# GWIn Documentation

***Release 0+untagged.322.g6e8d8e3***

**Collin Capano**

**Aug 03, 2018**

# Contents:

CHAPTER 1

# File input/output

## 1.1 HDF output file handler (`gwin.io.InferenceFile`)

The executable `gwin` will write a HDF file with all the samples from each walker along with the PSDs and some meta-data about the sampler. To read the output file:

```
from gwin.io import InferenceFile
fp = InferenceFile("example.h5", "r")
```

To get all samples for `distance` from the first walker you can do:

```
samples = fp.read_samples("distance", walkers=0)
print(samples.distance)
```

The *`InferenceFile.read_samples()`* method includes the options to thin the samples. By default the function will return samples beginning at the end of the burn-in to the last written sample, and will use the autocorrelation length (ACL) calculated by `gwin` to select the indepdedent samples. You can supply `thin_start`, `thin_end`, and `thin_interval` to override this. To read all samples you would do:

```
samples = fp.read_samples("distance", walkers=0,
                          thin_start=0, thin_end=-1, thin_interval=1)
print(samples.distance)
```

Some standard parameters that are derived from the variable arguments (listed via *`fp.variable_params`*) can also be retrieved. For example, if `fp.variable_params` includes `'mass1'` and `'mass2'`, then you can retrieve the chirp mass with:

```
samples = fp.read_samples("mchirp")
print(samples.mchirp)
```

Some standard parameters that are derived from the variable arguments (listed via *`fp.variable_params`*) can also be retrieved. For example, if `fp.variable_params` includes `'mass1'` and `'mass2'`, then you can retrieve the chirp mass with:

```
samples = fp.read_samples("mchirp")
print(samples.mchirp)
```

In this case, *fp.read_samples* will retrieve `mass1` and `mass2` (since they are needed to compute chirp mass); `samples.mchirp` then returns an array of the chirp mass computed from `mass1` and `mass2`.

For more information, including the list of predefined derived parameters, see the class reference for *InferenceFile*.

## 1.2 API Reference

| | |
|---|---|
| *InferenceFile*(path[, mode]) | A subclass of the h5py.File object that has extra functions for handling reading and writing the samples from the samplers. |
| *InferenceTXTFile*(path[, mode, delimiter]) | A class that has extra functions for handling reading the samples from posterior-only TXT files. |

Full module documentation is available at:

CHAPTER 2

# gwin package

## 2.1 Subpackages

### 2.1.1 gwin.io package

#### Submodules

#### gwin.io.hdf module

This modules defines functions for reading and writing samples that the inference samplers generate.

**class** gwin.io.hdf.**InferenceFile**(*path*, *mode=None*, ***kwargs*)
Bases: h5py._hl.files.File

A subclass of the h5py.File object that has extra functions for handling reading and writing the samples from the samplers.

**Parameters**

- **path** (*str*) – The path to the HDF file.
- **mode** (*{None, str}*) – The mode to open the file, eg. "w" for write and "r" for read.

**acl**
Returns the saved autocorelation length (ACL).

**Returns** **acl** – The ACL.

**Return type** {int, float}

**burn_in_iterations**
Returns number of iterations in the burn in.

**cmd**
Returns the (last) saved command line.

If the file was created from a run that resumed from a checkpoint, only the last command line used is returned.

**Returns** **cmd** – The command line that created this InferenceFile.

**Return type** string

**copy**(*other*, *parameters=None*, *parameter_names=None*, *posterior_only=False*, ***kwargs*)
Copies data in this file to another file.

The samples and stats to copy may be down selected using the given kwargs. All other data (the "metadata") are copied exactly.

**Parameters**

- **other** (*str or InferenceFile*) – The file to write to. May be either a string giving a filename, or an open hdf file. If the former, the file will be opened with the write attribute (note that if a file already exists with that name, it will be deleted).
- **parameters** (*list of str, optional*) – List of parameters to copy. If None, will copy all parameters.
- **parameter_names** (*dict, optional*) – Rename one or more parameters to the given name. The dictionary should map parameter -> parameter name. If None, will just use the original parameter names.
- **posterior_only** (*bool, optional*) – Write the samples and model stats as flattened arrays, and set other's posterior_only attribute. For example, if this file has a parameter's samples written to `{samples_group}/{param}/walker{x}`, then other will have all of the selected samples from all walkers written to `{samples_group}/{param}/`.
- ****kwargs** – All other keyword arguments are passed to *read_samples*.

**Returns** The open file handler to other.

**Return type** *InferenceFile*

**copy_metadata**(*other*)
Copies all metadata from this file to the other file.

Metadata is defined as all data that is not in either the samples or stats group.

**Parameters** **other** (InferenceFile) – An open inference file to write the data to.

**get_slice**(*thin_start=None*, *thin_interval=None*, *thin_end=None*)
Formats a slice using the given arguments that can be used to retrieve a thinned array from an InferenceFile.

**Parameters**

- **thin_start** (*{None, int}*) – The starting index to use. If None, will try to retrieve the *burn_in_iterations* from the given file. If no *burn_in_iterations* exists, will default to the start of the array.
- **thin_interval** (*{None, int}*) – The interval to use. If None, will try to retrieve the acl from the given file. If no acl attribute exists, will default to 1.
- **thin_end** (*{None, int}*) – The end index to use. If None, will retrieve to the end of the array.

**Returns** The slice needed.

**Return type** slice

**is_burned_in**
Returns whether or not the sampler is burned in.

**log_evidence**
Returns the log of the evidence and its error, if they exist in the file. Raises a KeyError otherwise.

**lognl**
Returns the log noise likelihood.

**model_name**
Returns the name of the model that was used.

**n_independent_samples**
Returns the number of independent samples stored in the file.

**name = 'hdf'**

**niterations**
Returns number of iterations performed.

**Returns** **niterations** – Number of iterations performed.

**Return type** int

**ntemps**
Returns number of temperatures used.

**nwalkers**
Returns number of walkers used.

**Returns** **nwalkesr** – Number of walkers used.

**Return type** int

**posterior_only**
Whether the file only contains flattened posterior samples.

**read_acceptance_fraction**(***kwargs*)
Returns the acceptance fraction that was written to the file.

**Parameters** ****kwargs** – All keyword arguments are passed to the sampler's *read_acceptance_fraction* function.

**Returns** The acceptance fraction.

**Return type** numpy.array

**read_acls**()
Returns all of the individual chains' acls. See the *read_acls* function of this file's sampler for more details.

**read_label**(*parameter*, *error_on_none=False*)
Returns the label for the parameter.

**Parameters**

- **parameter** (*str*) – Name of parameter to get a label for. Will first try to retrieve a label from this file's "label" attributes. If the parameter is not found there, will look for a label from pycbc.waveform.parameters.
- **error_on_none** (*{False, bool}*) – If True, will raise a ValueError if a label cannot be found, or if the label is None. Otherwise, the parameter will just be returned if no label can be found.

**Returns** **label** – A formatted string for the name of the paramter.

**Return type** str

**read_model_stats**(***kwargs*)
Reads model stats from self.

**Parameters** ****kwargs** – The keyword args are passed to the sampler's `read_model_stats` method.

**Returns** **stats** – Likelihood stats in the file, as a FieldArray. The fields of the array are the names of the stats that are in the `model_stats` group.

**Return type** {FieldArray, None}

**read_random_state**(*group=None*)
Reads the state of the random number generator from the file.

**Parameters** **group** (*str*) – Name of group to read random state from.

**Returns** A tuple with 5 elements that can be passed to numpy.set_state.

**Return type** tuple

**read_samples**(*parameters*, *samples_group=None*, ***kwargs*)
Reads samples from the file.

**Parameters**

- **parameters** (*(list of) strings*) – The parameter(s) to retrieve. A parameter can be the name of any field in *samples_group*, a virtual field or method of `FieldArray` (as long as the file contains the necessary fields to derive the virtual field or method), and/or a function of these.
- **samples_group** (*str*) – Group in HDF InferenceFile that parameters belong to.
- ****kwargs** – The rest of the keyword args are passed to the sampler's *read_samples* method.

**Returns** Samples for the given parameters, as an instance of a FieldArray.

**Return type** FieldArray

**resume_points**
The iterations at which a run was resumed from checkpoint.

**Returns** **resume_points** – An array of integers giving the points at which the run resumed.

**Return type** array or None

**Raises** `KeyError` – If the run never resumed from a checkpoint.

**sampler_class**
Returns the sampler class that was used.

**sampler_group = 'sampler_states'**

**sampler_name**
Returns the name of the sampler that was used.

**samples_group = 'samples'**

**samples_parser**
Returns the class to use to read/write samples from/to the file.

**sampling_params**
Returns the parameters that were used to sample.

**Returns** **sampling_params** – List of the sampling params.

**Return type** {list, str}

**static_params**
Returns a dictionary of the static_params. The keys are the argument names, values are the value they were set to.

**stats_group = 'model_stats'**

**variable_params**
Returns list of variable_params.

**Returns** **variable_params** – List of str that contain variable_params keys.

**Return type** {list, str}

**write_command_line**()
Writes command line to attributes.

The command line is written to the file's `attrs['cmd']`. If this attribute already exists in the file (this can happen when resuming from a checkpoint), `attrs['cmd']` will be a list storing the current command line and all previous command lines.

**write_data**(*strain_dict=None*, *stilde_dict=None*, *psd_dict=None*, *low_frequency_cutoff_dict=None*, *group=None*)
Writes the strain/stilde/psd.

**Parameters**

- **strain_dict** (*{None, dict}*) – A dictionary of strains. If None, no strain will be written.
- **stilde_dict** (*{None, dict}*) – A dictionary of stilde. If None, no stilde will be written.
- **psd_dict** (*{None, dict}*) – A dictionary of psds. If None, no psds will be written.
- **low_freuency_cutoff_dict** (*{None, dict}*) – A dictionary of low frequency cutoffs used for each detector in `psd_dict`; must be provided if `psd_dict` is not None.
- **group** (*{None, str}*) – The group to write the strain to. If None, will write to the top level.

**write_injections**(*injection_file*, *ifo*)
Writes injection parameters for an IFO to file.

**Parameters**

- **injection_file** (*str*) – Path to HDF injection file.
- **ifo** (*str*) – IFO name.

**write_psd**(*psds*, *low_frequency_cutoff*, *group=None*)
Writes PSD for each IFO to file.

**Parameters**

- **psds** (*{dict, FrequencySeries}*) – A dict of FrequencySeries where the key is the IFO.
- **low_frequency_cutoff** (*{dict, float}*) – A dict of the low-frequency cutoff where the key is the IFO. The minimum value will be stored as an attr in the File.
- **group** (*{None, str}*) – The group to write the strain to. If None, will write to the top level.

**write_random_state**(*group=None*, *state=None*)
Writes the state of the random number generator from the file.

**Parameters**

- **group** (*str*) – Name of group to read random state to.
- **state** (*tuple, optional*) – Specify the random state to write. If None, will use `numpy.random.get_state()`.

**write_resume_point**()
Keeps a list of the number of iterations that were in a file when a run was resumed from a checkpoint.

**write_stilde**(*stilde_dict*, *group=None*)
Writes stilde for each IFO to file.

**Parameters**

- **stilde** (*{dict, FrequencySeries}*) – A dict of FrequencySeries where the key is the IFO.
- **group** (*{None, str}*) – The group to write the strain to. If None, will write to the top level.

**write_strain**(*strain_dict*, *group=None*)
Writes strain for each IFO to file.

**Parameters**

- **strain** (*{dict, FrequencySeries}*) – A dict of FrequencySeries where the key is the IFO.
- **group** (*{None, str}*) – The group to write the strain to. If None, will write to the top level.

gwin.io.hdf.**check_integrity**(*filename*)
Checks the integrity of an InferenceFile.

Checks done are:

- can the file open?
- do all of the datasets in the samples group have the same shape?
- can the first and last sample in all of the datasets in the samples group be read?

If any of these checks fail, an IOError is raised.

**Parameters** **filename** (*str*) – Name of an InferenceFile to check.

**Raises**

- ValueError – If the given file does not exist.
- KeyError – If the samples group does not exist.
- IOError – If any of the checks fail.

### gwin.io.txt module

This modules defines functions for reading and samples that the inference samplers generate and are stored in an ASCII TXT file.

**class** gwin.io.txt.**InferenceTXTFile**(*path*, *mode=None*, *delimiter=None*)
Bases: object

A class that has extra functions for handling reading the samples from posterior-only TXT files.

**Parameters**

- **path** (*str*) – The path to the TXT file.
- **mode** (*{None, str}*) – The mode to open the file. Only accepts "r" or "rb" for reading.
- **delimiter** (*str*) – Delimiter to use for TXT file. Default is space-delimited.

**comments = ''**

**delimiter = ' '**

**name = 'txt'**

**classmethod write**(*output_file*, *samples*, *labels*, *delimiter=None*)
Writes a text file with samples.

**Parameters**

- **output_file** (*str*) – The path of the file to write.
- **samples** (*FieldArray*) – Samples to write to file.
- **labels** (*list*) – A list of strings to include as header in TXT file.
- **delimiter** (*str*) – Delimiter to use in TXT file.

#### Module contents

I/O utilities for GWIn

### 2.1.2 gwin.models package

#### Submodules

#### gwin.models.analytic module

This modules provides models that have analytic solutions for the log likelihood.

**class** gwin.models.analytic.**TestEggbox**(*variable_params*, ***kwargs*)
Bases: *gwin.models.base.BaseModel*

The test distribution is an 'eggbox' function:

$$\log \mathcal{L}(\Theta) = \left[2 + \prod_{i=1}^{n} \cos\left(\frac{\theta_i}{2}\right)\right]^5$$

The number of dimensions is set by the number of `variable_params` that are passed.

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied.
- ****kwargs** – All other keyword arguments are passed to `BaseModel`.

**name = 'test_eggbox'**

**class** gwin.models.analytic.**TestNormal**(*variable_params*, *mean=None*, *cov=None*, ***kwargs*)

Bases: *gwin.models.base.BaseModel*

The test distribution is an multi-variate normal distribution.

The number of dimensions is set by the number of `variable_params` that are passed. For details on the distribution used, see `scipy.stats.multivariate_normal`.

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied.
- **mean** (*array-like, optional*) – The mean values of the parameters. If None provide, will use 0 for all parameters.
- **cov** (*array-like, optional*) – The covariance matrix of the parameters. If None provided, will use unit variance for all parameters, with cross-terms set to 0.
- ****kwargs** – All other keyword arguments are passed to `BaseModel`.

### Examples

Create a 2D model with zero mean and unit variance:

```
>>> m = TestNormal(['x', 'y'])
```

Set the current parameters and evaluate the log posterior:

```
>>> m.update(x=-0.2, y=0.1)
>>> m.logposterior
-1.8628770664093453
```

See the current stats that were evaluated:

```
>>> m.current_stats
{'logjacobian': 0.0, 'loglikelihood': -1.8628770664093453, 'logprior': 0.0}
```

**name = 'test_normal'**

**class** gwin.models.analytic.**TestRosenbrock**(*variable_params*, ***kwargs*)

Bases: *gwin.models.base.BaseModel*

The test distribution is the Rosenbrock function:

$$\log \mathcal{L}(\Theta) = -\sum_{i=1}^{n-1}[(1-\theta_i)^2 + 100(\theta_{i+1} - \theta_i^2)^2]$$

The number of dimensions is set by the number of `variable_params` that are passed.

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied.
- ****kwargs** – All other keyword arguments are passed to `BaseModel`.

**name = 'test_rosenbrock'**

**class** gwin.models.analytic.**TestVolcano**(*variable_params*, ***kwargs*)
Bases: *gwin.models.base.BaseModel*

The test distribution is a two-dimensional 'volcano' function:

$$\Theta = \sqrt{\theta_1^2 + \theta_2^2} \log \mathcal{L}(\Theta) = 25 \left( e^{\frac{-\Theta}{35}} + \frac{1}{2\sqrt{2\pi}} e^{-\frac{(\Theta-5)^2}{8}} \right)$$

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied. Must have length 2.
- ****kwargs** – All other keyword arguments are passed to `BaseModel`.

**name = 'test_volcano'**

## gwin.models.base module

Base class for models.

**class** gwin.models.base.**BaseModel**(*variable_params*, *static_params=None*, *prior=None*, *sampling_transforms=None*)
Bases: `object`

Base class for all models.

Given some model $h$ with parameters $\Theta$, Bayes Theorem states that the probability of observing parameter values $\vartheta$ is:

$$p(\vartheta|h) = \frac{p(h|\vartheta)p(\vartheta)}{p(h)}.$$

Here:

- $p(\vartheta|h)$ is the **posterior** probability;
- $p(h|\vartheta)$ is the **likelihood**;
- $p(\vartheta)$ is the **prior**;
- $p(h)$ is the **evidence**.

This class defines properties and methods for evaluating the log likelihood, log prior, and log posteror. A set of parameter values is set using the `update` method. Calling the class's `log(likelihood|prior|posterior)` properties will then evaluate the model at those parameter values.

Classes that inherit from this class must implement a `_loglikelihood` function that can be called by `loglikelihood`.

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied.
- **static_params** (*dict, optional*) – A dictionary of parameter names -> values to keep fixed.
- **prior** (*callable, optional*) – A callable class or function that computes the log of the prior. If None provided, will use `_noprior`, which returns 0 for all parameter values.
- **sampling_params** (*list, optional*) – Replace one or more of the `variable_params` with the given parameters for sampling.

- **replace_parameters** (*list, optional*) – The `variable_params` to replace with sampling parameters. Must be the same length as `sampling_params`.
- **sampling_transforms** (*list, optional*) – List of transforms to use to go between the `variable_params` and the sampling parameters. Required if `sampling_params` is not None.
- **Properties** –
- **----------** –
- **logjacobian** – Returns the log of the jacobian needed to go from the parameter space of the `variable_params` to the sampling params.
- **logprior** – Returns the log of the prior.
- **loglikelihood** – A function that returns the log of the likelihood function.
- **logposterior** – A function that returns the log of the posterior.
- **loglr** – A function that returns the log of the likelihood ratio.
- **logplr** – A function that returns the log of the prior-weighted likelihood ratio.

**current_params**

**current_stats**
Return the `default_stats` as a dict.

This does no computation. It only returns what has already been calculated. If a stat hasn't been calculated, it will be returned as `numpy.nan`.

**Returns** Dictionary of stat names -> current stat values.

**Return type** dict

**default_stats**
The stats that `get_current_stats` returns by default.

**static extra_args_from_config**(*cp*, *section*, *skip_args=None*, *dtypes=None*)
Gets any additional keyword in the given config file.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- **section** (*str*) – The name of the section to read.
- **skip_args** (*list of str, optional*) – Names of arguments to skip.
- **dtypes** (*dict, optional*) – A dictionary of arguments -> data types. If an argument is found in the dict, it will be cast to the given datatype. Otherwise, the argument's value will just be read from the config file (and thus be a string).

**Returns** Dictionary of keyword arguments read from the config file.

**Return type** dict

**classmethod from_config**(*cp*, ***kwargs*)
Initializes an instance of this class from the given config file.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- ****kwargs** – All additional keyword arguments are passed to the class. Any provided keyword will over ride what is in the config file.

**get_current_stats**(*names=None*)
Return one or more of the current stats as a tuple.

This function does no computation. It only returns what has already been calculated. If a stat hasn't been calculated, it will be returned as `numpy.nan`.

**Parameters** **names** (*list of str, optional*) – Specify the names of the stats to retrieve. If `None` (the default), will return `default_stats`.

**Returns** The current values of the requested stats, as a tuple. The order of the stats is the same as the names.

**Return type** tuple

**logjacobian**
The log jacobian of the sampling transforms at the current postion.

If no sampling transforms were provided, will just return 0.

**Parameters** ****params** – The keyword arguments should specify values for all of the variable args and all of the sampling args.

**Returns** The value of the jacobian.

**Return type** float

**loglikelihood**
The log likelihood at the current parameters.

This will initially try to return the `current_stats.loglikelihood`. If that raises an `AttributeError`, will call `_loglikelihood`` to calculate it and store it to `current_stats`.

**logposterior**
Returns the log of the posterior of the current parameter values.

The logprior is calculated first. If the logprior returns `-inf` (possibly indicating a non-physical point), then the `loglikelihood` is not called.

**logprior**
Returns the log prior at the current parameters.

**name = None**

**static prior_from_config**(*cp*, *variable_params*, *prior_section*, *constraint_section*)
Gets arguments and keyword arguments from a config file.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- **variable_params** (*list*) – List of of model parameter names.
- **prior_section** (*str*) – Section to read prior(s) from.
- **constraint_section** (*str*) – Section to read constraint(s) from.

**Returns** The prior.

**Return type** pycbc.distributions.JointDistribution

**prior_rvs**(*size=1*, *prior=None*)
Returns random variates drawn from the prior.

If the `sampling_params` are different from the `variable_params`, the variates are transformed to the *sampling_params* parameter space before being returned.

**Parameters**

- **size** (*int, optional*) – Number of random values to return for each parameter. Default is 1.
- **prior** (*JointDistribution, optional*) – Use the given prior to draw values rather than the saved prior.

**Returns** A field array of the random values.

**Return type** FieldArray

**sampling_params**
Returns the sampling parameters.

If `sampling_transforms` is None, this is the same as the `variable_params`.

**static_params**
Returns the model's static arguments.

**update**(***params*)
Updates the current parameter positions and resets stats.

If any sampling transforms are specified, they are applied to the params before being stored.

**variable_params**
Returns the model parameters.

**class** gwin.models.base.**ModelStats**
Bases: object

Class to hold model's current stat values.

**getstats**(*names*, *default=nan*)
Get the requested stats as a tuple.

If a requested stat is not an attribute (implying it hasn't been stored), then the default value is returned for that stat.

**Parameters**

- **names** (*list of str*) – The names of the stats to get.
- **default** (*float, optional*) – What to return if a requested stat is not an attribute of self. Default is `numpy.nan`.

**Returns** A tuple of the requested stats.

**Return type** tuple

**getstatsdict**(*names*, *default=nan*)
Get the requested stats as a dictionary.

If a requested stat is not an attribute (implying it hasn't been stored), then the default value is returned for that stat.

**Parameters**

- **names** (*list of str*) – The names of the stats to get.
- **default** (*float, optional*) – What to return if a requested stat is not an attribute of self. Default is `numpy.nan`.

**Returns** A dictionary of the requested stats.

**Return type** dict

**statnames**
Returns the names of the stats that have been stored.

**class** gwin.models.base.**SamplingTransforms**(*variable_params*, *sampling_params*, *replace_parameters*, *sampling_transforms*)

Bases: object

Provides methods for transforming between sampling parameter space and model parameter space.

**apply**(*samples*, *inverse=False*)

Applies the sampling transforms to the given samples.

**Parameters**

- **samples** (*dict or FieldArray*) – The samples to apply the transforms to.
- **inverse** (*bool, optional*) – Whether to apply the inverse transforms (i.e., go from the sampling args to the variable_params). Default is False.

**Returns** The transformed samples, along with the original samples.

**Return type** dict or FieldArray

**classmethod from_config**(*cp*, *variable_params*)

Gets sampling transforms specified in a config file.

Sampling parameters and the parameters they replace are read from the sampling_params section, if it exists. Sampling transforms are read from the sampling_transforms section(s), using transforms.read_transforms_from_config.

An AssertionError is raised if no sampling_params section exists in the config file.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- **variable_params** (*list*) – List of parameter names of the original variable params.

**Returns** A sampling transforms class.

**Return type** *SamplingTransforms*

**logjacobian**(***params*)

Returns the log of the jacobian needed to transform pdfs in the variable_params parameter space to the sampling_params parameter space.

Let $\mathbf{x}$ be the set of variable parameters, $\mathbf{y} = f(\mathbf{x})$ the set of sampling parameters, and $p_x(\mathbf{x})$ a probability density function defined over $\mathbf{x}$. The corresponding pdf in $\mathbf{y}$ is then:

$$p_y(\mathbf{y}) = p_x(\mathbf{x}) \left|\det \mathbf{J}_{ij}\right| ,$$

where $\mathbf{J}_{ij}$ is the Jacobian of the inverse transform $\mathbf{x} = g(\mathbf{y})$. This has elements:

$$\mathbf{J}_{ij} = \frac{\partial g_i}{\partial y_j}$$

This function returns $\log \left|\det \mathbf{J}_{ij}\right|$.

**Parameters** ****params** – The keyword arguments should specify values for all of the variable args and all of the sampling args.

**Returns** The value of the jacobian.

**Return type** float

gwin.models.base.**read_sampling_params_from_config**(*cp*, *section_group=None*, *section='sampling_params'*)

Reads sampling parameters from the given config file.

Parameters are read from the `[({section_group}_){section}]` section. The options should list the variable args to transform; the parameters they point to should list the parameters they are to be transformed to for sampling. If a multiple parameters are transformed together, they should be comma separated. Example:

```
[sampling_params]
mass1, mass2 = mchirp, logitq
spin1_a = logitspin1_a
```

Note that only the final sampling parameters should be listed, even if multiple intermediate transforms are needed. (In the above example, a transform is needed to go from mass1, mass2 to mchirp, q, then another one needed to go from q to logitq.) These transforms should be specified in separate sections; see `transforms.read_transforms_from_config` for details.

**Parameters**

- **cp** (*WorkflowConfigParser*) – An open config parser to read from.
- **section_group** (*str, optional*) – Append `{section_group}_` to the section name. Default is None.
- **section** (*str, optional*) – The name of the section. Default is 'sampling_params'.

**Returns**

- **sampling_params** (*list*) – The list of sampling parameters to use instead.
- **replaced_params** (*list*) – The list of variable args to replace in the sampler.

### gwin.models.base_data module

Base classes for models with data.

**class** gwin.models.base_data.**BaseDataModel**(*variable_params*, *data*, *waveform_generator*, *waveform_transforms=None*, ***kwargs*)

Bases: *gwin.models.base.BaseModel*

Base class for models that require data and a waveform generator.

This adds propeties for the log of the likelihood that the data contain noise, `lognl`, and the log likelihood ratio `loglr`.

Classes that inherit from this class must define `_loglr` and `_lognl` functions, in addition to the `_loglikelihood` requirement inherited from `BaseModel`.

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied.
- **data** (*dict*) – A dictionary of data, in which the keys are the detector names and the values are the data.
- **waveform_generator** (*generator class*) – A generator class that creates waveforms.
- **waveform_transforms** (*list, optional*) – List of transforms to use to go from the variable args to parameters understood by the waveform generator.
- ****kwargs** – All other keyword arguments are passed to `BaseModel`.

**waveform_generator**
*dict* – The waveform generator that the class was initialized with.

**data**
*dict* – The data that the class was initialized with.

**Properties**

**----------**

**lognl**
Returns the log likelihood of the noise.

**loglr**
Returns the log of the likelihood ratio.

**logplr**
Returns the log of the prior-weighted likelihood ratio.

**See ``BaseModel`` for additional attributes and properties.**

**data**
Returns the data that was set.

**classmethod from_config**(*cp*, *data*, *delta_f=None*, *delta_t=None*, *gates=None*, *recalibration=None*, ***kwargs*)
Initializes an instance of this class from the given config file.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- **data** (*dict*) – A dictionary of data, in which the keys are the detector names and the values are the data. This is not retrieved from the config file, and so must be provided.
- **delta_f** (*float*) – The frequency spacing of the data; needed for waveform generation.
- **delta_t** (*float*) – The time spacing of the data; needed for time-domain waveform generators.
- **recalibration** (*dict of pycbc.calibration.Recalibrate, optional*) – Dictionary of detectors -> recalibration class instances for recalibrating data.
- **gates** (*dict of tuples, optional*) – Dictionary of detectors -> tuples of specifying gate times. The sort of thing returned by `pycbc.gate.gates_from_cli`.
- ****kwargs** – All additional keyword arguments are passed to the class. Any provided keyword will over ride what is in the config file.

**loglr**
The log likelihood ratio at the current parameters.

This will initially try to return the `current_stats.loglr`. If that raises an `AttributeError`, will call `_loglr`` to calculate it and store it to `current_stats`.

**lognl**
The log likelihood of the model assuming the data is noise.

This will initially try to return the `current_stats.lognl`. If that raises an `AttributeError`, will call `_lognl`` to calculate it and store it to `current_stats`.

**logplr**
Returns the log of the prior-weighted likelihood ratio at the current parameter values.

The logprior is calculated first. If the logprior returns `-inf` (possibly indicating a non-physical point), then `loglr` is not called.

**waveform_generator**
Returns the waveform generator that was set.

## gwin.models.gaussian_noise module

This module provides model classes that assume the noise is Gaussian.

**class** gwin.models.gaussian_noise.**GaussianNoise**(*variable_params*, *data*, *waveform_generator*, *f_lower*, *psds=None*, *f_upper=None*, *norm=None*, ***kwargs*)

Bases: *gwin.models.base_data.BaseDataModel*

Model that assumes data is stationary Gaussian noise.

With Gaussian noise the log likelihood functions for signal $\log p(d|\Theta)$ and for noise $\log p(d|n)$ are given by:

$$\log p(d|\Theta) = -\frac{1}{2}\sum_i \langle h_i(\Theta) - d_i | h_i(\Theta) - d_i \rangle$$

$$\log p(d|n) = -\frac{1}{2}\sum_i \langle d_i | d_i \rangle$$

where the sum is over the number of detectors, $d_i$ is the data in each detector, and $h_i(\Theta)$ is the model signal in each detector. The inner product is given by:

$$\langle a | b \rangle = 4\Re \int_0^\infty \frac{\tilde{a}(f)\tilde{b}(f)}{S_n(f)} \mathrm{d}f,$$

where $S_n(f)$ is the PSD in the given detector.

Note that the log prior-weighted likelihood ratio has one fewer term than the log posterior, since the $\langle d_i | d_i \rangle$ term cancels in the likelihood ratio:

$$\log \hat{\mathcal{L}} = \log p(\Theta) + \sum_i \left[ \langle h_i(\Theta) | d_i \rangle - \frac{1}{2} \langle h_i(\Theta) | h_i(\Theta) \rangle \right]$$

Upon initialization, the data is whitened using the given PSDs. If no PSDs are given the data and waveforms returned by the waveform generator are assumed to be whitened. The likelihood function of the noise,

$$p(d|n) = \frac{1}{2}\sum_i \langle d_i | d_i \rangle ,$$

is computed on initialization and stored as the `lognl` attribute.

By default, the data is assumed to be equally sampled in frequency, but unequally sampled data can be supported by passing the appropriate normalization using the `norm` keyword argument.

For more details on initialization parameters and definition of terms, see `BaseModel`.

**Parameters**

- **variable_params** (*(tuple of) string(s)*) – A tuple of parameter names that will be varied.
- **waveform_generator** (*generator class*) – A generator class that creates waveforms. This must have a `generate` function which takes parameter values as keyword arguments, a detectors attribute which is a dictionary of detectors keyed by their names, and an epoch which specifies the start time of the generated waveform.

- **data** (*dict*) – A dictionary of data, in which the keys are the detector names and the values are the data (assumed to be unwhitened). The list of keys must match the waveform generator's detectors keys, and the epoch of every data set must be the same as the waveform generator's epoch.
- **f_lower** (*float*) – The starting frequency to use for computing inner products.
- **psds** (*{None, dict}*) – A dictionary of FrequencySeries keyed by the detector names. The dictionary must have a psd for each detector specified in the data dictionary. If provided, the inner products in each detector will be weighted by 1/psd of that detector.
- **f_upper** (*{None, float}*) – The ending frequency to use for computing inner products. If not provided, the minimum of the largest frequency stored in the data and a given waveform will be used.
- **norm** (*{None, float or array}*) – An extra normalization weight to apply to the inner products. Can be either a float or an array. If `None`, `4*data.values()[0].delta_f` will be used.
- ****kwargs** – All other keyword arguments are passed to `BaseDataModel`.

### Examples

Create a signal, and set up the model using that signal:

```
>>> from pycbc import psd as pypsd
>>> from pycbc.waveform.generator import (FDomainDetFrameGenerator,
...                                       FDomainCBCGenerator)
>>> import gwin
>>> seglen = 4
>>> sample_rate = 2048
>>> N = seglen*sample_rate/2+1
>>> fmin = 30.
>>> m1, m2, s1z, s2z, tsig, ra, dec, pol, dist = (
...     38.6, 29.3, 0., 0., 3.1, 1.37, -1.26, 2.76, 3*500.)
>>> variable_params = ['tc']
>>> generator = FDomainDetFrameGenerator(
...     FDomainCBCGenerator, 0.,
...     variable_args=variable_params, detectors=['H1', 'L1'],
...     delta_f=1./seglen, f_lower=fmin,
...     approximant='SEOBNRv2_ROM_DoubleSpin',
...     mass1=m1, mass2=m2, spin1z=s1z, spin2z=s2z,
...     ra=ra, dec=dec, polarization=pol, distance=dist)
>>> signal = generator.generate(tc=tsig)
>>> psd = pypsd.aLIGOZeroDetHighPower(N, 1./seglen, 20.)
>>> psds = {'H1': psd, 'L1': psd}
>>> model = gwin.models.GaussianNoise(
...     variable_params, signal, generator, fmin, psds=psds)
```

Set the current position to the coalescence time of the signal:

```
>>> model.update(tc=tsig)
```

Now compute the log likelihood ratio and prior-weighted likelihood ratio; since we have not provided a prior, these should be equal to each other:

```
>>> print('{:.2f}'.format(model.loglr))
278.96
>>> print('{:.2f}'.format(model.logplr))
278.96
```

Print all of the default_stats:

```
>>> print(',\n'.join(['{}: {:.2f}'.format(s, v)
...                   for (s, v) in sorted(model.current_stats.items())]))
H1_cplx_loglr: 175.57+0.00j,
H1_optimal_snrsq: 351.13,
L1_cplx_loglr: 103.40+0.00j,
L1_optimal_snrsq: 206.79,
logjacobian: 0.00,
loglikelihood: 0.00,
loglr: 278.96,
logprior: 0.00
```

Compute the SNR; for this system and PSD, this should be approximately 24:

```
>>> from pycbc.conversions import snr_from_loglr
>>> x = snr_from_loglr(model.loglr)
>>> print('{:.2f}'.format(x))
23.62
```

Since there is no noise, the SNR should be the same as the quadrature sum of the optimal SNRs in each detector:

```
>>> x = (model.det_optimal_snrsq('H1') +
...      model.det_optimal_snrsq('L1'))**0.5
>>> print('{:.2f}'.format(x))
23.62
```

Using the same model, evaluate the log likelihood ratio at several points in time and check that the max is at tsig:

```
>>> import numpy
>>> times = numpy.arange(seglen*sample_rate)/float(sample_rate)
>>> loglrs = numpy.zeros(len(times))
>>> for (ii, t) in enumerate(times):
...     model.update(tc=t)
...     loglrs[ii] = model.loglr
>>> print('tsig: {:.3f}, time of max loglr: {:.3f}'.format(
...     tsig, times[loglrs.argmax()]))
tsig: 3.100, time of max loglr: 3.100
```

Create a prior and use it (see distributions module for more details):

```
>>> from pycbc import distributions
>>> uniform_prior = distributions.Uniform(tc=(tsig-0.2,tsig+0.2))
>>> prior = distributions.JointDistribution(variable_params, uniform_prior)
>>> model = gwin.models.GaussianNoise(variable_params,
...     signal_generator, 20., psds=psds, prior=prior)
>>> model.update(tc=tsig)
>>> print('{:.2f}'.format(model.logplr))
279.88
>>> print(',\n'.join(['{}: {:.2f}'.format(s, v)
...                   for (s, v) in sorted(model.current_stats.items())]))
```

(continues on next page)

(continued from previous page)

```
H1_cplx_loglr: 175.57+0.00j,
H1_optimal_snrsq: 351.13,
L1_cplx_loglr: 103.40+0.00j,
L1_optimal_snrsq: 206.79,
logjacobian: 0.00,
loglikelihood: 0.00,
loglr: 278.96,
logprior: 0.92
```

**det_cplx_loglr**(*det*)
Returns the complex log likelihood ratio in the given detector.

**Parameters** **det** (*str*) – The name of the detector.

**Returns** The complex log likelihood ratio.

**Return type** complex float

**det_lognl**(*det*)
Returns the log likelihood of the noise in the given detector.

**Parameters** **det** (*str*) – The name of the detector.

**Returns** The log likelihood of the noise in the requested detector.

**Return type** float

**det_optimal_snrsq**(*det*)
Returns the opitmal SNR squared in the given detector.

**Parameters** **det** (*str*) – The name of the detector.

**Returns** The opimtal SNR squared.

**Return type** float

**name = 'gaussian_noise'**

### gwin.models.marginalized_gaussian_noise module

This module provides model classes that assume the noise is Gaussian and allows for the likelihood to be marginalized over phase and/or time and/or distance.

**class** gwin.models.marginalized_gaussian_noise.**MarginalizedGaussianNoise**(*variable_params*, *data*, *waveform_generator*, *f_lower*, *psds=None*, *f_upper=None*, *norm=None*, *time_marginalization=Fals*
*distance_marginalization=Fal*
*phase_marginalization=Fa*
*marg_prior=None*, ***kwargs*)

Bases: *gwin.models.gaussian_noise.GaussianNoise*

The likelihood is analytically marginalized over phase and/or time and/or distance.

For the case of marginalizing over phase, the signal, can be written as:

$$\tilde{h}(f;\Theta,\phi) = A(f;\Theta)e^{i\Psi(f;\Theta)+i\phi},$$

where $\phi$ is an arbitrary phase constant. This phase constant can be analytically marginalized over with a uniform prior as follows: assuming the noise is stationary and Gaussian (see `GaussianNoise` for details), the posterior is:

$$\begin{aligned} p(\Theta,\phi|d) &\propto p(\Theta)p(\phi)p(d|\Theta,\phi) \\ &\propto p(\Theta)\frac{1}{2\pi}\exp\left[-\frac{1}{2}\sum_i^{N_D}\left\langle h_i(\Theta,\phi)-d_i, h_i(\Theta,\phi)-d_i\right\rangle\right]. \end{aligned}$$

Here, the sum is over the number of detectors $N_D$, $d_i$ and $h_i$ are the data and signal in detector $i$, respectively, and we have assumed a uniform prior on $phi \in [0, 2\pi)$. With the form of the signal model given above, the inner product in the exponent can be written as:

$$\begin{aligned} -\frac{1}{2}\left\langle h_i-d_i, h_i-d_i\right\rangle &= \left\langle h_i, d_i\right\rangle - \frac{1}{2}\left\langle h_i, h_i\right\rangle - \frac{1}{2}\left\langle d_i, d_i\right\rangle \\ &= \Re\left\{O(h_i^0, d_i)e^{-i\phi}\right\} - \frac{1}{2}\left\langle h_i^0, h_i^0\right\rangle - \frac{1}{2}\left\langle d_i, d_i\right\rangle, \end{aligned}$$

where:

$$\begin{aligned} h_i^0 &\equiv \tilde{h}_i(f;\Theta,\phi=0); \\ O(h_i^0, d_i) &\equiv 4\int_0^\infty \frac{\tilde{h}_i^*(f;\Theta,0)\tilde{d}_i(f)}{S_n(f)}\mathrm{d}f. \end{aligned}$$

Gathering all of the terms that are not dependent on $\phi$ together:

$$\alpha(\Theta,d) \equiv \exp\left[-\frac{1}{2}\sum_i\left\langle h_i^0, h_i^0\right\rangle + \left\langle d_i, d_i\right\rangle\right],$$

we can marginalize the posterior over $\phi$:

$$\begin{aligned} p(\Theta|d) &\propto p(\Theta)\alpha(\Theta,d)\frac{1}{2\pi}\int_0^{2\pi}\exp\left[\Re\left\{e^{-i\phi}\sum_i O(h_i^0,d_i)\right\}\right]\mathrm{d}\phi \\ &\propto p(\Theta)\alpha(\Theta,d)\frac{1}{2\pi}\int_0^{2\pi}\exp\left[x(\Theta,d)\cos(\phi)+y(\Theta,d)\sin(\phi)\right]\mathrm{d}\phi. \end{aligned}$$

The integral in the last line is equal to $2\pi I_0(\sqrt{x^2+y^2})$, where $I_0$ is the modified Bessel function of the first kind. Thus the marginalized log posterior is:

$$\log p(\Theta|d) \propto \log p(\Theta) + I_0\left(\left|\sum_i O(h_i^0,d_i)\right|\right) - \frac{1}{2}\sum_i\left[\left\langle h_i^0, h_i^0\right\rangle - \left\langle d_i, d_i\right\rangle\right]$$

For the case of marginalizing over distance, the signal can be written as,

$$\tilde{h}_j = \frac{1}{D}\tilde{h}_j^0$$

The distance can be analytically marginalized over with a uniform prior as follows: assuming the noise is stationary and Gaussian (see `GaussianNoise` for details), the likelihood is:

$$logL = -\frac{1}{2}\left\langle d-h|d-h\right\rangle$$

We see that :math: `<h|h>` is inversely proportional to distance squared and :math: `<h|d>` is inversely proportional to distance. The log likelihood is therefore

$$\log L = -\frac{1}{2}\left\langle d|d\right\rangle - \frac{1}{2D^2}\left\langle h|h\right\rangle + \frac{1}{D}\left\langle h|d\right\rangle$$

Consequently, the likelihood marginalised over distance is simply

$$\log L = \log\left(\int_0^D Lp(D)dD\right)$$

If we assume a flat prior

$$\log L = \log\left(\int_0^D \exp\log L dD\right)$$

For the case of marginalizing over time, the signal can be written as,

$$\tilde{h}_j = \tilde{h}_j^0 \exp\left(-2\pi i j \Delta f t\right)$$

The time can be analytically marginalized over with a uniform prior as follows: assuming the noise is stationary and Gaussian (see `GaussianNoise` for details), the likelihood is:

$$\log L = -\frac{1}{2}\left\langle d-h|d-h\right\rangle$$

We note that :math: `<h|h>` and :math: `<d|d>` are time independent while :math: `<d|h>` is dependent of time

$$\left\langle d|h\right\rangle(t) = 4\Delta f \sum_{j=0}^{N/2} \frac{\tilde{d}_j^* \tilde{h}_j^0}{S_j} \exp\left(-2\pi i j \Delta f t\right)$$

For integer timesteps :math: `t=kDelta t`

$$\left\langle d|h\right\rangle(k\Delta t) = 4\Delta f \sum_{j=0}^{N/2} \frac{\tilde{d}_j^* \tilde{h}_j^0}{S_j} \exp(-2\pi \frac{ijk}{N}$$

$$\left\langle d|h\right\rangle(k\Delta t) = 2\Delta f \sum_{j=0}^{N} \frac{\tilde{d}_j^* \tilde{h}_j^0}{S_j} \exp(-2\pi \frac{ijk}{N}$$

Using a FFT, this expression can be evaluated efficiently for all :math: `k`

$$\left\langle d|h\right\rangle(k\Delta t) = 2\Delta f FFT_k(\frac{dh}{S})$$

since :math:: `left<h|dright> = left<d|hright>^{*}`,

$$\left\langle d|h\right\rangle + \left\langle h|d\right\rangle = 4\Delta f FFT_k(\frac{dh}{S})$$

and so the likelihood marginalised over time is simply

$$\log L = \log\left(\int_0^T np.exp(.log(L)p(t))\right)$$

where p(t) is the prior. If we assume a flat prior then,

$$\log L = \log\left(\int_0^T np.exp(.log(L))\right)$$

**Parameters**

- **time_marginalization** (*bool, optional*) – A Boolean operator which determines if the likelihood is marginalized over time
- **phase_marginalization** (*bool, optional*) – A Boolean operator which determines if the likelihood is marginalized over phase
- **distance_marginalization** (*bool, optional*) – A Boolean operator which determines if the likelihood is marginalized over distance
- **marg_prior** (*list, optional*) – An instance of pycbc.distributions which returns a list of prior distributions to be used when marginalizing the likelihood
- ****kwargs** – All other keyword arguments are passed to `GaussianNoise`.

**classmethod from_config**(*cp*, *data*, *delta_f=None*, *delta_t=None*, *gates=None*, *recalibration=None*, ***kwargs*)

Initializes an instance of this class from the given config file.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- **data** (*dict*) – A dictionary of data, in which the keys are the detector names and the values are the data. This is not retrieved from the config file, and so must be provided.
- **delta_f** (*float*) – The frequency spacing of the data; needed for waveform generation.
- **delta_t** (*float*) – The time spacing of the data; needed for time-domain waveform generators.
- **recalibration** (*dict of pycbc.calibration.Recalibrate, optional*) – Dictionary of detectors -> recalibration class instances for recalibrating data.
- **gates** (*dict of tuples, optional*) – Dictionary of detectors -> tuples of specifying gate times. The sort of thing returned by `pycbc.gate.gates_from_cli`.
- ****kwargs** – All additional keyword arguments are passed to the class. Any provided keyword will over ride what is in the config file.

**name = 'marginalized_gaussian_noise'**

## Module contents

This package provides classes and functions for evaluating Bayesian statistics assuming various noise models.

**class** gwin.models.**CallModel**(*model*, *callstat*, *return_all_stats=True*)

Bases: `object`

Wrapper class for calling models from a sampler.

This class can be called like a function, with the parameter values to evaluate provided as a list in the same order as the model's `variable_params`. In that case, the model is updated with the provided parameters and then the `callstat` retrieved. If `return_all_stats` is set to `True`, then all of the stats specified by the model's `default_stats` will be returned as a tuple, in addition to the stat value.

The model's attributes are promoted to this class's namespace, so that any attribute and method of `model` may be called directly from this class.

This class must be initalized prior to the creation of a `Pool` object.

**Parameters**

- **model** (*Model instance*) – The model to call.
- **callstat** (*str*) – The statistic to call.
- **return_all_stats** (*bool, optional*) – Whether or not to return all of the other statistics along with the `callstat` value.

#### Examples

Create a wrapper around an instance of the `TestNormal` model, with the `callstat` set to `logposterior`:

```
>>> from gwin.models import TestNormal, CallModel
>>> model = TestNormal(['x', 'y'])
>>> call_model = CallModel(model, 'logposterior')
```

Now call on a set of parameter values:

```
>>> call_model([0.1, -0.2])
(-1.8628770664093453, (0.0, 0.0, -1.8628770664093453))
```

Note that a tuple of all of the model's `default_stats` were returned in addition to the `logposterior` value. We can shut this off by toggling `return_all_stats`:

```
>>> call_model.return_all_stats = False
>>> call_model([0.1, -0.2])
-1.8628770664093453
```

Attributes of the model can be called from the call model. For example:

```
>>> call_model.variable_params
('x', 'y')
```

gwin.models.**read_from_config**(*cp*, ***kwargs*)

Initializes a model from the given config file.

The section must have a `name` argument. The name argument corresponds to the name of the class to initialize.

**Parameters**

- **cp** (*WorkflowConfigParser*) – Config file parser to read.
- ****kwargs** – All other keyword arguments are passed to the `from_config` method of the class specified by the name argument.

**Returns** The initialized model.

**Return type** cls

### 2.1.3 gwin.results package

#### Submodules

#### gwin.results.scatter_histograms module

Module to generate figures with scatter plots and histograms.

gwin.results.scatter_histograms.**construct_kde**(*samples_array*, *use_kombine=False*)

Constructs a KDE from the given samples.

gwin.results.scatter_histograms.**create_axes_grid**(*parameters*, *labels=None*, *height_ratios=None*, *width_ratios=None*, *no_diagonals=False*)

Given a list of parameters, creates a figure with an axis for every possible combination of the parameters.

**Parameters**

- **parameters** (*list*) – Names of the variables to be plotted.
- **labels** (*{None, dict}, optional*) – A dictionary of parameters -> parameter labels.
- **height_ratios** (*{None, list}, optional*) – Set the height ratios of the axes; see matplotlib.gridspec.GridSpec for details.
- **width_ratios** (*{None, list}, optional*) – Set the width ratios of the axes; see matplotlib.gridspec.GridSpec for details.
- **no_diagonals** (*{False, bool}, optional*) – Do not produce axes for the same parameter on both axes.

**Returns**

- **fig** (*pyplot.figure*) – The figure that was created.
- **axis_dict** (*dict*) – A dictionary mapping the parameter combinations to the axis and their location in the subplots grid; i.e., the key, values are: `{('param1', 'param2'): (pyplot.axes, row index, column index)}`

gwin.results.scatter_histograms.**create_density_plot**(*xparam*, *yparam*, *samples*, *plot_density=True*, *plot_contours=True*, *percentiles=None*, *cmap='viridis'*, *contour_color=None*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *exclude_region=None*, *fig=None*, *ax=None*, *use_kombine=False*)

Computes and plots posterior density and confidence intervals using the given samples.

**Parameters**

- **xparam** (*string*) – The parameter to plot on the x-axis.
- **yparam** (*string*) – The parameter to plot on the y-axis.
- **samples** (*dict, numpy structured array, or FieldArray*) – The samples to plot.
- **plot_density** (*{True, bool}*) – Plot a color map of the density.
- **plot_contours** (*{True, bool}*) – Plot contours showing the n-th percentiles of the density.
- **percentiles** (*{None, float or array}*) – What percentile contours to draw. If None, will plot the 50th and 90th percentiles.
- **cmap** (*{'viridis', string}*) – The name of the colormap to use for the density plot.
- **contour_color** (*{None, string}*) – What color to make the contours. Default is white for density plots and black for other plots.
- **xmin** (*{None, float}*) – Minimum value to plot on x-axis.

- **xmax** (*{None, float}*) – Maximum value to plot on x-axis.
- **ymin** (*{None, float}*) – Minimum value to plot on y-axis.
- **ymax** (*{None, float}*) – Maximum value to plot on y-axis.
- **exclue_region** (*{None, str}*) – Exclude the specified region when plotting the density or contours. Must be a string in terms of `xparam` and `yparam` that is understandable by numpy's logical evaluation. For example, if `xparam = m_1` and `yparam = m_2`, and you want to exclude the region for which `m_2` is greater than `m_1`, then exclude region should be `'m_2 > m_1'`.
- **fig** (*{None, pyplot.figure}*) – Add the plot to the given figure. If None and ax is None, will create a new figure.
- **ax** (*{None, pyplot.axes}*) – Draw plot on the given axis. If None, will create a new axis from `fig`.
- **use_kombine** (*{False, bool}*) – Use kombine's KDE to calculate density. Otherwise, will use `scipy.stats.gaussian_kde`. Default is False.

**Returns**

- **fig** (*pyplot.figure*) – The figure the plot was made on.
- **ax** (*pyplot.axes*) – The axes the plot was drawn on.

`gwin.results.scatter_histograms.`**`create_marginalized_hist`**(*ax*, *values*, *label*, *percentiles=None*, *color='k'*, *fillcolor='gray'*, *linecolor='navy'*, *title=True*, *expected_value=None*, *expected_color='red'*, *rotated=False*, *plot_min=None*, *plot_max=None*)

Plots a 1D marginalized histogram of the given param from the given samples.

**Parameters**

- **ax** (*pyplot.Axes*) – The axes on which to draw the plot.
- **values** (*array*) – The parameter values to plot.
- **label** (*str*) – A label to use for the title.
- **percentiles** (*{None, float or array}*) – What percentiles to draw lines at. If None, will draw lines at `[5, 50, 95]` (i.e., the bounds on the upper 90th percentile and the median).
- **color** (*{'k', string}*) – What color to make the histogram; default is black.
- **fillcolor** (*{'gray', string, or None}*) – What color to fill the histogram with. Set to None to not fill the histogram. Default is 'gray'.
- **linecolor** (*{'navy', string}*) – What color to use for the percentile lines. Default is 'navy'.
- **title** (*{True, bool}*) – Add a title with the median value +/- uncertainty, with the max(min) `percentile` used for the +(-) uncertainty.
- **rotated** (*{False, bool}*) – Plot the histogram on the y-axis instead of the x. Default is False.

- **plot_min** (*{None, float}*) – The minimum value to plot. If None, will default to whatever `pyplot` creates.
- **plot_max** (*{None, float}*) – The maximum value to plot. If None, will default to whatever `pyplot` creates.
- **scalefac** (*{1., float}*) – Factor to scale the default font sizes by. Default is 1 (no scaling).

gwin.results.scatter_histograms.**create_multidim_plot**(*parameters*, *samples*, *labels=None*, *mins=None*, *maxs=None*, *expected_parameters=None*, *expected_parameters_color='r'*, *plot_marginal=True*, *plot_scatter=True*, *marginal_percentiles=None*, *contour_percentiles=None*, *zvals=None*, *show_colorbar=True*, *cbar_label=None*, *vmin=None*, *vmax=None*, *scatter_cmap='plasma'*, *plot_density=False*, *plot_contours=True*, *density_cmap='viridis'*, *contour_color=None*, *hist_color='black'*, *line_color=None*, *fill_color='gray'*, *use_kombine=False*, *fig=None*, *axis_dict=None*)

Generate a figure with several plots and histograms.

**Parameters**

- **parameters** (*list*) – Names of the variables to be plotted.
- **samples** (*FieldArray*) – A field array of the samples to plot.
- **labels** (*{None, list}, optional*) – A list of names for the parameters.
- **mins** (*{None, dict}, optional*) – Minimum value for the axis of each variable in `parameters`. If None, it will use the minimum of the corresponding variable in `samples`.
- **maxs** (*{None, dict}, optional*) – Maximum value for the axis of each variable in `parameters`. If None, it will use the maximum of the corresponding variable in `samples`.
- **expected_parameters** (*{None, dict}, optional*) – Expected values of `parameters`, as a dictionary mapping parameter names -> values. A cross will be plotted at the location of the expected parameters on axes that plot any of the expected parameters.
- **expected_parameters_color** (*{'r', string}, optional*) – What color to make the expected parameters cross.
- **plot_marginal** (*{True, bool}*) – Plot the marginalized distribution on the diagonals. If False, the diagonal axes will be turned off.
- **plot_scatter** (*{True, bool}*) – Plot each sample point as a scatter plot.

- **marginal_percentiles** (*{None, array}*) – What percentiles to draw lines at on the 1D histograms. If None, will draw lines at `[5, 50, 95]` (i.e., the bounds on the upper 90th percentile and the median).
- **contour_percentiles** (*{None, array}*) – What percentile contours to draw on the scatter plots. If None, will plot the 50th and 90th percentiles.
- **zvals** (*{None, array}*) – An array to use for coloring the scatter plots. If None, scatter points will be the same color.
- **show_colorbar** (*{True, bool}*) – Show the colorbar of zvalues used for the scatter points. A ValueError will be raised if zvals is None and this is True.
- **cbar_label** (*{None, str}*) – Specify a label to add to the colorbar.
- **vmin** (*{None, float}, optional*) – Minimum value for the colorbar. If None, will use the minimum of zvals.
- **vmax** (*{None, float}, optional*) – Maximum value for the colorbar. If None, will use the maxmimum of zvals.
- **scatter_cmap** (*{'plasma', string}*) – The color map to use for the scatter points. Default is 'plasma'.
- **plot_density** (*{False, bool}*) – Plot the density of points as a color map.
- **plot_contours** (*{True, bool}*) – Draw contours showing the 50th and 90th percentile confidence regions.
- **density_cmap** (*{'viridis', string}*) – The color map to use for the density plot.
- **contour_color** (*{None, string}*) – The color to use for the contour lines. Defaults to white for density plots, navy for scatter plots without zvals, and black otherwise.
- **use_kombine** (*{False, bool}*) – Use kombine's KDE to calculate density. Otherwise, will use `scipy.stats.gaussian_kde`. Default is False.

**Returns**

- **fig** (*pyplot.figure*) – The figure that was created.
- **axis_dict** (*dict*) – A dictionary mapping the parameter combinations to the axis and their location in the subplots grid; i.e., the key, values are: `{('param1', 'param2'): (pyplot.axes, row index, column index)}`

gwin.results.scatter_histograms.**get_scale_fac**(*fig*, *fiducial_width=8*, *fiducial_height=7*)

Gets a factor to scale fonts by for the given figure. The scale factor is relative to a figure with dimensions (`fiducial_width`, `fiducial_height`).

gwin.results.scatter_histograms.**reduce_ticks**(*ax*, *which*, *maxticks=3*)

Given a pyplot axis, resamples its `which`-axis ticks such that are at most `maxticks` left.

**Parameters**

- **ax** (*axis*) – The axis to adjust.
- **which** (*{'x' | 'y'}*) – Which axis to adjust.
- **maxticks** (*{3, int}*) – Maximum number of ticks to use.

**Returns** An array of the selected ticks.

**Return type** array

gwin.results.scatter_histograms.**remove_common_offset**(*arr*)

Given an array of data, removes a common offset > 1000, returning the removed value.

gwin.results.scatter_histograms.**set_marginal_histogram_title**(*ax*, *fmt*, *color*, *label=None*, *rotated=False*)

Sets the title of the marginal histograms.

**Parameters**

- **ax** (*Axes*) – The Axes instance for the plot.
- **fmt** (*str*) – The string to add to the title.
- **color** (*str*) – The color of the text to add to the title.
- **label** (*str*) – If title does not exist, then include label at beginning of the string.
- **rotated** (*bool*) – If True then rotate the text 270 degrees for sideways title.

#### Module contents

Results utilities for GWIn

### 2.1.4 gwin.sampler package

#### Submodules

#### gwin.sampler.base module

This modules provides classes and functions for using different sampler packages for parameter estimation.

**class** gwin.sampler.base.**BaseMCMCSampler**(*sampler*, *model*)

Bases: gwin.sampler.base._BaseSampler

This class is used to construct the MCMC sampler from the kombine-like packages.

**Parameters**

- **sampler** (*sampler instance*) – An instance of an MCMC sampler similar to kombine or emcee.
- **model** (*model class*) – A model from gwin.models.

**sampler**

The MCMC sampler instance used.

**p0**

*nwalkers x ndim array* – The initial position of the walkers. Set by using set_p0. If not set yet, a ValueError is raised when the attribute is accessed.

**pos**

*{None, array}* – An array of the current walker positions.

**acceptance_fraction**

Get the fraction of steps accepted by each walker as an array.

**classmethod compute_acfs**(*fp*, *start_index=None*, *end_index=None*, *per_walker=False*, *walkers=None*, *parameters=None*)

Computes the autocorrleation function of the model params in the given file.

By default, parameter values are averaged over all walkers at each iteration. The ACF is then calculated over the averaged chain. An ACF per-walker will be returned instead if per_walker=True.

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the samples from.
- **start_index** (*{None, int}*) – The start index to compute the acl from. If None, will try to use the number of burn-in iterations in the file; otherwise, will start at the first sample.
- **end_index** (*{None, int}*) – The end index to compute the acl to. If None, will go to the end of the current iteration.
- **per_walker** (*optional, bool*) – Return the ACF for each walker separately. Default is False.
- **walkers** (*optional, int or array*) – Calculate the ACF using only the given walkers. If None (the default) all walkers will be used.
- **parameters** (*optional, str or array*) – Calculate the ACF for only the given parameters. If None (the default) will calculate the ACF for all of the model params.

**Returns** A FieldArray of the ACF vs iteration for each parameter. If per-walker is True, the FieldArray will have shape nwalkers x niterations.

**Return type** FieldArray

**classmethod compute_acls**(*fp*, *start_index=None*, *end_index=None*)

Computes the autocorrleation length for all model params in the given file.

Parameter values are averaged over all walkers at each iteration. The ACL is then calculated over the averaged chain. If the returned ACL is inf, will default to the number of current iterations.

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the samples from.
- **start_index** (*{None, int}*) – The start index to compute the acl from. If None, will try to use the number of burn-in iterations in the file; otherwise, will start at the first sample.
- **end_index** (*{None, int}*) – The end index to compute the acl to. If None, will go to the end of the current iteration.

**Returns** A dictionary giving the ACL for each parameter.

**Return type** dict

**model_stats**

Returns the model stats as a FieldArray, with field names corresponding to the type of data returned by the model. The returned array has shape nwalkers x niterations. If no additional stats were returned to the sampler by the model, returns None.

**classmethod n_independent_samples**(*fp*)

Returns the number of independent samples stored in a file.

The number of independent samples are counted starting from after burn-in. If the sampler hasn't burned in yet, then 0 is returned.

**Parameters** **fp** (InferenceFile) – An open file handler to read.

**Returns** The number of independent samples.

**Return type** int

**name = None**

**nwalkers**
Get the number of walkers.

**p0**

**pos**

**static read_acceptance_fraction**(*fp*, *walkers=None*)
Reads the acceptance fraction from the given file.

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the samples from.
- **walkers** (*{None, (list of) int}*) – The walker index (or a list of indices) to retrieve. If None, samples from all walkers will be obtained.

**Returns** Array of acceptance fractions with shape (requested walkers,).

**Return type** array

**static read_acls**(*fp*)
Reads the acls of all the parameters in the given file.

**Parameters** **fp** (InferenceFile) – An open file handler to read the acls from.

**Returns** A dictionary of the ACLs, keyed by the parameter name.

**Return type** dict

**classmethod read_samples**(*fp*, *parameters*, *thin_start=None*, *thin_interval=None*, *thin_end=None*, *iteration=None*, *walkers=None*, *flatten=True*, *samples_group=None*, *array_class=None*)
Reads samples for the given parameter(s).

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the samples from.
- **parameters** (*(list of) strings*) – The parameter(s) to retrieve. A parameter can be the name of any field in `fp[fp.samples_group]`, a virtual field or method of `FieldArray` (as long as the file contains the necessary fields to derive the virtual field or method), and/or a function of these.
- **thin_start** (*int*) – Index of the sample to begin returning samples. Default is to read samples after burn in. To start from the beginning set thin_start to 0.
- **thin_interval** (*int*) – Interval to accept every i-th sample. Default is to use the `fp.acl`. If `fp.acl` is not set, then use all samples (set thin_interval to 1).
- **thin_end** (*int*) – Index of the last sample to read. If not given then `fp.niterations` is used.
- **iteration** (*int*) – Get a single iteration. If provided, will override the `thin_{start/interval/end}` arguments.
- **walkers** (*{None, (list of) int}*) – The walker index (or a list of indices) to retrieve. If None, samples from all walkers will be obtained.
- **flatten** (*{True, bool}*) – The returned array will be one dimensional, with all desired samples from all desired walkers concatenated together. If False, the returned array will have dimension requested walkers x requested iterations.
- **samples_group** (*{None, str}*) – The group in `fp` from which to retrieve the parameter fields. If None, searches in `fp.samples_group`.

- **array_class** (*{None, array class}*) – The type of array to return. The class must have a `from_kwargs` class method and a `parse_parameters` method. If None, will return a FieldArray.

**Returns** Samples for the given parameters, as an instance of a the given `array_class` (`FieldArray` if `array_class` is None).

**Return type** array_class

**sampler**

**samples**

Returns the samples in the chain as a FieldArray.

If the sampling args are not the same as the model params, the returned samples will have both the sampling and the model params.

The returned FieldArray has dimension [additional dimensions x] nwalkers x niterations.

**set_p0** (*samples_file=None*, *prior=None*)

Sets the initial position of the walkers.

**Parameters**

- **samples_file** (*InferenceFile, optional*) – If provided, use the last iteration in the given file for the starting positions.
- **prior** (*JointDistribution, optional*) – Use the given prior to set the initial positions rather than `model`'s prior.

**Returns** **p0** – An nwalkers x ndim array of the initial positions that were set.

**Return type** array

**write_acceptance_fraction** (*fp*)

Write acceptance_fraction data to file. Results are written to `fp[acceptance_fraction]`.

**Parameters** **fp** (*InferenceFile*) – A file handler to an open inference file.

**static write_acls** (*fp*, *acls*)

Writes the given autocorrelation lengths to the given file.

The ACL of each parameter is saved to `fp['acls/{param}']`. The maximum over all the parameters is saved to the file's 'acl' attribute.

**Parameters**

- **fp** (*InferenceFile*) – An open file handler to write the samples to.
- **acls** (*dict*) – A dictionary of ACLs keyed by the parameter.

**Returns** The maximum of the acls that was written to the file.

**Return type** ACL

**write_chain** (*fp*, *start_iteration=None*, *max_iterations=None*)

Writes the samples from the current chain to the given file.

Results are written to:

```
fp[fp.samples_group/{field}/(temp{k}/)walker{i}],
```

where `{i}` is the index of a walker, `{field}` is the name of each field returned by `model_stats`, and, if the sampler is multitempered, `{k}` is the temperature.

**Parameters**

- **fp** (InferenceFile) – A file handler to an open inference file.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the samples have not previously been written to file. The default (None) is to use the maximum size allowed by h5py.
- **samples_group** (*str*) – Name of samples group to write.

**write_metadata** (*fp*, ***kwargs*)
Writes metadata about this sampler to the given file. Metadata is written to the file's `attrs`.

**Parameters**

- **fp** (InferenceFile) – A file handler to an open inference file.
- ****kwargs** – All keyword args are written to the file's `attrs`.

**write_model_stats** (*fp*, *start_iteration=None*, *max_iterations=None*)
Writes the `model_stats` to the given file.

Results are written to:

```
fp[fp.stats_group/{field}/(temp{k}/)walker{i}],
```

where `{i}` is the index of a walker, `{field}` is the name of each field returned by `model_stats`, and, if the sampler is multitempered, `{k}` is the temperature. If nothing is returned by `model_stats`, this does nothing.

**Parameters**

- **fp** (InferenceFile) – A file handler to an open inference file.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the samples have not previously been written to file. The default (None) is to use the maximum size allowed by h5py.

**Returns** **stats** – The stats that were written, as a FieldArray. If there were no stats, returns None.

**Return type** {FieldArray, None}

**write_results** (*fp*, *start_iteration=None*, *max_iterations=None*, ***metadata*)
Writes metadata, samples, model stats, and acceptance fraction to the given file. Also computes and writes the autocorrleation lengths of the chains. See the various write function for details.

**Parameters**

- **fp** (InferenceFile) – A file handler to an open inference file.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the acceptance fraction has not previously been written to the file. The default (None) is to use the maximum size allowed by h5py.
- ****metadata** – All other keyword arguments are passed to `write_metadata`.

**static write_samples_group** (*fp*, *samples_group*, *parameters*, *samples*, *start_iteration=None*, *max_iterations=None*)
Writes samples to the given file.

Results are written to:

`fp[samples_group/{vararg}]`,

where `{vararg}` is the name of a model params. The samples are written as an `nwalkers x niterations` array.

**Parameters**

- **fp** (`InferenceFile`) – A file handler to an open inference file.
- **samples_group** (*str*) – Name of samples group to write.
- **parameters** (*list*) – The parameters to write to the file.
- **samples** (*FieldArray*) – The samples to write. Should be a FieldArray with fields containing the samples to write and shape nwalkers x niterations.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the samples have not previously been written to file. The default (None) is to use the maximum size allowed by h5py.

### gwin.sampler.emcee module

This modules provides classes and functions for using the emcee sampler packages for parameter estimation.

**class** gwin.sampler.emcee.**EmceeEnsembleSampler**(*model*, *nwalkers*, *pool=None*, *model_call=None*)

Bases: *gwin.sampler.base.BaseMCMCSampler*

This class is used to construct an MCMC sampler from the emcee package's EnsembleSampler.

**Parameters**

- **model** (*model*) – A model from `gwin.models`.
- **nwalkers** (*int*) – Number of walkers to use in sampler.
- **pool** (*function with map, Optional*) – A provider of a map function that allows a function call to be run over multiple sets of arguments and possibly maps them to cores/nodes/etc.

**chain**

Get all past samples as an nwalker x niterations x ndim array.

**clear_chain**()

Clears the chain and blobs from memory.

**classmethod from_cli**(*opts*, *model*, *pool=None*, *model_call=None*)

Create an instance of this sampler from the given command-line options.

**Parameters**

- **opts** (*ArgumentParser options*) – The options to parse.
- **model** (*LikelihoodEvaluator*) – The model to use with the sampler.

**Returns** An emcee sampler initialized based on the given arguments.

**Return type** *EmceeEnsembleSampler*

**lnpost**
Get the natural logarithm of the likelihood as an nwalkers x niterations array.

**name = 'emcee'**

**run**(*niterations*, ***kwargs*)
Advance the ensemble for a number of samples.

**Parameters** **niterations** (*int*) – Number of samples to get from sampler.

**Returns**

- **p** (*numpy.array*) – An array of current walker positions with shape (nwalkers, ndim).
- **lnpost** (*numpy.array*) – The list of log posterior probabilities for the walkers at positions p, with shape (nwalkers, ndim).
- *rstate* – The current state of the random number generator.

**set_p0**(*samples_file=None*, *prior=None*)
Sets the initial position of the walkers.

**Parameters**

- **samples_file** (*InferenceFile, optional*) – If provided, use the last iteration in the given file for the starting positions.
- **prior** (*JointDistribution, optional*) – Use the given prior to set the initial positions rather than `model`'s prior.

**Returns** **p0** – An nwalkers x ndim array of the initial positions that were set.

**Return type** array

**set_state_from_file**(*fp*)
Sets the state of the sampler back to the instance saved in a file.

**write_results**(*fp*, *start_iteration=None*, *max_iterations=None*, ***metadata*)
Writes metadata, samples, model stats, and acceptance fraction to the given file. See the write function for each of those for details.

**Parameters**

- **fp** (*InferenceFile*) – A file handler to an open inference file.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the samples have not previously been written to file. The default (None) is to use the maximum size allowed by h5py.
- ****metadata** – All other keyword arguments are passed to `write_metadata`.

**write_state**(*fp*)
Saves the state of the sampler in a file.

**class** gwin.sampler.emcee.**EmceePTSampler**(*model*, *ntemps*, *nwalkers*, *pool=None*, *model_call=None*)
Bases: *gwin.sampler.base.BaseMCMCSampler*

This class is used to construct a parallel-tempered MCMC sampler from the emcee package's PTSampler.

**Parameters**

- **model** (*model*) – A model from `gwin.models`.

- **ntemps** (*int*) – Number of temeratures to use in the sampler.
- **nwalkers** (*int*) – Number of walkers to use in sampler.
- **pool** (*function with map, Optional*) – A provider of a map function that allows a function call to be run over multiple sets of arguments and possibly maps them to cores/nodes/etc.

**classmethod calculate_logevidence**(*fp*, *thin_start=None*, *thin_end=None*, *thin_interval=None*)

Calculates the log evidence from the given file using emcee's thermodynamic integration.

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the stats from.
- **thin_start** (*int*) – Index of the sample to begin returning stats. Default is to read stats after burn in. To start from the beginning set thin_start to 0.
- **thin_interval** (*int*) – Interval to accept every i-th sample. Default is to use the `fp.acl`. If `fp.acl` is not set, then use all stats (set thin_interval to 1).
- **thin_end** (*int*) – Index of the last sample to read. If not given then `fp.niterations` is used.

**Returns**

- **lnZ** (*float*) – The estimate of log of the evidence.
- **dlnZ** (*float*) – The error on the estimate.

**chain**

Get all past samples as an ntemps x nwalker x niterations x ndim array.

**clear_chain**()

Clears the chain and blobs from memory.

**classmethod compute_acfs**(*fp*, *start_index=None*, *end_index=None*, *per_walker=False*, *walkers=None*, *parameters=None*, *temps=None*)

Computes the autocorrleation function of the model params in the given file.

By default, parameter values are averaged over all walkers at each iteration. The ACF is then calculated over the averaged chain for each temperature. An ACF per-walker will be returned instead if `per_walker=True`.

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the samples from.
- **start_index** (*{None, int}*) – The start index to compute the acl from. If None, will try to use the number of burn-in iterations in the file; otherwise, will start at the first sample.
- **end_index** (*{None, int}*) – The end index to compute the acl to. If None, will go to the end of the current iteration.
- **per_walker** (*optional, bool*) – Return the ACF for each walker separately. Default is False.
- **walkers** (*optional, int or array*) – Calculate the ACF using only the given walkers. If None (the default) all walkers will be used.
- **parameters** (*optional, str or array*) – Calculate the ACF for only the given parameters. If None (the default) will calculate the ACF for all of the model params.

- **temps** (*optional, (list of) int or 'all'*) – The temperature index (or list of indices) to retrieve. If None (the default), the ACF will only be computed for the coldest (= 0) temperature chain. To compute an ACF for all temperates pass 'all', or a list of all of the temperatures.

**Returns** A `FieldArray` of the ACF vs iteration for each parameter. If `per-walker` is True, the FieldArray will have shape `ntemps x nwalkers x niterations`. Otherwise, the returned array will have shape `ntemps x niterations`.

**Return type** FieldArray

**classmethod compute_acls**(*fp*, *start_index=None*, *end_index=None*)
Computes the autocorrleation length for all model params and temperatures in the given file.

Parameter values are averaged over all walkers at each iteration and temperature. The ACL is then calculated over the averaged chain. If the returned ACL is `inf`, will default to the number of current iterations.

**Parameters**

- **fp** (`InferenceFile`) – An open file handler to read the samples from.
- **start_index** (*{None, int}*) – The start index to compute the acl from. If None, will try to use the number of burn-in iterations in the file; otherwise, will start at the first sample.
- **end_index** (*{None, int}*) – The end index to compute the acl to. If None, will go to the end of the current iteration.

**Returns** A dictionary of ntemps-long arrays of the ACLs of each parameter.

**Return type** dict

**classmethod from_cli**(*opts*, *model*, *pool=None*, *model_call=None*)
Create an instance of this sampler from the given command-line options.

**Parameters**

- **opts** (*ArgumentParser options*) – The options to parse.
- **model** (*LikelihoodEvaluator*) – The model to use with the sampler.

**Returns** An emcee sampler initialized based on the given arguments.

**Return type** *EmceePTSampler*

**lnpost**
Get the natural logarithm of the likelihood + the prior as an ntemps x nwalkers x niterations array.

**model_stats**
Returns the log likelihood ratio and log prior as a FieldArray. The returned array has shape ntemps x nwalkers x niterations.

**name = 'emcee_pt'**

**ntemps**

**static read_acceptance_fraction**(*fp*, *temps=None*, *walkers=None*)
Reads the acceptance fraction from the given file.

**Parameters**

- **fp** (`InferenceFile`) – An open file handler to read the samples from.
- **temps** (*{None, (list of) int}*) – The temperature index (or a list of indices) to retrieve. If None, acfs from all temperatures and all walkers will be retrieved.

- **walkers** (*{None, (list of) int}*) – The walker index (or a list of indices) to retrieve. If None, samples from all walkers will be obtained.

**Returns** Array of acceptance fractions with shape (requested temps, requested walkers).

**Return type** array

**classmethod read_samples**(*fp*, *parameters*, *thin_start=None*, *thin_interval=None*, *thin_end=None*, *iteration=None*, *temps=0*, *walkers=None*, *flatten=True*, *samples_group=None*, *array_class=None*)

Reads samples for the given parameter(s).

**Parameters**

- **fp** (InferenceFile) – An open file handler to read the samples from.
- **parameters** (*(list of) strings*) – The parameter(s) to retrieve. A parameter can be the name of any field in `fp[fp.samples_group]`, a virtual field or method of `FieldArray` (as long as the file contains the necessary fields to derive the virtual field or method), and/or a function of these.
- **thin_start** (*int*) – Index of the sample to begin returning samples. Default is to read samples after burn in. To start from the beginning set thin_start to 0.
- **thin_interval** (*int*) – Interval to accept every i-th sample. Default is to use the `fp.acl`. If `fp.acl` is not set, then use all samples (set thin_interval to 1).
- **thin_end** (*int*) – Index of the last sample to read. If not given then `fp.niterations` is used.
- **iteration** (*int*) – Get a single iteration. If provided, will override the `thin_{start/interval/end}` arguments.
- **walkers** (*{None, (list of) int}*) – The walker index (or a list of indices) to retrieve. If None, samples from all walkers will be obtained.
- **temps** (*{None, (list of) int, 'all'}*) – The temperature index (or list of indices) to retrieve. If None, only samples from the coldest (= 0) temperature chain will be retrieved. To retrieve all temperates pass 'all', or a list of all of the temperatures.
- **flatten** (*{True, bool}*) – The returned array will be one dimensional, with all desired samples from all desired walkers concatenated together. If False, the returned array will have dimension requested temps x requested walkers x requested iterations.
- **samples_group** (*{None, str}*) – The group in `fp` from which to retrieve the parameter fields. If None, searches in `fp.samples_group`.
- **array_class** (*{None, array class}*) – The type of array to return. The class must have a `from_kwargs` class method and a `parse_parameters` method. If None, will return a FieldArray.

**Returns** Samples for the given parameters, as an instance of a the given `array_class` (`FieldArray` if `array_class` is None).

**Return type** array_class

**run**(*niterations*, ***kwargs*)

Advance the ensemble for a number of samples.

**Parameters** **niterations** (*int*) – Number of samples to get from sampler.

**Returns**

- **p** (*numpy.array*) – An array of current walker positions with shape (nwalkers, ndim).

- **lnpost** (*numpy.array*) – The list of log posterior probabilities for the walkers at positions p, with shape (nwalkers, ndim).
- *rstate* – The current state of the random number generator.

**set_p0** (*samples_file=None*, *prior=None*)
Sets the initial position of the walkers.

**Parameters**

- **samples_file** (InferenceFile, *optional*) – If provided, use the last iteration in the given file for the starting positions.
- **prior** (*JointDistribution, optional*) – Use the given prior to set the initial positions rather than model's prior.

**Returns** **p0** – An ntemps x nwalkers x ndim array of the initial positions that were set.

**Return type** array

**write_acceptance_fraction** (*fp*)
Write acceptance_fraction data to file. Results are written to fp[acceptance_fraction/temp{k}] where k is the temperature.

**Parameters** **fp** (InferenceFile) – A file handler to an open inference file.

**write_metadata** (*fp*, ***kwargs*)
Writes metadata about this sampler to the given file. Metadata is written to the file's attrs.

**Parameters**

- **fp** (InferenceFile) – A file handler to an open inference file.
- ****kwargs** – All keyword arguments are saved as separate arguments in the file attrs. If any keyword argument is a dictionary, the keyword will point to the list of keys in the the file's attrs. Each key is then stored as a separate attr with its corresponding value.

**write_results** (*fp*, *start_iteration=None*, *max_iterations=None*, ***metadata*)
Writes metadata, samples, model stats, and acceptance fraction to the given file. See the write function for each of those for details.

**Parameters**

- **fp** (InferenceFile) – A file handler to an open inference file.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the samples have not previously been written to file. The default (None) is to use the maximum size allowed by h5py.
- ****metadata** – All other keyword arguments are passed to write_metadata.

**static write_samples_group** (*fp*, *samples_group*, *parameters*, *samples*, *start_iteration=None*, *max_iterations=None*)
Writes samples to the given file.

Results are written to:

fp[samples_group/{vararg}],

where {vararg} is the name of a variable arg. The samples are written as an ntemps x nwalkers x niterations array.

**Parameters**

- **fp** (*InferenceFile*) – A file handler to an open inference file.
- **samples_group** (*str*) – Name of samples group to write.
- **parameters** (*list*) – The parameters to write to the file.
- **samples** (*FieldArray*) – The samples to write. Should be a FieldArray with fields containing the samples to write and shape nwalkers x niterations.
- **start_iteration** (*int, optional*) – Write results to the file's datasets starting at the given iteration. Default is to append after the last iteration in the file.
- **max_iterations** (*int, optional*) – Set the maximum size that the arrays in the hdf file may be resized to. Only applies if the samples have not previously been written to file. The default (None) is to use the maximum size allowed by h5py.

### gwin.sampler.kombine module

This modules provides classes and functions for using the kombine sampler packages for parameter estimation.

**class** gwin.sampler.kombine.**KombineSampler**(*model*, *nwalkers*, *transd=False*, *pool=None*, *model_call=None*, *update_interval=None*)

Bases: *gwin.sampler.base.BaseMCMCSampler*

This class is used to construct the MCMC sampler from the kombine package.

**Parameters**

- **model** (*model*) – A model from `gwin.models`.
- **nwalkers** (*int*) – Number of walkers to use in sampler.
- **transd** (*bool*) – If True, the sampler will operate across parameter spaces using a kombine.clustered_kde.TransdimensionalKDE proposal distribution. In this mode a masked array with samples in each of the possible sets of dimensions must be given for the initial ensemble distribution.
- **processes** (*{None, int}*) – Number of processes to use with multiprocessing. If None, all available cores are used.
- **update_interval** (*{None, int}*) – Make the sampler update the proposal densities every `update_interval` iterations.

**acceptance_fraction**

Get the fraction of steps accepted by each walker as an array.

**burn_in**()

Use kombine's `burnin` routine to advance the sampler.

If a minimum number of burn-in iterations was specified, this will run the burn-in until it has advanced at least as many steps as desired. The initial positions (p0) must be set prior to running.

For more details, see `kombine.sampler.burnin`.

**Returns**

- **p** (*numpy.array*) – An array of current walker positions with shape (nwalkers, ndim).
- **lnpost** (*numpy.array*) – The list of log posterior probabilities for the walkers at positions p, with shape (nwalkers, ndim).
- **lnprop** (*numpy.array*) – The list of log proposal densities for the walkers at positions p, with shape (nwalkers, ndim).

**chain**
Get all past samples as an nwalker x niterations x ndim array.

**clear_chain**()
Clears the chain and blobs from memory.

**classmethod from_cli**(*opts*, *model*, *pool=None*, *model_call=None*)
Create an instance of this sampler from the given command-line options.

**Parameters**

- **opts** (*ArgumentParser options*) – The options to parse.
- **model** (*Model*) – The model to use with the sampler.

**Returns** A kombine sampler initialized based on the given arguments.

**Return type** *KombineSampler*

**lnpost**
Get the natural logarithm of the likelihood as an nwalkers x niterations array.

**name = 'kombine'**

**run**(*niterations*, ***kwargs*)
Advance the sampler for a number of samples.

**Parameters** **niterations** (*int*) – Number of samples to get from sampler.

**Returns**

- **p** (*numpy.array*) – An array of current walker positions with shape (nwalkers, ndim).
- **lnpost** (*numpy.array*) – The list of log posterior probabilities for the walkers at positions p, with shape (nwalkers, ndim).
- **lnprop** (*numpy.array*) – The list of log proposal densities for the walkers at positions p, with shape (nwalkers, ndim).

**set_state_from_file**(*fp*)
Sets the state of the sampler back to the instance saved in a file.

In addition to the numpy random state, the current KDE used for the jump proposals is loaded.

**Parameters** **fp** (InferenceFile) – File with sampler state stored.

**write_state**(*fp*)
Saves the state of the sampler in a file.

In addition to the numpy random state, the current KDE used for the jump proposals is saved.

**Parameters** **fp** (InferenceFile) – File to store sampler state.

### gwin.sampler.mcmc module

This modules provides classes and functions for using a MCMC sampler for parameter estimation.

**class** gwin.sampler.mcmc.**MCMCSampler**(*model*)
Bases: *gwin.sampler.base.BaseMCMCSampler*

This class is used to construct the MCMC sampler.

**Parameters** **model** (*Model*) – A model from gwin.models.

**blobs**
This function should return the blobs with a shape nwalkers x niteration as requested by the BaseMCMCSampler class.

**chain**
This function should return the past samples as a [additional dimensions x] niterations x ndim array, where ndim are the number of sampling params, niterations the number of iterations, and additional dimensions are any additional dimensions used by the sampler (e.g, walkers, temperatures).

**clear_chain**()
This function should clear the current chain of samples from memory.

**classmethod from_cli**(*opts*, *model*, *pool=None*, *model_call=None*)
Create an instance of this sampler from the given command-line options.

**Parameters**

- **opts** (*ArgumentParser options*) – The options to parse.
- **model** (*Model*) – A model from `gwin.models`.

**Returns** A MCMC sampler initialized based on the given arguments.

**Return type** *MCMCSampler*

**lnpost**
This function should return the natural logarithm of the likelihood function used by the sampler as an [additional dimensions] x niterations array.

**name = 'mcmc'**

**niterations**
Get the current number of iterations.

**p0**
Since this is just a single chain, forces p0 to have shape (nparams,).

**run**(*niterations*)
This function should run the sampler.

**write_acceptance_fraction**(*fp*, *start_iteration=None*, *max_iterations=None*)
Write acceptance_fraction data to file. Results are written to `fp[acceptance_fraction]`.

**Parameters** **fp** (`InferenceFile`) – A file handler to an open inference file.

#### Module contents

This modules provides a list of implemented samplers for parameter estimation.

### 2.1.5 gwin.utils package

#### Submodules

#### gwin.utils.sphinx module

Sphinx/RST for GWIn

gwin.utils.sphinx.**rst_dict_table**(*dict_*, *key_format=<type 'str'>*, *val_format=<type 'str'>*, *header=None*)
Returns an RST-formatted table of keys and values from a `dict`

**Parameters**

- **dict** (dict) – data to display in table
- **key_format** (callable) – callable function with which to format keys
- **val_format** (callable) – callable function with which to format values
- **header** (None, tuple of str) – a 2-tuple of header for the two columns, or None to exclude a header line (default)

### Examples

```
>>> a = {'key1': 'value1', 'key2': 'value2'}
>>> print(rst_dict_table(a))
====  ======
key1  value1
key2  value2
====  ======
>>> print(rst_dict_table(a, key_format='``{}``'.format,
...                         val_format=':class:`{}`'.format,
...                         header=['Key', 'Value']))
========  ===============
Key       Value
========  ===============
``key1``  :class:`value1`
``key2``  :class:`value2`
========  ===============
```

### Module contents

Utilities for GWIn

## 2.2 Submodules

### 2.2.1 gwin.burn_in module

This modules provides classes and functions for determining when Markov Chains have burned in.

**class** gwin.burn_in.**BurnIn**(*function_names*, *min_iterations=0*)

Bases: object

Class to estimate the number of burn in iterations.

**Parameters**

- **function_names** (*list, optional*) – List of name of burn in functions to use. All names in the provided list muset be in the burn_in_functions dict. If none provided, will use no burn-in functions.
- **min_iterations** (*int, optional*) – Minimum number of burn in iterations to use. The burn in iterations returned by evaluate will be the maximum of this value and the values returned by the burn in functions provided in function_names. Default is 0.

### Examples

Initialize a *BurnIn* instance that will use *max_posterior* and *posterior_step* as the burn in criteria:

```
>>> import gwin
>>> burn_in = gwin.BurnIn(['max_posterior', 'posterior_step'])
```

Use this *BurnIn* instance to find the burn-in iteration of each walker in an inference result file:

```
>>> from pycbc.io import InferenceFile
>>> fp = InferenceFile('gwin.hdf', 'r')
>>> burn_in.evaluate(gwin.samplers[fp.sampler_name], fp)
array([11486, 11983, 11894, ..., 11793, 11888, 11981])
```

**evaluate** (*sampler*, *fp*)
Evaluates sampler's chains to find burn in.

**Parameters**

- **sampler** (*gwin.sampler*) – Sampler to determine burn in for. May be either an instance of a *gwin.sampler*, or the class itself.
- **fp** (InferenceFile) – Open inference hdf file containing the samples to load for determing burn in.

**Returns**

- **burnidx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in.

**update** (*sampler*, *fp*)
Evaluates burn in and saves the updated indices to the given file.

**Parameters**

- **sampler** (*gwin.sampler*) – Sampler to determine burn in for. May be either an instance of a *gwin.sampler*, or the class itself.
- **fp** (InferenceFile) – Open inference hdf file containing the samples to load for determing burn in.

**Returns**

- **burnidx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in.

gwin.burn_in.**half_chain** (*sampler*, *fp*)
Takes the second half of the iterations as post-burn in.

**Parameters**

- **sampler** (*gwin.sampler*) – This option is not used; it is just here give consistent API as the other burn in functions.
- **fp** (InferenceFile) – Open inference hdf file containing the samples to load for determing burn in.

**Returns**

- **burn_in_idx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in. By definition of this function, all values are set to True.

gwin.burn_in.**ks_test**(*sampler*, *fp*, *threshold=0.9*)
Burn in based on whether the p-value of the KS test between the samples at the last iteration and the samples midway along the chain for each parameter is > `threshold`.

**Parameters**

- **sampler** (*gwin.sampler*) – Sampler to determine burn in for. May be either an instance of a *gwin.sampler*, or the class itself.
- **fp** (`InferenceFile`) – Open inference hdf file containing the samples to load for determing burn in.
- **threshold** (*float*) – The thershold to use for the p-value. Default is 0.9.

**Returns**

- **burn_in_idx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in.

gwin.burn_in.**max_posterior**(*sampler*, *fp*)
Burn in based on samples being within dim/2 of maximum posterior.

**Parameters**

- **sampler** (*gwin.sampler*) – Sampler to determine burn in for. May be either an instance of a *gwin.sampler*, or the class itself.
- **fp** (`InferenceFile`) – Open inference hdf file containing the samples to load for determing burn in.

**Returns**

- **burn_in_idx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in.

gwin.burn_in.**n_acl**(*sampler*, *fp*, *nacls=10*)
Burn in based on ACL.

The sampler is considered burned in if the number of itertions is >= `nacls` times the maximum ACL over all parameters, as measured from the first iteration.

**Parameters**

- **sampler** (*pycbc.inference.sampler*) – Sampler to determine burn in for. May be either an instance of a `inference.sampler`, or the class itself.
- **fp** (`InferenceFile`) – Open inference hdf file containing the samples to load for determing burn in.
- **nacls** (*int*) – Number of ACLs to use for burn in. Default is 10.

**Returns**

- **burn_in_idx** (*array*) – Array of indices giving the burn-in index for each chain. By definition of this function, all chains reach burn in at the same iteration. Thus the returned array is the burn-in index repeated by the number of chains.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in. Since all chains obtain burn in at the same time, this is either an array of all False or True.

gwin.burn_in.**posterior_step**(*sampler*, *fp*)
Burn in based on the last time a chain made a jump > dim/2.

**Parameters**

- **sampler** (*gwin.sampler*) – Sampler to determine burn in for. May be either an instance of a *gwin.sampler*, or the class itself.
- **fp** (InferenceFile) – Open inference hdf file containing the samples to load for determing burn in.

**Returns**

- **burn_in_idx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in. By definition of this function, all values are set to True.

gwin.burn_in.**use_sampler**(*sampler*, *fp=None*)
Uses the sampler's burn_in function.

**Parameters**

- **sampler** (*gwin.sampler*) – Sampler to determine burn in for. Must be an instance of an *gwin.sampler* that has a `burn_in` function.
- **fp** (InferenceFile, *optional*) – This option is not used; it is just here give consistent API as the other burn in functions.

**Returns**

- **burn_in_idx** (*array*) – Array of indices giving the burn-in index for each chain.
- **is_burned_in** (*array*) – Array of booleans indicating whether each chain is burned in. Since the sampler's burn in function will run until all chains are burned, all values are set to True.

### 2.2.2 gwin.calibration module

Functions for adding calibration factors to waveform templates.

**class** gwin.calibration.**CubicSpline**(*minimum_frequency*, *maximum_frequency*, *n_points*, *ifo_name*)
Bases: *gwin.calibration.Recalibrate*

**apply_calibration**(*strain*)
Apply calibration model

This applies cubic spline calibration to the strain.

**Parameters** **strain** (*FrequencySeries*) – The strain to be recalibrated.

**Returns** **strain_adjusted** – The recalibrated strain.

**Return type** FrequencySeries

**name = 'cubic_spline'**

**class** gwin.calibration.**Recalibrate**(*ifo_name*)
Bases: object

**apply_calibration**(*strain*)
Apply calibration model

This method should be overwritten by subclasses

**Parameters** **strain** (*FrequencySeries*) – The strain to be recalibrated.

**Returns** **strain_adjusted** – The recalibrated strain.

**Return type** FrequencySeries

**classmethod from_config**(*cp*, *ifo*, *section*)
Read a config file to get calibration options and transfer functions which will be used to intialize the model.

**Parameters**

- **cp** (*WorkflowConfigParser*) – An open config file.
- **ifo** (*string*) – The detector (H1, L1) for which the calibration model will be loaded.
- **section** (*string*) – The section name in the config file from which to retrieve the calibration options.

**Returns** An instance of the class.

**Return type** instance

**map_to_adjust**(*strain*, *prefix='recalib_'*, ***params*)
Map an input dictionary of sampling parameters to the adjust_strain function by filtering the dictionary for the calibration parameters, then calling adjust_strain.

**Parameters**

- **strain** (*FrequencySeries*) – The strain to be recalibrated.
- **prefix** (*str*) – Prefix for calibration parameter names
- **params** (*dict*) – Dictionary of sampling parameters which includes calibration parameters.

**Returns** **strain_adjusted** – The recalibrated strain.

**Return type** FrequencySeries

**name = None**

### 2.2.3 gwin.entropy module

The module contains functions for calculating the Kullback-Leibler divergence.

gwin.entropy.**kl**(*samples1*, *samples2*, *pdf1=False*, *pdf2=False*, *bins=30*, *hist_min=None*, *hist_max=None*)
Computes the Kullback-Leibler divergence for a single parameter from two distributions.

**Parameters**

- **samples1** (*numpy.array*) – Samples or probability density function (must also set `pdf1=True`).
- **samples2** (*numpy.array*) – Samples or probability density function (must also set `pdf2=True`).
- **pdf1** (*bool*) – Set to `True` if `samples1` is a probability density funtion already.
- **pdf2** (*bool*) – Set to `True` if `samples2` is a probability density funtion already.
- **bins** (*int*) – Number of bins to use when calculating probability density function from a set of samples of the distribution.
- **hist_min** (*numpy.float64*) – Minimum of the distributions' values to use.
- **hist_max** (*numpy.float64*) – Maximum of the distributions' values to use.

**Returns** The Kullback-Leibler divergence value.

**Return type** numpy.float64

### 2.2.4 gwin.gelman_rubin module

This modules provides functions for evaluating the Gelman-Rubin convergence diagnostic statistic.

gwin.gelman_rubin.**gelman_rubin**(*chains*, *auto_burn_in=True*)

Calculates the univariate Gelman-Rubin convergence statistic which compares the evolution of multiple chains in a Markov-Chain Monte Carlo process and computes their difference to determine their convergence. The between-chain and within-chain variances are computed for each sampling parameter, and a weighted combination of the two is used to determine the convergence. As the chains converge, the point scale reduction factor should go to 1.

**Parameters**

- **chains** (*iterable*) – An iterable of numpy.array instances that contain the samples for each chain. Each chain has shape (nparameters, niterations).
- **auto_burn_in** (*bool*) – If True, then only use later half of samples provided.

**Returns** **psrf** – A numpy.array of shape (nparameters) that has the point estimates of the potential scale reduction factor.

**Return type** numpy.array

gwin.gelman_rubin.**walk**(*chains*, *start*, *end*, *step*)

Calculates Gelman-Rubin conervergence statistic along chains of data. This function will advance along the chains and calculate the statistic for each step.

**Parameters**

- **chains** (*iterable*) – An iterable of numpy.array instances that contain the samples for each chain. Each chain has shape (nparameters, niterations).
- **start** (*float*) – Start index of blocks to calculate all statistics.
- **end** (*float*) – Last index of blocks to calculate statistics.
- **step** (*float*) – Step size to take for next block.

**Returns**

- **starts** (*numpy.array*) – 1-D array of start indexes of calculations.
- **ends** (*numpy.array*) – 1-D array of end indexes of caluclations.
- **stats** (*numpy.array*) – Array with convergence statistic. It has shape (nparameters, ncalculations).

### 2.2.5 gwin.geweke module

Functions for computing the Geweke convergence statistic.

gwin.geweke.**geweke**(*x*, *seg_length*, *seg_stride*, *end_idx*, *ref_start*, *ref_end=None*, *seg_start=0*)

Calculates Geweke conervergence statistic for a chain of data. This function will advance along the chain and calculate the statistic for each step.

**Parameters**

- **x** (*numpy.array*) – A one-dimensional array of data.
- **seg_length** (*int*) – Number of samples to use for each Geweke calculation.
- **seg_stride** (*int*) – Number of samples to advance before next Geweke calculation.
- **end_idx** (*int*) – Index of last start.

- **ref_start** (*int*) – Index of beginning of end reference segment.
- **ref_end** (*int*) – Index of end of end reference segment. Default is None which will go to the end of the data array.
- **seg_start** (*int*) – What index to start computing the statistic. Default is 0 which will go to the beginning of the data array.

**Returns**

- **starts** (*numpy.array*) – The start index of the first segment in the chain.
- **ends** (*numpy.array*) – The end index of the first segment in the chain.
- **stats** (*numpy.array*) – The Geweke convergence diagnostic statistic for the segment.

### 2.2.6 gwin.option_utils module

This module contains standard options used for inference-related programs.

gwin.option_utils.**add_config_opts_to_parser**(*parser*)
Adds options for the configuration files to the given parser.

gwin.option_utils.**add_density_option_group**(*parser*)
Adds the options needed to configure contours and density colour map.

**Parameters** **parser** (*object*) – ArgumentParser instance.

gwin.option_utils.**add_inference_results_option_group**(*parser*, *include_parameters_group=True*)
Adds the options used to call gwin.results_from_cli function to an argument parser. These are options releated to loading the results from a run of pycbc_inference, for purposes of plotting and/or creating tables.

**Parameters**

- **parser** (*object*) – ArgumentParser instance.
- **include_parameters_group** (*bool*) – If true then include `--parameters-group` option.

gwin.option_utils.**add_low_frequency_cutoff_opt**(*parser*)
Adds the low-frequency-cutoff option to the given parser.

gwin.option_utils.**add_plot_posterior_option_group**(*parser*)
Adds the options needed to configure plots of posterior results.

**Parameters** **parser** (*object*) – ArgumentParser instance.

gwin.option_utils.**add_sampler_option_group**(*parser*)
Adds the options needed to set up an inference sampler.

**Parameters** **parser** (*object*) – ArgumentParser instance.

gwin.option_utils.**add_scatter_option_group**(*parser*)
Adds the options needed to configure scatter plots.

**Parameters** **parser** (*object*) – ArgumentParser instance.

gwin.option_utils.**config_parser_from_cli**(*opts*)
Loads a config file from the given options, applying any overrides specified. Specifically, config files are loaded from the `--config-files` options while overrides are loaded from `--config-overrides`.

gwin.option_utils.**data_from_cli**(*opts*)
Loads the data needed for a model from the given command-line options. Gates specifed on the command line are also applied.

**Parameters** **opts** (*ArgumentParser parsed args*) – Argument options parsed from a command line string (the sort of thing returned by `parser.parse_args`).

**Returns**

- **strain_dict** (*dict*) – Dictionary of instruments -> `TimeSeries` strain.
- **stilde_dict** (*dict*) – Dictionary of instruments -> `FrequencySeries` strain.
- **psd_dict** (*dict*) – Dictionary of instruments -> `FrequencySeries` psds.

gwin.option_utils.**expected_parameters_from_cli**(*opts*)
Parses the –expected-parameters arguments from the `plot_posterior` option group.

**Parameters** **opts** (*ArgumentParser*) – The parsed arguments from the command line.

**Returns** Dictionary of parameter name -> expected value. Only parameters that were specified in the –expected-parameters option will be included; if no parameters were provided, will return an empty dictionary.

**Return type** dict

gwin.option_utils.**get_file_type**(*filename*)
Returns I/O object to use for file.

**Parameters** **filename** (*str*) – Name of file.

**Returns** **file_type** – The type of inference file object to use.

**Return type** {InferenceFile, InferenceTXTFile}

gwin.option_utils.**injections_from_cli**(*opts*)
Gets injection parameters from the inference file(s).

**Parameters** **opts** (*argparser*) – Argparser object that has the command-line objects to parse.

**Returns** Array of the injection parameters from all of the input files given by `opts.input_file`.

**Return type** FieldArray

gwin.option_utils.**low_frequency_cutoff_from_cli**(*opts*)
Parses the low frequency cutoff from the given options.

**Returns** Dictionary of instruments -> low frequency cutoff.

**Return type** dict

gwin.option_utils.**parse_parameters_opt**(*parameters*)
Parses the –parameters opt in the results_reading_group.

**Parameters** **parameters** (*list of str or None*) – The parameters to parse.

**Returns**

- **parameters** (*list of str*) – The parameters.
- **labels** (*dict*) – A dictionary mapping parameters for which labels were provide to those labels.

gwin.option_utils.**plot_ranges_from_cli**(*opts*)
Parses the mins and maxs arguments from the `plot_posterior` option group.

**Parameters** **opts** (*ArgumentParser*) – The parsed arguments from the command line.

**Returns**

- **mins** (*dict*) – Dictionary of parameter name -> specified mins. Only parameters that were specified in the –mins option will be included; if no parameters were provided, will return an empty dictionary.
- **maxs** (*dict*) – Dictionary of parameter name -> specified maxs. Only parameters that were specified in the –mins option will be included; if no parameters were provided, will return an empty dictionary.

gwin.option_utils.**results_from_cli**(*opts*, *load_samples=True*, ***kwargs*)

Loads an inference result file along with any labels associated with it from the command line options.

**Parameters**

- **opts** (*ArgumentParser options*) – The options from the command line.
- **load_samples** (*{True, bool}*) – Load samples from the results file using the parameters, thin_start, and thin_interval specified in the options. The samples are returned as a FieldArray instance.
- ****kwargs** – All other keyword arguments are passed to the InferenceFile's read_samples function.

**Returns**

- **fp_all** (*pycbc.io.InferenceFile*) – The result file as an InferenceFile. If more than one input file, then it returns a list.
- **parameters_all** (*list*) – List of the parameters to use, parsed from the parameters option. If more than one input file, then it returns a list.
- **labels_all** (*list*) – List of labels to associate with the parameters. If more than one input file, then it returns a list.
- **samples_all** (*{None, FieldArray}*) – If load_samples, the samples as a FieldArray; otherwise, None. If more than one input file, then it returns a list.

gwin.option_utils.**sampler_from_cli**(*opts*, *model*, *pool=None*)

Parses the given command-line options to set up a sampler.

**Parameters**

- **opts** (*object*) – ArgumentParser options.
- **model** (*model*) – The model to use with the sampler.

**Returns** A sampler initialized based on the given arguments.

**Return type** gwin.sampler

gwin.option_utils.**validate_checkpoint_files**(*checkpoint_file*, *backup_file*)

Checks if the given checkpoint and/or backup files are valid.

The checkpoint file is considered valid if:

- it passes all tests run by `InferenceFile.check_integrity`;
- it has at least one sample written to it (indicating at least one checkpoint has happened).

The same applies to the backup file. The backup file must also have the same number of samples as the checkpoint file, otherwise, the backup is considered invalid.

If the checkpoint (backup) file is found to be valid, but the backup (checkpoint) file is not valid, then the checkpoint (backup) is copied to the backup (checkpoint). Thus, this function ensures that checkpoint and backup files are either both valid or both invalid.

**Parameters**

- **checkpoint_file** (*string*) – Name of the checkpoint file.
- **backup_file** (*string*) – Name of the backup file.

**Returns** **checkpoint_valid** – Whether or not the checkpoint (and backup) file may be used for loading samples.

**Return type** bool

### 2.2.7 gwin.workflow module

Module that contains functions for setting up the inference workflow.

gwin.workflow.**make_inference_1d_posterior_plots**(*workflow*, *inference_file*, *output_dir*, *parameters=None*, *analysis_seg=None*, *tags=None*)

gwin.workflow.**make_inference_acceptance_rate_plot**(*workflow*, *inference_file*, *output_dir*, *name='inference_rate'*, *analysis_seg=None*, *tags=None*)

Sets up the acceptance rate plot in the workflow.

**Parameters**

- **workflow** (*pycbc.workflow.Workflow*) – The core workflow instance we are populating
- **inference_file** (*pycbc.workflow.File*) – The file with posterior samples.
- **output_dir** (*str*) – The directory to store result plots and files.
- **name** (*str*) – The name in the [executables] section of the configuration file to use.
- **analysis_segs** (*{None, glue.segments.Segment}*) – The segment this job encompasses. If None then use the total analysis time from the workflow.
- **tags** (*{None, optional}*) – Tags to add to the inference executables.

**Returns** A list of result and output files.

**Return type** pycbc.workflow.FileList

gwin.workflow.**make_inference_inj_plots**(*workflow*, *inference_files*, *output_dir*, *parameters*, *name='inference_recovery'*, *analysis_seg=None*, *tags=None*)

Sets up the recovered versus injected parameter plot in the workflow.

**Parameters**

- **workflow** (*pycbc.workflow.Workflow*) – The core workflow instance we are populating
- **inference_files** (*pycbc.workflow.FileList*) – The files with posterior samples.
- **output_dir** (*str*) – The directory to store result plots and files.
- **parameters** (*list*) – A `list` of parameters. Each parameter gets its own plot.
- **name** (*str*) – The name in the [executables] section of the configuration file to use.
- **analysis_segs** (*{None, glue.segments.Segment}*) – The segment this job encompasses. If None then use the total analysis time from the workflow.

- **tags** (*{None, optional}*) – Tags to add to the inference executables.

**Returns** A list of result and output files.

**Return type** pycbc.workflow.FileList

gwin.workflow.**make_inference_posterior_plot**(*workflow*, *inference_file*, *output_dir*, *parameters=None*, *name='inference_posterior'*, *analysis_seg=None*, *tags=None*)

Sets up the corner plot of the posteriors in the workflow.

**Parameters**

- **workflow** (*pycbc.workflow.Workflow*) – The core workflow instance we are populating
- **inference_file** (*pycbc.workflow.File*) – The file with posterior samples.
- **output_dir** (*str*) – The directory to store result plots and files.
- **parameters** (*list*) – A list of parameters to plot.
- **name** (*str*) – The name in the [executables] section of the configuration file to use.
- **analysis_segs** (*{None, glue.segments.Segment}*) – The segment this job encompasses. If None then use the total analysis time from the workflow.
- **tags** (*{None, optional}*) – Tags to add to the inference executables.

**Returns** A list of result and output files.

**Return type** pycbc.workflow.FileList

gwin.workflow.**make_inference_prior_plot**(*workflow*, *config_file*, *output_dir*, *sections=None*, *name='inference_prior'*, *analysis_seg=None*, *tags=None*)

Sets up the corner plot of the priors in the workflow.

**Parameters**

- **workflow** (*pycbc.workflow.Workflow*) – The core workflow instance we are populating
- **config_file** (*pycbc.workflow.File*) – The WorkflowConfigParser parasable inference configuration file..
- **output_dir** (*str*) – The directory to store result plots and files.
- **sections** (*list*) – A list of subsections to use.
- **name** (*str*) – The name in the [executables] section of the configuration file to use.
- **analysis_segs** (*{None, glue.segments.Segment}*) – The segment this job encompasses. If None then use the total analysis time from the workflow.
- **tags** (*{None, optional}*) – Tags to add to the inference executables.

**Returns** A list of result and output files.

**Return type** pycbc.workflow.FileList

gwin.workflow.**make_inference_samples_plot**(*workflow*, *inference_file*, *output_dir*, *parameters=None*, *name='inference_samples'*, *analysis_seg=None*, *tags=None*)

gwin.workflow.**make_inference_summary_table**(*workflow*, *inference_file*, *output_dir*, *variable_params=None*, *name='inference_table'*, *analysis_seg=None*, *tags=None*)

Sets up the corner plot of the posteriors in the workflow.

**Parameters**

- **workflow** (*pycbc.workflow.Workflow*) – The core workflow instance we are populating
- **inference_file** (*pycbc.workflow.File*) – The file with posterior samples.
- **output_dir** (*str*) – The directory to store result plots and files.
- **variable_params** (*list*) – A list of parameters to use instead of [variable_params].
- **name** (*str*) – The name in the [executables] section of the configuration file to use.
- **analysis_segs** (*{None, glue.segments.Segment}*) – The segment this job encompasses. If None then use the total analysis time from the workflow.
- **tags** (*{None, optional}*) – Tags to add to the inference executables.

**Returns** A list of result and output files.

**Return type** pycbc.workflow.FileList

gwin.workflow.**setup_foreground_inference**(*workflow*, *coinc_file*, *single_triggers*, *tmpltbank_file*, *insp_segs*, *insp_data_name*, *insp_anal_name*, *dax_output*, *out_dir*, *tags=None*)

Creates workflow node that will run the inference workflow.

**Parameters**

- **workflow** (*pycbc.workflow.Workflow*) – The core workflow instance we are populating
- **coinc_file** (*pycbc.workflow.File*) – The file associated with coincident triggers.
- **single_triggers** (*list of pycbc.workflow.File*) – A list cointaining the file objects associated with the merged single detector trigger files for each ifo.
- **tmpltbank_file** (*pycbc.workflow.File*) – The file object pointing to the HDF format template bank
- **insp_segs** (*SegFile*) – The segment file containing the data read and analyzed by each inspiral job.
- **insp_data_name** (*str*) – The name of the segmentlist storing data read.
- **insp_anal_name** (*str*) – The name of the segmentlist storing data analyzed.
- **dax_output** (*str*) – The name of the output DAX file.
- **out_dir** (*path*) – The directory to store inference result plots and files
- **tags** (*{None, optional}*) – Tags to add to the inference executables

## 2.3 Module contents

GWIn is a package for parameter estimation of gravitational-wave data using Bayesian Inference.

CHAPTER 3

# Running on the command line

## 3.1 GWin on the command line (`gwin`)

### 3.1.1 Introduction

This page gives details on how to use the various parameter estimation executables and modules available in GWin. The `gwin` subpackage contains classes and functions for evaluating probability distributions, likelihoods, and running Bayesian samplers.

### 3.1.2 Sampling the parameter space (`gwin`)

#### Overview

The executable `gwin` is designed to sample the parameter space and save the samples in an HDF file. A high-level description of the `gwin` algorithm is

1. Read priors from a configuration file.
2. Setup the model to use. If the model uses data, then:
   (a) Read gravitational-wave strain from a gravitational-wave model or use recolored fake strain.
   (b) Estimate a PSD.
3. Run a sampler to estimate the posterior distribution of the model.

#### Options for samplers, models, and priors

For a full listing of all options run `gwin --help`. In this subsection we reference documentation for Python classes that contain more information about choices for samplers, likelihood models, and priors.

The user specifies the sampler on the command line with the `--sampler` option. A complete list of samplers is given in `gwin --help`. These samplers are described in `gwin.sampler.KombineSampler`, `gwin.sampler.EmceeEnsembleSampler`, and `gwin.sampler.EmceePTSampler`. In addition to `--sampler` the user will need to specify the number of walkers to use `--nwalkers`, and for parallel-tempered samplers the number of temperatures `--ntemps`. You also need to either specify the number of iterations to run for using `--niterations` **or** the number of independent samples to collect using `--n-independent-samples`. For the latter, a burn-in function must be specified using `--burn-in-function`. In this case, the program will run until the sampler has burned in, at which point the number of independent samples equals the number of walkers. If the number of independent samples desired is greater than the number of walkers, the program will continue to run until it has collected the specified number of independent samples (to do this, an autocorrelation length is computed at each checkpoint to determine how many iterations need to be skipped to obtain independent samples).

The user specifies a configuration file that defines the priors with the `--config-files` option. The syntax of the configuration file is described in the following subsection.

### Configuration file syntax

Configuration files follow the `ConfigParser` syntax. There are three required sections.

One is a `[model]` section that contains the name of the model class to use for evaluating the posterior. An example:

```
[model]
name = gaussian_noise
```

In this case, the `gwin.models.GaussianNoise` would be used. Examples of using this model on a BBH injection and on GW150914 are given below. Any name that starts with `test_` is an analytic test distribution that requires no data or waveform generation; see the section below on running on an analytic distribution for more details.

The other two required sections are `[variable_params]`, and `[static_params]`. The `[variable_params]` section contains a list of parameters in which the prior will be defined, and that will varied to obtain a posterior distribution. The `[static_params]` section contains a list of parameters that are held fixed through out the run.

Each parameter in `[variable_params]` must have a subsection in `[prior]`. To create a subsection use the `-` char, e.g. for one of the mass parameters do `[prior-mass1]`.

Each prior subsection must have a `name` option that identifies what prior to use. These distributions are described in `pycbc.distributions`.

| | |
|---|---|
| 'uniform_power_law' | pycbc.distributions.power_law.UniformPowerLaw |
| 'gaussian' | pycbc.distributions.gaussian.Gaussian |
| 'uniform_log10' | pycbc.distributions.uniform_log.UniformLog10 |
| 'uniform_solidangle' | pycbc.distributions.angular.UniformSolidAngle |
| 'uniform_sky' | pycbc.distributions.sky_location.UniformSky |
| 'cos_angle' | pycbc.distributions.angular.CosAngle |
| 'uniform_radius' | pycbc.distributions.power_law.UniformRadius |
| 'independent_chip_chieff' | pycbc.distributions.spins.IndependentChiPChiEff |
| 'sin_angle' | pycbc.distributions.angular.SinAngle |
| 'uniform' | pycbc.distributions.uniform.Uniform |
| 'uniform_angle' | pycbc.distributions.angular.UniformAngle |
| 'arbitrary' | pycbc.distributions.arbitrary.Arbitrary |
| 'fromfile' | pycbc.distributions.arbitrary.FromFile |
| 'uniform_component_masses' | pycbc.distributions.masses.UniformComponentMasses |

One or more of the `variable_params` may be transformed to a different parameter space for purposes of sampling. This is done by specifying a `[sampling_params]` section. This section specifies which `variable_params` to replace with which parameters for sampling. This must be followed by one or more `[sampling_transforms-{sampling_params}]` sections that provide the transform class to use. For example, the following would cause the sampler to sample in chirp mass (`mchirp`) and mass ratio (`q`) instead of `mass1` and `mass2`:

```
[sampling_params]
mass1, mass2: mchirp, q

[sampling_transforms-mchirp+q]
name = mass1_mass2_to_mchirp_q
```

For a list of all possible transforms see `pycbc.transforms`.

There can be any number of `variable_params` with any name. No parameter name is special (with the exception of parameters that start with `calib_`; see below). However, in order to generate waveforms, certain parameters must be provided for waveform generation. If you would like to specify a `variable_arg` that is not one of these parameters, then you must provide a `[waveforms_transforms-{param}]` section that provides a transform from the arbitrary `variable_params` to the needed waveform parameter(s) `{param}`. For example, in the following we provide a prior on `chirp_distance`. Since `distance`, not `chirp_distance`, is recognized by the CBC waveforms module, we provide a transform to go from `chirp_distance` to `distance`:

```
[variable_params]
chirp_distance =

[prior-chirp_distance]
name = uniform
min-chirp_distance = 1
max-chirp_distance = 200

[waveform_transforms-distance]
name = chirp_distance_to_distance
```

Any class in the transforms module may be used. A useful transform for these purposes is the `pycbc.transforms.CustomTransform`, which allows for arbitrary transforms using any function in the `pycbc.conversions`, `pycbc.coordinates`, or `pycbc.cosmology` modules, along with numpy math functions. For example, the following would use the I-Love-Q relationship `pycbc.conversions.dquadmon_from_lambda()` to relate the quadrupole moment of a neutron star `dquad_mon1` to its tidal deformation `lambda1`:

```
[variable_params]
lambda1 =

[waveform_transforms-dquad_mon1]
name = custom
inputs = lambda1
dquad_mon1 = dquadmon_from_lambda(lambda1)
```

The following table lists all parameters understood by the CBC waveform generator:

| Parameter | Description |
|---|---|
| `'mass1'` | The mass of the first component object in the binary (in solar masses). |
| `'mass2'` | The mass of the second component object in the binary (in solar masses). |
| `'spin1x'` | The x component of the first binary component's dimensionless spin. |
| `'spin1y'` | The y component of the first binary component's dimensionless spin. |

Table 1 – continued from pr

| Parameter | Description |
|---|---|
| 'spin1z' | The z component of the first binary component's dimensionless spin. |
| 'spin2x' | The x component of the second binary component's dimensionless spin. |
| 'spin2y' | The y component of the second binary component's dimensionless spin. |
| 'spin2z' | The z component of the second binary component's dimensionless spin. |
| 'eccentricity' | Eccentricity. |
| 'lambda1' | The dimensionless tidal deformability parameter of object 1. |
| 'lambda2' | The dimensionless tidal deformability parameter of object 2. |
| 'dquad_mon1' | Quadrupole-monopole parameter / m_1^5 -1. |
| 'dquad_mon2' | Quadrupole-monopole parameter / m_2^5 -1. |
| 'lambda_octu1' | The octupolar tidal deformability parameter of object 1. |
| 'lambda_octu2' | The octupolar tidal deformability parameter of object 2. |
| 'quadfmode1' | The quadrupolar f-mode angular frequency of object 1. |
| 'quadfmode2' | The quadrupolar f-mode angular frequency of object 2. |
| 'octufmode1' | The octupolar f-mode angular frequency of object 1. |
| 'octufmode2' | The octupolar f-mode angular frequency of object 2. |
| 'distance' | Luminosity distance to the binary (in Mpc). |
| 'coa_phase' | Coalesence phase of the binary (in rad). |
| 'inclination' | Inclination (rad), defined as the angle between the total angular momentum J and the line-of-sight. |
| 'long_asc_nodes' | Longitude of ascending nodes axis (rad). |
| 'mean_per_ano' | Mean anomaly of the periastron (rad). |
| 'delta_f' | The frequency step used to generate the waveform (in Hz). |
| 'f_lower' | The starting frequency of the waveform (in Hz). |
| 'approximant' | A string that indicates the chosen approximant. |
| 'f_ref' | The reference frequency. |
| 'phase_order' | The pN order of the orbital phase. The default of -1 indicates that all implemented orders are used. |
| 'spin_order' | The pN order of the spin corrections. The default of -1 indicates that all implemented orders are used |
| 'tidal_order' | The pN order of the tidal corrections. The default of -1 indicates that all implemented orders are used |
| 'amplitude_order' | The pN order of the amplitude. The default of -1 indicates that all implemented orders are used. |
| 'eccentricity_order' | The pN order of the eccentricity corrections.The default of -1 indicates that all implemented orders a |
| 'frame_axis' | Allow to choose among orbital_l, view and total_j |
| 'modes_choice' | Allow to turn on among orbital_l, view and total_j |
| 'side_bands' | Flag for generating sidebands |
| 'mode_array' | Choose which (l,m) modes to include when generating a waveform. Only if approximant supports thi |
| 'f_final' | The ending frequency of the waveform. The default (0) indicates that the choice is made by the respe |
| 'f_final_func' | Use the given frequency function to compute f_final based on the parameters of the waveform. |
| 'tc' | Coalescence time (s). |
| 'ra' | Right ascension (rad). |
| 'dec' | Declination (rad). |
| 'polarization' | Polarization (rad). |
| 'calib_delta_fc' | Change in cavity pole freq (Hz). |
| 'calib_delta_fs' | Change in optical spring freq (Hz). |
| 'calib_delta_qinv' | Change in inverse quality factor. |
| 'calib_kappa_c' | No description. |
| 'calib_kappa_tst_re' | No description. |
| 'calib_kappa_tst_im' | No description. |
| 'calib_kappa_pu_re' | No description. |
| 'calib_kappa_pu_im' | No description. |

Some common transforms are pre-defined in the code. These are: the mass parameters `mass1` and `mass2` can be substituted with `mchirp` and `eta` or `mchirp` and `q`. The component spin parameters `spin1x`, `spin1y`, and

`spin1z` can be substituted for polar coordinates `spin1_a`, `spin1_azimuthal`, and `spin1_polar` (ditto for `spin2`).

If any calibration parameters are used (prefix `calib_`), a `[calibration]` section must be included. This section must have a `name` option that identifies what calibration model to use. The models are described in `pycbc.calibration`. The `[calibration]` section must also include reference values `fc0`, `fs0`, and `qinv0`, as well as paths to ASCII transfer function files for the test mass actuation, penultimate mass actuation, sensing function, and digital filter for each IFO being used in the analysis. E.g. for an analysis using H1 only, the required options would be `h1-fc0`, `h1-fs0`, `h1-qinv0`, `h1-transfer-function-a-tst`, `h1-transfer-function-a-pu`, `h1-transfer-function-c`, `h1-transfer-function-d`.

Simple examples are given in the subsections below.

### Running on an analytic distribution

Several analytic distributions are available to run tests on. These can be run quickly on a laptop to check that a sampler is working properly.

This example demonstrates how to sample a 2D normal distribution with the `emcee` sampler. First, create the following configuration file (named `normal2d.ini`):

```
[model]
name = test_normal

[variable_params]
x =
y =

[prior-x]
name = uniform
min-x = -10
max-x = 10

[prior-y]
name = uniform
min-y = -10
max-y = 10
```

Then run:

```
gwin --verbose \
    --config-files normal2d.ini \
    --output-file normal2d.hdf \
    --sampler emcee \
    --niterations 50 \
    --nwalkers 5000 \
    --nprocesses 2
```

This will run the `emcee` sampler on the 2D analytic normal distribution with 5000 walkers for 100 iterations.

To plot the posterior distribution after the last iteration, run:

```
gwin_plot_posterior --verbose \
        --input-file normal2d.hdf \
        --output-file posterior-normal2d.png \
        --plot-scatter \
        --plot-contours \
```

(continues on next page)

(continued from previous page)

```
        --plot-marginal \
        --z-arg 'loglikelihood:$\log p(h|\vartheta)$' \
        --iteration -1
```

This will plot each walker's position as a single point colored by the log likelihood ratio at that point, with the 50th and 90th percentile contours drawn. See below for more information about using `gwin_plot_posterior`.

To make a movie showing how the walkers evolved, run:

```
gwin_plot_movie --verbose \
    --input-file normal2d.hdf \
    --output-prefix frames-normal2d \
    --movie-file normal2d_mcmc_evolution.mp4 \
    --cleanup \
    --plot-scatter \
    --plot-contours \
    --plot-marginal \
    --z-arg 'loglikelihood:$\log p(h|\vartheta)$' \
    --frame-step 1
```

**Note:** you need `ffmpeg` installed for the mp4 to be created. See below for more information on using `gwin_plot_movie`.

The number of dimensions of the distribution is set by the number of `variable_params` in the configuration file. The names of the `variable_params` do not matter, just that the prior sections use the same names (in this example `x` and `y` were used, but `foo` and `bar` would be equally valid). A higher (or lower) dimensional distribution can be tested by simply adding more (or less) `variable_params`.

Which analytic distribution is used is set by the `[model]` section in the configuration file. By setting to `test_normal` we used `gwin.models.TestNormal`. The other analytic distributions available are: `gwin.models.TestEggbox`, `gwin.models.TestRosenbrock`, and `gwin.models.TestVolcano`. As with `test_normal`, the dimensionality of these test distributions is set by the number of `variable_params` in the configuration file. The `test_volcano` distribution must be two dimensional, but all of the other distributions can have any number of dimensions. The configuration file syntax for the other test distributions is the same as in this example (aside from the name used in the model section). Indeed, with this configuration file one only needs to change the `name` argument in `[model]` argument to try (2D versions of) the other distributions.

### BBH software injection example

This example recovers the parameters of a precessing binary black-hole (BBH).

An example configuration file (named `gwin.ini`) is:

```
[model]
name = gaussian_noise

[variable_params]
; waveform parameters that will vary in MCMC
tc =
mass1 =
mass2 =
spin1_a =
spin1_azimuthal =
spin1_polar =
spin2_a =
spin2_azimuthal =
```

(continues on next page)

(continued from previous page)

```
spin2_polar =
distance =
coa_phase =
inclination =
polarization =
ra =
dec =

[static_params]
; waveform parameters that will not change in MCMC
approximant = IMRPhenomPv2
f_lower = 18
f_ref = 20

[prior-tc]
; coalescence time prior
name = uniform
min-tc = 1126259462.32
max-tc = 1126259462.52

[prior-mass1]
name = uniform
min-mass1 = 10.
max-mass1 = 80.

[prior-mass2]
name = uniform
min-mass2 = 10.
max-mass2 = 80.

[prior-spin1_a]
name = uniform
min-spin1_a = 0.0
max-spin1_a = 0.99

[prior-spin1_polar+spin1_azimuthal]
name = uniform_solidangle
polar-angle = spin1_polar
azimuthal-angle = spin1_azimuthal

[prior-spin2_a]
name = uniform
min-spin2_a = 0.0
max-spin2_a = 0.99

[prior-spin2_polar+spin2_azimuthal]
name = uniform_solidangle
polar-angle = spin2_polar
azimuthal-angle = spin2_azimuthal

[prior-distance]
; following gives a uniform volume prior
name = uniform_radius
min-distance = 10
max-distance = 1000

[prior-coa_phase]
```

(continues on next page)

(continued from previous page)

```
; coalescence phase prior
name = uniform_angle

[prior-inclination]
; inclination prior
name = sin_angle

[prior-ra+dec]
; sky position prior
name = uniform_sky

[prior-polarization]
; polarization prior
name = uniform_angle


;
;   Sampling transforms
;
[sampling_params]
; parameters on the left will be sampled in
; parametes on the right
mass1, mass2 : mchirp, q

[sampling_transforms-mchirp+q]
; inputs mass1, mass2
; outputs mchirp, q
name = mass1_mass2_to_mchirp_q
```

To generate an injection we will use `pycbc_create_injections`. This program takes a configuration file to set up the distributions from which to draw the injection parameters (run `pycbc_create_injections --help` for details). The syntax of that file is the same as the file used for `gwin`, so we could, if we wished, simply give the above configuartion file to `pycbc_create_injections`. However, to ensure we obtain a specific set of parameters, we will create another configuration file that fixes the injection parameters to specific values. Create the following file, calling it `injection.ini`:

```
[variable_params]


[static_params]
tc = 1126259462.420
mass1 = 37
mass2 = 32
ra = 2.2
dec = -1.25
inclincation = 2.5
coa_phase = 1.5
polarization = 1.75
distance = 100
f_ref = 20
f_lower = 18
approximant = IMRPhenomPv2
taper = start
```

This will create a non-spinning injection (if no spin parameters are provided, the injection will be non-spinning by default) using `IMRPhenomPv2`. (Note that we still need a `[variable_params]` section even though we are fixing all parameters.) Now run:

```
pycbc_create_injections --verbose \
    --config-files injection.ini \
    --ninjections 1 \
    --output-file injection.hdf \
    --variable-params-section variable_params \
    --static-args-section static_params \
    --dist-section prior
```

This will create a file called `injection.hdf` which contains the injection's parameters. This file can be passed to `gwin` with the `--injection-file` option. To run `gwin` on this injection in simulated fake data, set the following bash variables:

```
# injection parameters
TRIGGER_TIME_INT=1126259462

# sampler parameters
CONFIG_PATH=gwin.ini
OUTPUT_PATH=gwin.hdf
SEGLEN=8
PSD_INVERSE_LENGTH=4
IFOS="H1 L1"
STRAIN="H1:aLIGOZeroDetHighPower L1:aLIGOZeroDetHighPower"
SAMPLE_RATE=2048
F_MIN=20
N_UPDATE=500
N_WALKERS=5000
N_SAMPLES=5000
N_CHECKPOINT=1000
PROCESSING_SCHEME=cpu

# the following sets the number of cores to use; adjust as needed to
# your computer's capabilities
NPROCS=12

# start and end time of data to read in
GPS_START_TIME=$((${TRIGGER_TIME_INT} - ${SEGLEN}))
GPS_END_TIME=$((${TRIGGER_TIME_INT} + ${SEGLEN}))
```

Now run:

```
# run sampler
# specifies the number of threads for OpenMP
# Running with OMP_NUM_THREADS=1 stops lalsimulation
# from spawning multiple jobs that would otherwise be used
# by gwin and cause a reduced runtime.
OMP_NUM_THREADS=1 \
gwin --verbose \
    --seed 12 \
    --instruments ${IFOS} \
    --gps-start-time ${GPS_START_TIME} \
    --gps-end-time ${GPS_END_TIME} \
    --psd-model ${STRAIN} \
    --psd-inverse-length ${PSD_INVERSE_LENGTH} \
    --fake-strain ${STRAIN} \
    --fake-strain-seed 44 \
    --strain-high-pass ${F_MIN} \
    --sample-rate ${SAMPLE_RATE} \
```

(continues on next page)

(continued from previous page)

```
    --low-frequency-cutoff ${F_MIN} \
    --channel-name H1:FOOBAR L1:FOOBAR \
    --injection-file injection.hdf \
    --config-file ${CONFIG_PATH} \
    --output-file ${OUTPUT_PATH} \
    --processing-scheme ${PROCESSING_SCHEME} \
    --sampler kombine \
    --burn-in-function max_posterior \
    --update-interval ${N_UPDATE} \
    --nwalkers ${N_WALKERS} \
    --n-independent-samples ${N_SAMPLES} \
    --checkpoint-interval ${N_CHECKPOINT} \
    --nprocesses ${NPROCS} \
    --save-strain \
    --save-psd \
    --save-stilde \
    --force
```

While the code is running it will write results to `gwin.hdf.checkpoint` after every checkpoint interval (you'll see `Writing results to file` when a checkpoint occurs), with a backup kept in `gwin.hdf.bkup`. At each checkpoint, the number of independent samples that have been obtained to that point will be computed. If the number of independent samples is greater than or equal to `n-independent-samples`, the code will finish and exit. Upon finishing, `gwin.hdf.checkpoint` will be renamed to `gwin.hdf`, and `gwin.hdf.bkup` will be deleted.

If the job fails for any reason while running (say your computer loses power) you can resume from the last checkpoint by re-running the same command as above, but adding `--resume-from-checkpoint`. In this case, the code will automatically detect the checkpoint file, and pickup from where it last left off.

### GW150914 example

The configuration file `gwin.ini` used for the above injection is the same as what you need to analyze the data containing GW150914. You only need to change the command-line arguments to `gwin` to point it to the correct data. To do that, do one of the following:

- **If you are a LIGO member and are running on a LIGO Data Grid cluster:** you can use the LIGO data server to automatically obtain the frame files. Simply set the following environment variables:

```
FRAMES="--frame-type H1:H1_HOFT_C02 L1:L1_HOFT_C02"
CHANNELS="H1:H1:DCS-CALIB_STRAIN_C02 L1:L1:DCS-CALIB_STRAIN_C02"
```

- **If you are not a LIGO member, or are not running on a LIGO Data Grid cluster:** you need to obtain the data from the LIGO Open Science Center. First run the following commands to download the needed frame files to your working directory:

```
wget https://losc.ligo.org/s/events/GW150914/L-L1_LOSC_4_V2-1126257414-4096.gwf
wget https://losc.ligo.org/s/events/GW150914/H-H1_LOSC_4_V2-1126257414-4096.gwf
```

  Then set the following enviornment variables:

```
FRAMES="--frame-files H1:H-H1_LOSC_4_V2-1126257414-4096.gwf L1:L-L1_LOSC_4_V2-
↪1126257414-4096.gwf"
CHANNELS="H1:LOSC-STRAIN L1:LOSC-STRAIN"
```

Now run:

```
# trigger parameters
TRIGGER_TIME=1126259462.42

# data to use
# the longest waveform covered by the prior must fit in these times
SEARCH_BEFORE=6
SEARCH_AFTER=2

# use an extra number of seconds of data in addition to the data specified
PAD_DATA=8

# PSD estimation options
PSD_ESTIMATION="H1:median L1:median"
PSD_INVLEN=4
PSD_SEG_LEN=16
PSD_STRIDE=8
PSD_DATA_LEN=1024

# sampler parameters
CONFIG_PATH=gwin.ini
OUTPUT_PATH=gwin.hdf
IFOS="H1 L1"
SAMPLE_RATE=2048
F_HIGHPASS=15
F_MIN=20
N_UPDATE=500
N_WALKERS=5000
N_SAMPLES=5000
N_CHECKPOINT=1000
PROCESSING_SCHEME=cpu

# the following sets the number of cores to use; adjust as needed to
# your computer's capabilities
NPROCS=12

# get coalescence time as an integer
TRIGGER_TIME_INT=${TRIGGER_TIME%.*}

# start and end time of data to read in
GPS_START_TIME=$((${TRIGGER_TIME_INT} - ${SEARCH_BEFORE} - ${PSD_INVLEN}))
GPS_END_TIME=$((${TRIGGER_TIME_INT} + ${SEARCH_AFTER} + ${PSD_INVLEN}))

# start and end time of data to read in for PSD estimation
PSD_START_TIME=$((${TRIGGER_TIME_INT} - ${PSD_DATA_LEN}/2))
PSD_END_TIME=$((${TRIGGER_TIME_INT} + ${PSD_DATA_LEN}/2))

# run sampler
# specifies the number of threads for OpenMP
# Running with OMP_NUM_THREADS=1 stops lalsimulation
# to spawn multiple jobs that would otherwise be used
# by gwin and cause a reduced runtime.
OMP_NUM_THREADS=1 \
gwin --verbose \
    --seed 12 \
    --instruments ${IFOS} \
    --gps-start-time ${GPS_START_TIME} \
    --gps-end-time ${GPS_END_TIME} \
```

(continues on next page)

(continued from previous page)

```
    --channel-name ${CHANNELS} \
    ${FRAMES} \
    --strain-high-pass ${F_HIGHPASS} \
    --pad-data ${PAD_DATA} \
    --psd-estimation ${PSD_ESTIMATION} \
    --psd-start-time ${PSD_START_TIME} \
    --psd-end-time ${PSD_END_TIME} \
    --psd-segment-length ${PSD_SEG_LEN} \
    --psd-segment-stride ${PSD_STRIDE} \
    --psd-inverse-length ${PSD_INVLEN} \
    --sample-rate ${SAMPLE_RATE} \
    --low-frequency-cutoff ${F_MIN} \
    --config-file ${CONFIG_PATH} \
    --output-file ${OUTPUT_PATH} \
    --processing-scheme ${PROCESSING_SCHEME} \
    --sampler kombine \
    --burn-in-function max_posterior \
    --update-interval ${N_UPDATE} \
    --nwalkers ${N_WALKERS} \
    --n-independent-samples ${N_SAMPLES} \
    --checkpoint-interval ${N_CHECKPOINT} \
    --nprocesses ${NPROCS} \
    --save-strain \
    --save-psd \
    --save-stilde \
    --force
```

As discussed in the injection example above, the code will write results to `gwin.hdf.checkpoint` at every checkpoint interval, and will continue to run until it has obtained at least as many independent samples as specified by `n-independent-samples`. When this happens, `gwin.hdf.checkpoint` will be moved to `gwin.hdf` and the code will exit.

## 3.2 Plotting the posteriors (`gwin_plot_posterior`)

### 3.2.1 Overview

There is an executable that can plot the posteriors called `gwin_plot_posterior`. You can use `--plot-scatter` to plot a each sample as a point or `--plot-density` to plot a density map.

By default the plotting executables will plot all the parameters in the input file. In order to specify a different set of variables to plot, use the `--parameters` option. Examples for how to use this option are shown below.

By default the plotting executables will plot samples beginning at the end of the burn in. If the burn-in was skipped, then it starts from the first sample. It will then use a sample every autocorrelation length along the chain. Examples on how to plot a specific iteration or change how the thinning is performed are shown in the examples below.

You may plot a z-axis on the 2-D histograms using the `--z-arg` option. For a list of options use `gwin_plot_posterior --help`.

### 3.2.2 Plotting a specific iteration

An example of plotting the posteriors at a specific iteration (the last one):

```
ITER=-1
INPUT_FILE=gwin.hdf
OUTPUT_FILE=scatter.png
gwin_plot_posterior \
    --iteration ${ITER} \
    --input-file ${INPUT_FILE} \
    --output-file ${OUTPUT_FILE} \
    --plot-scatter \
    --plot-marginal \
    --z-arg 'snr_from_loglr(loglr):$\rho$' \
    --parameters "ra*12/pi:$\alpha$ (h)" \
                 "dec*180/pi:$\delta$ (deg)" \
                 "polarization*180/pi:$\psi$ (deg)" \
                 mass1 mass2 spin1_a spin1_azimuthal spin1_polar \
                 spin2_a spin2_azimuthal spin2_polar \
                 "inclination*180/pi:$\iota$ (deg)" distance \
                 "coa_phase*180/pi:$\phi_0$ (deg)" tc
```

### 3.2.3 Plotting a thinned chain of samples

There are also options for thinning the chains of samples from the command line, an example starting at the 6000-th iteration and taking every 2000-th iteration until the 12000-th iteration:

```
THIN_START=5999
THIN_INTERVAL=2000
THIN_END=11999
INPUT_FILE=gwin.hdf
OUTPUT_FILE=scatter.png
gwin_plot_posterior \
    --input-file ${INPUT_FILE} \
    --output-file ${OUTPUT_FILE} \
    --plot-scatter \
    --thin-start ${THIN_START} \
    --thin-interval ${THIN_INTERVAL} \
    --thin-end ${THIN_END} \
    --plot-marginal \
    --z-arg 'snr_from_loglr(loglr):$\rho$' \
    --parameters "ra*12/pi:$\alpha$ (h)" \
                 "dec*180/pi:$\delta$ (deg)" \
                 "polarization*180/pi:$\psi$ (deg)" \
                 mass1 mass2 spin1_a spin1_azimuthal spin1_polar \
                 spin2_a spin2_azimuthal spin2_polar \
                 "inclination*180/pi:$\iota$ (deg)" distance \
                 "coa_phase*180/pi:$\phi_0$ (deg)" tc
```

## 3.3 Making a movie (`gwin_plot_movie`)

`gwin_plot_movie` is an executable similar to `_plot_posterior` that allows you to combine plots in to a small movie. Most options for `_plot_movie` are the same as `_plot_posterior` with a few differences. Again, the plotting executables will plot all the parameters in the input file unless the option `--parameters` is used to specify a set of parameters that you want to see. An example plotting every 20-th iteration into a directory called "movies":

```
INPUT_FILE=gwin.hdf
START_SAMPLE=1
END_SAMPLE=12000
FRAME_STEP=20
OUTPUT_PREFIX=frame
NPROCESSES=10
MOVIE_FILE=~/src/gwin/movies/movie.mp4
DPI=100

gwin_plot_movie \
    --input-file ${INPUT_FILE} \
    --start-sample ${START_SAMPLE} \
    --end-sample ${END_SAMPLE} \
    --frame-step ${FRAME_STEP} \
    --output-prefix ${OUTPUT_PREFIX} \
    --nprocesses ${NPROCESSES} \
    --movie-file ${MOVIE_FILE} \
    --cleanup \
    --plot-scatter \
    --plot-marginal \
    --z-arg 'snr_from_loglr(loglr):$\rho$' \
    --dpi ${DPI} \
    --parameters mass1 mass2 spin1_a spin1_azimuthal spin1_polar \
                 spin2_a spin2_azimuthal spin2_polar
```

This will create a 24-second movie for a selection of parameters. The option `--cleanup` deletes the individual frame files prefixed as specified by the variable `OUTPUT_PREFIX`. This is optional. For a list of options use `gwin_plot_movie --help`.

## 3.4 `gwin_make_workflow`: A parameter estimation workflow generator

### 3.4.1 Introduction

The executable `gwin_make_workflow` is a workflow generator to setup a parameter estimation analysis.

### 3.4.2 Workflow configuration file

A simple workflow configuration file

```
[workflow]
; basic information used by the workflow generator
file-retention-level = all_triggers
h1-channel-name = H1:DCS-CALIB_STRAIN_C02
l1-channel-name = L1:DCS-CALIB_STRAIN_C02

[workflow-ifos]
; the IFOs to analyze
h1 =
l1 =

[workflow-datafind]
```

(continues on next page)

(continued from previous page)

```
; how the workflow generator should get frame data
datafind-h1-frame-type = H1_HOFT_C02
datafind-l1-frame-type = L1_HOFT_C02
datafind-method = AT_RUNTIME_SINGLE_FRAMES
datafind-check-segment-gaps = raise_error
datafind-check-frames-exist = raise_error
datafind-check-segment-summary = no_test

[workflow-inference]
; how the workflow generator should setup inference nodes
num-events = 1
plot-1d-mass = mass1 mass2 mchirp q
plot-1d-orientation = ra dec tc polarization inclination coa_phase
plot-1d-distance = distance redshift

[executables]
; paths to executables to use in workflow
inference = ${which:run_gwin}
inference_posterior = ${which:gwin_plot_posterior}
inference_prior = ${which:gwin_plot_prior}
inference_rate = ${which:gwin_plot_acceptance_rate}
inference_samples = ${which:gwin_plot_samples}
inference_table = ${which:gwin_table_summary}
plot_spectrum = ${which:pycbc_plot_psd_file}
results_page = ${which:pycbc_make_html_page}

[datafind]
; datafind options
urltype = file

[inference]
; command line options use --help for more information
sample-rate = 2048
low-frequency-cutoff = 20
strain-high-pass = 15
pad-data = 8
psd-estimation = median
psd-segment-length = 16
psd-segment-stride = 8
psd-inverse-length = 4
processing-scheme = cpu
checkpoint-interval = 1000
sampler = kombine
nwalkers = 5000
update-interval = 500
n-independent-samples = 5000
burn-in-function = max_posterior
save-psd =
save-strain =
save-stilde =
nprocesses = 12
resume-from-checkpoint =

[pegasus_profile-inference]
; pegasus profile for inference nodes
condor|request_memory = 20G
condor|request_cpus = 12
```

(continues on next page)

(continued from previous page)

```
[inference_posterior]
; command line options use --help for more information
plot-density =
plot-contours =
plot-marginal =

[inference_prior]
; command line options use --help for more information

[inference_rate]
; command line options use --help for more information

[inference_samples]
; command line options use --help for more information

[inference_table]
; command line options use --help for more information

[plot_spectrum]
; command line options use --help for more information

[results_page]
; command line options use --help for more information
analysis-title = "GWin Test"
```

### 3.4.3 Inference configuration file

You will also need a configuration file with sections that tells `gwin` how to construct the priors. A simple inference configuration file is

```
[model]
name = gaussian_noise

[variable_params]
; parameters to vary in inference sampler
tc =
mass1 =
mass2 =
distance =
coa_phase =
inclination =
ra =
dec =
polarization =

[static_params]
; parameters that do not vary in inference sampler
approximant = SEOBNRv2_ROM_DoubleSpin
f_lower = 28.0

[prior-tc]
; how to construct prior distribution
name = uniform
min-tc = 1126259462.2
```

(continues on next page)

(continued from previous page)

```
max-tc = 1126259462.6

[prior-mass1]
; how to construct prior distribution
name = uniform
min-mass1 = 10.
max-mass1 = 80.

[prior-mass2]
; how to construct prior distribution
name = uniform
min-mass2 = 10.
max-mass2 = 80.

[prior-distance]
; how to construct prior distribution
name = uniform
min-distance = 10
max-distance = 500

[prior-coa_phase]
; how to construct prior distribution
name = uniform_angle
; uniform_angle defaults to [0,2pi), so we
; don't need to specify anything here

[prior-inclination]
; how to construct prior distribution
name = sin_angle

[prior-ra+dec]
; how to construct prior distribution
name = uniform_sky

[prior-polarization]
; how to construct prior distribution
name = uniform_angle
```

A simple configuration file for parameter estimation on the ringdown is:

```
[model]
name = gaussian_noise

[variable_params]
; parameters to vary in inference sampler
tc =
f_0 =
tau =
amp =
phi =

[static_params]
; parameters that do not vary in inference sampler
approximant = FdQNM
ra = 2.21535724066
dec = -1.23649695537
polarization = 0.
```

(continues on next page)

(continued from previous page)

```
f_lower = 28.0
f_final = 512

[prior-tc]
; how to construct prior distribution
name = uniform
min-tc = 1126259462.4
max-tc = 1126259462.5

[prior-f_0]
; how to construct prior distribution
name = uniform
min-f_0 = 200.
max-f_0 = 300.

[prior-tau]
; how to construct prior distribution
name = uniform
min-tau = 0.0008
max-tau = 0.020

[prior-amp]
; how to construct prior distribution
name = uniform
min-amp = 0
max-amp = 1e-20

[prior-phi]
; how to construct prior distribution
name = uniform_angle
```

If you want to use another variable parameter in the inference sampler then add its name to `[variable_params]` and add a prior section like shown above.

The number of likelihood calculations grows exponentially with the number of variable parameters, and so in order to keep computational costs to a minimum, we can choose to marginalize over time and/or distance and/or phase. This removes the need to sample over those dimensions without affecting the other posterior PDFs.

We can turn on likehood marginalization by specifying the `marginalized_gaussian_noise` class in the `[model]` section. For example, we can modify the simple configuration file above to include distance marginalization,

```
[model]
name = marginalized_gaussian_noise
distance_marginalization =

[marginalized_prior-distance]
; how to construct prior distribution for distance marginalization
name = uniform
min-distance = 50.
max-distance = 5000.

[variable_params]
; parameters to vary in inference sampler
tc =
mass1 =
mass2 =
```

(continues on next page)

(continued from previous page)

```
distance =
coa_phase =
inclination =
ra =
dec =
polarization =

[static_params]
; parameters that do not vary in inference sampler
approximant = SEOBNRv2_ROM_DoubleSpin
f_lower = 28.0

[prior-tc]
; how to construct prior distribution
name = uniform
min-tc = 1126259462.2
max-tc = 1126259462.6

[prior-mass1]
; how to construct prior distribution
name = uniform
min-mass1 = 10.
max-mass1 = 80.

[prior-mass2]
; how to construct prior distribution
name = uniform
min-mass2 = 10.
max-mass2 = 80.

[prior-distance]
; how to construct prior distribution
name = uniform
min-distance = 10
max-distance = 500

[prior-coa_phase]
; how to construct prior distribution
name = uniform_angle
; uniform_angle defaults to [0,2pi), so we
; don't need to specify anything here

[prior-inclination]
; how to construct prior distribution
name = sin_angle

[prior-ra+dec]
; how to construct prior distribution
name = uniform_sky

[prior-polarization]
; how to construct prior distribution
name = uniform_angle
```

If you want to also marginalize over time and/or phase, then add its name to the `[model]` section and add a corresponding prior under the section `[marginalized_prior-${VARIABLE}]` like shown above. If this was the case, then for this trivial example, you would no longer need to sample over the phase and coalescence time.

When working on real data, it is necessary to marginalise over calibration uncertainty. The model and parameters describing the calibration uncertainty can be passed in another ini file, e.g.:

```
# Details of set up as given in the O1 Binary Black Hole Paper https://arxiv.org/abs/
↪1606.04856
[calibration]
h1_model = cubic_spline
h1_minimum_frequency = 10
h1_maximum_frequency = 1024
h1_n_points = 5
l1_model = cubic_spline
l1_minimum_frequency = 10
l1_maximum_frequency = 1024
l1_n_points = 5

[variable_params]
recalib_amplitude_h1_0 =
recalib_amplitude_h1_1 =
recalib_amplitude_h1_2 =
recalib_amplitude_h1_3 =
recalib_amplitude_h1_4 =
recalib_phase_h1_0 =
recalib_phase_h1_1 =
recalib_phase_h1_2 =
recalib_phase_h1_3 =
recalib_phase_h1_4 =
recalib_amplitude_l1_0 =
recalib_amplitude_l1_1 =
recalib_amplitude_l1_2 =
recalib_amplitude_l1_3 =
recalib_amplitude_l1_4 =
recalib_phase_l1_0 =
recalib_phase_l1_1 =
recalib_phase_l1_2 =
recalib_phase_l1_3 =
recalib_phase_l1_4 =

[prior-recalib_amplitude_h1_0]
name = gaussian
recalib_amplitude_h1_0_mean = 0
recalib_amplitude_h1_0_var = 0.0023

[prior-recalib_amplitude_h1_1]
name = gaussian
recalib_amplitude_h1_1_mean = 0
recalib_amplitude_h1_1_var = 0.0023

[prior-recalib_amplitude_h1_2]
name = gaussian
recalib_amplitude_h1_2_mean = 0
recalib_amplitude_h1_2_var = 0.0023

[prior-recalib_amplitude_h1_3]
name = gaussian
recalib_amplitude_h1_3_mean = 0
recalib_amplitude_h1_3_var = 0.0023

[prior-recalib_amplitude_h1_4]
```

(continues on next page)

(continued from previous page)

```
name = gaussian
recalib_amplitude_h1_4_mean = 0
recalib_amplitude_h1_4_var = 0.0023

[prior-recalib_amplitude_l1_0]
name = gaussian
recalib_amplitude_l1_0_mean = 0
recalib_amplitude_l1_0_var = 0.0068

[prior-recalib_amplitude_l1_1]
name = gaussian
recalib_amplitude_l1_1_mean = 0
recalib_amplitude_l1_1_var = 0.0068

[prior-recalib_amplitude_l1_2]
name = gaussian
recalib_amplitude_l1_2_mean = 0
recalib_amplitude_l1_2_var = 0.0068

[prior-recalib_amplitude_l1_3]
name = gaussian
recalib_amplitude_l1_3_mean = 0
recalib_amplitude_l1_3_var = 0.0068

[prior-recalib_amplitude_l1_4]
name = gaussian
recalib_amplitude_l1_4_mean = 0
recalib_amplitude_l1_4_var = 0.0068

[prior-recalib_phase_h1_0]
name = gaussian
recalib_phase_h1_0_mean = 0
recalib_phase_h1_0_var = 0.0030

[prior-recalib_phase_h1_1]
name = gaussian
recalib_phase_h1_1_mean = 0
recalib_phase_h1_1_var = 0.0030

[prior-recalib_phase_h1_2]
name = gaussian
recalib_phase_h1_2_mean = 0
recalib_phase_h1_2_var = 0.0030

[prior-recalib_phase_h1_3]
name = gaussian
recalib_phase_h1_3_mean = 0
recalib_phase_h1_3_var = 0.0030

[prior-recalib_phase_h1_4]
name = gaussian
recalib_phase_h1_4_mean = 0
recalib_phase_h1_4_var = 0.0030

[prior-recalib_phase_l1_0]
name = gaussian
recalib_phase_l1_0_mean = 0
```

(continues on next page)

(continued from previous page)

```
recalib_phase_l1_0_var = 0.0054

[prior-recalib_phase_l1_1]
name = gaussian
recalib_phase_l1_1_mean = 0
recalib_phase_l1_1_var = 0.0054

[prior-recalib_phase_l1_2]
name = gaussian
recalib_phase_l1_2_mean = 0
recalib_phase_l1_2_var = 0.0054

[prior-recalib_phase_l1_3]
name = gaussian
recalib_phase_l1_3_mean = 0
recalib_phase_l1_3_var = 0.0054

[prior-recalib_phase_l1_4]
name = gaussian
recalib_phase_l1_4_mean = 0
recalib_phase_l1_4_var = 0.0054
```

### 3.4.4 Generate the workflow

To generate a workflow you will need your configuration files. The workflow creates a single config file `inference.ini` from all the files specified in `INFERENCE_CONFIG_PATH`. We set the following enviroment variables for this example:

```
# name of the workflow
WORKFLOW_NAME="r1"

# path to output dir
OUTPUT_DIR=output

# input configuration files
CONFIG_PATH=workflow.ini
INFERENCE_CONFIG_PATH="gwin.ini calibration.ini"
```

Specify a directory to save the HTML pages:

```
# directory that will be populated with HTML pages
HTML_DIR=${HOME}/public_html/inference_test
```

If you want to run on the loudest triggers from a PyCBC coincident search workflow then run:

```
# run workflow generator on triggers from workflow
gwin_make_workflow --workflow-name ${WORKFLOW_NAME} \
    --config-files ${CONFIG_PATH} \
    --inference-config-file ${INFERENCE_CONFIG_PATH} \
    --output-dir ${OUTPUT_DIR} \
    --output-file ${WORKFLOW_NAME}.dax \
    --output-map ${OUTPUT_MAP_PATH} \
    --bank-file ${BANK_PATH} \
    --statmap-file ${STATMAP_PATH} \
    --single-detector-triggers ${SNGL_H1_PATHS} ${SNGL_L1_PATHS}
```

(continues on next page)

(continued from previous page)

```
    --config-overrides workflow:start-time:${WORKFLOW_START_TIME} \
                       workflow:end-time:${WORKFLOW_END_TIME} \
                       workflow-inference:data-seconds-before-trigger:8 \
                       workflow-inference:data-seconds-after-trigger:8 \
                       results_page:output-path:${HTML_DIR} \
                       results_page:analysis-subtitle:${WORKFLOW_NAME}
```

Where `${BANK_FILE}` is the path to the template bank HDF file, `${STATMAP_FILE}` is the path to the combined statmap HDF file, `${SNGL_H1_PATHS}` and `${SNGL_L1_PATHS}` are the paths to the merged single-detector HDF files, and `${WORKFLOW_START_TIME}` and `${WORKFLOW_END_TIME}` are the start and end time of the coincidence workflow.

Else you can run from a specific GPS end time with the `--gps-end-time` option like:

```
# run workflow generator on specific GPS end time
gwin_make_workflow --workflow-name ${WORKFLOW_NAME} \
    --config-files ${CONFIG_PATH} \
    --inference-config-file ${INFERENCE_CONFIG_PATH} \
    --output-dir ${OUTPUT_DIR} \
    --output-file ${WORKFLOW_NAME}.dax \
    --output-map ${OUTPUT_MAP_PATH} \
    --gps-end-time ${GPS_END_TIME} \
    --config-overrides workflow:start-time:$((${GPS_END_TIME}-16)) \
                       workflow:end-time:$((${GPS_END_TIME}+16)) \
                       workflow-inference:data-seconds-before-trigger:2 \
                       workflow-inference:data-seconds-after-trigger:2 \
                       inference:psd-start-time:$((${GPS_END_TIME}-300)) \
                       inference:psd-end-time:$((${GPS_END_TIME}+748)) \
                       results_page:output-path:${HTML_DIR} \
                       results_page:analysis-subtitle:${WORKFLOW_NAME}
```

Where `${GPS_END_TIME}` is the GPS end time of the trigger.

For the CBC example above define the environment variables `GPS_END_TIME=1126259462` and `OUTPUT_MAP_PATH=output.map`.

### 3.4.5 Plan and execute the workflow

Finally plan and submit the workflow with:

```
# submit workflow
pycbc_submit_dax --dax ${WORKFLOW_NAME}.dax \
    --accounting-group ligo.dev.o3.cbc.explore.test \
    --enable-shared-filesystem
```

## 3.5 `gwin_make_inj_workflow`: A parameter estimation workflow generator

### 3.5.1 Introduction

The executable `gwin_make_inj_workflow` is a workflow generator to setup a parameter estimation analysis.

### 3.5.2 Workflow configuration file

A simple workflow configuration file:

```
[workflow]
; basic information used by the workflow generator
file-retention-level = all_triggers
h1-channel-name = H1:DCS-CALIB_STRAIN_C02
l1-channel-name = L1:DCS-CALIB_STRAIN_C02

[workflow-ifos]
; the IFOs to analyze
h1 =
l1 =

[workflow-inference]
; how the workflow generator should setup inference nodes
num-injections = 3
plot-group-mass = mass1 mass2 mchirp q
plot-group-orientation =  inclination polarization ra dec
plot-group-distance = distance redshift
plot-group-time = tc coa_phase

[executables]
; paths to executables to use in workflow
create_injections = ${which:pycbc_create_injections}
inference = ${which:run_gwin}
inference_intervals = ${which:gwin_plot_inj_intervals}
inference_posterior = ${which:gwin_plot_posterior}
inference_rate = ${which:gwin_plot_acceptance_rate}
inference_recovery = ${which:gwin_plot_inj_recovery}
inference_samples = ${which:gwin_plot_samples}
inference_table = ${which:gwin_table_summary}
results_page = ${which:pycbc_make_html_page}

[create_injections]
; command line options use --help for more information
ninjections = 1
dist-section = prior
variable-args-section = variable_params
static-args-section = static_params

[inference]
; command line options use --help for more information
processing-scheme = cpu
sampler = kombine
nwalkers = 5000
checkpoint-interval = 1000
n-independent-samples = 5000
update-interval = 500
burn-in-function = max_posterior
nprocesses = 24
fake-strain = aLIGOZeroDetHighPower
psd-model = aLIGOZeroDetHighPower
pad-data = 8
strain-high-pass = 15
sample-rate = 2048
low-frequency-cutoff = 20
```

(continues on next page)

(continued from previous page)

```
config-overrides = static_params:approximant:TaylorF2
save-psd =
save-strain =
save-stilde =
resmume-from-checkpoint =

[pegasus_profile-inference]
; pegasus profile for inference nodes
condor|request_memory = 20G
condor|request_cpus = 24

[inference_intervals]
; command line options use --help for more information

[inference_posterior]
; command line options use --help for more information
plot-scatter =
plot-contours =
plot-marginal =
z-arg = logposterior

[inference_rate]
; command line options use --help for more information

[inference_recovery]
; command line options use --help for more information

[inference_samples]
; command line options use --help for more information

[inference_table]
; command line options use --help for more information

[results_page]
; command line options use --help for more information
analysis-title = "PyCBC Inference Test"
```

Use the `ninjections` option in the `[workflow-inference]` section to set the number of injections in the analysis.

### 3.5.3 Generate the workflow

To generate a workflow you will need your configuration files. We set the following enviroment variables for this example:

```
# name of the workflow
WORKFLOW_NAME="r1"

# path to output dir
OUTPUT_DIR=output

# input configuration files
CONFIG_PATH=workflow.ini
INFERENCE_CONFIG_PATH=gwin.ini
```

Specify a directory to save the HTML pages:

```
# directory that will be populated with HTML pages
HTML_DIR=${HOME}/public_html/inference_test
```

If you want to run with a test likelihood function use:

```
# option for using test likelihood functions
DATA_TYPE=analytical
```

Otherwise if you want to run with simulated data use:

```
# option for using simulated data
DATA_TYPE=simulated_data
```

### 3.5.4 Plan and execute the workflow

Finally plan and submit the workflow with:

```
# submit workflow
pycbc_submit_dax --dax ${WORKFLOW_NAME}.dax \
    --accounting-group ligo.dev.o3.cbc.explore.test \
    --enable-shared-filesystem
```

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## g

# Index

## D



## E



## F



## G



## H



## I



## K



## L

## M



## N



## P

## R



## S



## T



## U



## V



## W