Flarestack Documentation

Release 2.0-beta.1

Robert Stein

May 28, 2019

CONTENTS:

CHAPTER

ONE

BASE PDFS

1.1 Time PDFs

class flarestack.core.time_pdf.**Box**(*t_pdf_dict*, *season*)

The simplest time-dependent case for a Time PDF. Used for a source that is uniformly emitting for a fixed period of time. Requires arguments of Pre-Window and Post_window, and gives a box from Pre-Window days before the reference time to Post-Window days after the reference time.

__init__(*t_pdf_dict*, *season*)

Initialize self. See help(type(self)) for accurate signature.

effective_injection_time(*source*)

Calculates the effective injection time for the given PDF. The livetime is measured in days, but here is converted to seconds.

Parameters **source** – Source to be considered

Returns Effective Livetime in seconds

flare_time_mask(*source*)

In this case, the interesting period for Flare Searches is the period of overlap of the flare and the box. Thus, for a given season, return the source and data

Returns Start time (MJD) and End Time (MJD) for flare search period

raw_injection_time(*source*)

Calculates the 'raw injection time' which is the injection time assuming a detector with 100% uptime. Useful for calculating source emission times for source-frame energy estimation.

Parameters **source** – Source to be considered

Returns Time in seconds for 100% uptime

sig_t0(*source*)

Calculates the starting time for the window, equal to the source reference time in MJD minus the length of the pre-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window Start

sig_t1(*source*)

Calculates the starting time for the window, equal to the source reference time in MJD plus the length of the post-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window End

signal_f (*t*, *source*)

In this case, the signal PDF is a uniform PDF for a fixed duration of time. It is normalised with the length of the box in LIVETIME rather than days, to give an integral of 1.

Parameters

- **t** – Time
- **source** – Source to be considered

Returns Value of normalised box function at t

signal_integral (*t*, *source*)

In this case, the signal PDF is a uniform PDF for a fixed duration of time. Thus, the integral is simply a linear function increasing between t0 (box start) and t1 (box end). After t1, the integral is equal to 1, while it is equal to 0 for t < t0.

Parameters

- **t** – Time
- **source** – Source to be considered

Returns Value of normalised box function at t

class flarestack.core.time_pdf.**FixedEndBox** (*t_pdf_dict*, *season*)

The simplest time-dependent case for a Time PDF. Used for a source that is uniformly emitting for a fixed period of time. In this case, the start and end time for the box is unique for each source. The sources must have a field "Start Time (MJD)" and another "End Time (MJD)", specifying the period of the Time PDF.

__init__ (*t_pdf_dict*, *season*)

Initialize self. See help(type(self)) for accurate signature.

sig_t0 (*source*)

Calculates the starting time for the window, equal to the source reference time in MJD minus the length of the pre-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window Start

sig_t1 (*source*)

Calculates the starting time for the window, equal to the source reference time in MJD plus the length of the post-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window End

class flarestack.core.time_pdf.**FixedRefBox** (*t_pdf_dict*, *season*)

The simplest time-dependent case for a Time PDF. Used for a source that is uniformly emitting for a fixed period of time. In this case, the start and end time for the box is unique for each source. The sources must have a field "Start Time (MJD)" and another "End Time (MJD)", specifying the period of the Time PDF.

__init__ (*t_pdf_dict*, *season*)

Initialize self. See help(type(self)) for accurate signature.

sig_t0 (*source*)

Calculates the starting time for the window, equal to the source reference time in MJD minus the length of the pre-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window Start

sig_t1(*source*)
Calculates the starting time for the window, equal to the source reference time in MJD plus the length of the post-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window End

class flarestack.core.time_pdf.**Steady**(*t_pdf_dict*, *season*)
The time-independent case for a Time PDF. Requires no additional arguments in the dictionary for __init__. Used for a steady source that is continuously emitting.

effective_injection_time(*source*)
Calculates the effective injection time for the given PDF. The livetime is measured in days, but here is converted to seconds.

Parameters **source** – Source to be considered

Returns Effective Livetime in seconds

flare_time_mask(*source*)
In this case, the interesting period for Flare Searches is the entire season. Thus returns the start and end times for the season.

Returns Start time (MJD) and End Time (MJD) for flare search period

raw_injection_time(*source*)
Calculates the 'raw injection time' which is the injection time assuming a detector with 100% uptime. Useful for calculating source emission times for source-frame energy estimation.

Parameters **source** – Source to be considered

Returns Time in seconds for 100% uptime

sig_t0(*source*)
Calculates the starting time for the window, equal to the source reference time in MJD minus the length of the pre-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window Start

sig_t1(*source*)
Calculates the starting time for the window, equal to the source reference time in MJD plus the length of the post-reference-time window (in days).

Parameters **source** – Source to be considered

Returns Time of Window End

signal_f(*t*, *source*)
In the case of a steady source, the signal PDF is a uniform PDF in time. It is thus simply equal to the season_f, normalised with the length of the season to give an integral of 1. It is thus equal to the background PDF.

Parameters

- **t** – Time
- **source** – Source to be considered

Returns Value of normalised box function at t

signal_integral(*t*, *source*)
In the case of a steady source, the signal PDF is a uniform PDF in time. Thus, the integral is simply a

linear function increasing between t0 (box start) and t1 (box end). After t1, the integral is equal to 1, while it is equal to 0 for t < t0.

Parameters

- **t** – Time
- **source** – Source to be considered

Returns Value of normalised box function at t

class flarestack.core.time_pdf.**TimePDF**(*t_pdf_dict*, *season*)

__init__(*t_pdf_dict*, *season*)
Initialize self. See help(type(self)) for accurate signature.

background_f(*t*, *source*)
In all cases, we assume that the background is uniform in time. Thus, the background PDF is just a normalised version of the season_f box function.

Parameters

- **t** – Time
- **source** – Source to be considered

Returns Value of normalised box function at t

classmethod create(*t_pdf_dict*, *season*)

inverse_interpolate(*source*)
Calculates the values for the integral of the signal PDF within the season. Then rescales these values, such that the start of the season yields 0, and then end of the season yields 1. Creates a function to interpolate between these values. Then, for a number between 0 and 1, the interpolated function will return the MJD time at which that fraction of the cumulative distribution was reached.

Parameters **source** – Source to be considered

Returns Interpolated function

product_integral(*t*, *source*)
Calculates the product of the given signal PDF with the season box function. Thus gives 0 everywhere outside the season, and otherwise the value of the normalised integral. The season function is offset by 1e-9, to ensure that f(t1) is equal to 1. (i.e the function is equal to 1 at the end of the box).

Parameters

- **t** – Time
- **source** – Source to be considered

Returns Product of signal integral and season

classmethod register_subclass(*time_pdf_name*)

simulate_times(*source*, *n_s*)
Randomly draws times for n_s events for a given source, all lying within the current season. The values are based on an interpolation of the integrated time PDF.

Parameters

- **source** – Source being considered
- **n_s** – Number of event times to be simulated

Returns Array of times in MJD for a given source

subclasses = {'Box': <class 'flarestack.core.time_pdf.Box'>, 'FixedEndBox': <class '

flarestack.core.time_pdf.**box_func**(*t*, *t0*, *t1*)
Box function that is equal to 1 between t0 and t1, and 0 otherwise. Equal to 0.5 at to and t1.

Parameters

- **t** – Time to be evaluated
- **t0** – Start time of box
- **t1** – End time of box

Returns Value of Box function at t

flarestack.core.time_pdf.**read_t_pdf_dict**(*t_pdf_dict*)
Ensures backwards compatibility for t_pdf_dict objects

1.2 Energy PDFs

This script contains the EnergyPDF classes, that are used for weighting events based on a given energy PDF.

class flarestack.core.energy_pdf.**EnergyPDF**(*e_pdf_dict*)

__init__(*e_pdf_dict*)
Initialize self. See help(type(self)) for accurate signature.

classmethod create(*e_pdf_dict*)

static f(*energy*)

fluence_integral()
Performs an integral for fluence over a given energy range. This is gives the total energy per unit area per second that is radiated.

flux_integral()
Integrates over energy PDF to give integrated flux (dN/dT)

integrate_over_E(*f*, *lower=None*, *upper=None*)
Uses Newton's method to integrate function f over the energy range. By default, uses 100GeV to 10PeV, unless otherwise specified. Uses 1000 logarithmically-spaced bins to calculate integral.

Parameters **f** – Function to be integrated

Returns Integral of function

classmethod register_subclass(*energy_pdf_name*)
Adds a new subclass of EnergyPDF, with class name equal to "energy_pdf_name".

return_energy_parameters()

subclasses = {'PowerLaw': <class 'flarestack.core.energy_pdf.PowerLaw'>, 'Spline': <

class flarestack.core.energy_pdf.**PowerLaw**(*e_pdf_dict=None*)
A Power Law energy PDF. Takes an argument of gamma in the dictionary for the init function, where gamma is the spectral index of the Power Law.

__init__(*e_pdf_dict=None*)
Creates a PowerLaw object, which is an energy PDF based on a power law. The power law is generated from e_pdf_dict, which can specify a spectral index (Gamma), as well as an optional minimum energy (E Min) and a maximum energy (E Max)

Parameters **e_pdf_dict** – Dictionary containing parameters

f (*energy*)

fluence_integral ()
Performs an integral for fluence over a given energy range. This is gives the total energy per unit area per second that is radiated.

flux_integral ()
Integrates over energy PDF to give integrated flux (dN/dT)

return_energy_parameters ()

return_injected_parameters ()

weight_mc (*mc*, *gamma=None*)
Returns an array containing the weights for each MC event, given that the spectral index gamma has been chosen. Weights each event as (E/GeV)^-gamma, and multiplies this by the pre-existing MC oneweight value, to give the overall oneweight.

Parameters

- **mc** – Monte Carlo
- **gamma** – Spectral Index (default is value in e_pdf_dict)

Returns Weights Array

class flarestack.core.energy_pdf.**Spline** (*e_pdf_dict={}*)
A Power Law energy PDF. Takes an argument of gamma in the dictionary for the init function, where gamma is the spectral index of the Power Law.

__init__ (*e_pdf_dict={}*)
Creates a PowerLaw object, which is an energy PDF based on a power law. The power law is generated from e_pdf_dict, which can specify a spectral index (Gamma), as well as an optional minimum energy (E Min) and a maximum energy (E Max)

Parameters **e_pdf_dict** – Dictionary containing parameters

weight_mc (*mc*)
Returns an array containing the weights for each MC event, given that the spectral index gamma has been chosen. Weights each event using the energy spline, and multiplies this by the pre-existing MC oneweight value, to give the overall oneweight.

Parameters **mc** – Monte Carlo

Returns Weights Array

flarestack.core.energy_pdf.**read_e_pdf_dict** (*e_pdf_dict*)
Ensures backwards compatibility of e_pdf_dict objects.

Parameters **e_pdf_dict** – Energy PDF dictionary

Returns Updated Energy PDF dictionary compatible with new format

1.3 Spatial PDFs

This script contains the EnergyPDF classes, that are used for weighting events based on a given energy PDF.

class flarestack.core.energy_pdf.**EnergyPDF** (*e_pdf_dict*)

__init__(*e_pdf_dict*)
Initialize self. See help(type(self)) for accurate signature.

classmethod create(*e_pdf_dict*)

static f(*energy*)

fluence_integral()
Performs an integral for fluence over a given energy range. This is gives the total energy per unit area per second that is radiated.

flux_integral()
Integrates over energy PDF to give integrated flux (dN/dT)

integrate_over_E(*f*, *lower=None*, *upper=None*)
Uses Newton's method to integrate function f over the energy range. By default, uses 100GeV to 10PeV, unless otherwise specified. Uses 1000 logarithmically-spaced bins to calculate integral.

Parameters **f** – Function to be integrated

Returns Integral of function

classmethod register_subclass(*energy_pdf_name*)
Adds a new subclass of EnergyPDF, with class name equal to "energy_pdf_name".

return_energy_parameters()

subclasses = {'PowerLaw': <class 'flarestack.core.energy_pdf.PowerLaw'>, 'Spline': <

class flarestack.core.energy_pdf.**PowerLaw**(*e_pdf_dict=None*)
A Power Law energy PDF. Takes an argument of gamma in the dictionary for the init function, where gamma is the spectral index of the Power Law.

__init__(*e_pdf_dict=None*)
Creates a PowerLaw object, which is an energy PDF based on a power law. The power law is generated from e_pdf_dict, which can specify a spectral index (Gamma), as well as an optional minimum energy (E Min) and a maximum energy (E Max)

Parameters **e_pdf_dict** – Dictionary containing parameters

f(*energy*)

fluence_integral()
Performs an integral for fluence over a given energy range. This is gives the total energy per unit area per second that is radiated.

flux_integral()
Integrates over energy PDF to give integrated flux (dN/dT)

return_energy_parameters()

return_injected_parameters()

weight_mc(*mc*, *gamma=None*)
Returns an array containing the weights for each MC event, given that the spectral index gamma has been chosen. Weights each event as (E/GeV)^-gamma, and multiplies this by the pre-existing MC oneweight value, to give the overall oneweight.

Parameters

- **mc** – Monte Carlo
- **gamma** – Spectral Index (default is value in e_pdf_dict)

Returns Weights Array

class flarestack.core.energy_pdf.**Spline**(*e_pdf_dict={}*)

A Power Law energy PDF. Takes an argument of gamma in the dictionary for the init function, where gamma is the spectral index of the Power Law.

__init__(*e_pdf_dict={}*)

Creates a PowerLaw object, which is an energy PDF based on a power law. The power law is generated from e_pdf_dict, which can specify a spectral index (Gamma), as well as an optional minimum energy (E Min) and a maximum energy (E Max)

Parameters **e_pdf_dict** – Dictionary containing parameters

weight_mc(*mc*)

Returns an array containing the weights for each MC event, given that the spectral index gamma has been chosen. Weights each event using the energy spline, and multiplies this by the pre-existing MC oneweight value, to give the overall oneweight.

Parameters **mc** – Monte Carlo

Returns Weights Array

flarestack.core.energy_pdf.**read_e_pdf_dict**(*e_pdf_dict*)

Ensures backwards compatibility of e_pdf_dict objects.

Parameters **e_pdf_dict** – Energy PDF dictionary

Returns Updated Energy PDF dictionary compatible with new format

CHAPTER

TWO

COMPOSITE PDF OBJECTS

2.1 Injector

class flarestack.core.injector.**BaseInjector**(*season*, *sources*, ***kwargs*)
Base Injector Class

__init__(*season*, *sources*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

calculate_n_exp()

calculate_n_exp_single(*source*)

classmethod create(*season*, *sources*, ***kwargs*)

create_dataset(*scale*, *pull_corrector*)
Create a dataset based on scrambled data for background, and Monte Carlo simulation for signal. Returns the composite dataset. The source flux can be scaled by the scale parameter.

Parameters **scale** – Ratio of Injected Flux to source flux

Returns Simulated dataset

static get_dec_and_omega(*source*)

get_expectation(*source*, *scale*)

get_n_exp_single(*source*)

inject_signal(*scale*)

classmethod register_subclass(*inj_name*)
Adds a new subclass of EnergyPDF, with class name equal to "energy_pdf_name".

subclasses = {}

update_sources(*sources*)
Reuses an injector with new sources

Parameters **sources** – Sources to be added

class flarestack.core.injector.**EffectiveAreaInjector**(*season*, *sources*, ***kwargs*)
Class for injecting signal events by relying on effective areas rather than pre-existing Monte Carlo simulation. This Injector should be used for analysing public data, as no MC is provided.

__init__(*season*, *sources*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

calculate_energy_proxy(*source*)

calculate_n_exp_single(*source*)

calculate_single_source(*source*, *scale*)

inject_signal(*scale*)

class flarestack.core.injector.**LowMemoryInjector**(*season*, *sources*, ***kwargs*)
For large numbers of sources O(~100), saving MC masks becomes increasingly burdensome. As a solution, the LowMemoryInjector should be used instead. It will be somewhat slower, but will have much more reasonable memory consumption.

__init__(*season*, *sources*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

calculate_n_exp()

get_band_mask(*source*, *min_dec*, *max_dec*)

load_band_mask(*index*)

make_injection_band_mask()

class flarestack.core.injector.**MCInjector**(*season*, *sources*, ***kwargs*)
Core Injector Class, returns a dataset on which calculations can be performed. This base class is tailored for injection of MC into mock background. This can be either MC background, or scrambled real data.

__init__(*season*, *sources*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

calculate_fluence(*source*, *scale*, *source_mc*, *band_mask*, *omega*)
Function to calculate the fluence for a given source, and multiply the oneweights by this. After this step, the oneweight sum is equal to the expected neutrino number.

Parameters

- **source** – Source to be calculated
- **scale** – Flux scale
- **source_mc** – MC that is close to source
- **band_mask** – Closeness mask for MC
- **omega** – Solid angle covered by MC mask

Returns Modified source MC

calculate_n_exp_single(*source*)

calculate_single_source(*source*, *scale*)
Calculate the weighted MC for a single source, given a flux scale and a distance scale.

Parameters

- **source** –
- **scale** –

Returns

get_band_mask(*source*, *min_dec*, *max_dec*)

inject_signal(*scale*)
Randomly select simulated events from the Monte Carlo dataset to simulate a signal for each source. The source flux can be scaled by the scale parameter.

Parameters **scale** – Ratio of Injected Flux to source flux.

Returns Set of signal events for the given IC Season.

select_mc_band(*source*)
For a given source, selects MC events within a declination band of width +/- 5 degrees that contains the source. Then returns the MC data subset containing only those MC events.

Parameters **source** – Source to be simulated

Returns mc (cut): Simulated events which lie within the band

Returns omega: Solid Angle of the chosen band

Returns band_mask: The mask which removes events outside band

subclasses = {'low_memory_injector': <class 'flarestack.core.injector.LowMemoryInjecto

class flarestack.core.injector.**MockUnblindedInjector**(*season*, *sources=nan*, ***kwargs*)
If the data is not really to be unblinded, then MockUnblindedInjector should be called. In this case, the create_dataset function simply returns one background scramble.

__init__(*season*, *sources=nan*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

create_dataset(*scale*, *pull_corrector*)
Returns a background scramble

Returns Scrambled data

class flarestack.core.injector.**TrueUnblindedInjector**(*season*, *sources*, ***kwargs*)
If the data is unblinded, then UnblindedInjector should be called. In this case, the create_dataset function simply returns the unblinded dataset.

__init__(*season*, *sources*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

create_dataset(*scale*, *pull_corrector*)

flarestack.core.injector.**read_injector_dict**(*inj_dict*)
Ensures that injection dictionaries remain backwards-compatible

Parameters **inj_dict** – Injection Dictionary

Returns Injection Dictionary compatible with new format

2.2 Log Likelihood

class flarestack.core.llh.**FixedEnergyLLH**(*season*, *sources*, *llh_dict*)

__init__(*season*, *sources*, *llh_dict*)
Initialize self. See help(type(self)) for accurate signature.

calculate_test_statistic(*params*, *weights*, ***kwargs*)
Calculates the test statistic, given the parameters. Uses numexpr for faster calculations.

Parameters

- **params** – Parameters from Minimisation
- **weights** – Normalised fraction of n_s allocated to each source

Returns 2 * llh value (Equal to Test Statistic)

create_acceptance_function(*acc_path*)

create_energy_functions()
Creates the acceptance function, which parameterises signal acceptance as a function of declination, and the energy weighting function, which gives the energy signal-over-background ratio

Returns Acceptance function, energy_weighting_function

create_energy_weighting_function(*SoB_path*)

create_kwargs(*data*, *pull_corrector*, *weight_f=None*)
Creates a likelihood function to minimise, based on the dataset.

Parameters **data** – Dataset

Returns LLH function that can be minimised

fit_energy = False

class flarestack.core.llh.**LLH**(*season*, *sources*, *llh_dict*)
Base class LLH.

__init__(*season*, *sources*, *llh_dict*)
Initialize self. See help(type(self)) for accurate signature.

static assume_background(*n_s*, *n_coincident*, *n_all*)
To save time with likelihood calculation, it can be assumed that all events defined as "non-coincident", because of distance in space and time to the source, are in fact background events. This is equivalent to setting S=0 for all non-coincident events. IN this case, the likelihood can be calculated as the product of the number of non-coincident events, and the likelihood of an event which has S=0.

Parameters

- **n_s** – Array of expected number of events
- **n_coincident** – Number of events that were not assumed to have S=0
- **n_all** – The total number of events

Returns Log Likelihood value for the given

background_pdf(*source*, *cut_data*)
Calculates the value of the background spatial PDF for a given source for each event in the coincident data subsample. Thus is done by calling the self.bkg_spline spline function, which was fitted to the Sin(Declination) distribution of the data.

If there is a signal Time PDF given, then the background time PDF is also calculated for each event. This is assumed to be a normalised uniform distribution for the season.

Returns either the background spatial PDF values, or the product of the background spatial and time PDFs.

Parameters

- **source** – Source to be considered
- **cut_data** – Subset of Dataset with coincident events

Returns Array of Background Spacetime PDF values

background_spatial(*cut_data*)

calculate_test_statistic(*params*, *weights*, ***kwargs*)

classmethod create(*season*, *sources*, *llh_dict*)

create_background_function()

create_energy_functions()
Creates the acceptance function, which parameterises signal acceptance as a function of declination, and the energy weighting function, which gives the energy signal-over-background ratio

Returns Acceptance function, energy_weighting_function

create_kwargs(*data*, *pull_corrector*, *weight_f=None*)

create_llh_function(*data*, *pull_corrector*, *weight_f=None*)
Creates a likelihood function to minimise, based on the dataset.

Parameters **data** – Dataset

Returns LLH function that can be minimised

classmethod get_injected_parameters(*mh_dict*)

classmethod get_parameters(*llh_dict*)

classmethod register_subclass(*llh_name*)
Adds a new subclass of EnergyPDF, with class name equal to "energy_pdf_name".

static return_injected_parameters(*mh_dict*)

static return_llh_parameters(*llh_dict*)

select_spatially_coincident_data(*data*, *sources*)
Checks each source, and only identifies events in data which are both spatially and time-coincident with the source. Spatial coincidence is defined as a +/- 5 degree box centered on the given source. Time coincidence is determined by the parameters of the LLH Time PDF. Produces a mask for the dataset, which removes all events which are not coincident with at least one source.

Parameters

- **data** – Dataset to be tested
- **sources** – Sources to be tested

Returns Mask to remove

signal_pdf(*source*, *cut_data*)
Calculates the value of the signal spatial PDF for a given source for each event in the coincident data subsample. If there is a Time PDF given, also calculates the value of the signal Time PDF for each event. Returns either the signal spatial PDF values, or the product of the signal spatial and time PDFs.

Parameters

- **source** – Source to be considered
- **cut_data** – Subset of Dataset with coincident events

Returns Array of Signal Spacetime PDF values

subclasses = {'fixed_energy': <class 'flarestack.core.llh.FixedEnergyLLH'>, 'spatial'

class flarestack.core.llh.**SpatialLLH**(*season*, *sources*, *llh_dict*)
Most basic LLH, in which only spatial, and optionally also temporal, information is included. No Energy PDF is used, and no energy weighting is applied.

__init__(*season*, *sources*, *llh_dict*)
Initialize self. See help(type(self)) for accurate signature.

calculate_test_statistic(*params*, *weights*, ***kwargs*)
Calculates the test statistic, given the parameters. Uses numexpr for faster calculations.

Parameters

- **params** – Parameters from Minimisation
- **weights** – Normalised fraction of n_s allocated to each source

Returns 2 * llh value (Equal to Test Statistic)

create_energy_function()
In the most simple case of spatial-only weighting, you would neglect the energy weighting of events. Then, you can simply assume that the detector acceptance is roughly proportional to the data rate, i.e assuming that the incident background atmospheric neutrino flux is uniform. Thus the acceptance of the detector is simply the background spatial PDF (which is a spline fitted to data as a function of declination). This method does, admittedly neglect the fact that background in the southern hemisphere is mainly composed of muon bundles, rather than atmospheric neutrinos. Still, it's slighty better than assuming a uniform detector acceptance

Returns 1D linear interpolation

create_llh_function(*data*, *pull_corrector*, *weight_f=None*)
Creates a likelihood function to minimise, based on the dataset.

Parameters

- **data** – Dataset
- **pull_corrector** – pull_corrector

Returns LLH function that can be minimised

fit_energy = False

class flarestack.core.llh.**StandardLLH**(*season*, *sources*, *llh_dict*)

__init__(*season*, *sources*, *llh_dict*)
Initialize self. See help(type(self)) for accurate signature.

calculate_test_statistic(*params*, *weights*, ***kwargs*)
Calculates the test statistic, given the parameters. Uses numexpr for faster calculations.

Parameters

- **params** – Parameters from Minimisation
- **weights** – Normalised fraction of n_s allocated to each source

Returns 2 * llh value (Equal to Test Statistic)

create_SoB_energy_cache(*cut_data*)
Evaluates the Log(Signal/Background) values for all coincident data. For each value of gamma in self.gamma_support_points, calculates the Log(Signal/Background) values for the coincident data. Then saves each weight array to a dictionary.

Parameters **cut_data** – Subset of the data containing only coincident events

Returns Dictionary containing SoB values for each event for each

gamma value.

create_acceptance_function()
Creates a 2D linear interpolation of the acceptance of the detector for the given season, as a function of declination and gamma. Returns this interpolation function.

Returns 2D linear interpolation

create_energy_functions()
Creates the acceptance function, which parameterises signal acceptance as a function of declination, and the energy weighting function, which gives the energy signal-over-background ratio

Returns Acceptance function, energy_weighting_function

create_kwargs(*data*, *pull_corrector*, *weight_f=None*)
Creates a likelihood function to minimise, based on the dataset.

Parameters **data** – Dataset

Returns LLH function that can be minimised

estimate_energy_weights(*gamma*, *energy_SoB_cache*)
Quickly estimates the value of Signal/Background for Gamma. Uses pre-calculated values for first and second derivatives. Uses a Taylor series to estimate S(gamma), unless SoB has already been calculated for a given gamma.

Parameters

- **gamma** – Spectral Index
- **energy_SoB_cache** – Weight cache

Returns Estimated value for S(gamma)

fit_energy = True

new_acceptance(*source*, *params=None*)
Calculates the detector acceptance for a given source, using the 2D interpolation of the acceptance as a function of declination and gamma. If gamma IS NOT being fit, uses the default value of gamma for weighting (determined in __init__). If gamma IS being fit, it will be the last entry in the parameter array, and is the acceptance uses this value.

Parameters

- **source** – Source to be considered
- **params** – Parameter array

Returns Value for the acceptance of the detector, in the given

season, for the source

static return_injected_parameters(*mh_dict*)

static return_llh_parameters(*llh_dict*)

class flarestack.core.llh.**StandardMatrixLLH**(*season*, *sources*, *llh_dict*)

create_kwargs(*data*, *pull_corrector*, *weight_f=None*)
Creates a likelihood function to minimise, based on the dataset.

Parameters **data** – Dataset

Returns LLH function that can be minimised

class flarestack.core.llh.**StandardOverlappingLLH**(*season*, *sources*, *llh_dict*)

calculate_test_statistic(*params*, *weights*, ***kwargs*)
Calculates the test statistic, given the parameters. Uses numexpr for faster calculations.

Parameters

- **params** – Parameters from Minimisation

- **weights** – Normalised fraction of n_s allocated to each source

Returns 2 * llh value (Equal to Test Statistic)

create_kwargs (*data*, *pull_corrector*, *weight_f=None*)
Creates a likelihood function to minimise, based on the dataset.

Parameters **data** – Dataset

Returns LLH function that can be minimised

flarestack.core.llh.**generate_dynamic_flare_class** (*season*, *sources*, *llh_dict*)

flarestack.core.llh.**read_llh_dict** (*llh_dict*)
Ensures that llh dictionaries remain backwards-compatible

Parameters **llh_dict** – LLH Dictionary

Returns LLH Dictionary compatible with new format

CHAPTER

THREE

UTILS

3.1 IceCube Utils

flarestack.icecube_utils.dataset_loader.**convert_grl**(*season*)

flarestack.icecube_utils.dataset_loader.**data_loader**(*data_path*, *floor=True*, *cut_fields=True*)

Helper function to load data for a given season/set of season. Adds sinDec field if this is not available, and combines multiple years of data is appropriate (different sets of data from the same icecube configuration should be given as a list)

Parameters

- **data_path** – Path to data or list of paths to data
- **cut_fields** – Boolean to remove unused fields from datasets on loading

Returns Loaded Dataset (experimental or MC)

flarestack.icecube_utils.dataset_loader.**grl_loader**(*season*)

flarestack.icecube_utils.dataset_loader.**verify_grl_with_data**(*seasons*)

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

f

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

U

V

W