XLALComputeOptimalSNR2FromMmunu`. Parameters correspond to the standard ones used to describe a CW (see e.g. Eqs. (16), (26), (30) of <https://dcc.ligo.org/T0900149-v6/public>).

Mind that this function returns *squared* SNR (Eq. (76) of <https://dcc.ligo.org/T0900149-v6/public>), which can be directly related to the expected F-statistic as $\langle 2\mathcal{F} \rangle = 4 + \text{SNR}^2$.

Parameters

- **Alpha** (*float*) – Right ascension (equatorial longitude) of the signal in radians.
- **Delta** (*float*) – Declination (equatorial latitude) of the signal in radians.
- **psi** (*float*) – Polarization angle.
- **h0** (*float, optional*) – Nominal GW amplitude. Must be given together with *cosi* and conflicts with *aPlus* and *aCross*. *None*
- **cosi** (*float, optional*) – Cosine of the source inclination w.r.t. line of sight. Must be given together with *h0* and conflicts with *aPlus* and *aCross*. *None*
- **aPlus** (*float, optional*) – Plus polarization amplitude. Must be given with *aCross* and conflicts with *h0* and *cosi*. *None*
- **aCross** (*float, optional*) – Cross polarization amplitude. Must be given with *aPlus* and conflicts with *h0* and *cosi*. *None*

Returns

SNR² – Squared signal-to-noise ratio of a CW signal consistent with the specified parameters in the specified detector network.

Return type

float

compute_h0_from_snr2(*Alpha, Delta, psi, phi, cosi, snr2*)

Convert the SNR^2 of a CW signal to a corresponding amplitude h_0 given the source orientation. Parameters correspond to the standard ones used to describe a CW (see e.g. Eqs. (16), (26), (30) of <https://dcc.ligo.org/T0900149-v6/public>).

This function returns “inverts” Eq. (77) of <https://dcc.ligo.org/T0900149-v6/public> by computing the overall prefactor on h_0 using `self.compute_snr2(h0=1, ...)`.

Parameters

- **Alpha** (*float*) – Right ascension (equatorial longitude) of the signal in radians.
- **Delta** (*float*) – Declination (equatorial latitude) of the signal in radians.
- **psi** (*float*) – Polarization angle.
- **cosi** (*float*) – Cosine of the source inclination w.r.t. line of sight. Must be given together with *h0* and conflicts with *aPlus* and *aCross*.

- **snr2** (*float*) – Squared signal-to-noise ratio of a CW signal in the specified detector network.

Returns

h0 – Nominal GW amplitude.

Return type

float

compute_twoF(*args, **kwargs)

Compute the expected $2\mathcal{F}$ value of a CW signal from the result of *compute_snr2*.

$$\langle 2\mathcal{F} \rangle = 4 + \text{SNR}^2$$

$$\sigma_{2\mathcal{F}} = \sqrt{8 + 4\text{SNR}^2}$$

Input parameters are passed untouched to *self.compute_snr2*. See corresponding docstring for a list of valid parameters.

Returns

- *expected_2F* – Expected value of a non-central chi-squared distribution with four degrees of freedom and non-centrality parameter given by SNR^2 .
- *stdev_2F* – Standard deviation of a non-central chi-squared distribution with four degrees of freedom and non-centrality parameter given by SNR^2 .

compute_Mmunu(*Alpha*, *Delta*)

Compute Mmunu matrix at a specific sky position using the detector states (and possible noise weights) given at initialization time. If no noise weights were given, unit weights are assumed.

Parameters

- **Alpha** (*float*) – Right ascension (equatorial longitude) of the signal in radians.
- **Delta** (*float*) – Declination (equatorial latitude) of the signal in radians.

Returns

Mmunu – Mmunu matrix encoding the response of the given detector network to a CW at the specified sky position.

Return type

lalpulsar.AntennaPatternMatrix

class pyfstat.snr.DetectorStates

Bases: object

Python interface to XLALGetMultiDetectorStates and XLALGetMultiDetectorStatesFromMultiSFTs.

get_multi_detector_states(*timestamps*, *Tsft*, *detectors=None*, *time_offset=None*)

Parameters

- **timestamps** (*array-like or dict*) – GPS timestamps at which detector states will be retrieved. If array, use the same set of timestamps for all detectors, which must be explicitly given by the user via *detectors*. If dictionary, each key should correspond to a valid detector name to be parsed by XLALParseMultiLALDetector and the associated value should be an array-like set of GPS timestamps for each individual detector.
- **Tsft** (*float*) – Timespan covered by each timestamp. It does not need to coincide with the separation between consecutive timestamps.

- **detectors** (*list[str] or comma-separated string, optional*) – List of detectors to be parsed using `XLALParseMultiLALDetector`. Conflicts with dictionary of *timestamps*, required otherwise. `None`
- **time_offset** (*float, optional*) – Timestamp offset to retrieve detector states. Defaults to LALSuite’s default of using the central time of an STF (SFT’s timestamp + $T_{\text{sft}}/2$). `None`

Returns

multi_detector_states – Resulting multi-detector states produced by `XLALGetMultiDetectorStates`

Return type

`lalpulsar.MultiDetectorStateSeries`

get_multi_detector_states_from_sfts(*sftfilepath, central_frequency, time_offset=None, frequency_wing_bins=1, sft_constraint=None, return_sfts=False*)

Parameters

- **sftfilepath** (*str*) – Path to SFT files in a format compatible with `XLALSFTdataFind`.
- **central_frequency** (*float*) – Frequency [Hz] around which SFT data will be retrieved. This option is only relevant if further information is to be retrieved from the SFTs (i.e. *return_sfts=True*).
- **time_offset** (*float, optional*) – Timestamp offset to retrieve detector states. Defaults to LALSuite’s default of using the central time of an STF (SFT’s timestamp + $T_{\text{sft}}/2$). `None`
- **frequency_wing_bins** (*int, optional*) – Frequency bins around the central frequency to retrieve from SFT data. Bin size is determined using the SFT baseline time as obtained from the catalog. This option is only relevant if further information is to be retrieved from the SFTs (i.e. *return_sfts=True*). 1
- **sft_constraint** (*lalpulsar.SFTConstraint, optional*) – Optional argument to specify further constraints in `XLALSFTdataFind`. `None`
- **return_sfts** (*bool, optional*) – If `True`, also return the loaded SFTs. This is useful to compute further quantities such as noise weights. `False`

Returns

- **multi_detector_states** (*lalpulsar.MultiDetectorStateSeries*) – Resulting multi-detector states produced by `XLALGetMultiDetectorStatesFromMultiSFTs`
- **multi_sfts** (*lalpulsar.MultiSFTVector*) – Only if *return_sfts* is `True`. `MultiSFTVector` produced by `XLALLoadMultiSFTs` along the specified frequency band.

2.1.12 pyfstat.tcw_fstat_map_funcs module

Additional helper functions dealing with transient-CW $F(t_0, \tau)$ maps.

See Prix, Giampanis & Messenger (PRD 84, 023007, 2011): <https://arxiv.org/abs/1104.1704> for the algorithm in general and Keitel & Ashton (CQG 35, 205003, 2018): <https://arxiv.org/abs/1805.05652> for a detailed discussion of the GPU implementation.

```
class pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap(N_t0Range=None, N_tauRange=None,  
                                                    transientFstatMap_t=None, from_file=None)
```

Bases: object

Simplified object class for a $F(t_0, \tau)$ F-stat map.

This is based on LALSuite's `transientFstatMap_t` type, replacing the gsl matrix with a numpy array.

Here, t_0 is a transient start time, τ is a transient duration parameter, and $F(t_0, \tau)$ is the F-statistic (not $2F$)! evaluated for a signal with those parameters (and an implicit window function, which is not stored inside this object).

The 'map' covers a range of different (t_0, τ) pairs.

F_mn

2D array of F values (not $2F$!), with m an index over start-times t_0 , and n an index over duration parameters τ , in steps of dt_0 in $[t_0, t_0+t_0Band]$, and $d\tau$ in $[\tau, \tau+\tauBand]$.

Type

np.ndarray

maxF

Maximum of F (not $2F$!) over the array.

Type

float

t0_ML

Maximum likelihood estimate of the transient start time t_0 .

Type

float

tau_ML

Maximum likelihood estimate of the transient duration τ .

Type

float

lnBtSG

Natural log of the marginalised transient Bayes factor. NOTE: This is always initialised as *nan*, and you have to call `get_lnBtSG()` to get its actual value.

Type

float

The class can be initialized with either a pair of (`N_t0Range`, `N_tauRange`), from a `lalpulsar` object, or reading from a file.

Parameters

- **N_t0Range** (*int, optional*) – Number of t_0 values covered. None
- **N_tauRange** (*int, optional*) – Number of τ values covered. None
- **transientFstatMap_t** (*lalpulsar.transientFstatMap_t, optional*) – pre-allocated matrix from `lalpulsar` to initialise from. None
- **from_file** (*str or None, optional*) – Text file, compatible with `lalpulsar.write_transientFstatMap_to_fp()` format, to load and initialise from. None

read_from_file(file)

Read F_{mn} map from a text file and set all other fields.

Apart from optional header lines (# comments), the format has to be consistent with `lalpulsar.write_transientFstatMap_to_fp()` and the `write_F_mn_to_file()` method of this class itself: with the columns $[t0[s], \tau[s], 2F]$. NOTE that the file is expected to provide 2F, so the values will be halved to obtain F for storage in this class.

Parameters

file (*str*) – Name of the file to load from.

get_maxF_idx()

Gets the 2D-unravellled index pair of the maximum of the F_{mn} map

Returns

idx – The m,n indices of the map entry with maximal F value.

Return type

tuple

get_lnBtSG()

Compute (natural log of the) transient-CW Bayes-factor $B_tSG = P(x|H_{yptS})/P(x|H_{yptG})$.

Here HypG = Gaussian noise hypothesis, HypS = transient-CW signal hypothesis.

B_{tSG} is marginalized over start-time and timescale of transient CW signal, using given type and parameters of transient window range.

This is a python port of the *lalpulsar.ComputeTransientBstat* implementation, replacing *for* loops by numpy operations.

write_F_mn_to_file(tCWfile, windowRange, header=[])

Format a 2D transient-F-stat matrix over $(t0, \tau)$ and write as a text file.

Apart from the optional extra header lines, the format is consistent with `lalpulsar.write_transientFstatMap_to_fp()`, with the columns $[t0[s], \tau[s], 2F]$. NOTE that the output is 2F, not F like stored in this class itself!

Parameters

- **tCWfile** (*str*) – Name of the file to write to.
- **windowRange** (*lalpulsar.transientWindowRange_t*) – A *lalpulsar* structure containing the transient parameters.
- **header** (*list, optional*) – A list of additional header lines to print at the start of the file. []

```
pyfstat.tcw_fstat_map_funcs.fstatmap_versions = {'lal': <function <lambda>>, 'pycuda':  
<function <lambda>>}
```

Dictionary of the actual callable transient F-stat map functions this module supports.

Actual runtime availability depends on the corresponding external modules being available.

```
pyfstat.tcw_fstat_map_funcs.init_transient_fstat_map_features(feature='lal',  
                                                             cudaDeviceName=None)
```

Initialization of available modules (or ‘features’) for computing transient F-stat maps.

Currently, two implementations are supported and checked for through the `_optional_import()` method:

1. *lal*: requires both *lal* and *lalpulsar* packages to be importable.

2. *pycuda*: requires the *pycuda* package to be importable along with its modules *driver*, *gpuarray*, *tools* and *compiler*.

Parameters

- **feature** (*str*, *optional*) – Set the transient F-stat map implementation. 'lal'
- **cudaDeviceName** (*str* or *None*, *optional*) – Request a CUDA device with this name. Partial matches are allowed. *None*

Returns

- **features** (*dict*) – A dictionary of available method names, to match *fstatmap_versions*. Each key's value is set to *True* only if all required modules are importable on this system.
- **gpu_context** (*pycuda.driver.Context* or *None*) – A CUDA device context object, if assigned.

```
pyfstat.tcw_fstat_map_funcs.call_compute_transient_fstat_map(version, features,
                                                             multiFstatAtoms=None,
                                                             windowRange=None, BtSG=False)
```

Call a version of the ComputeTransientFstatMap function.

This checks if the requested *version* is available, and if so, executes the computation of a transient F-statistic map over the *windowRange*.

Parameters

- **version** (*str*) – Name of the method to call (currently supported: 'lal' or 'pycuda').
- **features** (*dict*) – Dictionary of available features, as obtained from *init_transient_fstat_map_features()*.
- **multiFstatAtoms** (*lalpulsar.MultiFstatAtomVector* or *None*, *optional*) – The time-dependent F-stat atoms previously computed by *ComputeFstat*. *None*
- **windowRange** (*lalpulsar.transientWindowRange_t* or *None*, *optional*) – The structure defining the transient parameters. *None*
- **BtSG** (*boolean*, *optional*) – If true, also compute the lnBtSG transient Bayes factor statistic, using the appropriate implementation for each feature, and store it in *FstatMap.lnBtSG*. *False*

Returns

- **FstatMap** (*pyTransientFstatMap* or *lalpulsar.transientFstatMap_t*) – The output of the called function, including the evaluated transient F-statistic map over the *windowRange*.
- **timingFstatMap** (*float*) – Execution time of the called function.

```
pyfstat.tcw_fstat_map_funcs.lalpulsar_compute_transient_fstat_map(multiFstatAtoms,
                                                                    windowRange, BtSG=False)
```

Wrapper for the standard *lalpulsar* function for computing a transient F-statistic map.

See https://lscsoft.docs.ligo.org/lalsuite/lalpulsar/_transient_c_w_utils_8h.html for the wrapped function.

Parameters

- **multiFstatAtoms** (*lalpulsar.MultiFstatAtomVector*) – The time-dependent F-stat atoms previously computed by *ComputeFstat*.
- **windowRange** (*lalpulsar.transientWindowRange_t*) – The structure defining the transient parameters.
- **BtSG** (*boolean*, *optional*) – If true, also compute the lnBtSG transient Bayes factor statistic, using the corresponding *lalpulsar* function, and store it in *FstatMap.lnBtSG*. *False*

Returns

FstatMap – The computed results, see the class definition for details.

Return type

pyTransientFstatMap

`pyfstat.tcw_fstat_map_funcs.reshape_FstatAtomsVector(atomsVector)`

Make a dictionary of ndarrays out of an F-stat atoms ‘vector’ structure.

Parameters

atomsVector (*lalpulsar.FstatAtomVector*) – The atoms in a ‘vector’-like structure: iterating over timestamps as the higher hierarchical level, with a set of ‘atoms’ quantities defined at each timestamp.

Returns

atomsDict – A dictionary with an entry for each quantity, which then is a 1D ndarray over timestamps for that one quantity.

Return type

dict

`pyfstat.tcw_fstat_map_funcs.pycuda_compute_transient_fstat_map(multiFstatAtoms, windowRange, BtSG=False)`

GPU version of computing a transient F-statistic map.

This is based on XLALComputeTransientFstatMap from LALSuite, (C) 2009 Reinhard Prix, licensed under GPL.

The ‘map’ consists of F-statistics evaluated over a range of different (*t0,tau*) pairs (transient start-times and duration parameters).

This is a high-level wrapper function; the actual CUDA computations are performed in one of the functions *pycuda_compute_transient_fstat_map_rect()* or *pycuda_compute_transient_fstat_map_exp()*, depending on the window function defined in *windowRange*.

Parameters

- **multiFstatAtoms** (*lalpulsar.MultiFstatAtomVector*) – The time-dependent F-stat atoms previously computed by *ComputeFstat*.
- **windowRange** (*lalpulsar.transientWindowRange_t*) – The structure defining the transient parameters.
- **BtSG** (*boolean, optional*) – If true, also compute the lnBtSG transient Bayes factor statistic, using a CPU python port of the corresponding *lalpulsar* function, and store it in *FstatMap.lnBtSG*. False

Returns

FstatMap – The computed results, see the class definition for details.

Return type

pyTransientFstatMap

`pyfstat.tcw_fstat_map_funcs.pycuda_compute_transient_fstat_map_rect(atomsInputMatrix, windowRange, tCWparams)`

GPU computation of the transient F-stat map for rectangular windows.

As discussed in Keitel & Ashton (CQG 35, 205003, 2018): <https://arxiv.org/abs/1805.05652> this version only does GPU parallization for the outer loop, keeping the partial sums of the inner loop local to each individual kernel using the ‘memory trick’.

Parameters

- **atomsInputMatrix** (*np.ndarray*) – A 2D array of stacked named columns containing the F-stat atoms.
- **windowRange** (*lalpulsar.transientWindowRange_t*) – The structure defining the transient parameters.
- **tcWparams** (*dict*) – A dictionary of miscellaneous parameters.

Returns

F_mn – A 2D array of the computed transient F-stat map over the $[t_0, \tau]$ range.

Return type

np.ndarray

`pyfstat.tcw_fstat_map_funcs.pycuda_compute_transient_fstat_map_exp(atomsInputMatrix, windowRange, tcWparams)`

GPU computation of the transient F-stat map for exponential windows.

As discussed in Keitel & Ashton (CQG 35, 205003, 2018): <https://arxiv.org/abs/1805.05652> this version does full GPU parallization of both the inner and outer loop.

Parameters

- **atomsInputMatrix** (*np.ndarray*) – A 2D array of stacked named columns containing the F-stat atoms.
- **windowRange** (*lalpulsar.transientWindowRange_t*) – The structure defining the transient parameters.
- **tcWparams** (*dict*) – A dictionary of miscellaneous parameters.

Returns

F_mn – A 2D array of the computed transient F-stat map over the $[t_0, \tau]$ range.

Return type

np.ndarray

2.1.13 Module contents

EXAMPLES

This part of the documentation covers a suite of examples intended to demonstrate the various applications of PyFstat in searching for [continuous gravitational waves \(CWs\)](#), using different grid- and MCMC-based methods, as well as some of the additional helper utilities included in the package.

The examples are simple stand-alone python scripts that can be run after *installing the PyFstat package* and downloading the example itself from these pages (or from [github](#)).

The complete set of examples can be accessed by cloning or downloading the PyFstat repository from [github](#) or [Zenodo](#) (make sure to target the proper PyFstat version). After that, [run_all_examples.py](#), included in the package, can be executed to run all examples one by one. This can be useful to have a general view of all the tools provided by PyFstat.

Alternatively, they can be run interactively through Binder:

Note though that the examples have currently not yet been optimized the interactive notebook experience, meaning for example that you'll need to use the jupyter file browser to find the plots which are saved to files instead of being directly displayed.

See also the in-progress new set of bottom-up [tutorials](#) .

3.1 Grid searches for isolated CW

Fully-coherent F-statistic grid search for isolated CW sources. The examples consist of directed searches (i.e. fixed sky poistions) considering an isolated CW source with 0, 1, and 2 spindown parameters.

3.1.1 Directed grid search: Linear spindown

Search for CW signal including one spindown parameter using a parameter space grid (i.e. no MCMC).

```
8  import os
9
10 import numpy as np
11
12 import pyfstat
13
14 label = "PyFstatExampleGridSearchF0F1"
15 outdir = os.path.join("PyFstat_example_data", label)
16 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
17
18 # Properties of the GW data
```

(continues on next page)

(continued from previous page)

```

19 sqrtSX = 1e-23
20 tstart = 10000000000
21 duration = 10 * 86400
22 tend = tstart + duration
23 tref = 0.5 * (tstart + tend)
24 IF0s = "H1"
25
26 # parameters for injected signals
27 depth = 20
28 inj = {
29     "tref": tref,
30     "F0": 30.0,
31     "F1": -1e-10,
32     "F2": 0,
33     "Alpha": 1.0,
34     "Delta": 1.5,
35     "h0": sqrtSX / depth,
36     "cosi": 0.0,
37 }
38
39 data = pyfstat.Writer(
40     label=label,
41     outdir=outdir,
42     tstart=tstart,
43     duration=duration,
44     sqrtSX=sqrtSX,
45     detectors=IF0s,
46     **inj,
47 )
48 data.make_data()
49
50 m = 0.01
51 dF0 = np.sqrt(12 * m) / (np.pi * duration)
52 dF1 = np.sqrt(180 * m) / (np.pi * duration**2)
53 dF2 = 1e-17
54 N = 100
55 DeltaF0 = N * dF0
56 DeltaF1 = N * dF1
57 F0s = [inj["F0"] - DeltaF0 / 2.0, inj["F0"] + DeltaF0 / 2.0, dF0]
58 F1s = [inj["F1"] - DeltaF1 / 2.0, inj["F1"] + DeltaF1 / 2.0, dF1]
59 F2s = [inj["F2"]]
60 Alphas = [inj["Alpha"]]
61 Deltas = [inj["Delta"]]
62 search = pyfstat.GridSearch(
63     label=label,
64     outdir=outdir,
65     sftfilepath=data.sftfilepath,
66     F0s=F0s,
67     F1s=F1s,
68     F2s=F2s,
69     Alphas=Alphas,
70     Deltas=Deltas,

```

(continues on next page)

(continued from previous page)

```

71     tref=tref,
72     minStartTime=tstart,
73     maxStartTime=tend,
74 )
75 search.run()
76
77 # report details of the maximum point
78 max_dict = search.get_max_twoF()
79 logger.info(
80     "max2F={:.4f} from GridSearch, offsets from injection: {:s}.".format(
81         max_dict["twoF"],
82         ", ".join(
83             [
84                 "{:.4e} in {:s}".format(max_dict[key] - inj[key], key)
85                 for key in max_dict.keys()
86                 if not key == "twoF"
87             ]
88         ),
89     )
90 )
91 search.generate_loudest()
92
93 logger.info("Plotting 2F(F0)...")
94 search.plot_1D(xkey="F0", xlabel="freq [Hz]", ylabel="$2\\mathcal{F}$")
95 logger.info("Plotting 2F(F1)...")
96 search.plot_1D(xkey="F1")
97 logger.info("Plotting 2F(F0,F1)...")
98 search.plot_2D(xkey="F0", ykey="F1", colorbar=True)
99
100 logger.info("Making gridcorner plot...")
101 F0_vals = np.unique(search.data["F0"]) - inj["F0"]
102 F1_vals = np.unique(search.data["F1"]) - inj["F1"]
103 twoF = search.data["twoF"].reshape((len(F0_vals), len(F1_vals)))
104 xyz = [F0_vals, F1_vals]
105 labels = [
106     "$f - f_0$",
107     "$\\dot{f} - \\dot{f}_0$",
108     "$\\widetilde{2\\mathcal{F}}$",
109 ]
110 fig, axes = pyfstat.gridcorner(
111     twoF, xyz, projection="log_mean", labels=labels, whspace=0.1, factor=1.8
112 )
113 fig.savefig(os.path.join(outdir, label + "_projection_matrix.png"))

```

Total running time of the script: (0 minutes 0.000 seconds)

3.1.2 Directed grid search: Monochromatic source

Search for a monochromatic (no spindown) signal using a parameter space grid (i.e. no MCMC).

```

8  import os
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12
13 import pyfstat
14
15 label = "PyFstatExampleGridSearchF0"
16 outdir = os.path.join("PyFstat_example_data", label)
17 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
18
19 # Properties of the GW data
20 sqrtS = "1e-23"
21 IFOs = "H1"
22 # IFOs = "H1,L1"
23 sqrtSX = ",".join(np.repeat(sqrtS, len(IFOs.split(","))))
24 tstart = 10000000000
25 duration = 100 * 86400
26 tend = tstart + duration
27 tref = 0.5 * (tstart + tend)
28
29 # parameters for injected signals
30 depth = 70
31 inj = {
32     "tref": tref,
33     "F0": 30.0,
34     "F1": 0,
35     "F2": 0,
36     "Alpha": 1.0,
37     "Delta": 1.5,
38     "h0": float(sqrtS) / depth,
39     "cosi": 0.0,
40 }
41
42 data = pyfstat.Writer(
43     label=label,
44     outdir=outdir,
45     tstart=tstart,
46     duration=duration,
47     sqrtSX=sqrtSX,
48     detectors=IFOs,
49     **inj,
50 )
51 data.make_data()
52
53 m = 0.001
54 dF0 = np.sqrt(12 * m) / (np.pi * duration)
55 DeltaF0 = 800 * dF0
56 F0s = [inj["F0"] - DeltaF0 / 2.0, inj["F0"] + DeltaF0 / 2.0, dF0]

```

(continues on next page)

(continued from previous page)

```

57 F1s = [inj["F1"]]
58 F2s = [inj["F2"]]
59 Alphas = [inj["Alpha"]]
60 Deltas = [inj["Delta"]]
61 search = pyfstat.GridSearch(
62     label=label,
63     outdir=outdir,
64     sftfilepattern=data.sftfilepath,
65     F0s=F0s,
66     F1s=F1s,
67     F2s=F2s,
68     Alphas=Alphas,
69     Deltas=Deltas,
70     tref=tref,
71     minStartTime=tstart,
72     maxStartTime=tend,
73 )
74 search.run()
75
76 # report details of the maximum point
77 max_dict = search.get_max_twoF()
78 logger.info(
79     "max2F={:.4f} from GridSearch, offsets from injection: {:s}.".format(
80         max_dict["twoF"],
81         ", ".join(
82             [
83                 "{:.4e} in {:s}".format(max_dict[key] - inj[key], key)
84                 for key in max_dict.keys()
85                 if not key == "twoF"
86             ]
87         ),
88     )
89 )
90 search.generate_loudest()
91
92 logger.info("Plotting 2F(F0)...")
93 fig, ax = plt.subplots()
94 frequencies = search.data["F0"]
95 twoF = search.data["twoF"]
96 # mismatch = np.sign(x-inj["F0"])*(duration * np.pi * (x - inj["F0"]))**2 / 12.0
97 ax.plot(frequencies, twoF, "k", lw=1)
98 DeltaF = frequencies - inj["F0"]
99 sinc = np.sin(np.pi * DeltaF * duration) / (np.pi * DeltaF * duration)
100 A = np.abs((np.max(twoF) - 4) * sinc**2 + 4)
101 ax.plot(frequencies, A, "-r", lw=1)
102 ax.set_ylabel("$\\widetilde{{2\\mathcal{F}}}$")
103 ax.set_xlabel("Frequency")
104 ax.set_xlim(F0s[0], F0s[1])
105 dF0 = np.sqrt(12 * 1) / (np.pi * duration)
106 xticks = [inj["F0"] - 10 * dF0, inj["F0"], inj["F0"] + 10 * dF0]
107 ax.set_xticks(xticks)
108 xticklabels = ["$f_0$ {-} 10\\Delta f$", "$f_0$", "$f_0$ {+} 10\\Delta f$"]

```

(continues on next page)

(continued from previous page)

```

109 ax.set_xticklabels(xticklabels)
110 plt.tight_layout()
111 fig.savefig(os.path.join(outdir, label + "_1D.png"), dpi=300)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.1.3 Targeted grid search with line-robust BSGL statistic

Search for a monochromatic (no spindown) signal using a parameter space grid (i.e. no MCMC) and the line-robust BSGL statistic to distinguish an astrophysical signal from an artifact in a single detector.

```

11 import os
12
13 import numpy as np
14
15 import pyfstat
16
17 label = "PyFstatExampleGridSearchBSGL"
18 outdir = os.path.join("PyFstat_example_data", label)
19 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
20
21 F0 = 30.0
22 F1 = 0
23 F2 = 0
24 Alpha = 1.0
25 Delta = 1.5
26
27 # Properties of the GW data - first we make data for two detectors,
28 # both including Gaussian noise and a coherent 'astrophysical' signal.
29 depth = 70
30 sqrtS = "1e-23"
31 h0 = float(sqrtS) / depth
32 cosi = 0
33 IF0s = "H1,L1"
34 sqrtSX = ",".join(np.repeat(sqrtS, len(IF0s.split(","))))
35 tstart = 10000000000
36 duration = 100 * 86400
37 tend = tstart + duration
38 tref = 0.5 * (tstart + tend)
39
40 data = pyfstat.Writer(
41     label=label,
42     outdir=outdir,
43     tref=tref,
44     tstart=tstart,
45     duration=duration,
46     F0=F0,
47     F1=F1,
48     F2=F2,
49     Alpha=Alpha,
50     Delta=Delta,

```

(continues on next page)

(continued from previous page)

```

51     h0=h0,
52     cosi=cosi,
53     sqrtSX=sqrtSX,
54     detectors=IFOs,
55     SFTWindowType="tukey",
56     SFTWindowParam=0.001,
57     Band=1,
58 )
59 data.make_data()
60
61 # Now we add an additional single-detector artifact to H1 only.
62 # For simplicity, this is modelled here as a fully modulated CW-like signal,
63 # just restricted to the single detector.
64 SFTs_H1 = data.sftfilepath.split(";")[0]
65 extra_writer = pyfstat.Writer(
66     label=label,
67     outdir=outdir,
68     tref=tref,
69     F0=F0 + 0.01,
70     F1=F1,
71     F2=F2,
72     Alpha=Alpha,
73     Delta=Delta,
74     h0=10 * h0,
75     cosi=cosi,
76     sqrtSX=0, # don't add yet another set of Gaussian noise
77     noiseSFTs=SFTs_H1,
78 )
79 extra_writer.make_data()
80
81 # use the combined data from both Writers
82 sftfilepattern = os.path.join(outdir, "*" + label + "*sft")
83
84 # set up search parameter ranges
85 dF0 = 0.0001
86 DeltaF0 = 1000 * dF0
87 F0s = [F0 - DeltaF0 / 2.0, F0 + DeltaF0 / 2.0, dF0]
88 F1s = [F1]
89 F2s = [F2]
90 Alphas = [Alpha]
91 Deltas = [Delta]
92
93 # first search: standard F-statistic
94 # This should show a weak peak from the coherent signal
95 # and a larger one from the "line artifact" at higher frequency.
96 searchF = pyfstat.GridSearch(
97     label=label + "_twoF",
98     outdir=outdir,
99     sftfilepattern=sftfilepattern,
100     F0s=F0s,
101     F1s=F1s,
102     F2s=F2s,

```

(continues on next page)

(continued from previous page)

```

103     Alphas=Alphas,
104     Deltas=Deltas,
105     tref=tref,
106     minStartTime=tstart,
107     maxStartTime=tend,
108 )
109 searchF.run()
110
111 logger.info("Plotting 2F(F0)...")
112 searchF.plot_1D(xkey="F0")
113
114 # second search: line-robust statistic BSGL activated
115 searchBSGL = pyfstat.GridSearch(
116     label=label + "_BSGL",
117     outdir=outdir,
118     sftfilepattern=sftfilepattern,
119     F0s=F0s,
120     F1s=F1s,
121     F2s=F2s,
122     Alphas=Alphas,
123     Deltas=Deltas,
124     tref=tref,
125     minStartTime=tstart,
126     maxStartTime=tend,
127     BSGL=True,
128 )
129 searchBSGL.run()
130
131 # The actual output statistic is log10BSGL.
132 # The peak at the higher frequency from the "line artifact" should now
133 # be massively suppressed.
134 logger.info("Plotting log10BSGL(F0)...")
135 searchBSGL.plot_1D(xkey="F0")

```

Total running time of the script: (0 minutes 0.000 seconds)

3.1.4 Directed grid search: Quadratic spindown

Search for CW signal including two spindown parameters using a parameter space grid (i.e. no MCMC).

```

8  import os
9
10 import numpy as np
11
12 import pyfstat
13
14 label = "PyFstatExampleGridSearchF0F1F2"
15 outdir = os.path.join("PyFstat_example_data", label)
16 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
17
18 # Properties of the GW data

```

(continues on next page)

(continued from previous page)

```

19 sqrtSX = 1e-23
20 tstart = 10000000000
21 duration = 10 * 86400
22 tend = tstart + duration
23 tref = 0.5 * (tstart + tend)
24 IF0s = "H1"
25
26 # parameters for injected signals
27 depth = 20
28 inj = {
29     "tref": tref,
30     "F0": 30.0,
31     "F1": -1e-10,
32     "F2": 0,
33     "Alpha": 1.0,
34     "Delta": 1.5,
35     "h0": sqrtSX / depth,
36     "cosi": 0.0,
37 }
38 data = pyfstat.Writer(
39     label=label,
40     outdir=outdir,
41     tstart=tstart,
42     duration=duration,
43     sqrtSX=sqrtSX,
44     detectors=IF0s,
45     **inj,
46 )
47 data.make_data()
48
49 m = 0.01
50 dF0 = np.sqrt(12 * m) / (np.pi * duration)
51 dF1 = np.sqrt(180 * m) / (np.pi * duration**2)
52 dF2 = 1e-17
53 N = 100
54 DeltaF0 = N * dF0
55 DeltaF1 = N * dF1
56 DeltaF2 = N * dF2
57 F0s = [inj["F0"] - DeltaF0 / 2.0, inj["F0"] + DeltaF0 / 2.0, dF0]
58 F1s = [inj["F1"] - DeltaF1 / 2.0, inj["F1"] + DeltaF1 / 2.0, dF1]
59 F2s = [inj["F2"] - DeltaF2 / 2.0, inj["F2"] + DeltaF2 / 2.0, dF2]
60 Alphas = [inj["Alpha"]]
61 Deltas = [inj["Delta"]]
62 search = pyfstat.GridSearch(
63     label=label,
64     outdir=outdir,
65     sftfilepattern=data.sftfilepath,
66     F0s=F0s,
67     F1s=F1s,
68     F2s=F2s,
69     Alphas=Alphas,
70     Deltas=Deltas,

```

(continues on next page)

(continued from previous page)

```

71     tref=tref,
72     minStartTime=tstart,
73     maxStartTime=tend,
74 )
75 search.run()
76
77 # report details of the maximum point
78 max_dict = search.get_max_twoF()
79 logger.info(
80     "max2F={:.4f} from GridSearch, offsets from injection: {:s}.".format(
81         max_dict["twoF"],
82         ", ".join(
83             [
84                 "{:.4e} in {:s}.".format(max_dict[key] - inj[key], key)
85                 for key in max_dict.keys()
86                 if not key == "twoF"
87             ]
88         ),
89     )
90 )
91 search.generate_loudest()
92
93 # FIXME: workaround for matplotlib "Exceeded cell block limit" errors
94 agg_chunksize = 10000
95
96 logger.info("Plotting 2F(F0)...")
97 search.plot_1D(
98     xkey="F0", xlabel="freq [Hz]", ylabel="$2\\mathcal{F}$", agg_chunksize=agg_chunksize
99 )
100 logger.info("Plotting 2F(F1)...")
101 search.plot_1D(xkey="F1", agg_chunksize=agg_chunksize)
102 logger.info("Plotting 2F(F2)...")
103 search.plot_1D(xkey="F2", agg_chunksize=agg_chunksize)
104 logger.info("Plotting 2F(Alpha)...")
105 search.plot_1D(xkey="Alpha", agg_chunksize=agg_chunksize)
106 logger.info("Plotting 2F(Delta)...")
107 search.plot_1D(xkey="Delta", agg_chunksize=agg_chunksize)
108 # 2D plots will currently not work for >2 non-trivial (gridded) search dimensions
109 # search.plot_2D(xkey="F0", ykey="F1", colorbar=True)
110 # search.plot_2D(xkey="F0", ykey="F2", colorbar=True)
111 # search.plot_2D(xkey="F1", ykey="F2", colorbar=True)
112
113 logger.info("Making gridcorner plot...")
114 F0_vals = np.unique(search.data["F0"]) - inj["F0"]
115 F1_vals = np.unique(search.data["F1"]) - inj["F1"]
116 F2_vals = np.unique(search.data["F2"]) - inj["F2"]
117 twoF = search.data["twoF"].reshape((len(F0_vals), len(F1_vals), len(F2_vals)))
118 xyz = [F0_vals, F1_vals, F2_vals]
119 labels = [
120     "$f - f_0$",
121     "$\\dot{f} - \\dot{f}_0$",
122     "$\\ddot{f} - \\ddot{f}_0$",

```

(continues on next page)

(continued from previous page)

```

123     "$\\widetilde{2}\\mathcal{F}}$",
124 ]
125 fig, axes = pyfstat.gridcorner(
126     twoF, xyz, projection="log_mean", labels=labels, whspace=0.1, factor=1.8
127 )
128 fig.savefig(os.path.join(outdir, label + "_projection_matrix.png"))

```

Total running time of the script: (0 minutes 0.000 seconds)

3.2 MCMC searches for isolated CW signals

Application of MCMC coherent and semicoherent F-statistic algorithms to the search of isolated CW signals.

3.2.1 MCMC search: Semicoherent F-statistic with initialisation

Directed MCMC search for an isolated CW signal using the fully-coherent F-statistic. Prior to the burn-in stage, walkers are initialized with a certain scattering factor.

```

9  import os
10
11  import numpy as np
12
13  import pyfstat
14
15  label = "PyFstatExampleMCMCSearchUsingInitialisation"
16  outdir = os.path.join("PyFstat_example_data", label)
17  logger = pyfstat.set_up_logger(label=label, outdir=outdir)
18
19  # Properties of the GW data
20  data_parameters = {
21      "sqrtSX": 1e-23,
22      "tstart": 10000000000,
23      "duration": 100 * 86400,
24      "detectors": "H1",
25  }
26  tend = data_parameters["tstart"] + data_parameters["duration"]
27  mid_time = 0.5 * (data_parameters["tstart"] + tend)
28
29  # Properties of the signal
30  depth = 10
31  signal_parameters = {
32      "F0": 30.0,
33      "F1": -1e-10,
34      "F2": 0,
35      "Alpha": np.radians(83.6292),
36      "Delta": np.radians(22.0144),
37      "tref": mid_time,
38      "h0": data_parameters["sqrtSX"] / depth,
39      "cosi": 1.0,
40  }

```

(continues on next page)

(continued from previous page)

```

41 data = pyfstat.Writer(
42     label=label, outdir=outdir, **data_parameters, **signal_parameters
43 )
44 data.make_data()
45
46
47 # The predicted twoF (expectation over noise realizations) can be accessed by
48 twoF = data.predict_fstat()
49 logger.info("Predicted twoF value: {}\n".format(twoF))
50
51 DeltaF0 = 1e-7
52 DeltaF1 = 1e-13
53 VF0 = (np.pi * data_parameters["duration"] * DeltaF0) ** 2 / 3.0
54 VF1 = (np.pi * data_parameters["duration"] ** 2 * DeltaF1) ** 2 * 4 / 45.0
55 logger.info("\nV={:1.2e}, VF0={:1.2e}, VF1={:1.2e}\n".format(VF0 * VF1, VF0, VF1))
56
57 theta_prior = {
58     "F0": {
59         "type": "unif",
60         "lower": signal_parameters["F0"] - DeltaF0 / 2.0,
61         "upper": signal_parameters["F0"] + DeltaF0 / 2.0,
62     },
63     "F1": {
64         "type": "unif",
65         "lower": signal_parameters["F1"] - DeltaF1 / 2.0,
66         "upper": signal_parameters["F1"] + DeltaF1 / 2.0,
67     },
68 }
69 for key in "F2", "Alpha", "Delta":
70     theta_prior[key] = signal_parameters[key]
71
72 ntemps = 1
73 log10beta_min = -1
74 nwalkers = 100
75 nsteps = [100, 100]
76
77 mcmc = pyfstat.MCMCSearch(
78     label=label,
79     outdir=outdir,
80     sftfilepattern=data.sftfilepath,
81     theta_prior=theta_prior,
82     tref=mid_time,
83     minStartTime=data_parameters["tstart"],
84     maxStartTime=tend,
85     nsteps=nsteps,
86     nwalkers=nwalkers,
87     ntemps=ntemps,
88     log10beta_min=log10beta_min,
89 )
90 mcmc.setup_initialisation(100, scatter_val=1e-10)
91 mcmc.run(
92     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters}

```

(continues on next page)

(continued from previous page)

```

93 )
94 mcmc.print_summary()
95 mcmc.plot_corner(add_prior=True, truths=signal_parameters)
96 mcmc.plot_prior_posterior(injection_parameters=signal_parameters)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.2.2 MCMC search: Semicohherent F-statistic

Directed MCMC search for an isolated CW signal using the semicoherent F-statistic.

```

8  import os
9
10 import numpy as np
11
12 import pyfstat
13
14 label = "PyFstatExampleSemiCoherentMCMCSearch"
15 outdir = os.path.join("PyFstat_example_data", label)
16 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
17
18 # Properties of the GW data
19 data_parameters = {
20     "sqrtSX": 1e-23,
21     "tstart": 10000000000,
22     "duration": 100 * 86400,
23     "detectors": "H1",
24 }
25 tend = data_parameters["tstart"] + data_parameters["duration"]
26 mid_time = 0.5 * (data_parameters["tstart"] + tend)
27
28 # Properties of the signal
29 depth = 10
30 signal_parameters = {
31     "F0": 30.0,
32     "F1": -1e-10,
33     "F2": 0,
34     "Alpha": np.radians(83.6292),
35     "Delta": np.radians(22.0144),
36     "tref": mid_time,
37     "h0": data_parameters["sqrtSX"] / depth,
38     "cosi": 1.0,
39 }
40
41 data = pyfstat.Writer(
42     label=label, outdir=outdir, **data_parameters, **signal_parameters
43 )
44 data.make_data()
45
46 # The predicted twoF (expectation over noise realizations) can be accessed by
47 twoF = data.predict_fstat()

```

(continues on next page)

(continued from previous page)

```

48 logger.info("Predicted twoF value: {}".format(twoF))
49
50 DeltaF0 = 1e-7
51 DeltaF1 = 1e-13
52 VF0 = (np.pi * data_parameters["duration"] * DeltaF0) ** 2 / 3.0
53 VF1 = (np.pi * data_parameters["duration"] ** 2 * DeltaF1) ** 2 * 4 / 45.0
54 logger.info("\nV={:1.2e}, VF0={:1.2e}, VF1={:1.2e}\n".format(VF0 * VF1, VF0, VF1))
55
56 theta_prior = {
57     "F0": {
58         "type": "unif",
59         "lower": signal_parameters["F0"] - DeltaF0 / 2.0,
60         "upper": signal_parameters["F0"] + DeltaF0 / 2.0,
61     },
62     "F1": {
63         "type": "unif",
64         "lower": signal_parameters["F1"] - DeltaF1 / 2.0,
65         "upper": signal_parameters["F1"] + DeltaF1 / 2.0,
66     },
67 }
68 for key in "F2", "Alpha", "Delta":
69     theta_prior[key] = signal_parameters[key]
70
71 ntemps = 1
72 log10beta_min = -1
73 nwalkers = 100
74 nsteps = [300, 300]
75
76 mcmc = pyfstat.MCMCSemiCoherentSearch(
77     label=label,
78     outdir=outdir,
79     nsegs=10,
80     sftfilepattern=data.sftfilepath,
81     theta_prior=theta_prior,
82     tref=mid_time,
83     minStartTime=data_parameters["tstart"],
84     maxStartTime=tend,
85     nsteps=nsteps,
86     nwalkers=nwalkers,
87     ntemps=ntemps,
88     log10beta_min=log10beta_min,
89 )
90 mcmc.transform_dictionary = dict(
91     F0=dict(subtractor=signal_parameters["F0"], symbol="$f-f^{\mathrm{s}}$"),
92     F1=dict(
93         subtractor=signal_parameters["F1"], symbol="$\dot{f}-\dot{f}^{\mathrm{s}}$")
94 ),
95 )
96 mcmc.run(
97     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters}
98 )
99 mcmc.print_summary()

```

(continues on next page)

(continued from previous page)

```

100 mcmc.plot_corner(add_prior=True, truths=signal_parameters)
101 mcmc.plot_prior_posterior(injection_parameters=signal_parameters)
102 mcmc.plot_chainconsumer(truth=signal_parameters)
103 mcmc.plot_cumulative_max(
104     savefig=True,
105     custom_ax_kwargs={"title": "Cumulative 2F for the best MCMC candidate"},
106 )

```

Total running time of the script: (0 minutes 0.000 seconds)

3.2.3 MCMC search: Fully coherent F-statistic

Directed MCMC search for an isolated CW signal using the fully coherent F-statistic.

```

9  import os
10
11  import numpy as np
12
13  import pyfstat
14  from pyfstat.utils import get_predict_fstat_parameters_from_dict
15
16  label = "PyFstatExampleFullyCoherentMCMCSearch"
17  outdir = os.path.join("PyFstat_example_data", label)
18  logger = pyfstat.set_up_logger(label=label, outdir=outdir)
19
20  # Properties of the GW data
21  data_parameters = {
22      "sqrtSX": 1e-23,
23      "tstart": 10000000000,
24      "duration": 100 * 86400,
25      "detectors": "H1",
26  }
27  tend = data_parameters["tstart"] + data_parameters["duration"]
28  mid_time = 0.5 * (data_parameters["tstart"] + tend)
29
30  # Properties of the signal
31  depth = 10
32  signal_parameters = {
33      "F0": 30.0,
34      "F1": -1e-10,
35      "F2": 0,
36      "Alpha": np.radians(83.6292),
37      "Delta": np.radians(22.0144),
38      "tref": mid_time,
39      "h0": data_parameters["sqrtSX"] / depth,
40      "cosi": 1.0,
41  }
42
43  data = pyfstat.Writer(
44      label=label, outdir=outdir, **data_parameters, **signal_parameters
45  )

```

(continues on next page)

(continued from previous page)

```

46 data.make_data()
47
48 # The predicted twoF (expectation over noise realizations) can be accessed by
49 twoF = data.predict_fstat()
50 logger.info("Predicted twoF value: {}".format(twoF))
51
52 DeltaF0 = 1e-7
53 DeltaF1 = 1e-13
54 VF0 = (np.pi * data_parameters["duration"] * DeltaF0) ** 2 / 3.0
55 VF1 = (np.pi * data_parameters["duration"] ** 2 * DeltaF1) ** 2 * 4 / 45.0
56 logger.info("\nV={}, VF0={}, VF1={}\n".format(VF0 * VF1, VF0, VF1))
57
58 theta_prior = {
59     "F0": {
60         "type": "unif",
61         "lower": signal_parameters["F0"] - DeltaF0 / 2.0,
62         "upper": signal_parameters["F0"] + DeltaF0 / 2.0,
63     },
64     "F1": {
65         "type": "unif",
66         "lower": signal_parameters["F1"] - DeltaF1 / 2.0,
67         "upper": signal_parameters["F1"] + DeltaF1 / 2.0,
68     },
69 }
70 for key in "F2", "Alpha", "Delta":
71     theta_prior[key] = signal_parameters[key]
72
73 ntemps = 2
74 log10beta_min = -0.5
75 nwalkers = 100
76 nsteps = [300, 300]
77
78 mcmc = pyfstat.MCMCSearch(
79     label=label,
80     outdir=outdir,
81     sftfilepattern=data.sftfilepath,
82     theta_prior=theta_prior,
83     tref=mid_time,
84     minStartTime=data_parameters["tstart"],
85     maxStartTime=tend,
86     nsteps=nsteps,
87     nwalkers=nwalkers,
88     ntemps=ntemps,
89     log10beta_min=log10beta_min,
90 )
91 mcmc.transform_dictionary = dict(
92     F0=dict(subtractor=signal_parameters["F0"], symbol="$f-f^{\mathrm{s}}$"),
93     F1=dict(
94         subtractor=signal_parameters["F1"], symbol="$\dot{f}-\dot{f}^{\mathrm{s}}$"
95     ),
96 )
97 mcmc.run(

```

(continues on next page)

(continued from previous page)

```

98     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters}
99 )
100 mcmc.print_summary()
101 mcmc.plot_corner(add_prior=True, truths=signal_parameters)
102 mcmc.plot_prior_posterior(injection_parameters=signal_parameters)
103
104 mcmc.generate_loudest()
105
106 # plot cumulative 2F, first building a dict as required for PredictFStat
107 d, maxtwoF = mcmc.get_max_twoF()
108 for key, val in mcmc.theta_prior.items():
109     if key not in d:
110         d[key] = val
111 d["h0"] = data.h0
112 d["cosi"] = data.cosi
113 d["psi"] = data.psi
114 PFS_input = get_predict_fstat_parameters_from_dict(d)
115 mcmc.plot_cumulative_max(PFS_input=PFS_input)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.2.4 MCMC search with fully coherent BSGL statistic

Targeted MCMC search for an isolated CW signal using the fully coherent line-robust BSGL-statistic.

```

9  import os
10
11  import numpy as np
12
13  import pyfstat
14
15  label = "PyFstatExampleFullyCoherentMCMCSearchBSGL"
16  outdir = os.path.join("PyFstat_example_data", label)
17  logger = pyfstat.set_up_logger(label=label, outdir=outdir)
18
19  # Properties of the GW data - first we make data for two detectors,
20  # both including Gaussian noise and a coherent 'astrophysical' signal.
21  data_parameters = {
22      "sqrtSX": 1e-23,
23      "tstart": 10000000000,
24      "duration": 100 * 86400,
25      "detectors": "H1,L1",
26      "SFTWindowType": "tukey",
27      "SFTWindowParam": 0.001,
28  }
29  tend = data_parameters["tstart"] + data_parameters["duration"]
30  mid_time = 0.5 * (data_parameters["tstart"] + tend)
31
32  # Properties of the signal
33  depth = 10
34  signal_parameters = {

```

(continues on next page)

(continued from previous page)

```

35     "F0": 30.0,
36     "F1": -1e-10,
37     "F2": 0,
38     "Alpha": np.radians(83.6292),
39     "Delta": np.radians(22.0144),
40     "tref": mid_time,
41     "h0": data_parameters["sqrtSX"] / depth,
42     "cosi": 1.0,
43 }
44
45 data = pyfstat.Writer(
46     label=label, outdir=outdir, **data_parameters, **signal_parameters
47 )
48 data.make_data()
49
50 # The predicted twoF (expectation over noise realizations) can be accessed by
51 twoF = data.predict_fstat()
52 logger.info("Predicted twoF value: {}\n".format(twoF))
53
54 # Now we add an additional single-detector artifact to H1 only.
55 # For simplicity, this is modelled here as a fully modulated CW-like signal,
56 # just restricted to the single detector.
57 SFTs_H1 = data.sftfilepath.split(";")[0]
58 data_parameters_line = data_parameters.copy()
59 signal_parameters_line = signal_parameters.copy()
60 data_parameters_line["detectors"] = "H1"
61 data_parameters_line["sqrtSX"] = 0 # don't add yet another set of Gaussian noise
62 signal_parameters_line["F0"] += 1e-6
63 signal_parameters_line["h0"] *= 10.0
64 extra_writer = pyfstat.Writer(
65     label=label,
66     outdir=outdir,
67     **data_parameters_line,
68     **signal_parameters_line,
69     noiseSFTs=SFTs_H1,
70 )
71 extra_writer.make_data()
72
73 # use the combined data from both Writers
74 sftfilepattern = os.path.join(outdir, "*" + label + "*sft")
75
76 # MCMC prior ranges
77 DeltaF0 = 1e-5
78 DeltaF1 = 1e-13
79 theta_prior = {
80     "F0": {
81         "type": "unif",
82         "lower": signal_parameters["F0"] - DeltaF0 / 2.0,
83         "upper": signal_parameters["F0"] + DeltaF0 / 2.0,
84     },
85     "F1": {
86         "type": "unif",

```

(continues on next page)

(continued from previous page)

```

87         "lower": signal_parameters["F1"] - DeltaF1 / 2.0,
88         "upper": signal_parameters["F1"] + DeltaF1 / 2.0,
89     },
90 }
91 for key in "F2", "Alpha", "Delta":
92     theta_prior[key] = signal_parameters[key]
93
94 # MCMC sampler settings - relatively cheap setup, may not converge perfectly
95 ntemps = 2
96 log10beta_min = -0.5
97 nwalkers = 50
98 nsteps = [100, 100]
99
100 # we'll want to plot results relative to the injection parameters
101 transform_dict = dict(
102     F0=dict(subtractor=signal_parameters["F0"], symbol="$f-f^{\mathrm{s}}$"),
103     F1=dict(
104         subtractor=signal_parameters["F1"], symbol="$\dot{f}-\dot{f}^{\mathrm{s}}$")
105 ),
106 )
107
108 # first search: standard F-statistic
109 # This should show a weak peak from the coherent signal
110 # and a larger one from the "line artifact" at higher frequency.
111 mcmc_F = pyfstat.MCMCSearch(
112     label=label + "_twoF",
113     outdir=outdir,
114     sftfilepattern=sftfilepattern,
115     theta_prior=theta_prior,
116     tref=mid_time,
117     minStartTime=data_parameters["tstart"],
118     maxStartTime=tend,
119     nsteps=nsteps,
120     nwalkers=nwalkers,
121     ntemps=ntemps,
122     log10beta_min=log10beta_min,
123     BSGl=False,
124 )
125 mcmc_F.transform_dictionary = transform_dict
126 mcmc_F.run(
127     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters}
128 )
129 mcmc_F.print_summary()
130 mcmc_F.plot_corner(add_prior=True, truths=signal_parameters)
131 mcmc_F.plot_prior_posterior(injection_parameters=signal_parameters)
132
133 # second search: line-robust statistic BSGl activated
134 mcmc_F = pyfstat.MCMCSearch(
135     label=label + "_BSGL",
136     outdir=outdir,
137     sftfilepattern=sftfilepattern,
138     theta_prior=theta_prior,

```

(continues on next page)

(continued from previous page)

```

139     tref=mid_time,
140     minStartTime=data_parameters["tstart"],
141     maxStartTime=tend,
142     nsteps=nsteps,
143     nwalkers=nwalkers,
144     ntemps=ntemps,
145     log10beta_min=log10beta_min,
146     BSG=True,
147 )
148 mcmc_F.transform_dictionary = transform_dict
149 mcmc_F.run(
150     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters}
151 )
152 mcmc_F.print_summary()
153 mcmc_F.plot_corner(add_prior=True, truths=signal_parameters)
154 mcmc_F.plot_prior_posterior(injection_parameters=signal_parameters)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.3 Comparison between MCMC and Grid searches

Run an MCMC and a Grid search on the same data to perform a consistency check.

3.3.1 MCMC search v.s. grid search

An example to compare MCMCSearch and GridSearch on the same data.

```

8  import os
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12
13 import pyfstat
14
15 # flip this switch for a more expensive 4D (F0,F1,Alpha,Delta) run
16 # instead of just (F0,F1)
17 # (still only a few minutes on current laptops)
18 sky = False
19
20 label = "PyFstatExampleSimpleMCMCvsGridComparison"
21 outdir = os.path.join("PyFstat_example_data", label)
22 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
23 if sky:
24     outdir += "AlphaDelta"
25
26 # parameters for the data set to generate
27 tstart = 10000000000
28 duration = 30 * 86400
29 Tsft = 1800
30 detectors = "H1,L1"

```

(continues on next page)

(continued from previous page)

```

31 sqrtSX = 1e-22
32
33 # parameters for injected signals
34 inj = {
35     "tref": tstart,
36     "F0": 30.0,
37     "F1": -1e-10,
38     "F2": 0,
39     "Alpha": 0.5,
40     "Delta": 1,
41     "h0": 0.05 * sqrtSX,
42     "cosi": 1.0,
43 }
44
45 # latex-formatted plotting labels
46 labels = {
47     "F0": "$f$ [Hz]",
48     "F1": "$\\dot{f}$ [Hz/s]",
49     "2F": "$2\\mathcal{F}$",
50     "Alpha": "$\\alpha$",
51     "Delta": "$\\delta$",
52 }
53 labels["max2F"] = "$\\max\\$, $" + labels["2F"]
54
55
56 def plot_grid_vs_samples(grid_res, mcmc_res, xkey, ykey):
57     """local plotting function to avoid code duplication in the 4D case"""
58     plt.plot(grid_res[xkey], grid_res[ykey], ".", label="grid")
59     plt.plot(mcmc_res[xkey], mcmc_res[ykey], ".", label="mcmc")
60     plt.plot(inj[xkey], inj[ykey], "*k", label="injection")
61     grid_maxidx = np.argmax(grid_res["twoF"])
62     mcmc_maxidx = np.argmax(mcmc_res["twoF"])
63     plt.plot(
64         grid_res[xkey][grid_maxidx],
65         grid_res[ykey][grid_maxidx],
66         "+g",
67         label=labels["max2F"] + "(grid)",
68     )
69     plt.plot(
70         mcmc_res[xkey][mcmc_maxidx],
71         mcmc_res[ykey][mcmc_maxidx],
72         "xm",
73         label=labels["max2F"] + "(mcmc)",
74     )
75     plt.xlabel(labels[xkey])
76     plt.ylabel(labels[ykey])
77     plt.legend()
78     plotfilename_base = os.path.join(outdir, "grid_vs_mcmc_{:s}{:s}".format(xkey, ykey))
79     plt.savefig(plotfilename_base + ".png")
80     if xkey == "F0" and ykey == "F1":
81         plt.xlim(zoom[xkey])
82         plt.ylim(zoom[ykey])

```

(continues on next page)

(continued from previous page)

```

83     plt.savefig(plotfilename_base + "_zoom.png")
84     plt.close()
85
86
87     def plot_2F_scatter(res, label, xkey, ykey):
88         """local plotting function to avoid code duplication in the 4D case"""
89         markersize = 3 if label == "grid" else 1
90         sc = plt.scatter(res[xkey], res[ykey], c=res["twoF"], s=markersize)
91         cb = plt.colorbar(sc)
92         plt.xlabel(labels[xkey])
93         plt.ylabel(labels[ykey])
94         cb.set_label(labels["2F"])
95         plt.title(label)
96         plt.plot(inj[xkey], inj[ykey], "*k", label="injection")
97         maxidx = np.argmax(res["twoF"])
98         plt.plot(
99             res[xkey][maxidx],
100             res[ykey][maxidx],
101             "+r",
102             label=labels["max2F"],
103         )
104         plt.legend()
105         plotfilename_base = os.path.join(
106             outdir, "{:s}_{:s}{:s}_2F".format(label, xkey, ykey)
107         )
108         plt.xlim([min(res[xkey]), max(res[xkey])])
109         plt.ylim([min(res[ykey]), max(res[ykey])])
110         plt.savefig(plotfilename_base + ".png")
111         plt.close()
112
113
114     if __name__ == "__main__":
115         logger.info("Generating SFTs with injected signal...")
116         writer = pyfstat.Writer(
117             label=label + "SimulatedSignal",
118             outdir=outdir,
119             tstart=tstart,
120             duration=duration,
121             detectors=detectors,
122             sqrtSX=sqrtSX,
123             Tsft=Tsft,
124             **inj,
125             Band=1, # default band estimation would be too narrow for a wide grid/prior
126         )
127         writer.make_data()
128
129         # set up square search grid with fixed (F0,F1) mismatch
130         # and (optionally) some ad-hoc sky coverage
131         m = 0.001
132         dF0 = np.sqrt(12 * m) / (np.pi * duration)
133         dF1 = np.sqrt(180 * m) / (np.pi * duration**2)
134         DeltaF0 = 500 * dF0

```

(continues on next page)

(continued from previous page)

```

135 DeltaF1 = 200 * dF1
136 if sky:
137     # cover less range to keep runtime down
138     DeltaF0 /= 10
139     DeltaF1 /= 10
140 F0s = [inj["F0"] - DeltaF0 / 2.0, inj["F0"] + DeltaF0 / 2.0, dF0]
141 F1s = [inj["F1"] - DeltaF1 / 2.0, inj["F1"] + DeltaF1 / 2.0, dF1]
142 F2s = [inj["F2"]]
143 search_keys = ["F0", "F1"] # only the ones that aren't 0-width
144 if sky:
145     dSky = 0.01 # rather coarse to keep runtime down
146     DeltaSky = 10 * dSky
147     Alphas = [inj["Alpha"] - DeltaSky / 2.0, inj["Alpha"] + DeltaSky / 2.0, dSky]
148     Deltas = [inj["Delta"] - DeltaSky / 2.0, inj["Delta"] + DeltaSky / 2.0, dSky]
149     search_keys += ["Alpha", "Delta"]
150 else:
151     Alphas = [inj["Alpha"]]
152     Deltas = [inj["Delta"]]
153 search_keys_label = "".join(search_keys)
154
155 logger.info("Performing GridSearch...")
156 gridsearch = pyfstat.GridSearch(
157     label="GridSearch" + search_keys_label,
158     outdir=outdir,
159     sftfilepattern=writer.sftfilepath,
160     F0s=F0s,
161     F1s=F1s,
162     F2s=F2s,
163     Alphas=Alphas,
164     Deltas=Deltas,
165     tref=inj["tref"],
166 )
167 gridsearch.run()
168 gridsearch.print_max_twoF()
169 gridsearch.generate_loudest()
170
171 # do some plots of the GridSearch results
172 if not sky: # this plotter can't currently deal with too large result arrays
173     logger.info("Plotting 1D 2F distributions...")
174     for key in search_keys:
175         gridsearch.plot_1D(xkey=key, xlabel=labels[key], ylabel=labels["2F"])
176
177 logger.info("Making GridSearch {s} corner plot...".format("-".join(search_keys)))
178 vals = [np.unique(gridsearch.data[key]) - inj[key] for key in search_keys]
179 twoF = gridsearch.data["twoF"].reshape([len(kval) for kval in vals])
180 corner_labels = [
181     "$f - f_0$ [Hz]",
182     "$\\dot{f} - \\dot{f}_0$ [Hz/s]",
183 ]
184 if sky:
185     corner_labels.append("$\\alpha - \\alpha_0$")
186     corner_labels.append("$\\delta - \\delta_0$")

```

(continues on next page)

(continued from previous page)

```

187 corner_labels.append(labels["2F"])
188 gridcorner_fig, gridcorner_axes = pyfstat.gridcorner(
189     twoF, vals, projection="log_mean", labels=corner_labels, whspace=0.1, factor=1.8
190 )
191 gridcorner_fig.savefig(os.path.join(outdir, gridsearch.label + "_corner.png"))
192 plt.close(gridcorner_fig)
193
194 logger.info("Performing MCMCSearch...")
195 # set up priors in F0 and F1 (over)covering the grid ranges
196 if sky: # MCMC will still be fast in 4D with wider range than grid
197     DeltaF0 *= 50
198     DeltaF1 *= 50
199     theta_prior = {
200         "F0": {
201             "type": "unif",
202             "lower": inj["F0"] - DeltaF0 / 2.0,
203             "upper": inj["F0"] + DeltaF0 / 2.0,
204         },
205         "F1": {
206             "type": "unif",
207             "lower": inj["F1"] - DeltaF1 / 2.0,
208             "upper": inj["F1"] + DeltaF1 / 2.0,
209         },
210         "F2": inj["F2"],
211     }
212 if sky:
213     # also implicitly covering twice the grid range here
214     theta_prior["Alpha"] = {
215         "type": "unif",
216         "lower": inj["Alpha"] - DeltaSky,
217         "upper": inj["Alpha"] + DeltaSky,
218     }
219     theta_prior["Delta"] = {
220         "type": "unif",
221         "lower": inj["Delta"] - DeltaSky,
222         "upper": inj["Delta"] + DeltaSky,
223     }
224 else:
225     theta_prior["Alpha"] = inj["Alpha"]
226     theta_prior["Delta"] = inj["Delta"]
227 # ptemcee sampler settings - in a real application we might want higher values
228 ntemps = 2
229 log10beta_min = -1
230 nwalkers = 100
231 nsteps = [200, 200] # [burnin, production]
232
233 mcmcsearch = pyfstat.MCMCSearch(
234     label="MCMCSearch" + search_keys_label,
235     outdir=outdir,
236     sftfilepattern=writer.sftfilepath,
237     theta_prior=theta_prior,
238     tref=inj["tref"],

```

(continues on next page)

(continued from previous page)

```

239     nsteps=nsteps,
240     nwalkers=nwalkers,
241     ntemps=ntemps,
242     log10beta_min=log10beta_min,
243 )
244 # walker plot is generated during main run of the search class
245 mcmcsearch.run(
246     walker_plot_args={"plot_det_stat": True, "injection_parameters": inj}
247 )
248 mcmcsearch.print_summary()
249
250 # call some built-in plotting methods
251 # these can all highlight the injection parameters, too
252 logger.info("Making MCMCSearch {:s} corner plot...".format("-".join(search_keys)))
253 mcmcsearch.plot_corner(truths=inj)
254 logger.info("Making MCMCSearch prior-posterior comparison plot...")
255 mcmcsearch.plot_prior_posterior(injection_parameters=inj)
256
257 # NOTE: everything below here is just custom commandline output and plotting
258 # for this particular example, which uses the PyFstat outputs,
259 # but isn't very instructive if you just want to learn the main usage of the package.
260
261 # some informative command-line output comparing search results and injection
262 # get max of GridSearch, contains twoF and all Doppler parameters in the dict
263 max_dict_grid = gridsearch.get_max_twoF()
264 # same for MCMCSearch, here twoF is separate, and non-sampled parameters are not
265 # included either
266 max_dict_mcmc, max_2F_mcmc = mcmcsearch.get_max_twoF()
267 logger.info(
268     "max2F={:.4f} from GridSearch, offsets from injection: {:s}".format(
269         max_dict_grid["twoF"],
270         ", ".join(
271             [
272                 "{:.4e} in {:s}".format(max_dict_grid[key] - inj[key], key)
273                 for key in search_keys
274             ]
275         ),
276     )
277 )
278 logger.info(
279     "max2F={:.4f} from MCMCSearch, offsets from injection: {:s}".format(
280         max_2F_mcmc,
281         ", ".join(
282             [
283                 "{:.4e} in {:s}".format(max_dict_mcmc[key] - inj[key], key)
284                 for key in search_keys
285             ]
286         ),
287     )
288 )
289 # get additional point and interval estimators
290 stats_dict_mcmc = mcmcsearch.get_summary_stats()

```

(continues on next page)

(continued from previous page)

```

290 logger.info(
291     "mean from MCMCSearch: offset from injection by      {:s},"
292     " or in fractions of 2sigma intervals: {:s}.".format(
293         ", ".join(
294             [
295                 "{:.4e} in {:s}".format(
296                     stats_dict_mcmc[key]["mean"] - inj[key], key
297                 )
298                 for key in search_keys
299             ]
300         ),
301         ", ".join(
302             [
303                 "{:.2f}% in {:s}".format(
304                     100
305                     * np.abs(stats_dict_mcmc[key]["mean"] - inj[key])
306                     / (2 * stats_dict_mcmc[key]["std"]),
307                     key,
308                 )
309                 for key in search_keys
310             ]
311         ),
312     )
313 )
314 logger.info(
315     "median from MCMCSearch: offset from injection by      {:s},"
316     " or in fractions of 90% confidence intervals: {:s}.".format(
317         ", ".join(
318             [
319                 "{:.4e} in {:s}".format(
320                     stats_dict_mcmc[key]["median"] - inj[key], key
321                 )
322                 for key in search_keys
323             ]
324         ),
325         ", ".join(
326             [
327                 "{:.2f}% in {:s}".format(
328                     100
329                     * np.abs(stats_dict_mcmc[key]["median"] - inj[key])
330                     / (
331                         stats_dict_mcmc[key]["upper90"]
332                         - stats_dict_mcmc[key]["lower90"]
333                     ),
334                     key,
335                 )
336                 for key in search_keys
337             ]
338         ),
339     )
340 )
341

```

(continues on next page)

(continued from previous page)

```

342 # do additional custom plotting
343 logger.info("Loading grid and MCMC search results for custom comparison plots...")
344 gridfile = os.path.join(outdir, gridsearch.label + "_NA_GridSearch.txt")
345 if not os.path.isfile(gridfile):
346     raise RuntimeError(
347         "Failed to load GridSearch results from file '{:s}',"
348         " something must have gone wrong!".format(gridfile)
349     )
350 grid_res = pyfstat.utils.read_txt_file_with_header(gridfile)
351 mcmc_file = os.path.join(outdir, mcmcsearch.label + "_samples.dat")
352 if not os.path.isfile(mcmc_file):
353     raise RuntimeError(
354         "Failed to load MCMCSearch results from file '{:s}',"
355         " something must have gone wrong!".format(mcmc_file)
356     )
357 mcmc_res = pyfstat.utils.read_txt_file_with_header(mcmc_file)
358
359 zoom = {
360     "F0": [inj["F0"] - 10 * dF0, inj["F0"] + 10 * dF0],
361     "F1": [inj["F1"] - 5 * dF1, inj["F1"] + 5 * dF1],
362 }
363
364 # we'll use the two local plotting functions defined above
365 # to avoid code duplication in the sky case
366 logger.info("Creating MCMC-grid comparison plots...")
367 plot_grid_vs_samples(grid_res, mcmc_res, "F0", "F1")
368 plot_2F_scatter(grid_res, "grid", "F0", "F1")
369 plot_2F_scatter(mcmc_res, "mcmc", "F0", "F1")
370 if sky:
371     plot_grid_vs_samples(grid_res, mcmc_res, "Alpha", "Delta")
372     plot_2F_scatter(grid_res, "grid", "Alpha", "Delta")
373     plot_2F_scatter(mcmc_res, "mcmc", "Alpha", "Delta")

```

Total running time of the script: (0 minutes 0.000 seconds)

3.4 Multi-stage MCMC follow up

Application of MCMC F-statistic algorithms to the follow up of a CW signal candidate.

3.4.1 Follow up example

Multi-stage MCMC follow up of a CW signal produced by an isolated source using a ladder of coherent times.

```

8  import os
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12
13 import pyfstat
14

```

(continues on next page)

(continued from previous page)

```

15 label = "PyFstatExampleSemiCoherentDirectedFollowup"
16 outdir = os.path.join("PyFstat_example_data", label)
17 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
18
19 # Properties of the GW data
20 data_parameters = {
21     "sqrtSX": 1e-23,
22     "tstart": 10000000000,
23     "duration": 100 * 86400,
24     "detectors": "H1",
25 }
26 tend = data_parameters["tstart"] + data_parameters["duration"]
27 mid_time = 0.5 * (data_parameters["tstart"] + tend)
28
29 # Properties of the signal
30 depth = 40
31 signal_parameters = {
32     "F0": 30.0,
33     "F1": -1e-10,
34     "F2": 0,
35     "Alpha": np.radians(83.6292),
36     "Delta": np.radians(22.0144),
37     "tref": mid_time,
38     "h0": data_parameters["sqrtSX"] / depth,
39     "cosi": 1.0,
40 }
41
42 data = pyfstat.Writer(
43     label=label, outdir=outdir, **data_parameters, **signal_parameters
44 )
45 data.make_data()
46
47 # The predicted twoF (expectation over noise realizations) can be accessed by
48 twoF = data.predict_fstat()
49 logger.info("Predicted twoF value: {}\n".format(twoF))
50
51 # Search
52 VF0 = VF1 = 1e5
53 DeltaF0 = np.sqrt(VF0) * np.sqrt(3) / (np.pi * data_parameters["duration"])
54 DeltaF1 = np.sqrt(VF1) * np.sqrt(180) / (np.pi * data_parameters["duration"] ** 2)
55 theta_prior = {
56     "F0": {
57         "type": "unif",
58         "lower": signal_parameters["F0"] - DeltaF0 / 2.0,
59         "upper": signal_parameters["F0"] + DeltaF0 / 2,
60     },
61     "F1": {
62         "type": "unif",
63         "lower": signal_parameters["F1"] - DeltaF1 / 2.0,
64         "upper": signal_parameters["F1"] + DeltaF1 / 2,
65     },
66 }

```

(continues on next page)

(continued from previous page)

```

67 for key in "F2", "Alpha", "Delta":
68     theta_prior[key] = signal_parameters[key]
69
70
71 ntemps = 3
72 log10beta_min = -0.5
73 nwalkers = 100
74 nsteps = [100, 100]
75
76 mcmc = pyfstat.MCMCFollowUpSearch(
77     label=label,
78     outdir=outdir,
79     sftfilepattern=data.sftfilepath,
80     theta_prior=theta_prior,
81     tref=mid_time,
82     minStartTime=data_parameters["tstart"],
83     maxStartTime=tend,
84     nwalkers=nwalkers,
85     nsteps=nsteps,
86     ntemps=ntemps,
87     log10beta_min=log10beta_min,
88 )
89
90 NstarMax = 1000
91 Nsegs0 = 100
92 walkers_fig, walkers_axes = plt.subplots(nrows=2, figsize=(3.4, 3.5))
93 mcmc.run(
94     NstarMax=NstarMax,
95     Nsegs0=Nsegs0,
96     plot_walkers=True,
97     walker_plot_args={
98         "labelpad": 0.01,
99         "plot_det_stat": False,
100         "fig": walkers_fig,
101         "axes": walkers_axes,
102         "injection_parameters": signal_parameters,
103     },
104 )
105 walkers_fig.savefig(os.path.join(outdir, label + "_walkers.png"))
106 plt.close(walkers_fig)
107
108 mcmc.print_summary()
109 mcmc.plot_corner(add_prior=True, truths=signal_parameters)
110 mcmc.plot_prior_posterior(injection_parameters=signal_parameters)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.5 Binary-modulated CW searches

Application of MCMC F-statistic algorithms to the search for continuous gravitational wave sources in binary systems.

3.5.1 Binary CW example: Semicoherent MCMC search

MCMC search of a CW signal produced by a source in a binary system using the semicoherent F-statistic.

```

9  import os
10
11  import numpy as np
12
13  import pyfstat
14
15  # If False, sky priors are used
16  directed_search = True
17  # If False, ecc and argp priors are used
18  known_eccentricity = True
19
20  label = "PyFstatExampleSemiCoherentBinarySearchUsingMCMC"
21  outdir = os.path.join("PyFstat_example_data", label)
22  logger = pyfstat.set_up_logger(label=label, outdir=outdir)
23
24  # Properties of the GW data
25  data_parameters = {
26      "sqrtSX": 1e-23,
27      "tstart": 10000000000,
28      "duration": 10 * 86400,
29      "detectors": "H1",
30  }
31  tend = data_parameters["tstart"] + data_parameters["duration"]
32  mid_time = 0.5 * (data_parameters["tstart"] + tend)
33
34  # Properties of the signal
35  depth = 0.1
36  signal_parameters = {
37      "F0": 30.0,
38      "F1": 0,
39      "F2": 0,
40      "Alpha": 0.15,
41      "Delta": 0.45,
42      "tp": mid_time,
43      "argp": 0.3,
44      "asini": 10.0,
45      "ecc": 0.1,
46      "period": 45 * 24 * 3600.0,
47      "tref": mid_time,
48      "h0": data_parameters["sqrtSX"] / depth,
49      "cosi": 1.0,
50  }
51
52  data = pyfstat.BinaryModulatedWriter(

```

(continues on next page)

(continued from previous page)

```

53     label=label, outdir=outdir, **data_parameters, **signal_parameters
54 )
55 data.make_data()
56
57 theta_prior = {
58     "F0": signal_parameters["F0"],
59     "F1": signal_parameters["F1"],
60     "F2": signal_parameters["F2"],
61     "asini": {
62         "type": "unif",
63         "lower": 0.9 * signal_parameters["asini"],
64         "upper": 1.1 * signal_parameters["asini"],
65     },
66     "period": {
67         "type": "unif",
68         "lower": 0.9 * signal_parameters["period"],
69         "upper": 1.1 * signal_parameters["period"],
70     },
71     "tp": {
72         "type": "unif",
73         "lower": mid_time - signal_parameters["period"] / 2.0,
74         "upper": mid_time + signal_parameters["period"] / 2.0,
75     },
76 }
77
78 if directed_search:
79     for key in "Alpha", "Delta":
80         theta_prior[key] = signal_parameters[key]
81 else:
82     theta_prior.update(
83         {
84             "Alpha": {
85                 "type": "unif",
86                 "lower": signal_parameters["Alpha"] - 0.01,
87                 "upper": signal_parameters["Alpha"] + 0.01,
88             },
89             "Delta": {
90                 "type": "unif",
91                 "lower": signal_parameters["Delta"] - 0.01,
92                 "upper": signal_parameters["Delta"] + 0.01,
93             },
94         }
95     )
96
97
98 if known_eccentricity:
99     for key in "ecc", "argp":
100         theta_prior[key] = signal_parameters[key]
101 else:
102     theta_prior.update(
103         {
104             "ecc": {

```

(continues on next page)

(continued from previous page)

```

105         "type": "unif",
106         "lower": signal_parameters["ecc"] - 5e-2,
107         "upper": signal_parameters["ecc"] + 5e-2,
108     },
109     "argp": {
110         "type": "unif",
111         "lower": signal_parameters["argp"] - np.pi / 2,
112         "upper": signal_parameters["argp"] + np.pi / 2,
113     },
114 }
115 )
116
117 ntemps = 3
118 log10beta_min = -1
119 nwalkers = 150
120 nsteps = [100, 200]
121
122 mcmc = pyfstat.MCMCSemiCoherentSearch(
123     label=label,
124     outdir=outdir,
125     nsegs=10,
126     sftfilepattern=data.sftfilepath,
127     theta_prior=theta_prior,
128     tref=signal_parameters["tref"],
129     minStartTime=data_parameters["tstart"],
130     maxStartTime=tend,
131     nsteps=nsteps,
132     nwalkers=nwalkers,
133     ntemps=ntemps,
134     log10beta_min=log10beta_min,
135     binary=True,
136 )
137
138 mcmc.run(
139     plot_walkers=True,
140     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters},
141 )
142 mcmc.plot_corner(add_prior=True, truths=signal_parameters)
143 mcmc.plot_prior_posterior(injection_parameters=signal_parameters)
144 mcmc.print_summary()

```

Total running time of the script: (0 minutes 0.000 seconds)

3.5.2 Binary CW example: Comparison between MCMC and grid search

Comparison of the semicoherent F-statistic MCMC search algorithm to a simple grid search across the parameter space corresponding to a CW source in a binary system.

```

10 import os
11
12 import matplotlib.pyplot as plt
13 import numpy as np
14
15 import pyfstat
16
17 # Set to false to include eccentricity
18 circular_orbit = False
19
20 label = "PyFstatExampleBinaryMCMCvsGridComparison" + (
21     "CircularOrbit" if circular_orbit else ""
22 )
23 outdir = os.path.join("PyFstat_example_data", label)
24 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
25
26 # Parameters to generate a data set
27 data_parameters = {
28     "sqrtSX": 1e-22,
29     "tstart": 10000000000,
30     "duration": 90 * 86400,
31     "detectors": "H1,L1",
32     "Tsft": 3600,
33     "Band": 4,
34 }
35
36 # Injected signal parameters
37 tend = data_parameters["tstart"] + data_parameters["duration"]
38 mid_time = 0.5 * (data_parameters["tstart"] + tend)
39 depth = 10.0
40 signal_parameters = {
41     "tref": data_parameters["tstart"],
42     "F0": 40.0,
43     "F1": 0,
44     "F2": 0,
45     "Alpha": 0.5,
46     "Delta": 0.5,
47     "period": 85 * 24 * 3600.0,
48     "asini": 4.0,
49     "tp": mid_time * 1.05,
50     "argp": 0.0 if circular_orbit else 0.54,
51     "ecc": 0.0 if circular_orbit else 0.7,
52     "h0": data_parameters["sqrtSX"] / depth,
53     "cosi": 1.0,
54 }
55
56
57 logger.info("Generating SFTs with injected signal...")

```

(continues on next page)

(continued from previous page)

```

58 writer = pyfstat.BinaryModulatedWriter(
59     label=label + "SimulatedSignal",
60     outdir=outdir,
61     **data_parameters,
62     **signal_parameters,
63 )
64 writer.make_data()
65 logger.info("")
66
67 logger.info("Performing Grid Search...")
68
69 # Create ad-hoc grid and compute Fstatistic around injection point
70 # There's no class supporting a binary search in the same way as
71 # grid_based_searches.GridSearch, so we do it by hand constructing
72 # a grid and using core.ComputeFstat.
73 half_points_per_dimension = 2
74 search_keys = ["period", "asini", "tp", "argp", "ecc"]
75 search_keys_label = [
76     r"$P$ [s]",
77     r"$a_p$ [s]",
78     r"$t_{p}$ [s]",
79     r"$\omega$ [rad]",
80     r"$e$",
81 ]
82
83 grid_arrays = np.meshgrid(
84     *[
85         signal_parameters[key]
86         * (
87             1
88             + 0.01
89             * np.arange(-half_points_per_dimension, half_points_per_dimension + 1, 1)
90         )
91         for key in search_keys
92     ]
93 )
94 grid_points = np.hstack(
95     [grid_arrays[i].reshape(-1, 1) for i in range(len(grid_arrays))]
96 )
97
98 compute_f_stat = pyfstat.ComputeFstat(
99     sftfilepattern=writer.sftfilepath,
100     tref=signal_parameters["tref"],
101     binary=True,
102     minCoverFreq=-0.5,
103     maxCoverFreq=-0.5,
104 )
105 twoF_values = np.zeros(grid_points.shape[0])
106 for ind in range(grid_points.shape[0]):
107     point = grid_points[ind]
108     twoF_values[ind] = compute_f_stat.get_fullycoherent_twoF(
109         F0=signal_parameters["F0"],

```

(continues on next page)

(continued from previous page)

```

110     F1=signal_parameters["F1"],
111     F2=signal_parameters["F2"],
112     Alpha=signal_parameters["Alpha"],
113     Delta=signal_parameters["Delta"],
114     period=point[0],
115     asini=point[1],
116     tp=point[2],
117     argp=point[3],
118     ecc=point[4],
119 )
120 logger.info(f"2Fstat computed on {grid_points.shape[0]} points")
121 logger.info("")
122 logger.info("Plotting results...")
123 dim = len(search_keys)
124 fig, ax = plt.subplots(dim, 1, figsize=(10, 10))
125 for ind in range(dim):
126     a = ax.ravel()[ind]
127     a.grid()
128     a.set(xlabel=search_keys_label[ind], ylabel=r"$2 \mathcal{F}$", ylabel="log")
129     a.plot(grid_points[:, ind], twoF_values, "o")
130     a.axvline(signal_parameters[search_keys[ind]], label="Injection", color="orange")
131 plt.tight_layout()
132 fig.savefig(os.path.join(outdir, "grid_twoF_per_dimension.png"))
133
134
135 logger.info("Performing MCMCSearch...")
136 # Fixed points in frequency and sky parameters
137 theta_prior = {
138     "F0": signal_parameters["F0"],
139     "F1": signal_parameters["F1"],
140     "F2": signal_parameters["F2"],
141     "Alpha": signal_parameters["Alpha"],
142     "Delta": signal_parameters["Delta"],
143 }
144
145 # Set up priors for the binary parameters
146 for key in search_keys:
147     theta_prior.update(
148         {
149             key: {
150                 "type": "unif",
151                 "lower": 0.999 * signal_parameters[key],
152                 "upper": 1.001 * signal_parameters[key],
153             }
154         }
155     )
156 if circular_orbit:
157     for key in ["ecc", "argp"]:
158         theta_prior[key] = 0
159         search_keys.remove(key)
160
161 # ptemcee sampler settings - in a real application we might want higher values

```

(continues on next page)

(continued from previous page)

```

162 ntemps = 2
163 log10beta_min = -1
164 nwalkers = 100
165 nsteps = [100, 100] # [burnin, production]
166
167 mcmcsearch = pyfstat.MCMCSearch(
168     label=label + "MCMCSearch",
169     outdir=outdir,
170     sftfilepattern=writer.sftfilepath,
171     theta_prior=theta_prior,
172     tref=signal_parameters["tref"],
173     nsteps=nsteps,
174     nwalkers=nwalkers,
175     ntemps=ntemps,
176     log10beta_min=log10beta_min,
177     binary=True,
178 )
179 # walker plot is generated during main run of the search class
180 mcmcsearch.run(
181     plot_walkers=True,
182     walker_plot_args={"plot_det_stat": True, "injection_parameters": signal_parameters},
183 )
184 mcmcsearch.print_summary()
185
186 # call some built-in plotting methods
187 # these can all highlight the injection parameters, too
188 logger.info("Making MCMCSearch {:s} corner plot...".format("-".join(search_keys)))
189 mcmcsearch.plot_corner(truths=signal_parameters)
190 logger.info("Making MCMCSearch prior-posterior comparison plot...")
191 mcmcsearch.plot_prior_posterior(injection_parameters=signal_parameters)
192 logger.info("")
193
194 logger.info("==" * 20)
195 logger.info("Quantitative comparisons:")
196 logger.info("==" * 20)
197
198 # some informative command-line output comparing search results and injection
199 # get max twoF and binary Doppler parameters
200 max_grid_index = np.argmax(twoF_values)
201 max_grid_2F = twoF_values[max_grid_index]
202 max_grid_parameters = grid_points[max_grid_index]
203
204 # same for MCMCSearch, here twoF is separate, and non-sampled parameters are not_
↳ included either
205 max_dict_mcmc, max_2F_mcmc = mcmcsearch.get_max_twoF()
206 logger.info(
207     "Grid Search:\n\tmax2F={:.4f}\n\tOffsets from injection parameters (relative_
↳ error): {:s}.".format(
208         max_grid_2F,
209         ", ".join(
210             [
211                 "\n\t\t{1:s}: {0:.4e} ({2:.4f}%)".format(

```

(continues on next page)

(continued from previous page)

```

212         max_grid_parameters[search_keys.index(key)]
213         - signal_parameters[key],
214         key,
215         100
216         * (
217             max_grid_parameters[search_keys.index(key)]
218             - signal_parameters[key]
219         )
220         / signal_parameters[key],
221     )
222     for key in search_keys
223 ]
224 ),
225 )
226 )
227 logger.info(
228     "Max 2F candidate from MCMC Search:\n\tmax2F={:.4f}"
229     "\n\tOffsets from injection parameters (relative error): {:s}.".format(
230         max_2F_mcmc,
231         ", ".join(
232             [
233                 "\n\t\t{1:s}: {0:.4e} ({2:.4f}%)".format(
234                     max_dict_mcmc[key] - signal_parameters[key],
235                     key,
236                     100
237                     * (max_dict_mcmc[key] - signal_parameters[key])
238                     / signal_parameters[key],
239                 )
240                 for key in search_keys
241             ]
242         ),
243     )
244 )
245 # get additional point and interval estimators
246 stats_dict_mcmc = mcmcsearch.get_summary_stats()
247 logger.info(
248     "Mean from MCMCSearch:\n\tOffset from injection parameters (relative error): {:s}"
249     "\n\tExpressed as fractions of 2sigma intervals: {:s}.".format(
250         ", ".join(
251             [
252                 "\n\t\t{1:s}: {0:.4e} ({2:.4f}%)".format(
253                     stats_dict_mcmc[key]["mean"] - signal_parameters[key],
254                     key,
255                     100
256                     * (stats_dict_mcmc[key]["mean"] - signal_parameters[key])
257                     / signal_parameters[key],
258                 )
259                 for key in search_keys
260             ]
261         ),
262         ", ".join(
263             [

```

(continues on next page)

(continued from previous page)

```

264         "\n\t\t{1:s}: {0:.4f}%".format(
265             100
266             * np.abs(stats_dict_mcmc[key]["mean"] - signal_parameters[key])
267             / (2 * stats_dict_mcmc[key]["std"]),
268             key,
269         )
270     for key in search_keys
271 ]
272 ),
273 )
274 )
275 logger.info(
276     "Median from MCMCSearch:\n\tOffset from injection parameters (relative error): {s},
↪ "
277     "\n\tExpressed as fractions of 90% confidence intervals: {s}.".format(
278         ", ".join(
279             [
280                 "\n\t\t{1:s}: {0:.4e} ({2:.4f}%)".format(
281                     stats_dict_mcmc[key]["median"] - signal_parameters[key],
282                     key,
283                     100
284                     * (stats_dict_mcmc[key]["median"] - signal_parameters[key])
285                     / signal_parameters[key],
286                 )
287                 for key in search_keys
288             ]
289         ),
290         ", ".join(
291             [
292                 "\n\t\t{1:s}: {0:.4f}%".format(
293                     100
294                     * np.abs(stats_dict_mcmc[key]["median"] - signal_parameters[key])
295                     / (
296                         stats_dict_mcmc[key]["upper90"]
297                         - stats_dict_mcmc[key]["lower90"]
298                     ),
299                     key,
300                 )
301                 for key in search_keys
302             ]
303         ),
304     )
305 )

```

Total running time of the script: (0 minutes 0.000 seconds)

3.6 Glitch robust CW searches

Extension of standard CW search methods to increase its robustness against astrophysical glitches. See [arXiv:1805.03314 \[gr-qc\]](https://arxiv.org/abs/1805.03314).

3.6.1 MCMC search on data presenting a glitch

Executes a directed MCMC semicoherent F-statistic search on data presenting a glitch. This is intended to show the impact of glitches on vanilla CW searches.

```

10 import os
11
12 import numpy as np
13 from PyFstat_example_make_data_for_search_on_1_glitch import (
14     F0,
15     F1,
16     F2,
17     Alpha,
18     Delta,
19     duration,
20     outdir,
21     tref,
22     tstart,
23 )
24
25 import pyfstat
26
27 label = "PyFstatExampleStandardDirectedMCMCSearchOn1Glitch"
28 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
29
30 Nstar = 100000
31 F0_width = np.sqrt(Nstar) * np.sqrt(12) / (np.pi * duration)
32 F1_width = np.sqrt(Nstar) * np.sqrt(180) / (np.pi * duration**2)
33
34 theta_prior = {
35     "F0": {"type": "unif", "lower": F0 - F0_width / 2.0, "upper": F0 + F0_width / 2.0},
36     "F1": {"type": "unif", "lower": F1 - F1_width / 2.0, "upper": F1 + F1_width / 2.0},
37     "F2": F2,
38     "Alpha": Alpha,
39     "Delta": Delta,
40 }
41
42 ntemps = 2
43 log10beta_min = -0.5
44 nwalkers = 100
45 nsteps = [500, 2000]
46
47 mcmc = pyfstat.MCMCSearch(
48     label=label,
49     outdir=outdir,
50     sftfilepattern=os.path.join(outdir, "*1glitch*sft"),
51     theta_prior=theta_prior,

```

(continues on next page)

(continued from previous page)

```

52     tref=tref,
53     minStartTime=tstart,
54     maxStartTime=tstart + duration,
55     nsteps=nsteps,
56     nwalkers=nwalkers,
57     ntemps=ntemps,
58     log10beta_min=log10beta_min,
59 )
60
61 mcmc.transform_dictionary["F0"] = dict(subtractor=F0, symbol="$f-f^{\mathrm{s}}$")
62 mcmc.transform_dictionary["F1"] = dict(
63     subtractor=F1, symbol="$\dot{f}-\dot{f}^{\mathrm{s}}$")
64 )
65
66 mcmc.run()
67 mcmc.print_summary()
68 mcmc.plot_corner()
69 mcmc.plot_cumulative_max(savefig=True)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.6.2 Glitch examples: Make data

Generate the data to run examples on glitch-robust searches.

```

8  import os
9
10 import numpy as np
11
12 from pyfstat import GlitchWriter, Writer
13
14 label = "PyFstatExampleGlitchRobustSearch"
15 outdir = os.path.join("PyFstat_example_data", label)
16
17 # First, we generate data with a reasonably strong smooth signal
18
19 # Define parameters of the Crab pulsar as an example
20 F0 = 30.0
21 F1 = -1e-10
22 F2 = 0
23 Alpha = np.radians(83.6292)
24 Delta = np.radians(22.0144)
25
26 # Signal strength
27 h0 = 5e-24
28 cosi = 0
29
30 # Properties of the GW data
31 sqrtSX = 1e-22
32 tstart = 10000000000
33 duration = 50 * 86400

```

(continues on next page)

(continued from previous page)

```

34 tend = tstart + duration
35 tref = tstart + 0.5 * duration
36 IFO = "H1"
37
38 data = Writer(
39     label=label + "0glitch",
40     outdir=outdir,
41     tref=tref,
42     tstart=tstart,
43     F0=F0,
44     F1=F1,
45     F2=F2,
46     duration=duration,
47     Alpha=Alpha,
48     Delta=Delta,
49     h0=h0,
50     cosi=cosi,
51     sqrtSX=sqrtSX,
52     detectors=IFO,
53 )
54 data.make_data()
55
56 # Next, taking the same signal parameters, we include a glitch half way through
57 dtglitch = duration / 2.0
58 delta_F0 = 5e-6
59 delta_F1 = 0
60
61 glitch_data = GlitchWriter(
62     label=label + "1glitch",
63     outdir=outdir,
64     tref=tref,
65     tstart=tstart,
66     F0=F0,
67     F1=F1,
68     F2=F2,
69     duration=duration,
70     Alpha=Alpha,
71     Delta=Delta,
72     h0=h0,
73     cosi=cosi,
74     sqrtSX=sqrtSX,
75     detectors=IFO,
76     dtglitch=dtglitch,
77     delta_F0=delta_F0,
78     delta_F1=delta_F1,
79 )
80 glitch_data.make_data()
81
82 # Making data with two glitches
83
84 dtglitch_2 = [duration / 4.0, 4 * duration / 5.0]
85 delta_phi_2 = [0, 0]

```

(continues on next page)

(continued from previous page)

```

86 delta_F0_2 = [4e-6, 3e-7]
87 delta_F1_2 = [0, 0]
88 delta_F2_2 = [0, 0]
89
90 two_glitch_data = GlitchWriter(
91     label=label + "2glitch",
92     outdir=outdir,
93     tref=tref,
94     tstart=tstart,
95     F0=F0,
96     F1=F1,
97     F2=F2,
98     duration=duration,
99     Alpha=Alpha,
100     Delta=Delta,
101     h0=h0,
102     cosi=cosi,
103     sqrtSX=sqrtSX,
104     detectors=IFO,
105     dtglitch=dtglitch_2,
106     delta_phi=delta_phi_2,
107     delta_F0=delta_F0_2,
108     delta_F1=delta_F1_2,
109     delta_F2=delta_F2_2,
110 )
111 two_glitch_data.make_data()

```

Total running time of the script: (0 minutes 0.000 seconds)

3.6.3 Glitch robust grid search

Grid search employing a signal hypothesis allowing for a glitch to be present in the data. The setup corresponds to a targeted search, and the simulated signal contains a single glitch.

```

10 import os
11 import time
12
13 import numpy as np
14 from PyFstat_example_make_data_for_search_on_1_glitch import (
15     F0,
16     F1,
17     F2,
18     Alpha,
19     Delta,
20     delta_F0,
21     dtglitch,
22     duration,
23     outdir,
24     tref,
25     tstart,
26 )

```

(continues on next page)

(continued from previous page)

```

27
28 import pyfstat
29
30 label = "PyFstatExampleGlitchRobustDirectedGridSearchOn1Glitch"
31 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
32
33 Nstar = 10000
34 F0_width = np.sqrt(Nstar) * np.sqrt(12) / (np.pi * duration)
35 F1_width = np.sqrt(Nstar) * np.sqrt(180) / (np.pi * duration**2)
36 N = 20
37 F0s = [F0 - F0_width / 2.0, F0 + F0_width / 2.0, F0_width / N]
38 F1s = [F1 - F1_width / 2.0, F1 + F1_width / 2.0, F1_width / N]
39 F2s = [F2]
40 Alphas = [Alpha]
41 Deltas = [Delta]
42
43 max_delta_F0 = 1e-5
44 tglitches = [tstart + 0.1 * duration, tstart + 0.9 * duration, 0.8 * float(duration) / N]
45 delta_F0s = [0, max_delta_F0, max_delta_F0 / N]
46 delta_F1s = [0]
47
48
49 t1 = time.time()
50 search = pyfstat.GridGlitchSearch(
51     label,
52     outdir,
53     os.path.join(outdir, "1glitch*sft"),
54     F0s=F0s,
55     F1s=F1s,
56     F2s=F2s,
57     Alphas=Alphas,
58     Deltas=Deltas,
59     tref=tref,
60     minStartTime=tstart,
61     maxStartTime=tstart + duration,
62     tglitches=tglitches,
63     delta_F0s=delta_F0s,
64     delta_F1s=delta_F1s,
65 )
66 search.run()
67 dT = time.time() - t1
68
69 F0_vals = np.unique(search.data["F0"]) - F0
70 F1_vals = np.unique(search.data["F1"]) - F1
71 delta_F0s_vals = np.unique(search.data["delta_F0"]) - delta_F0
72 tglitch_vals = np.unique(search.data["tglitch"])
73 tglitch_vals_days = (tglitch_vals - tstart) / 86400.0 - dtglitch / 86400.0
74
75 logger.info("Making gridcorner plot...")
76 twoF = search.data["twoF"].reshape(
77     (len(F0_vals), len(F1_vals), len(delta_F0s_vals), len(tglitch_vals))
78 )

```

(continues on next page)

(continued from previous page)

```

79 xyz = [F0_vals * 1e6, F1_vals * 1e12, delta_F0s_vals * 1e6, tglitch_vals_days]
80 labels = [
81     "$f - f_{\\mathrm{s}}$\\n[$\\mu$Hz]",
82     "$\\dot{f} - \\dot{f}_{\\mathrm{s}}$\\n[$p$Hz/s]",
83     "$\\delta f - \\delta f_{\\mathrm{s}}$\\n[$\\mu$Hz]",
84     "$t^{\\mathrm{g}} - t^{\\mathrm{g}}_{\\mathrm{s}}$\\n[d]",
85     "$t^{\\mathrm{g}} - t^{\\mathrm{g}}_{\\mathrm{s}}$\\n[d]",
86     "$\\widehat{2\\mathcal{F}}$",
87 ]
88 fig, axes = pyfstat.gridcorner(
89     twoF,
90     xyz,
91     projection="log_mean",
92     labels=labels,
93     showDvals=False,
94     lines=[0, 0, 0, 0],
95     label_offset=0.25,
96     max_n_ticks=4,
97 )
98 fig.savefig("{}_{}_projection_matrix.png".format(outdir, label), bbox_inches="tight")
99
100
101 logger.info(f"Prior widths = {F0_width}, {F1_width}")
102 logger.info(f"Actual run time = {dT} s")
103 logger.info(f"Actual number of grid points = {search.data.shape[0]}")

```

Total running time of the script: (0 minutes 0.000 seconds)

3.6.4 Glitch robust MCMC search

MCMC search employing a signal hypothesis allowing for a glitch to be present in the data. The setup corresponds to a targeted search, and the simulated signal contains a single glitch.

```

10 import os
11 import time
12
13 import numpy as np
14 from PyFstat_example_make_data_for_search_on_1_glitch import (
15     F0,
16     F1,
17     F2,
18     Alpha,
19     Delta,
20     delta_F0,
21     dtglitch,
22     duration,
23     outdir,
24     tref,
25     tstart,
26 )
27

```

(continues on next page)

(continued from previous page)

```

28 import pyfstat
29
30 label = "PyFstatExampleGlitchRobustDirectedMCMCSearchOn1Glitch"
31 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
32
33 Nstar = 10000
34 F0_width = np.sqrt(Nstar) * np.sqrt(12) / (np.pi * duration)
35 F1_width = np.sqrt(Nstar) * np.sqrt(180) / (np.pi * duration**2)
36
37 theta_prior = {
38     "F0": {"type": "unif", "lower": F0 - F0_width / 2.0, "upper": F0 + F0_width / 2.0},
39     "F1": {"type": "unif", "lower": F1 - F1_width / 2.0, "upper": F1 + F1_width / 2.0},
40     "F2": F2,
41     "delta_F0": {"type": "unif", "lower": 0, "upper": 1e-5},
42     "delta_F1": 0,
43     "tglitch": {
44         "type": "unif",
45         "lower": tstart + 0.1 * duration,
46         "upper": tstart + 0.9 * duration,
47     },
48     "Alpha": Alpha,
49     "Delta": Delta,
50 }
51
52 ntemps = 3
53 log10beta_min = -0.5
54 nwalkers = 100
55 nsteps = [250, 250]
56
57 mcmc = pyfstat.MCMCGlitchSearch(
58     label=label,
59     outdir=outdir,
60     sftfilepattern=os.path.join(outdir, "*1glitch*sft"),
61     theta_prior=theta_prior,
62     tref=tref,
63     minStartTime=tstart,
64     maxStartTime=tstart + duration,
65     nsteps=nsteps,
66     nwalkers=nwalkers,
67     ntemps=ntemps,
68     log10beta_min=log10beta_min,
69     nglitch=1,
70 )
71 mcmc.transform_dictionary["F0"] = dict(
72     subtractor=F0, multiplier=1e6, symbol="$f-f_{\\mathrm{s}}$"
73 )
74 mcmc.unit_dictionary["F0"] = "$\\mu$Hz"
75 mcmc.transform_dictionary["F1"] = dict(
76     subtractor=F1, multiplier=1e12, symbol="$\\dot{f}-\\dot{f}_{\\mathrm{s}}$"
77 )
78 mcmc.unit_dictionary["F1"] = "$p$Hz/s"
79 mcmc.transform_dictionary["delta_F0"] = dict(

```

(continues on next page)

(continued from previous page)

```

80     multiplier=1e6, subtractor=delta_F0, symbol="$\\delta f-\\delta f_{\\mathrm{s}}$"
81 )
82 mcmc.unit_dictionary["delta_F0"] = "$\\mu$Hz/s"
83 mcmc.transform_dictionary["tglitch"]["subtractor"] = tstart + dtglitch
84 mcmc.transform_dictionary["tglitch"][
85     "label"
86 ] = "$t^{\\mathrm{g}}-t^{\\mathrm{g}}_{\\mathrm{s}}$[d]"
87
88 t1 = time.time()
89 mcmc.run(save_loudest=False) # uses CFSv2 which doesn't support glitch parameters
90 dT = time.time() - t1
91 mcmc.print_summary()
92
93 logger.info("Making corner plot...")
94 mcmc.plot_corner(
95     label_offset=0.25,
96     truths={"F0": F0, "F1": F1, "delta_F0": delta_F0, "tglitch": tstart + dtglitch},
97     quantiles=(0.16, 0.5, 0.84),
98     hist_kwargs=dict(lw=1.5, zorder=-1),
99     truth_color="C3",
100 )
101
102 mcmc.plot_cumulative_max(savefig=True)
103
104 logger.info(f"Prior widths = {F0_width}, {F1_width}")
105 logger.info(f"Actual run time = {dT} s")

```

Total running time of the script: (0 minutes 0.000 seconds)

3.7 Transient CW searches

F-statistic based searches for transient CW signals. See [arXiv:1104.1704 \[gr-qc\]](https://arxiv.org/abs/1104.1704).

3.7.1 Long transient search examples: Make data

An example to generate data with a long transient signal.

This can be run either stand-alone (will just generate SFT files and nothing else); or it is also being imported from `PyFstat_example_long_transient_MCMC_search.py`

```

12 import os
13
14 import pyfstat
15
16 label = "PyFstatExampleLongTransientSearch"
17 outdir = os.path.join("PyFstat_example_data", label)
18 logger = pyfstat.set_up_logger(outdir=outdir, label=label)
19
20 F0 = 30.0
21 F1 = -1e-10

```

(continues on next page)

(continued from previous page)

```

22 F2 = 0
23 Alpha = 0.5
24 Delta = 1
25
26 tstart = 1000000000
27 duration = 200 * 86400
28
29 transient_tstart = tstart + 0.25 * duration
30 transient_duration = 0.5 * duration
31 transientWindowType = "rect"
32 tref = tstart
33
34 h0 = 1e-23
35 cosi = 0
36 psi = 0
37 phi = 0
38 sqrtSX = 1e-22
39 detectors = "H1,L1"
40
41 if __name__ == "__main__":
42     transient = pyfstat.Writer(
43         label=label,
44         outdir=outdir,
45         tref=tref,
46         tstart=tstart,
47         duration=duration,
48         F0=F0,
49         F1=F1,
50         F2=F2,
51         Alpha=Alpha,
52         Delta=Delta,
53         h0=h0,
54         cosi=cosi,
55         detectors=detectors,
56         sqrtSX=sqrtSX,
57         transientStartTime=transient_tstart,
58         transientTau=transient_duration,
59         transientWindowType=transientWindowType,
60     )
61     transient.make_data()
62     logger.info(f"Predicted 2F from injection Writer: {transient.predict_fstat()}")

```

Total running time of the script: (0 minutes 0.000 seconds)

3.7.2 Short transient search examples: Make data

An example to generate data with a short transient signal.

This can be run either stand-alone (will just generate SFT files and nothing else); or it is also being imported from `PyFstat_example_short_transient_grid_search.py` and `PyFstat_example_short_transient_MCMC_search.py`

```

14 import os
15
16 import pyfstat
17
18 label = "PyFstatExampleShortTransientSearch"
19 outdir = os.path.join("PyFstat_example_data", label)
20 logger = pyfstat.set_up_logger(outdir=outdir, label=label)
21
22
23 F0 = 30.0
24 F1 = -1e-10
25 F2 = 0
26 Alpha = 0.5
27 Delta = 1
28
29 tstart = 10000000000
30 duration = 2 * 86400
31
32 transient_tstart = tstart + 0.25 * duration
33 transient_duration = 0.5 * duration
34 transientWindowType = "rect"
35 tref = tstart
36
37 h0 = 1e-23
38 cosi = 0
39 psi = 0
40 phi = 0
41 sqrtSX = 1e-22
42 detectors = "H1,L1"
43
44 Tsft = 1800
45
46 if __name__ == "__main__":
47     transient = pyfstat.Writer(
48         label=label,
49         outdir=outdir,
50         tref=tref,
51         tstart=tstart,
52         duration=duration,
53         F0=F0,
54         F1=F1,
55         F2=F2,
56         Alpha=Alpha,
57         Delta=Delta,
58         h0=h0,
59         cosi=cosi,
60         detectors=detectors,

```

(continues on next page)

(continued from previous page)

```

61     sqrtSX=sqrtSX,
62     transientStartTime=transient_tstart,
63     transientTau=transient_duration,
64     transientWindowType=transientWindowType,
65     Tsft=Tsft,
66     Band=0.1,
67 )
68 transient.make_data()
69 logger.info(f"Predicted 2F from injection Writer: {transient.predict_fstat()}")

```

Total running time of the script: (0 minutes 0.000 seconds)

3.7.3 Short transient grid search

An example grid-based search for a short transient signal. By default, the standard persistent-CW 2F-statistic and the transient max2F statistic are compared.

You can turn on either *BSGL* = *True* or *BtSG* = *True* (not both!) to test alternative statistics.

This is also ready to use on a GPU, if you have one available and *pycuda* installed. Just change to *tCWFstatMapVersion* = "*pycuda*".

```

17 import os
18
19 import numpy as np
20 import PyFstat_example_make_data_for_short_transient_search as data
21
22 import pyfstat
23
24 tCWFstatMapVersion = "lal"
25
26 if __name__ == "__main__":
27     label = "PyFstatExampleShortTransientGridSearch"
28     logger = pyfstat.set_up_logger(label=label, outdir=data.outdir)
29     if not os.path.isdir(data.outdir) or not np.any(
30         [f.endswith(".sft") for f in os.listdir(data.outdir)]
31     ):
32         raise RuntimeError(
33             "Please first run PyFstat_example_make_data_for_short_transient_search.py !"
34         )
35
36     maxStartTime = data.tstart + data.duration
37
38     m = 0.001
39     dF0 = np.sqrt(12 * m) / (np.pi * data.duration)
40     DeltaF0 = 100 * dF0
41     F0s = [data.F0 - DeltaF0 / 2.0, data.F0 + DeltaF0 / 2.0, dF0]
42     F1s = [data.F1]
43     F2s = [data.F2]
44     Alphas = [data.Alpha]
45     Deltas = [data.Delta]
46

```

(continues on next page)

(continued from previous page)

```

47 BSGL = False
48 BtSG = False
49
50 logger.info("Standard CW search:")
51 search1 = pyfstat.GridSearch(
52     label=label + f"CW{'BSGL' if BSGL else ''}",
53     outdir=data.outdir,
54     sftfilepattern=os.path.join(data.outdir, f"*{data.label}*sft"),
55     F0s=F0s,
56     F1s=F1s,
57     F2s=F2s,
58     Alphas=Alphas,
59     Deltas=Deltas,
60     tref=data.tref,
61     BSGL=BSGL,
62 )
63 search1.run()
64 search1.print_max_twoF()
65 search1.plot_1D(
66     xkey="F0", xlabel="freq [Hz]", ylabel=search1.tex_labels[search1.detstat]
67 )
68
69 logger.info("with t0,tau bands:")
70 label = f"CW{'_BSGL' if BSGL else ''}{'_BtSG' if BtSG else ''}_FstatMap_
↪ {tCWFstatMapVersion}"
71 search2 = pyfstat.TransientGridSearch(
72     label=label,
73     outdir=data.outdir,
74     sftfilepattern=os.path.join(data.outdir, f"*{data.label}*sft"),
75     F0s=F0s,
76     F1s=F1s,
77     F2s=F2s,
78     Alphas=Alphas,
79     Deltas=Deltas,
80     tref=data.tref,
81     transientWindowType="rect",
82     t0Band=data.duration - 2 * data.Tsft,
83     tauBand=data.duration,
84     outputTransientFstatMap=True,
85     tCWFstatMapVersion=tCWFstatMapVersion,
86     BSGL=BSGL,
87     BtSG=BtSG,
88 )
89 search2.run()
90 search2.print_max_twoF()
91 search2.plot_1D(
92     xkey="F0", xlabel="freq [Hz]", ylabel=search2.tex_labels[search2.detstat]
93 )

```

Total running time of the script: (0 minutes 0.000 seconds)

3.7.4 Long transient MCMC search

MCMC search for a long transient CW signal.

By default, the standard persistent-CW 2F-statistic and the transient max2F statistic are compared.

You can turn on either $BSGL = True$ or $BtSG = True$ (not both!) to test alternative statistics.

```

13 import os
14
15 import numpy as np
16 import PyFstat_example_make_data_for_long_transient_search as data
17
18 import pyfstat
19 from pyfstat.utils import get_predict_fstat_parameters_from_dict
20
21 if not os.path.isdir(data.outdir) or not np.any(
22     [f.endswith(".sft") for f in os.listdir(data.outdir)]
23 ):
24     raise RuntimeError(
25         "Please first run PyFstat_example_make_data_for_long_transient_search.py !"
26     )
27
28 label = "PyFstatExampleLongTransientMCMCSearch"
29 logger = pyfstat.set_up_logger(label=label, outdir=data.outdir)
30
31 tstart = data.tstart
32 duration = data.duration
33
34 inj = {
35     "tref": data.tstart,
36     "F0": data.F0,
37     "F1": data.F1,
38     "F2": data.F2,
39     "Alpha": data.Alpha,
40     "Delta": data.Delta,
41     "transient_tstart": data.transient_tstart,
42     "transient_duration": data.transient_duration,
43 }
44
45 DeltaF0 = 6e-7
46 DeltaF1 = 1e-13
47
48 # to make the search cheaper, we exactly target the transientStartTime
49 # to the injected value and only search over TransientTau
50 theta_prior = {
51     "F0": {
52         "type": "unif",
53         "lower": inj["F0"] - DeltaF0 / 2.0,
54         "upper": inj["F0"] + DeltaF0 / 2.0,
55     },
56     "F1": {
57         "type": "unif",
58         "lower": inj["F1"] - DeltaF1 / 2.0,

```

(continues on next page)

(continued from previous page)

```

59     "upper": inj["F1"] + DeltaF1 / 2.0,
60 },
61     "F2": inj["F2"],
62     "Alpha": inj["Alpha"],
63     "Delta": inj["Delta"],
64     "transient_tstart": tstart + 0.25 * duration,
65     "transient_duration": {
66         "type": "halfnorm",
67         "loc": 0.001 * duration,
68         "scale": 0.5 * duration,
69     },
70 }
71
72 ntemps = 2
73 log10beta_min = -1
74 nwalkers = 100
75 nsteps = [100, 100]
76
77 transientWindowType = "rect"
78 BSGl = False
79 BtSG = False
80
81 mcmc = pyfstat.MCMCTransientSearch(
82     label=label + ("BSGL" if BSGl else "") + ("BtSG" if BtSG else ""),
83     outdir=data.outdir,
84     sftfilepattern=os.path.join(data.outdir, f"*{data.label}*sft"),
85     theta_prior=theta_prior,
86     tref=inj["tref"],
87     nsteps=nsteps,
88     nwalkers=nwalkers,
89     ntemps=ntemps,
90     log10beta_min=log10beta_min,
91     transientWindowType=transientWindowType,
92     BSGl=BSGL,
93     BtSG=BtSG,
94 )
95 mcmc.run(walker_plot_args={"plot_det_stat": True, "injection_parameters": inj})
96 mcmc.print_summary()
97 mcmc.plot_corner(add_prior=True, truths=inj)
98 mcmc.plot_prior_posterior(injection_parameters=inj)
99
100 # plot cumulative 2F, first building a dict as required for PredictFStat
101 d, maxtwoF = mcmc.get_max_twoF()
102 for key, val in mcmc.theta_prior.items():
103     if key not in d:
104         d[key] = val
105 d["h0"] = data.h0
106 d["cosi"] = data.cosi
107 d["psi"] = data.psi
108 PFS_input = get_predict_fstat_parameters_from_dict(
109     d, transientWindowType=transientWindowType
110 )

```

(continues on next page)

(continued from previous page)

```
mcmc.plot_cumulative_max(PFS_input=PFS_input)
```

Total running time of the script: (0 minutes 0.000 seconds)

3.7.5 Short transient MCMC search

MCMC search for a Short transient CW signal.

```

8  import os
9
10 import numpy as np
11 import PyFstat_example_make_data_for_short_transient_search as data
12
13 import pyfstat
14 from pyfstat.utils import get_predict_fstat_parameters_from_dict
15
16 if __name__ == "__main__":
17     label = "PyFstatExampleShortTransientMCMCSearch"
18     logger = pyfstat.set_up_logger(label=label, outdir=data.outdir)
19
20     if not os.path.isdir(data.outdir) or not np.any(
21         [f.endswith(".sft") for f in os.listdir(data.outdir)]
22     ):
23         raise RuntimeError(
24             "Please first run PyFstat_example_make_data_for_short_transient_search.py !"
25         )
26
27     inj = {
28         "tref": data.tstart,
29         "F0": data.F0,
30         "F1": data.F1,
31         "F2": data.F2,
32         "Alpha": data.Alpha,
33         "Delta": data.Delta,
34         "transient_tstart": data.transient_tstart,
35         "transient_duration": data.transient_duration,
36     }
37
38     DeltaF0 = 1e-2
39     DeltaF1 = 1e-9
40
41     theta_prior = {
42         "F0": {
43             "type": "unif",
44             "lower": inj["F0"] - DeltaF0 / 2.0,
45             "upper": inj["F0"] + DeltaF0 / 2.0,
46         },
47         "F1": {
48             "type": "unif",
49             "lower": inj["F1"] - DeltaF1 / 2.0,
50             "upper": inj["F1"] + DeltaF1 / 2.0,

```

(continues on next page)

(continued from previous page)

```

51         },
52         "F2": inj["F2"],
53         "Alpha": inj["Alpha"],
54         "Delta": inj["Delta"],
55         "transient_tstart": {
56             "type": "unif",
57             "lower": data.tstart,
58             "upper": data.tstart + data.duration - 2 * data.Tsft,
59         },
60         "transient_duration": {
61             "type": "unif",
62             "lower": 2 * data.Tsft,
63             "upper": data.duration - 2 * data.Tsft,
64         },
65     }
66
67     ntemps = 2
68     log10beta_min = -1
69     nwalkers = 100
70     nsteps = [200, 200]
71
72     BSGL = False
73     transientWindowType = "rect"
74
75     mcmc = pyfstat.MCMCTransientSearch(
76         label=label + ("BSGL" if BSGL else ""),
77         outdir=data.outdir,
78         sftfilepattern=os.path.join(data.outdir, f"*{data.label}*sft"),
79         theta_prior=theta_prior,
80         tref=inj["tref"],
81         nsteps=nsteps,
82         nwalkers=nwalkers,
83         ntemps=ntemps,
84         log10beta_min=log10beta_min,
85         transientWindowType=transientWindowType,
86         BSGL=BSGL,
87     )
88     mcmc.run(walker_plot_args={"plot_det_stat": True, "injection_parameters": inj})
89     mcmc.print_summary()
90     mcmc.plot_corner(add_prior=True, truths=inj)
91     mcmc.plot_prior_posterior(injection_parameters=inj)
92
93     # plot cumulative 2F, first building a dict as required for PredictFStat
94     d, maxtwoF = mcmc.get_max_twoF()
95     for key, val in mcmc.theta_prior.items():
96         if key not in d:
97             d[key] = val
98     d["h0"] = data.h0
99     d["cosi"] = data.cosi
100    d["psi"] = data.psi
101    PFS_input = get_predict_fstat_parameters_from_dict(
102        d, transientWindowType=transientWindowType

```

(continues on next page)

(continued from previous page)

```

103 )
104 mcmc.plot_cumulative_max(PFS_input=PFS_input)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.8 Other examples

Summary of use cases of the different tools provided by PyFstat.

3.8.1 Compute a spectrogram

Compute the spectrogram of a set of SFTs. This is useful to produce visualizations of the Doppler modulation of a CW signal.

```

9  import os
10
11  import matplotlib.pyplot as plt
12
13  import pyfstat
14
15  # not github-action compatible
16  # plt.rcParams["font.family"] = "serif"
17  # plt.rcParams["font.size"] = 18
18  # plt.rcParams["text.usetex"] = True
19
20  # workaround deprecation warning
21  # see https://github.com/matplotlib/matplotlib/issues/21723
22  plt.rcParams["axes.grid"] = False
23
24  label = "PyFstatExampleSpectrogram"
25  outdir = os.path.join("PyFstat_example_data", label)
26  logger = pyfstat.set_up_logger(label=label, outdir=outdir)
27
28  depth = 5
29
30  data_parameters = {
31      "sqrtSX": 1e-23,
32      "tstart": 10000000000,
33      "duration": 2 * 365 * 86400,
34      "detectors": "H1",
35      "Tsft": 1800,
36  }
37
38  signal_parameters = {
39      "F0": 100.0,
40      "F1": 0,
41      "F2": 0,
42      "Alpha": 0.0,
43      "Delta": 0.5,
44      "tp": data_parameters["tstart"],

```

(continues on next page)

(continued from previous page)

```

45     "asini": 25.0,
46     "period": 50 * 86400,
47     "tref": data_parameters["tstart"],
48     "h0": data_parameters["sqrtSX"] / depth,
49     "cosi": 1.0,
50 }
51
52 # making data
53 data = pyfstat.BinaryModulatedWriter(
54     label=label, outdir=outdir, **data_parameters, **signal_parameters
55 )
56 data.make_data()
57
58 logger.info("Loading SFT data and computing normalized power...")
59 freqs, times, sft_data = pyfstat.utils.get_sft_as_arrays(data.sftfilepath)
60 sft_power = sft_data["H1"].real ** 2 + sft_data["H1"].imag ** 2
61 normalized_power = (
62     2 * sft_power / (data_parameters["Tsft"] * data_parameters["sqrtSX"] ** 2)
63 )
64
65 plotfile = os.path.join(outdir, label + ".png")
66 logger.info(f"Plotting to file: {plotfile}")
67 fig, ax = plt.subplots(figsize=(0.8 * 16, 0.8 * 9))
68 ax.set(xlabel="Time [days]", ylabel="Frequency [Hz]", ylim=(99.98, 100.02))
69 c = ax.pcolormesh(
70     (times["H1"] - times["H1"][0]) / 86400,
71     freqs,
72     normalized_power,
73     cmap="inferno_r",
74     shading="nearest",
75 )
76 fig.colorbar(c, label="Normalized Power")
77 plt.tight_layout()
78 fig.savefig(plotfile)

```

Total running time of the script: (0 minutes 0.000 seconds)

3.8.2 Cumulative coherent 2F

Compute the cumulative coherent F-statistic of a signal candidate.

```

9  import os
10
11  import numpy as np
12
13  import pyfstat
14  from pyfstat.utils import get_predict_fstat_parameters_from_dict
15
16  label = "PyFstatExampleTwoFCumulative"
17  outdir = os.path.join("PyFstat_example_data", label)
18  logger = pyfstat.set_up_logger(label=label, outdir=outdir)

```

(continues on next page)

(continued from previous page)

```

19
20 # Properties of the GW data
21 gw_data = {
22     "sqrtSX": 1e-23,
23     "tstart": 10000000000,
24     "duration": 100 * 86400,
25     "detectors": "H1,L1",
26     "Band": 4,
27     "Tsft": 1800,
28 }
29
30 # Properties of the signal
31 depth = 100
32 phase_parameters = {
33     "F0": 30.0,
34     "F1": -1e-10,
35     "F2": 0,
36     "Alpha": np.radians(83.6292),
37     "Delta": np.radians(22.0144),
38     "tref": gw_data["tstart"],
39     "asini": 10,
40     "period": 10 * 3600 * 24,
41     "tp": gw_data["tstart"] + gw_data["duration"] / 2.0,
42     "ecc": 0,
43     "argp": 0,
44 }
45 amplitude_parameters = {
46     "h0": gw_data["sqrtSX"] / depth,
47     "cosi": 1,
48     "phi": np.pi,
49     "psi": np.pi / 8,
50 }
51
52 PFS_input = get_predict_fstat_parameters_from_dict(
53     **phase_parameters, **amplitude_parameters
54 )
55
56 # Let me grab tref here, since it won't really be needed in phase_parameters
57 tref = phase_parameters.pop("tref")
58 data = pyfstat.BinaryModulatedWriter(
59     label=label,
60     outdir=outdir,
61     tref=tref,
62     **gw_data,
63     **phase_parameters,
64     **amplitude_parameters,
65 )
66 data.make_data()
67
68 # The predicted twoF (expectation over noise realizations) can be accessed by
69 twoF = data.predict_fstat()
70 logger.info("Predicted twoF value: {}\n".format(twoF))

```

(continues on next page)

(continued from previous page)

```

71
72 # Create a search object for each of the possible SFT combinations
73 # (H1 only, L1 only, H1 + L1).
74 ifo_constraints = ["L1", "H1", None]
75 compute_fstat_per_ifo = [
76     pyfstat.ComputeFstat(
77         sftfilepattern=os.path.join(
78             data.outdir,
79             (f"{ifo_constraint[0]}*.sft" if ifo_constraint is not None else "*.sft"),
80         ),
81         tref=data.tref,
82         binary=phase_parameters.get("asini", 0),
83         minCoverFreq=-0.5,
84         maxCoverFreq=-0.5,
85     )
86     for ifo_constraint in ifo_constraints
87 ]
88
89 for ind, compute_f_stat in enumerate(compute_fstat_per_ifo):
90     compute_f_stat.plot_twoF_cumulative(
91         label=label + (f"_{ifo_constraints[ind]}" if ind < 2 else "_H1L1"),
92         outdir=outdir,
93         savefig=True,
94         CFS_input=phase_parameters,
95         PFS_input=PFS_input,
96         custom_ax_kwargs={
97             "title": "How does 2F accumulate over time?",
98             "label": "Cumulative 2F"
99         } + (f" {ifo_constraints[ind]}" if ind < 2 else " H1 + L1"),
100     ),
101 )

```

Total running time of the script: (0 minutes 0.000 seconds)

3.8.3 Software injection into pre-existing data files

Add a software injection into a set of SFTs.

In this case, the set of SFTs is generated using Makefakedata_v5, but the same procedure can be applied to any other set of SFTs (including real detector data).

```

12 import os
13
14 import numpy as np
15
16 import pyfstat
17
18 label = "PyFstatExampleInjectionIntoNoiseSFTs"
19 outdir = os.path.join("PyFstat_example_data", label)
20 logger = pyfstat.set_up_logger(label=label, outdir=outdir)
21
22 tstart = 1269734418

```

(continues on next page)

(continued from previous page)

```

23 duration_Tsft = 100
24 Tsft = 1800
25 randSeed = 69420
26 IFO = "H1"
27 h0 = 1000
28 cosi = 0
29 F0 = 30
30 Alpha = 0
31 Delta = 0
32
33 Band = 2.0
34
35 # create sfts with a strong signal in them
36 # window options are optional here
37 noise_and_signal_writer = pyfstat.Writer(
38     label="PyFstatTestSFTsNoiseAndSignal",
39     outdir=outdir,
40     h0=h0,
41     cosi=cosi,
42     F0=F0,
43     Alpha=Alpha,
44     Delta=Delta,
45     tstart=tstart,
46     duration=duration_Tsft * Tsft,
47     Tsft=Tsft,
48     Band=Band,
49     detectors=IFO,
50     randSeed=randSeed,
51     SFTWindowType="tukey",
52     SFTWindowParam=0.001,
53 )
54 noise_and_signal_writer.make_data()
55
56 # compute Fstat
57 coherent_search = pyfstat.ComputeFstat(
58     tref=noise_and_signal_writer.tref,
59     sftfilepath=noise_and_signal_writer.sftfilepath,
60     minCoverFreq=-0.5,
61     maxCoverFreq=0.5,
62 )
63 FS_1 = coherent_search.get_fullycoherent_twoF(
64     noise_and_signal_writer.F0,
65     noise_and_signal_writer.F1,
66     noise_and_signal_writer.F2,
67     noise_and_signal_writer.Alpha,
68     noise_and_signal_writer.Delta,
69 )
70
71 # create noise sfts
72 # window options are again optional for this step
73 noise_writer = pyfstat.Writer(
74     label="PyFstatTestSFTsOnlyNoise",

```

(continues on next page)

(continued from previous page)

```

75     outdir=outdir,
76     h0=0,
77     F0=F0,
78     tstart=tstart,
79     duration=duration_Tsft * Tsft,
80     Tsft=Tsft,
81     Band=Band,
82     detectors=IF0,
83     randSeed=randSeed,
84     SFTWindowType="tukey",
85     SFTWindowParam=0.001,
86 )
87 noise_writer.make_data()
88
89 # then inject a strong signal
90 # window options *must* match those previously used for the noiseSFTs
91 add_signal_writer = pyfstat.Writer(
92     label="PyFstatTestSFTsWithAddedSignal",
93     outdir=outdir,
94     F0=F0,
95     Alpha=Alpha,
96     Delta=Delta,
97     h0=h0,
98     cosi=cosi,
99     tstart=tstart,
100    duration=duration_Tsft * Tsft,
101    Tsft=Tsft,
102    Band=Band,
103    detectors=IF0,
104    sqrtSX=0,
105    noiseSFTs=noise_writer.sftfilepath,
106    SFTWindowType="tukey",
107    SFTWindowParam=0.001,
108 )
109 add_signal_writer.make_data()
110
111 # compute Fstat
112 coherent_search = pyfstat.ComputeFstat(
113     tref=add_signal_writer.tref,
114     sftfilepattern=add_signal_writer.sftfilepath,
115     minCoverFreq=-0.5,
116     maxCoverFreq=-0.5,
117 )
118 FS_2 = coherent_search.get_fullycoherent_twoF(
119     add_signal_writer.F0,
120     add_signal_writer.F1,
121     add_signal_writer.F2,
122     add_signal_writer.Alpha,
123     add_signal_writer.Delta,
124 )
125
126 logger.info("Base case Fstat: {}".format(FS_1))

```

(continues on next page)

(continued from previous page)

```

127 logger.info("Noise + Signal Fstat: {}".format(FS_2))
128 logger.info("Relative Difference: {}".format(np.abs(FS_2 - FS_1) / FS_1))

```

Total running time of the script: (0 minutes 0.000 seconds)

3.8.4 Randomly sampling parameter space points

Application of dedicated classes to sample software injection parameters according to the specified parameter space priors.

```

8  import os
9
10 import matplotlib.pyplot as plt
11 import numpy as np
12
13 from pyfstat import (
14     AllSkyInjectionParametersGenerator,
15     InjectionParametersGenerator,
16     Writer,
17     isotropic_amplitude_distribution,
18     set_up_logger,
19 )
20
21 label = "PyFstatExampleInjectionParametersGenerator"
22 outdir = os.path.join("PyFstat_example_data", label)
23 logger = set_up_logger(label=label, outdir=outdir)
24
25 # Properties of the GW data
26 gw_data = {
27     "sqrtSX": 1e-23,
28     "tstart": 10000000000,
29     "duration": 86400,
30     "detectors": "H1,L1",
31     "Band": 1,
32     "Tsft": 1800,
33 }
34
35 logger.info("Drawing random signal parameters...")
36
37 # Draw random signal phase parameters.
38 # The AllSkyInjectionParametersGenerator covers [Alpha,Delta] priors automatically.
39 # The rest can be a mix of nontrivial prior distributions and fixed values.
40 phase_params_generator = AllSkyInjectionParametersGenerator(
41     priors={
42         "F0": {"stats.uniform": {"loc": 29.0, "scale": 2.0}},
43         "F1": -1e-10,
44         "F2": 0,
45     },
46     seed=23,
47 )
48 phase_parameters = phase_params_generator.draw()

```

(continues on next page)

(continued from previous page)

```

49 phase_parameters["tref"] = gw_data["tstart"]
50
51 # Draw random signal amplitude parameters.
52 # Here we use the plain InjectionParametersGenerator class.
53 amplitude_params_generator = InjectionParametersGenerator(
54     priors={
55         "h0": {"stats.norm": {"loc": 1e-24, "scale": 1e-26}},
56         **isotropic_amplitude_distribution,
57     },
58     seed=42,
59 )
60 amplitude_parameters = amplitude_params_generator.draw()
61
62 # Now we can pass the parameter dictionaries to the Writer class and make SFTs.
63 data = Writer(
64     label=label,
65     outdir=outdir,
66     **gw_data,
67     **phase_parameters,
68     **amplitude_parameters,
69 )
70 data.make_data()
71
72 # Now we draw many phase parameters and check the sky distribution
73 Ndraws = 10000
74 phase_parameters = phase_params_generator.draw_many(size=Ndraws)
75 Alphas = phase_parameters["Alpha"]
76 Deltas = phase_parameters["Delta"]
77 plotfile = os.path.join(outdir, label + "_allsky.png")
78 logger.info(f"Plotting sky distribution of {Ndraws} points to file: {plotfile}")
79 plt.subplot(111, projection="aitoff")
80 plt.plot(Alphas - np.pi, Deltas, ".", markersize=1)
81 plt.savefig(plotfile, dpi=300)
82 plt.close()
83 plotfile = os.path.join(outdir, label + "_alpha_hist.png")
84 logger.info(f"Plotting Alpha distribution of {Ndraws} points to file: {plotfile}")
85 plt.hist(Alphas, 50)
86 plt.xlabel("Alpha")
87 plt.ylabel("draws")
88 plt.savefig(plotfile, dpi=100)
89 plt.close()
90 plotfile = os.path.join(outdir, label + "_delta_hist.png")
91 logger.info(f"Plotting Delta distribution of {Ndraws} points to file: {plotfile}")
92 plt.hist(Deltas, 50)
93 plt.xlabel("Delta")
94 plt.ylabel("draws")
95 plt.savefig(plotfile, dpi=100)
96 plt.close()
97 plotfile = os.path.join(outdir, label + "_sindelta_hist.png")
98 logger.info(f"Plotting sin(Delta) distribution of {Ndraws} points to file: {plotfile}")
99 plt.hist(np.sin(Deltas), 50)
100 plt.xlabel("sin(Delta)")

```

(continues on next page)

(continued from previous page)

```
101 plt.ylabel("draws")
102 plt.savefig(plotfile, dpi=100)
103 plt.close()
```

Total running time of the script: (0 minutes 0.000 seconds)

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

BIBLIOGRAPHY

[JKS1998] Jaranowski, Krolak, Schuz Phys. Rev. D58 063001, 1998

PYTHON MODULE INDEX

p

- [pyfstat](#), 75
- [pyfstat.core](#), 20
- [pyfstat.grid_based_searches](#), 34
- [pyfstat.gridcorner](#), 41
- [pyfstat.injection_parameters](#), 42
- [pyfstat.logging](#), 45
- [pyfstat.make_sfts](#), 46
- [pyfstat.mcmc_based_searches](#), 55
- [pyfstat.optimal_setup_functions](#), 66
- [pyfstat.snr](#), 67
- [pyfstat.tcw_fstat_map_funcs](#), 70
- [pyfstat.utils](#), 20
 - [pyfstat.utils.atoms](#), 9
 - [pyfstat.utils.cli](#), 10
 - [pyfstat.utils.converting](#), 11
 - [pyfstat.utils.ephemeris](#), 12
 - [pyfstat.utils.formatting](#), 13
 - [pyfstat.utils.gsl](#), 14
 - [pyfstat.utils.importing](#), 14
 - [pyfstat.utils.io](#), 14
 - [pyfstat.utils.predict](#), 16
 - [pyfstat.utils.runlalsuite](#), 17
 - [pyfstat.utils.sft](#), 19

A

AllSkyInjectionParametersGenerator (class in *pyfstat.injection_parameters*), 44
 assumeSqrtSX (*pyfstat.snr.SignalToNoiseRatio* attribute), 67

B

BaseSearchClass (class in *pyfstat.core*), 20
 BinaryModulatedWriter (class in *pyfstat.make_sfts*), 49

C

calculate_fmin_Band() (*pyfstat.make_sfts.Writer* method), 48
 calculate_twoF_cumulative() (*pyfstat.core.ComputeFstat* method), 28
 call_compute_transient_fstat_map() (in module *pyfstat.tcw_fstat_map_funcs*), 73
 check_cached_data_okay_to_use() (*pyfstat.make_sfts.Writer* method), 49
 check_if_samples_are_railing() (*pyfstat.mcmc_based_searches.MCMCSearch* method), 60
 check_old_data_is_okay_to_use() (*pyfstat.grid_based_searches.GridSearch* method), 35
 compute_evidence() (*pyfstat.mcmc_based_searches.MCMCSearch* method), 61
 compute_glitch_fstat_single() (*pyfstat.core.SemiCoherentGlitchSearch* method), 34
 compute_h0_from_snr2() (*pyfstat.snr.SignalToNoiseRatio* method), 68
 compute_Mmunu() (*pyfstat.snr.SignalToNoiseRatio* method), 69
 compute_snr2() (*pyfstat.snr.SignalToNoiseRatio* method), 68
 compute_twoF() (*pyfstat.snr.SignalToNoiseRatio* method), 69
 ComputeFstat (class in *pyfstat.core*), 22

concatenate_sft_files() (*pyfstat.make_sfts.FrequencyModulatedArtifactWriter* method), 53
 convert_aPlus_aCross_to_h0_cosi() (in module *pyfstat.utils.converting*), 12
 convert_array_to_gsl_matrix() (in module *pyfstat.utils.gsl*), 14
 convert_h0_cosi_to_aPlus_aCross() (in module *pyfstat.utils.converting*), 11
 copy_FstatAtomVector() (in module *pyfstat.utils.atoms*), 10
 custom_prior() (in module *pyfstat.injection_parameters*), 42

D

DefunctClass (class in *pyfstat.core*), 34
 DeprecatedClass (class in *pyfstat.core*), 34
 detector_states (*pyfstat.snr.SignalToNoiseRatio* attribute), 67
 DetectorStates (class in *pyfstat.snr*), 69
 DMoff_NO_SPIN (class in *pyfstat.grid_based_searches*), 40
 draw() (*pyfstat.injection_parameters.InjectionParametersGenerator* method), 44
 draw_many() (*pyfstat.injection_parameters.InjectionParametersGenerator* method), 44

E

EarthTest (class in *pyfstat.grid_based_searches*), 40
 estimate_min_max_CoverFreq() (*pyfstat.core.ComputeFstat* method), 24
 export_samples_to_disk() (*pyfstat.mcmc_based_searches.MCMCSearch* method), 60
 extract_singleIFOMultiFatoms_from_multiAtoms() (in module *pyfstat.utils.atoms*), 9

F

F_mn (*pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap* attribute), 71
 fmt_detstat (*pyfstat.grid_based_searches.GridSearch* attribute), 35

FrequencyAmplitudeModulatedArtifactWriter
 (class in `pyfstat.make_sfts`), 54
 FrequencyModulatedArtifactWriter (class in `pyfstat.make_sfts`), 52
 FrequencySlidingWindow (class in `pyfstat.grid_based_searches`), 40
 from_sfts() (`pyfstat.snr.SignalToNoiseRatio` class method), 67
 fstatmap_versions (in module `pyfstat.tcw_fstat_map_funcs`), 72

G

generate_loudest() (`pyfstat.grid_based_searches.GridSearch` method), 38
 generate_loudest() (`pyfstat.mcmc_based_searches.MCMCSearch` method), 61
 generate_loudest_file() (in module `pyfstat.utils.runlalsuite`), 18
 get_commandline_from_SFTDescriptor() (in module `pyfstat.utils.sft`), 19
 get_covering_band() (in module `pyfstat.utils.runlalsuite`), 17
 get_dictionary_from_lines() (in module `pyfstat.utils.converting`), 11
 get_doppler_params_output_format() (in module `pyfstat.utils.formatting`), 13
 get_ephemeris_files() (in module `pyfstat.utils.ephemeris`), 12
 get_frequency() (`pyfstat.make_sfts.FrequencyModulatedArtifactWriter` method), 53
 get_fullycoherent_detstat() (`pyfstat.core.ComputeFstat` method), 24
 get_fullycoherent_log10BSGL() (`pyfstat.core.ComputeFstat` method), 26
 get_fullycoherent_single_IF0_twoFs() (`pyfstat.core.ComputeFstat` method), 26
 get_fullycoherent_twoF() (`pyfstat.core.ComputeFstat` method), 26
 get_h0() (`pyfstat.make_sfts.FrequencyAmplitudeModulatedArtifactWriter` method), 55
 get_h0() (`pyfstat.make_sfts.FrequencyModulatedArtifactWriter` method), 53
 get_lnBtSG() (`pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap` method), 72
 get_max_det_stat() (`pyfstat.grid_based_searches.GridSearch` method), 37
 get_max_twoF() (`pyfstat.grid_based_searches.GridSearch` method), 37
 get_max_twoF() (`pyfstat.mcmc_based_searches.MCMCSearch` method), 60
 get_maxF_idx() (`pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap` method), 72
 get_multi_detector_states() (`pyfstat.snr.DetectorStates` method), 69
 get_multi_detector_states_from_sfts() (`pyfstat.snr.DetectorStates` method), 70
 get_Nstar_estimate() (in module `pyfstat.optimal_setup_functions`), 66
 get_official_sft_filename() (in module `pyfstat.utils.sft`), 19
 get_optimal_setup() (in module `pyfstat.optimal_setup_functions`), 66
 get_output_file_header() (`pyfstat.core.BaseSearchClass` method), 21
 get_p_value() (`pyfstat.mcmc_based_searches.MCMCSearch` method), 61
 get_parameters_dict_from_file_header() (in module `pyfstat.utils.io`), 15
 get_predict_fstat_parameters_from_dict() (in module `pyfstat.utils.predict`), 17
 get_saved_data_dictionary() (`pyfstat.mcmc_based_searches.MCMCSearch` method), 60
 get_semicohherent_det_stat() (`pyfstat.core.SemiCoherentSearch` method), 31
 get_semicohherent_log10BSGL() (`pyfstat.core.SemiCoherentSearch` method), 32
 get_semicohherent_nglitch_twoF() (`pyfstat.core.SemiCoherentGlitchSearch` method), 33
 get_semicohherent_single_IF0_twoFs() (`pyfstat.core.SemiCoherentSearch` method), 32
 get_semicohherent_twoF() (`pyfstat.core.SemiCoherentSearch` method), 32
 get_sft_as_arrays() (in module `pyfstat.utils.sft`), 19
 get_summary_stats() (`pyfstat.mcmc_based_searches.MCMCSearch` method), 60
 get_transient_detstats() (`pyfstat.core.ComputeFstat` method), 27
 get_transient_fstat_map_filename() (`pyfstat.grid_based_searches.TransientGridSearch` method), 39
 get_transient_log10BSGL() (`pyfstat.core.ComputeFstat` method), 27
 get_transient_maxTwoFstat() (`pyfstat.core.ComputeFstat` method), 27
 glitch_symbol_dictionary (`pyfstat.mcmc_based_searches.MCMCGlitchSearch` attribute), 62

- GlitchWriter (class in `pyfstat.make_sfts`), 52
 gps_time_and_string_formats_as_LAL (pyfstat.make_sfts.Writer attribute), 48
 gps_to_datestr_utc() (in module `pyfstat.utils.converting`), 11
 gridcorner() (in module `pyfstat.gridcorner`), 42
 GridGlitchSearch (class in `pyfstat.grid_based_searches`), 40
 GridSearch (class in `pyfstat.grid_based_searches`), 34
 GridUniformPriorSearch (class in `pyfstat.grid_based_searches`), 40
- I**
- idx_array_slice() (in module `pyfstat.gridcorner`), 42
 init_compute_fstatistic() (pyfstat.core.ComputeFstat method), 24
 init_run_setup() (pyfstat.mcmc_based_searches.MCMCFollowUpSearch method), 63
 init_semi_coherent_parameters() (pyfstat.core.SemiCoherentSearch method), 31
 init_transient_fstat_map_features() (in module `pyfstat.tcw_fstat_map_funcs`), 72
 initializer() (in module `pyfstat.utils.importing`), 14
 InjectionParametersGenerator (class in `pyfstat.injection_parameters`), 43
- L**
- lalpulsar_compute_transient_fstat_map() (in module `pyfstat.tcw_fstat_map_funcs`), 73
 last_supported_version (pyfstat.core.DefunctClass attribute), 34
 last_supported_version (pyfstat.grid_based_searches.DMoff_NO_SPIN attribute), 41
 last_supported_version (pyfstat.grid_based_searches.EarthTest attribute), 40
 last_supported_version (pyfstat.grid_based_searches.FrequencySlidingWindow attribute), 40
 last_supported_version (pyfstat.grid_based_searches.GridUniformPriorSearch attribute), 40
 last_supported_version (pyfstat.grid_based_searches.SliceGridSearch attribute), 40
 last_supported_version (pyfstat.grid_based_searches.SlidingWindow attribute), 40
 LineWriter (class in `pyfstat.make_sfts`), 50
 lnBtSG (pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap attribute), 71
 log_mean() (in module `pyfstat.gridcorner`), 41
- M**
- make_cff() (pyfstat.make_sfts.GlitchWriter method), 52
 make_cff() (pyfstat.make_sfts.Writer method), 49
 make_data() (pyfstat.make_sfts.FrequencyModulatedArtifactWriter method), 54
 make_data() (pyfstat.make_sfts.Writer method), 49
 make_ith_sft() (pyfstat.make_sfts.FrequencyModulatedArtifactWriter method), 54
 match_commandlines() (in module `pyfstat.utils.cli`), 10
 max_slice() (in module `pyfstat.gridcorner`), 41
 maxF (pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap attribute), 71
 MCMCFollowUpSearch (class in `pyfstat.mcmc_based_searches`), 63
 MCMCGlitchSearch (class in `pyfstat.mcmc_based_searches`), 62
 MCMCSearch (class in `pyfstat.mcmc_based_searches`), 56
 MCMCSemiCoherentSearch (class in `pyfstat.mcmc_based_searches`), 62
 MCMCTransientSearch (class in `pyfstat.mcmc_based_searches`), 64
 mfd (pyfstat.make_sfts.LineWriter attribute), 51
 mfd (pyfstat.make_sfts.Writer attribute), 48
 module
 pyfstat, 75
 pyfstat.core, 20
 pyfstat.grid_based_searches, 34
 pyfstat.gridcorner, 41
 pyfstat.injection_parameters, 42
 pyfstat.logging, 45
 pyfstat.make_sfts, 46
 pyfstat.mcmc_based_searches, 55
 pyfstat.optimal_setup_functions, 66
 pyfstat.snr, 67
 pyfstat.tcw_fstat_map_funcs, 70
 pyfstat.utils, 20
 pyfstat.utils.atoms, 9
 pyfstat.utils.cli, 10
 pyfstat.utils.converting, 11
 pyfstat.utils.ephemeris, 12
 pyfstat.utils.formatting, 13
 pyfstat.utils.gsl, 14
 pyfstat.utils.importing, 14
 pyfstat.utils.io, 14
 pyfstat.utils.predict, 16
 pyfstat.utils.runlalsuite, 17
 pyfstat.utils.sft, 19
- N**
- noise_weights (pyfstat.snr.SignalToNoiseRatio attribute), 67

P

`parse_list_of_numbers()` (in module `pyfstat.utils.converting`), 11

`plot_1D()` (`pyfstat.grid_based_searches.GridSearch` method), 36

`plot_2D()` (`pyfstat.grid_based_searches.GridSearch` method), 36

`plot_chainconsumer()` (`pyfstat.mcmc_based_searches.MCMCSearch` method), 59

`plot_corner()` (`pyfstat.mcmc_based_searches.MCMCSearch` method), 58

`plot_cumulative_max()` (`pyfstat.mcmc_based_searches.MCMCGLitchSearch` method), 62

`plot_cumulative_max()` (`pyfstat.mcmc_based_searches.MCMCSearch` method), 59

`plot_prior_posterior()` (`pyfstat.mcmc_based_searches.MCMCSearch` method), 59

`plot_twoF_cumulative()` (`pyfstat.core.ComputeFstat` method), 29

`pprint_init_params_dict()` (`pyfstat.core.BaseSearchClass` method), 20

`pr_welcome` (`pyfstat.core.DefunctClass` attribute), 34

`pre_compute_evolution()` (`pyfstat.make_sfts.FrequencyModulatedArtifactWriter` method), 53

`predict_fstat()` (in module `pyfstat.utils.predict`), 16

`predict_fstat()` (`pyfstat.make_sfts.Writer` method), 49

`predict_twoF_cumulative()` (`pyfstat.core.ComputeFstat` method), 28

`print_max_twoF()` (`pyfstat.grid_based_searches.GridSearch` method), 38

`print_summary()` (`pyfstat.mcmc_based_searches.MCMCSearch` method), 61

`projection_1D()` (in module `pyfstat.gridcorner`), 42

`projection_2D()` (in module `pyfstat.gridcorner`), 42

`pycuda_compute_transient_fstat_map()` (in module `pyfstat.tcw_fstat_map_funcs`), 74

`pycuda_compute_transient_fstat_map_exp()` (in module `pyfstat.tcw_fstat_map_funcs`), 75

`pycuda_compute_transient_fstat_map_rect()` (in module `pyfstat.tcw_fstat_map_funcs`), 74

`pyfstat`
module, 75

`pyfstat.core`
module, 20

`pyfstat.grid_based_searches`
module, 34

`pyfstat.gridcorner`
module, 41

`pyfstat.injection_parameters`
module, 42

`pyfstat.logging`
module, 45

`pyfstat.make_sfts`
module, 46

`pyfstat.mcmc_based_searches`
module, 55

`pyfstat.optimal_setup_functions`
module, 66

`pyfstat.snr`
module, 67

`pyfstat.tcw_fstat_map_funcs`
module, 70

`pyfstat.utils`
module, 20

`pyfstat.utils.atoms`
module, 9

`pyfstat.utils.cli`
module, 10

`pyfstat.utils.converting`
module, 11

`pyfstat.utils.ephemeris`
module, 12

`pyfstat.utils.formatting`
module, 13

`pyfstat.utils.gsl`
module, 14

`pyfstat.utils.importing`
module, 14

`pyfstat.utils.io`
module, 14

`pyfstat.utils.predict`
module, 16

`pyfstat.utils.runlalsuite`
module, 17

`pyfstat.utils.sft`
module, 19

`pyTransientFstatMap` (class in `pyfstat.tcw_fstat_map_funcs`), 70

R

`read_evidence_file_to_dict()` (`pyfstat.mcmc_based_searches.MCMCSearch` static method), 61

`read_from_file()` (`pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap` method), 71

`read_par()` (in module `pyfstat.utils.io`), 14

`read_par()` (`pyfstat.core.BaseSearchClass` method), 21

`read_parameters_dict_lines_from_file_header()` (in module `pyfstat.utils.io`), 15

`read_setup_input_file()` (pyfs-
tat.mcmc_based_searches.MCMCFollowUpSearch
method), 64
`read_txt_file_with_header()` (in module *pyfs-*
tat.utils.io), 14
`required_signal_parameters` (pyfs-
tat.make_sfts.LineWriter attribute), 52
`required_signal_parameters` (pyfs-
tat.make_sfts.Writer attribute), 48
`reshape_FstatAtomsVector()` (in module *pyfs-*
tat.tcw_fstat_map_funcs), 74
`round_to_n()` (in module *pyfstat.utils.formatting*), 13
`run()` (*pyfstat.grid_based_searches.GridSearch*
method), 35
`run()` (*pyfstat.grid_based_searches.TransientGridSearch*
method), 39
`run()` (*pyfstat.mcmc_based_searches.MCMCFollowUpSearch*
method), 63
`run()` (*pyfstat.mcmc_based_searches.MCMCSearch*
method), 58
`run_commandline()` (in module *pyfstat.utils.cli*), 10
`run_makefakedata()` (*pyfstat.make_sfts.Writer*
method), 49
`run_makefakedata_v4()` (pyfs-
tat.make_sfts.FrequencyModulatedArtifactWriter
method), 54

S

`safe_X_less_plt()` (in module *pyfstat.utils.importing*),
14
`save_array_to_disk()` (pyfs-
tat.grid_based_searches.GridSearch *method*),
36
`SearchForSignalWithJumps` (class in *pyfstat.core*), 33
`SemiCoherentGlitchSearch` (class in *pyfstat.core*), 33
`SemiCoherentSearch` (class in *pyfstat.core*), 30
`set_ephemeris_files()` (pyfs-
tat.core.BaseSearchClass *method*), 20
`set_out_file()` (pyfs-
tat.grid_based_searches.GridSearch *method*),
38
`set_up_logger()` (in module *pyfstat.logging*), 45
`setup_initialisation()` (pyfs-
tat.mcmc_based_searches.MCMCSearch
method), 57
`signal_parameter_labels` (*pyfstat.make_sfts.Writer*
attribute), 48
`signal_parameters_labels` (pyfs-
tat.make_sfts.LineWriter attribute), 52
`SignalToNoiseRatio` (class in *pyfstat.snr*), 67
`SliceGridSearch` (class in *pyfs-*
tat.grid_based_searches), 40
`SlidingWindow` (class in *pyfstat.grid_based_searches*),
40

`symbol_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCGlitchSearch
attribute), 62
`symbol_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCSearch
attribute), 57
`symbol_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCTransientSearch
attribute), 65

T

`t0_ML` (*pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap*
attribute), 71
`tau_ML` (*pyfstat.tcw_fstat_map_funcs.pyTransientFstatMap*
attribute), 71
`tend` (*pyfstat.make_sfts.Writer* property), 48
`tex_labels` (*pyfstat.grid_based_searches.GridSearch*
attribute), 35
`tex_labels0` (*pyfstat.grid_based_searches.GridSearch*
attribute), 35
`texify_float()` (in module *pyfstat.utils.formatting*), 13
`transform_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCGlitchSearch
attribute), 62
`transform_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCSearch
attribute), 57
`transform_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCTransientSearch
attribute), 65
`TransientGridSearch` (class in *pyfs-*
tat.grid_based_searches), 38
`translate_keys_to_lal()` (pyfs-
tat.core.BaseSearchClass *static* *method*),
21

U

`uniform_sky_declination()` (in module *pyfs-*
tat.injection_parameters), 43
`unit_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCGlitchSearch
attribute), 62
`unit_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCSearch
attribute), 57
`unit_dictionary` (pyfs-
tat.mcmc_based_searches.MCMCTransientSearch
attribute), 65

W

`write_atoms_to_file()` (*pyfstat.core.ComputeFstat*
method), 30

`write_evidence_file_from_dict()` (*pyfs-*
tat.mcmc_based_searches.MCMCSearch
method), [61](#)

`write_F_mn_to_file()` (*pyfs-*
tat.tcw_fstat_map_funcs.pyTransientFstatMap
method), [72](#)

`write_par()` (*pyfstat.mcmc_based_searches.MCMCSearch*
method), [61](#)

`write_prior_table()` (*pyfs-*
tat.mcmc_based_searches.MCMCSearch
method), [61](#)

`Writer` (*class in pyfstat.make_sfts*), [46](#)