dddm Documentation

Release 4.0.0

J. R. Angevaare

Apr 18, 2024

SETUP AND BASICS

Github page: https://github.com/JoranAngevaare/dddm

This code is for Angevaare, J. R., et al. "Complementarity of direct detection experiments in search of light Dark Matter." Journal of Cosmology and Astroparticle Physics 2022.10 (2022): 004. (https://iopscience.iop.org/article/10.1088/1475-7516/2022/10/004).

CHAPTER

ONE

CONTENT

1.1 Setting up dddm

For most practical purposes, one can simply run *pip install dddm* to install it via https://pypi.org/project/dddm/.

For some implementations (MultiNest), one is adviced to follow instructions as on this script: https://github.com/JoranAngevaare/dddm/blob/master/.github/scripts/install_on_linux.sh

That said, as ultranest is an implemented and working alternative method to MultiNest, a pip setup may prove much simpler.

1.2 dddm package

1.2.1 Subpackages

dddm.detectors package

Submodules

dddm.detectors.examples module

class dddm.detectors.examples.**ArgonSimple**(*n_energy_bins=10*, *e_min_kev=0*, *e_max_kev=100*)

Bases: `Experiment`

background_function(*energies_in_kev*)

Assume background free detector

cut_efficiency: Union[int, float] = 0.8

detection_efficiency: Union[int, float] = 0.8

detector_name: str = 'Ar_simple'

energy_threshold_kev: Union[int, float] = 30

exposure_tonne_year: Union[int, float] = 10

interaction_type: str = 'SI'

location: str = 'XENON'

resolution(*energies_in_kev*)
Simple square root dependency of the energy resolution

target_material: str = 'Ar'

class dddm.detectors.examples.**GermaniumSimple**(*n_energy_bins=10*, *e_min_kev=0*, *e_max_kev=100*)
Bases: *Experiment*

background_function(*energies_in_kev*)
Assume background free detector

cut_efficiency: Union[int, float] = 0.8

detection_efficiency: Union[int, float] = 0.9

detector_name: str = 'Ge_simple'

energy_threshold_kev: Union[int, float] = 10

exposure_tonne_year: Union[int, float] = 3

interaction_type: str = 'SI'

location: str = 'SUF'

resolution(*energies_in_kev*)
Simple resolution model

target_material: str = 'Ge'

class dddm.detectors.examples.**XenonSimple**(*n_energy_bins=10*, *e_min_kev=0*, *e_max_kev=100*)
Bases: *Experiment*

background_function(*energies_in_kev*)
Assume background free detector

cut_efficiency: Union[int, float] = 0.8

detection_efficiency: Union[int, float] = 0.5

detector_name: str = 'Xe_simple'

energy_threshold_kev: Union[int, float] = 10

exposure_tonne_year: Union[int, float] = 5

interaction_type: str = 'SI'

location: str = 'XENON'

resolution(*energies_in_kev*)
Simple square root dependency of the energy resolution

target_material: str = 'Xe'

dddm.detectors.experiment module

class dddm.detectors.experiment.**Experiment**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: object

Base class of experiments. To use, subclass and set the required attributes

background_function(*energies_in_kev: ndarray*) → ndarray

Return background at <energies [keV>

property config

cut_efficiency: Union[int, float] = None

detection_efficiency: Union[int, float] = None

property detector_hash

detector_name: str = None

e_max_kev: Union[int, float] = None

e_min_kev: Union[int, float] = None

property effective_exposure

energy_threshold_kev: Union[int, float] = None

exposure_tonne_year: Union[int, float] = None

interaction_type: str = 'SI'

location: str = None

n_energy_bins: int = 50

resolution(*energies_in_kev: ndarray*) → ndarray

Return resolution at <energies [keV]>

target_material: str = None

dddm.detectors.super_cdms module

class dddm.detectors.super_cdms.**SuperCdmsHvGeMigdal**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.85

detection_efficiency: Union[int, float] = 0.5

detector_name: str = 'SuperCDMS_HV_Ge_Migdal'

property energy_threshold_kev

exposure_tonne_year: Union[int, float] = 0.044

interaction_type: str = 'migdal_SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Ge'

class dddm.detectors.super_cdms.**SuperCdmsHvGeNr**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.85

detection_efficiency: Union[int, float] = 0.85

detector_name: str = 'SuperCDMS_HV_Ge_NR'

energy_threshold_kev: Union[int, float] = 0.04

exposure_tonne_year: Union[int, float] = 0.044

interaction_type: str = 'SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Ge'

class dddm.detectors.super_cdms.**SuperCdmsHvSiMigdal**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.85

detection_efficiency: Union[int, float] = 0.675

detector_name: str = 'SuperCDMS_HV_Si_Migdal'

property **energy_threshold_kev**

exposure_tonne_year: Union[int, float] = 0.0096

interaction_type: str = 'migdal_SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Si'

class dddm.detectors.super_cdms.**SuperCdmsHvSiNr**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.85

detection_efficiency: Union[int, float] = 0.85

detector_name: str = 'SuperCDMS_HV_Si_NR'

energy_threshold_kev: Union[int, float] = 0.078

exposure_tonne_year: Union[int, float] = 0.0096

interaction_type: str = 'SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Si'

class dddm.detectors.super_cdms.**SuperCdmsIzipGeMigdal**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.75

detection_efficiency: Union[int, float] = 0.5

detector_name: str = 'SuperCDMS_iZIP_Ge_Migdal'

property energy_threshold_kev

exposure_tonne_year: Union[int, float] = 0.056

interaction_type: str = 'migdal_SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Ge'

class dddm.detectors.super_cdms.**SuperCdmsIzipGeNr**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.75

detection_efficiency: Union[int, float] = 0.85

detector_name: str = 'SuperCDMS_iZIP_Ge_NR'

energy_threshold_kev: Union[int, float] = 0.272

exposure_tonne_year: Union[int, float] = 0.056

interaction_type: str = 'SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Ge'

class dddm.detectors.super_cdms.**SuperCdmsIzipSiMigdal**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.75

detection_efficiency: Union[int, float] = 0.675

detector_name: str = 'SuperCDMS_iZIP_Si_Migdal'

property energy_threshold_kev

exposure_tonne_year: Union[int, float] = 0.0048

interaction_type: str = 'migdal_SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Si'

class dddm.detectors.super_cdms.**SuperCdmsIzipSiNr**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseSuperCdms

background_function(*energies_in_kev*)

Flat bg rate

cut_efficiency: Union[int, float] = 0.75

detection_efficiency: Union[int, float] = 0.85

detector_name: str = 'SuperCDMS_iZIP_Si_NR'

energy_threshold_kev: Union[int, float] = 0.166

exposure_tonne_year: Union[int, float] = 0.0048

interaction_type: str = 'SI'

resolution(*energies_in_kev*)

Flat resolution

target_material: str = 'Si'

dddm.detectors.xenon_nt module

class dddm.detectors.xenon_nt.**XenonNtMigdal**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseXenonNt

background_function(*energies_in_kev*)

Returns
ER background for Xe detector in events/keV/t/yr

cut_efficiency: Union[int, float] = 0.82

detection_efficiency: Union[int, float] = 1

detector_name: str = 'XENONnT_Migdal'

energy_threshold_kev: Union[int, float] = 1

interaction_type: str = 'migdal_SI'

resolution(*energies_in_kev*)

Assume the same as the 1T resolution

class dddm.detectors.xenon_nt.**XenonNtNr**(*n_energy_bins=50*, *e_min_kev=0*, *e_max_kev=5*)

Bases: _BaseXenonNt

background_function(*energies_in_kev*)

Returns
NR background for Xe detector in events/keV/t/yr

cut_efficiency: Union[int, float] = 0.83

detection_efficiency: Union[int, float] = 1

detector_name: str = 'XENONnT_NR'

energy_threshold_kev: Union[int, float] = 1.6

interaction_type: str = 'SI'

resolution(*energies_in_kev*)

Use _get_nr_resolution to calculate the energy resolution.

Parameters
energies_in_kev – NR energies to evaluate the resolution function at

Returns

Module contents

dddm.plotting package

Submodules

dddm.plotting.confidence_figures module

Utility for opening and displaying results from multinest optimization class dddm.plotting.confidence_figures.**DDDMResult**(*path*, *sampler='multinest'*)

Bases: object

Parse results from fitting from nested sampling

config_summary(*get_props=('detector', 'mass', 'sigma', 'nlive', 'halo_model', 'notes', 'n_parameters')*) → DataFrame

property detector

get_from_config(*to_get: str*, *if_not_available=None*)

get_samples()

property halo_model

property mass

property n_parameters

property nlive

property notes

result: dict = None

result_summary() → DataFrame

setup()

property sigma

summary() → DataFrame

class dddm.plotting.confidence_figures.**ResultsManager**(*pattern=None*, *sampler='multinest'*)

Bases: object

add_result(*path: str*)

apply_mask(*mask*)

build_df()

property df

Lazy alias

register_pattern(*pattern*, *show_tqdm=True*)

result_cache: list = None

result_df: DataFrame = None

class dddm.plotting.confidence_figures.**SeabornPlot**(*result: DDDMResult*)

Bases: object

best_fit() → tuple

plot_bench(*c='cyan'*, ***kwargs*)

plot_best_fit(***kwargs*) → None

plot_kde(***kwargs*)

plot_samples(***kwargs*) → None

plot_sigma_contours(*nsigma=2*, ***kwargs*)

property samples: ndarray

samples_to_df() → DataFrame

dddm.plotting.plot_basics module

Some basic functions for plotting et cetera. Used to for instance to check that the likelihood function is well behaved

dddm.plotting.plot_basics.**error_bar_hist**(*ax*, *data*, *data_range=None*, *nbins=50*, ***kwargs*)

dddm.plotting.plot_basics.**get_color_from_range**(*val*, *_range=(0, 1)*, *it=0*)

dddm.plotting.plot_basics.**hist_data**(*data*, *data_range=None*, *nbins=50*)

dddm.plotting.plot_basics.**ll_element_wise**(*x*, *y*, *clip_val=-10000.0*)

dddm.plotting.plot_basics.**open_pickle_figure**(*name*)

dddm.plotting.plot_basics.**pickle_dump_figure**(*name*)

dddm.plotting.plot_basics.**plot_spectrum**(*data*, *color='blue'*, *label='label'*, *linestyle='none'*, *plot_error=True*)

dddm.plotting.plot_basics.**plt_ll_mass_det**(*det_class=<class 'dddm.detectors.examples.XenonSimple'>*, *bins=10*, *m=50*, *sig=1e-45*)

dddm.plotting.plot_basics.**plt_ll_mass_spec**(*det_class=<class 'dddm.detectors.examples.XenonSimple'>*, *bins=10*, *m=50*, *sig=1e-45*)

dddm.plotting.plot_basics.**plt_ll_sigma_det**(*det_class=<class 'dddm.detectors.examples.XenonSimple'>*, *bins=10*, *m=50*, *sig=1e-45*)

dddm.plotting.plot_basics.**plt_ll_sigma_mass**(*spec_clas*, *vary*, *det_class=<class 'dddm.detectors.examples.XenonSimple'>*, *bins=10*, *m=50*, *sig=1e-45*)

dddm.plotting.plot_basics.**plt_ll_sigma_spec**(*det_class=<class 'dddm.detectors.examples.XenonSimple'>*, *bins=10*, *m=50*, *sig=1e-45*)

dddm.plotting.plot_basics.**plt_priors**(*itot=100*)

dddm.plotting.plot_basics.**save_canvas**(*name*, *save_dir='./figures'*, *dpi=200*, *tight_layout=False*, *pickle_dump=True*)

Wrapper for saving current figure

dddm.plotting.plot_basics.**show_ll_function**(*npoints=10000.0*, *clip_val=-10000.0*, *min_val=0.1*)

dddm.plotting.plot_basics.**simple_hist**(*y: ndarray*)

dddm.plotting.seaborn_utils module

" Small script to extract the results from seaborn to calculate confidence intervals

I'm sorry for this script, I wanted to have something robust but I couldn't find it anyware. Seaborn is doing a great job, so let's use it's functionality.

This work is mostly based on: https://github.com/mwaskom/seaborn/blob/ff0fc76b4b65c7bcc1d2be2244e4ca1a92e4e740/seaborn/distributions.py

dddm.plotting.seaborn_utils.**one_sigma_area**(*x*, *y*, *clf=True*, ***kwargs*)

Module contents

dddm.recoil_rates package

Submodules

dddm.recoil_rates.detector_spectrum module

Introduce detector effects into the expected detection spectrum

class dddm.recoil_rates.detector_spectrum.**DetectorSpectrum**(*dark_matter_model: Union[SHM, ShieldedSHM]*, *experiment: Experiment*)

Bases: *GenSpectrum*

Convolve a recoil spectrum with the detector effects:

- background levels
- energy resolution
- energy threshold

static above_threshold(*rates: ndarray*, *e_bin_edges: ndarray*, *e_thr: Union[float, int]*)

Apply threshold to the rates. We are right edge inclusive bin edges : **|bin0|bin1|bin2|** e_thr : | bin0 -> 0 bin1 -> fraction of bin1 > e_thr bin2 -> full content

Parameters

- **rates** – bins with the number of counts
- **e_bin_edges** – 2d array of the left, right bins
- **e_thr** – energy threshold

Returns

rates with energy threshold applied

dddm.recoil_rates.halo module

For a given detector get a WIMPrate for a given detector (not taking into account any detector effects

```
class dddm.recoil_rates.halo.SHM(v_0=None, v_esc=None, rho_dm=None)
```

Bases: `object`

class used to pass a halo model to the rate computation must contain: :param v_esc – escape velocity (multiplied by units) :param rho_dm – density in mass/volume of dark matter at the Earth (multiplied by units) The standard halo model also allows variation of v_0 :param v_0 – v0 of the velocity distribution (multiplied by units) :function velocity_dist – function taking v,t giving normalised velocity distribution in earth rest-frame.

parameter_dict()

Return a dict of readable parameters of the current settings

velocity_dist(*v*, *t*)

Get the velocity distribution in units of per velocity, :param v: v is in units of velocity :return: observed velocity distribution at earth

dddm.recoil_rates.halo_shielded module

```
class dddm.recoil_rates.halo_shielded.ShieldedSHM(location, file_folder='./verne_files', v_0=None,
                                                  v_esc=None, rho_dm=None,
                                                  log_cross_section=None, log_mass=None)
```

Bases: `object`

class used to pass a halo model to the rate computation based on the earth shielding effect as calculated by Verne must contain:

:param v_esc – escape velocity (multiplied by units) :param rho_dm – density in mass/volume of dark matter at the Earth (multiplied by units)

The standard halo model also allows variation of v_0

:param v_0 – v0 of the velocity distribution (multiplied by units) :function velocity_dist – function taking v,t giving normalised

velocity distribution in earth rest-frame.

load_f()

load the velocity distribution. If there is no velocity
distribution shaved, load one.

Returns

parameter_dict()

Return a dict of readable parameters of the current settings

property rho_dm_nodim

property v_0_nodim

property v_esc_nodim

velocity_dist(*v*, *t*)

Get the velocity distribution in units of per velocity, :param v: v is in units of velocity :return: observed velocity distribution at earth

dddm.recoil_rates.spectrum module

class dddm.recoil_rates.spectrum.**GenSpectrum**(*dark_matter_model: Union[SHM, ShieldedSHM]*, *experiment: Experiment*)

Bases: `object`

property darkelf_class

get_bin_edges()

get_counts(*wimp_mass: Union[int, float]*, *cross_section: Union[int, float]*, *poisson=False*) → array

Parameters

- **wimp_mass** – wimp mass (not log)
- **cross_section** – cross-section of the wimp nucleon interaction (not log)
- **poisson** – type bool, add poisson True or False

Returns
array of counts/bin

get_data(*wimp_mass: Union[int, float]*, *cross_section: Union[int, float]*, *poisson=False*, *return_counts=False*) → Union[DataFrame, ndarray]

Parameters

- **wimp_mass** – wimp mass (not log)
- **cross_section** – cross-section of the wimp nucleon interaction (not log)
- **poisson** – type bool, add poisson True or False
- **return_counts** – instead of a dataframe, return counts only

Returns
pd.DataFrame containing events binned in energy

required_detector_fields = ['name', 'material', 'type', 'exp_eff']

set_negative_to_zero(*counts: ndarray*)

spectrum_simple(*energy_bins: Union[list, tuple, ndarray]*, *wimp_mass: Union[int, float]*, *cross_section: Union[int, float]*)

Compute the spectrum for a given mass and cross-section :param wimp_mass: wimp mass (not log) :param cross_section: cross-section of the wimp nucleon interaction

(not log)

Returns
returns the rate

Module contents

dddm.samplers package

Submodules

dddm.samplers.emcee module

Do a likelihood fit. The class MCMCStatModel is used for fitting applying the MCMC algorithm emcee.

MCMC is:
slower than the nestle package; and harder to use since one has to choose the 'right' initial parameters

Nevertheless, the walkers give great insight in how the likelihood-function is felt by the steps that the walkers make

`class dddm.samplers.emcee.MCMCStatModel`(*wimp_mass: Union[float, int], cross_section: Union[float, int], spectrum_class: Union[DetectorSpectrum, GenSpectrum], prior: dict, tmp_folder: str, fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k'), detector_name=None, verbose=False, notes='default', nwalkers=50, nsteps=100, remove_frac=0.2, emcee_thin=15*)

Bases: `StatModel`

`run()`

`save_results`(*save_to_dir='emcee', force_index=False*)

`set_sampler`(*mult=True*)
init the MCMC sampler

`show_corner()`

`show_walkers()`

dddm.samplers.multi_detectors module

`class dddm.samplers.multi_detectors.CombinedMultinest`(*wimp_mass: Union[float, int], cross_section: Union[float, int], spectrum_class: List[Union[DetectorSpectrum, GenSpectrum]], prior: dict, tmp_folder: str, results_dir: Optional[str] = None, fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k'), detector_name=None, verbose=False, notes='default', nlive=1024, tol=0.1*)

Bases: `_CombinedInference`, `MultiNestSampler`

`class dddm.samplers.multi_detectors.CombinedNestle`(*wimp_mass: Union[float, int], cross_section: Union[float, int], spectrum_class: List[Union[DetectorSpectrum, GenSpectrum]], prior: dict, tmp_folder: str, results_dir: Optional[str] = None, fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k'), detector_name=None, verbose=False, notes='default', nlive=1024, tol=0.1*)

Bases: _CombinedInference, NestleSampler

class dddm.samplers.multi_detectors.**CombinedUltraNest**(*wimp_mass: Union[float, int], cross_section: Union[float, int], spectrum_class: List[Union[DetectorSpectrum, GenSpectrum]], prior: dict, tmp_folder: str, results_dir: Optional[str] = None, fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k'), detector_name=None, verbose=False, notes='default', nlive=1024, tol=0.1*)

Bases: _CombinedInference, UltraNestSampler

dddm.samplers.nestle module

class dddm.samplers.nestle.**NestleSampler**(*wimp_mass: Union[float, int], cross_section: Union[float, int], spectrum_class: Union[DetectorSpectrum, GenSpectrum], prior: dict, tmp_folder: str, results_dir: Optional[str] = None, fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k'), detector_name=None, verbose=False, notes='default', nlive=1024, tol=0.1*)

Bases: MultiNestSampler

get_summary()

run()

save_results(*force_index=False*)

show_corner()

dddm.samplers.pymultinest module

Do a likelihood fit. The class NestedSamplerStatModel is used for fitting applying the bayesian algorithm nestle/multinest

class dddm.samplers.pymultinest.**MultiNestSampler**(*wimp_mass: Union[float, int], cross_section: Union[float, int], spectrum_class: Union[DetectorSpectrum, GenSpectrum], prior: dict, tmp_folder: str, results_dir: Optional[str] = None, fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k'), detector_name=None, verbose=False, notes='default', nlive=1024, tol=0.1*)

Bases: StatModel

check_did_run()

check_did_save()

get_save_dir(*force_index=False, _hash=None*) → str

get_summary()

log_prior_transform_nested(*x*, *x_name*)

log_probability_nested(*parameter_vals*, *parameter_names*)

Parameters
`parameter_vals` – the values of the model/benchmark considered as the truth

:param parameter_values: the values of the parameters that are being varied :param parameter_names: the names of the parameter_values :return:

run()

save_results(*force_index=False*)

show_corner()

Module contents

1.2.2 Submodules

1.2.3 dddm.context module

Setup the file structure for the software. Specifies several folders: software_dir: path of installation

dddm.context.**base_context**()

1.2.4 dddm.priors module

dddm.priors.**get_priors**(*priors_from='Evans_2019'*)

Returns
dictionary of priors, type and values

1.2.5 dddm.statistics module

Statistical model giving likelihoods for detecting a spectrum given a benchmark to compare it with.

class dddm.statistics.**StatModel**(*wimp_mass: Union[float, int]*, *cross_section: Union[float, int]*, *spectrum_class: Union[DetectorSpectrum, GenSpectrum]*, *prior: dict*, *tmp_folder: str*, *fit_parameters=('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density', 'k')*, *detector_name=None*, *verbose=False*, *notes='default'*)

Bases: `object`

allow_multiple_detectors = False

property bench_is_set

benchmark_values = None

check_spectrum()
Lazy alias for eval_spectrum

property density: Union[int, float]

eval_benchmark()

eval_spectrum(*values: Union[list, tuple, ndarray], parameter_names: Union[List[str], Tuple[str]]*)

For given values and parameter names, return the spectrum one would have with these parameters. The values and parameter names should be array like objects of the same length. Usually, one fits either two ('log_mass', 'log_cross_section') or five parameters ('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density'). :param values: array like object of :param parameter_names: names of parameters :return: a spectrum as specified by the parameter_names

get_logger(*tmp_folder*, *verbosity*)

known_parameters = ('log_mass', 'log_cross_section', 'v_0', 'v_esc', 'density')

property log_cross_section

property log_mass

log_prior(*value*, *variable_name*)

Compute the prior of variable_name for a given value :param value: value of variable name :param variable_name: name of the 'value'. This name should be in the config of the class under the priors with a similar content as the priors as specified in the get_prior function. :return: prior of value

log_probability(*parameter_vals*, *parameter_names*)

Parameters

- **parameter_vals** – the values of the model/benchmark considered as the truth
- **parameter_names** – the names of the parameter_values

Returns

read_priors_mean(*prior_name*) → Union[int, float]

set_benchmark()

Set up the benchmark used in this statistical model. Likelihood of other models can be evaluated for this 'truth'

set_fit_parameters(*params*)

Write the fit parameters to the config

set_models()

Update the dm model with with the required settings from the prior

total_log_prior(*parameter_vals*, *parameter_names*)

For each of the parameter names, read the prior

Parameters

- **parameter_vals** – the values of the model/benchmark considered as the truth
- **parameter_names** – the names of the parameter_values

Returns

property v_0: Union[int, float]

property v_esc: Union[int, float]

1.2.6 dddm.test_utils module

`dddm.test_utils.test_context()`

just returns the base contexts, might be different one day

1.2.7 dddm.utils module

Basic functions for saving et cetera

`dddm.utils.deterministic_hash(thing, length=10)`

Return a base32 lowercase string of length determined from hashing a container hierarchy

`dddm.utils.exporter(export_self=False)`

Export utility modified from https://stackoverflow.com/a/41895194 Returns export decorator, __all__ list stolen from https://github.com/AxFoundation/strax/blob/d3608efc77acd52e1d5a208c3092b6b45b27a6e2/strax/utils.py#46

`dddm.utils.is_installed(module)`

Try to import <module>, return False if not installed

`dddm.utils.is_windows()`

`dddm.utils.print_versions(modules=('dddm', 'numpy', 'numba', 'wimprates'), print_output=True, include_python=True, return_string=False, include_git=True)`

Print versions of modules installed.

Parameters

- **modules** – Modules to print, should be str, tuple or list. E.g. print_versions(modules=('numpy', 'dddm',))
- **return_string** – optional. Instead of printing the message, return a string
- **include_git** – Include the current branch and latest commit hash

Returns

optional, the message that would have been printed

`dddm.utils.to_str_tuple(x: Union[str, bytes, list, tuple, Series, ndarray]) → Tuple[str]`

Convert any sensible instance to a tuple of strings stolen from https://github.com/AxFoundation/strax/blob/d3608efc77acd52e1d5a208c3092b6b45b27a6e2/strax/utils.py#242

1.2.8 Module contents

CHAPTER TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

A

B

C

D

E

G

H

I

K

L

M

N

O

P

R

S

T

V

X