lenstronomy Documentation

Release 1.11.1

Simon Birrer

Mar 08, 2023

Contents

1 Getting started **3**

2 Example notebooks **5**

3 Affiliated packages **7**

4 Mailing list and Slack channel **9**

5 Contribution **11**

6 Attribution **13**

6.1 Contents: 13

Python Module Index **403**

Index **407**

`lenstronomy` is a multi-purpose software package to model strong gravitational lenses. `lenstronomy` finds application for time-delay cosmography and measuring the expansion rate of the Universe, for quantifying lensing substructure to infer dark matter properties, morphological quantification of galaxies, quasar-host galaxy decomposition and much more. A (incomplete) list of publications making use of lenstronomy can be found at this link.

The development is coordinated on GitHub and contributions are welcome. The documentation of `lenstronomy` is available at readthedocs.org and the package is distributed through PyPI and conda-forge. `lenstronomy` is an affiliated package of astropy.

lenstronomy releases are distributed through PyPI and conda-forge. Instructions for installing lenstronomy and its dependencies can be found in the Installation section of the documentation. Specific instructions for settings and installation requirements for special cases that can provide speed-ups, we also refer to the Installation page.

CHAPTER 1

Getting started

The starting guide jupyter notebook leads through the main modules and design features of `lenstronomy`. The modular design of `lenstronomy` allows the user to directly access a lot of tools and each module can also be used as stand-alone packages.

If you are new to gravitational lensing, check out the mini lecture series giving an introduction to gravitational lensing with interactive Jupyter notebooks in the cloud.

CHAPTER 2

Example notebooks

We have made an extension module available at https://github.com/lenstronomy/lenstronomy-tutorials. You can find simple example notebooks for various cases. The latest versions of the notebooks should be compatible with the recent pip version of lenstronomy.

CHAPTER 3

Affiliated packages

Multiple affiliated packages that make use of lenstronomy can be found here (not complete) and further packages are under development by the community.

CHAPTER 4

Mailing list and Slack channel

You can join the `lenstronomy` mailing list by signing up on the google groups page.

The email list is meant to provide a communication platform between users and developers. You can ask questions, and suggest new features. New releases will be announced via this mailing list.

We also have a Slack channel for the community. Please send us an email such that we can add you to the channel.

If you encounter errors or problems with `lenstronomy`, please let us know!

CHAPTER 5

Contribution

Check out the contributing page and become an author of `lenstronomy`! A big shout-out to the current list of contributors and developers!

CHAPTER 6

Attribution

The design concept of `lenstronomy` is reported by Birrer & Amara 2018 and is based on Birrer et al 2015. The current JOSS software publication is presented by Birrer et al. 2021. Please cite Birrer & Amara 2018 and Birrer et al. 2021 when you use lenstronomy in a publication and link to https://github.com/lenstronomy/lenstronomy. Please also cite Birrer et al 2015 when you make use of the `lenstronomy` work-flow or the Shapelet source reconstruction and make sure to cite also the relevant work that was implemented in `lenstronomy`, as described in the release paper and the documentation. Don't hesitate to reach out to the developers if you have questions!

6.1 Contents:

6.1.1 Installation

This page outlines how to install one of the officially distributed lenstronomy releases and its dependencies, or install and test the latest development version.

From PyPI

All lenstronomy releases are distributed through the Python Package Index (PyPI). To install the latest version use pip:

At the command line with pip:

```
$ pip install lenstronomy
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv lenstronomy
$ pip install lenstronomy
```

From conda-forge

All lenstronomy releases are also distributed for conda through the conda-forge channel. To install the latest version for your active conda environment:

```
$ conda install -c conda-forge lenstronomy
```

You can also clone the github repository for development purposes.

Requirements

Make sure the standard python libraries as specified in the requirements. The standard usage does not require all libraries to be installed, in particular the different posterior samplers are only required when being used.

In the following, a few specific cases are mentioned that may require special attention in the installation and settings, in particular when it comes to MPI and HPC applications.

MPI

MPI support is provided for several sampling techniques for parallel computing. A specific version of the library schwimmbad is required for the correct support of the moving of the likelihood elements from one processor to another with MPI. Pay attention ot the requirements.

NUMBA

Just-in-time (jit) compilation with numba can provide significant speed-up for certain calculations. There are specific settings for the settings provided as per default, but these may need to be adopted when running on a HPC cluster. You can define your own configuration file in your $XDG_CONFIG_HOME/lenstronomy/config.yaml file. E.g. (check your system for the path):

```
$ ~/.conf/lenstronomy/config.yaml
```

following the format of the default configuration which is here.

FASTELL

The fastell4py package, originally from Barkana (fastell), is required to run the PEMD (power-law elliptical mass distribution) lens model and can be cloned from: https://github.com/sibirrer/fastell4py (needs a fortran compiler). We recommend using the EPL model as it is a pure python version of the same profile.

```
$ sudo apt-get install gfortran
$ git clone https://github.com/sibirrer/fastell4py.git <desired location>
$ cd <desired location>
$ python setup.py install --user
```

Check installation by running tests

You can check your installation with pytest:

```
$ cd <lenstronomy_repo>
$ py.test
```

Or you can run a partial test with:

```
$ cd <lenstronomy_repo>
$ py.test/test/test_LensModel/
```

You can also run the tests with tox in a virtual environment with:

```
$ cd <lenstronomy_repo>
$ tox
```

Note: tox might have trouble with the PyMultiNest installation and the cmake part of it.

6.1.2 Usage

To use lenstronomy in a project:

```
import lenstronomy
```

Getting started

The starting guide jupyter notebook leads through the main modules and design features of `lenstronomy`. The modular design of `lenstronomy` allows the user to directly access a lot of tools and each module can also be used as stand-alone packages.

Example notebooks

We have made an extension module available at https://github.com/lenstronomy/lenstronomy-tutorials. You can find simple example notebooks for various cases. The latest versions of the notebooks should be compatible with the recent pip version of lenstronomy.

6.1.3 Contributing to lenstronomy

Contributor Guidelines

GitHub Workflow

Fork and Clone the lenstronomy Repository

You should only need to do this step once

First *fork* the lenstronomy repository. A fork is your own remote copy of the repository on GitHub. To create a fork:

1. Go to the lenstronomy GitHub Repository
2. Click the **Fork** button (in the top-right-hand corner)
3. Choose where to create the fork, typically your personal GitHub account

Next *clone* your fork. Cloning creates a local copy of the repository on your computer to work with. To clone your fork:

```
git clone https://github.com/<your-account>/lenstronomy.git
```

Finally add the `lenstronomyproject` repository as a *remote*. This will allow you to fetch changes made to the codebase. To add the `lenstronomyproject` remote:

```
cd lenstronomy
git remote add lenstronomyproject https://github.com/lenstronomy/lenstronomy.git
```

Install your local lenstronomy version

To enable that your new code gets accessible by python also outside of the development environment, make sure all previous versions of lenstronomy are uninstalled and then install your version of lenstronomy (aka add the software to the python path)

```
cd lenstronomy
python setup.py develop --user
```

Alternatively, create virtual environments for the development (recommended for advanced usage with multiple branches).

Create a branch for your new feature

Create a *branch* off the *lenstronomyproject* main branch. Working on unique branches for each new feature simplifies the development, review and merge processes by maintaining logical separation. To create a feature branch:

```
git fetch lenstronomyproject
git checkout -b <your-branch-name> lenstronomyproject/main
```

Hack away!

Write the new code you would like to contribute and *commit* it to the feature branch on your local repository. Ideally commit small units of work often with clear and descriptive commit messages describing the changes you made. To commit changes to a file:

```
git add file_containing_your_contribution
git commit -m 'Your clear and descriptive commit message'
```

Push the contributions in your feature branch to your remote fork on GitHub:

```
git push origin <your-branch-name>
```

Note: The first time you *push* a feature branch you will probably need to use *–set-upstream origin* to link to your remote fork:

```
git push --set-upstream origin <your-branch-name>
```

Open a Pull Request

When you feel that work on your new feature is complete, you should create a *Pull Request*. This will propose your work to be merged into the main lenstronomy repository.

1. Go to lenstronomy Pull Requests
2. Click the green **New pull request** button

3. Click **compare across forks**
4. Confirm that the base fork is `lenstronomy/lenstronomy` and the base branch is `main`
5. Confirm the head fork is `<your-account>/lenstronomy` and the compare branch is `<your-branch-name>`
6. Give your pull request a title and fill out the the template for the description
7. Click the green **Create pull request** button

Updating your branch

As you work on your feature, new commits might be made to the `lenstronomy/lenstronomy` main branch. You will need to update your branch with these new commits before your pull request can be accepted. You can achieve this in a few different ways:

- If your pull request has no conflicts, click **Update branch**
- If your pull request has conflicts, click **Resolve conflicts**, manually resolve the conflicts and click **Mark as resolved**
- *merge* the `lenstronomyproject` main branch from the command line:

```
git fetch lenstronomyproject
git merge lenstronomyproject/main
```

- *rebase* your feature branch onto the `lenstronomy` main branch from the command line:

```
git fetch lenstronomyproject
git rebase lenstronomyproject/main
```

Warning: It is bad practice to *rebase* commits that have already been pushed to a remote such as your fork. Rebasing creates new copies of your commits that can cause the local and remote branches to diverge. `git push --force` will **overwrite** the remote branch with your newly rebased local branch. This is strongly discouraged, particularly when working on a shared branch where you could erase a collaborators commits.

For more information about resolving conflicts see the GitHub guides:

- Resolving a merge conflict on GitHub
- Resolving a merge conflict using the command line
- About Git rebase

More Information

More information regarding the usage of GitHub can be found in the GitHub Guides.

Coding Guidelines

Before your pull request can be merged into the codebase, it will be reviewed by one of the lenstronomy developers and required to pass a number of automated checks. Below are a minimum set of guidelines for developers to follow:

General Guidelines

- lenstronomy is compatible with Python>=3.7 (see setup.cfg). lenstronomy *does not* support backwards compatibility with Python 2.x; *six*, *__future__* and *2to3* should not be used.
- All contributions should follow the PEP8 Style Guide for Python Code. We recommend using flake8 to check your code for PEP8 compliance.
- Importing lenstronomy should only depend on having NumPy, SciPy and Astropy installed.
- Code is grouped into submodules based e.g. LensModel, LightModel or ImSim. There is also a Util submodule for general utility functions.
- For more information see the Astropy Coding Guidelines.

Unit Tests

Pull requests will require existing unit tests to pass before they can be merged. Additionally, new unit tests should be written for all new public methods and functions. Unit tests for each submodule are contained in subdirectories called `tests` and you can run them locally using `python setup.py test`. For more information see the Astropy Testing Guidelines.

Docstrings

All public classes, methods and functions require docstrings. You can build documentation locally by installing sphinx and calling `python setup.py build_docs`. Docstrings should include the following sections:

- Description
- Parameters
- Notes
- Examples
- References

For more information see the Astropy guide to Writing Documentation.

This page is inspired by the Contributions guidelines of the Skypy project.

6.1.4 Mailing list and Slack channel

You can join the **lenstronomy** mailing list by signing up on the google groups page.

The email list is meant to provide a communication platform between users and developers. You can ask questions, and suggest new features. New releases will be announced via this mailing list.

We also have a Slack channel for the community. Please send us an email such that we can add you to the channel.

If you encounter errors or problems with **lenstronomy**, please let us know! You can open an issue, make a post on the Slack channel or write an email to the lenstronomy developers.

We are also encouraging you to reach out with feature requests, general or specific feedback and questions about use cases.

6.1.5 Credits

Current maintainers

- Simon Birrer <sibirrer@gmail.com> sibirrer
- Anowar Shajib ajshajib
- Daniel Gilman dangilman

Contact the lenstronomy developers via email if you have questions.

Contributors (alphabetic)

- Jelle Aalbers JelleAalbers
- Joel Akeret jakeret
- Adam Amara aamara
- Vikram Bhamre vikramb1
- Xuheng Ding dartoon
- Sydney Erickson smericks
- Andreas Filipp andreasfilipp
- Pierre Fleury pierrefleury
- Kevin Fusshoeller
- Aymeric Galan aymgal
- Matthew R. Gomer mattgomer
- Natalie B. Hogg nataliehogg
- Tyler Hughes
- Daniel Johnson DanJohnson98
- Felix A. Kuhn
- Zhiyuan Ma Jerry-Ma
- Felix Mayor
- Martin Millon martin-millon
- Robert Morgan rmorgan10
- Anna Nierenberg amn3142
- Brian Nord bnord
- Jackson O'Donnell jhod0
- Maverick S. H. Oh Maverick-Oh
- Giulia Pagano
- Ji Won Park jiwoncpark
- Thomas Schmidt Thomas-01
- Dominique Sluse

- Luca Teodori lucateo
- Nicolas Tessore ntessore
- Madison Ueland mueland
- Lyne Van de Vyvere LyneVdV
- Sebastian Wagner-Carena swagnercarena
- Cyril Welschen
- Ewoud Wempe ewoudwempe
- Lilan Yang ylilan
- Nan Zhang nanz6

Past development lead

The initial source code of lenstronomy was developed by Simon Birrer (sibirrer) in 2014-2018 and made public in 2018. From 2018-2022 the development of lenstronomy was hosted on Simon Birrer's repository with increased contributions from many people. The lenstronomy development moved to the project repository in 2022.

Lenstronomy logo

The lenstronomy logo was designed by Zoe Alexander zoe-blyss.

6.1.6 Published work with lenstronomy

In this section you can find the concept papers **lenstronomy** is based on and a list of science publications that made use of **lenstronomy** before 09/2022. For a more complete and current list of publications using lenstronomy we refer to the NASA/ADS query (this incudes all publications citing lenstronomy papers, which is not the same as publications making active use of the software).

Core lenstronomy methodology and software publications

- **lenstronomy: Multi-purpose gravitational lens modelling software package; Birrer & Amara 2018** *This is the lenstronomy software paper. Please cite this paper whenever you make use of lenstronomy. The paper gives a design overview and highlights some use cases.*
- **lenstronomy II: A gravitational lensing software ecosystem; Birrer et al. 2021** *JOSS software publication. Please cite this paper whenever you make use of lenstronomy.*
- **Gravitational Lens Modeling with Basis Sets; Birrer et al. 2015** *This is the method paper lenstronomy is primary based on. Please cite this paper whenever you publish results with lenstronomy by using Shapelet basis sets and/or the PSO and MCMC chain.*

Related software publications

- **A versatile tool for cluster lensing source reconstruction. I. methodology and illustration on sources in the Hubble Frontie** *reconstructing the intrinsic size-mass relation of strongly lensed sources in clusters*
- **SLITronomy: towards a fully wavelet-based strong lensing inversion technique; Galan et al. 2020** *This is the method paper presenting SLITromomy, an improved version of the SLIT algorithm fully implemented and compatible with lenstronomy.*

- **deeplenstronomy: A dataset simulation package for strong gravitational lensing;** Morgan et al. 2021a
  Software to simulating large datasets for applying deep learning to strong gravitational lensing.
- **Galaxy shapes of Light (GaLight): a 2D modeling of galaxy images;** Ding et al. 2021b *Tool to perform two-dimensional model fitting of optical and near-infrared images to characterize surface brightness distributions.*
- **LensingETC: a tool to optimize multi-filter imaging campaigns of galaxy-scale strong lensing systems;** Shajib et al. 2022b
  A Python package to select an optimal observing strategy for multi-filter imaging campaigns of strong lensing systems.
- **Using wavelets to capture deviations from smoothness in galaxy-scale strong lenses;** Galan et al. 2022
  Presenting a new software 'herculens'. The code structure and part of the modeling routines of herculens are based on lenstronomy.

6.1.7 Scientific publication before 09/2022

Measuring the Hubble constant

- **The mass-sheet degeneracy and time-delay cosmography: analysis of the strong lens RXJ1131-1231;** Birrer et al. 2016
  This paper performs a cosmographic analysis and applies the Shapelet basis set scaling to marginalize over a major lensing degeneracy.
- **H0LiCOW - IX. Cosmographic analysis of the doubly imaged quasar SDSS 1206+4332 and a new measurement of the Hu**
  This paper performs a cosmographic analysis with power-law and composite models and covers a range in complexity in the source reconstruction
- **Astrometric requirements for strong lensing time-delay cosmography;** Birrer & Treu 2019 *Derives requirements on how well the image positions of time-variable sources has to be known to perform a time-delay cosmographic measurement*
- **H0LiCOW XIII. A 2.4% measurement of H0 from lensed quasars: 5.3σ tension between early and late-Universe probes;** V
  Joint analysis of the six H0LiCOW lenses including the lenstronomy analysis of J1206
- **STRIDES: A 3.9 per cent measurement of the Hubble constant from the strongly lensed system DES J0408-5354;** Shajib e
  most precise single lensing constraint on the Hubble constant. This analysis includes two source planes and three lensing planes
- **TDCOSMO. I. An exploration of systematic uncertainties in the inference of H0 from time-delay cosmography** Millon et a
  mock lenses to test accuracy on the recovered H0 value
- **Lens modelling of the strongly lensed Type Ia supernova iPTF16geu** Moertsell et al. 2020 *Modeling of a lensed supernova to measure the Hubble constant*
- **The impact of line-of-sight structures on measuring H0 with strong lensing time-delays** Li, Becker and Dye 2020
  Point source position and time-delay modeling of quads
- **TDCOSMO III: Dark matter substructure meets dark energy – the effects of (sub)halos on strong-lensing measurements o**
  Full line-of-sight halo rendering and time-delay analysis on mock images
- **TDCOSMO IV: Hierarchical time-delay cosmography – joint inference of the Hubble constant and galaxy density profiles**
  lenstronomy.Galkin for kinematics calculation that folds in the hierarchical analysis
- **TDCOSMO V: strategies for precise and accurate measurements of the Hubble constant with strong lensing** Birrer & Tre
  lenstronomy.Galkin for kinematics calculation that folds in the hierarchical analysis for a forecast for future Hubble constant constraints
- **Large-Scale Gravitational Lens Modeling with Bayesian Neural Networks for Accurate and Precise Inference of the Hubb**
  BBN lens model inference using lenstronomy through 'baobab <https://github.com/jiwoncpark/baobab>'_ for training set generation.

- **Improved time-delay lens modelling and H0 inference with transient sources** Ding et al. 2021a
  Simulations and models with and without lensed point sources to perform a time-delay cosmography analysis.
- **Gravitational lensing H0 tension from ultralight axion galactic cores** Blum & Teodori 2021 *Investigating the detectability of a cored component with mock imaging modeling and comparison of kinematic modeling.*
- **The Hubble constant from strongly lensed supernovae with standardizable magnifications** Birrer, Dhawan, Shajib 2021
  Methodology and forecast to use standardizable magnifications to break the mass-sheet degeneracy and hierarchically measure H0.
- **AI-driven spatio-temporal engine for finding gravitationally lensed supernovae** Ramanah et al. 2021
  Simulated images with time series of lensed supernovae.
- **Systematic errors induced by the elliptical power-law model in galaxy-galaxy strong lens modeling** Cao et al. 2021
  Computing lensing quantities from mass maps.
- **TDCOSMO. VII. Boxyness/discyness in lensing galaxies** [Detectability and impact on H0 Van de Vyvere et al. 2021] *Assessment of boxy and discy lens model on the inference of H0.*
- **TDCOSMO. IX. Systematic comparison between lens modelling software programs: time delay prediction for WGD 2038**
  modeling of a time-delay lens and comprehensive analysis between two modeling codes.
- **Forecast of observing time delay of the strongly lensed quasars with Muztagh-Ata 1.93m telescope** Zhu et al. 2022a
  Using lenstronomy to reproduce a lens and simulate the observed images based on parameters fitted by other work.
- **Consequences of the lack of azimuthal freedom in the modeling of lensing galaxies** van de Vyvere et al. 2022
  Implemented a model 'ElliSLICE' to describe radial changes in ellipticities and investigating assumptiosn on azimuthal freedom in the reconstruction.

Dark Matter substructure

- **Lensing substructure quantification in RXJ1131-1231: a 2 keV lower bound on dark matter thermal relic mass;** Birrer et
  This paper quantifies the substructure content of a lens by a sub-clump scanning procedure and the application of Approximate Bayesian Computing.
- **Probing the nature of dark matter by forward modelling flux ratios in strong gravitational lenses;** Gilman et al. 2018
  - *
- **Probing dark matter structure down to 10**7 solar masses: flux ratio statistics in gravitational lenses with line-of-sight ha**
  - *
- **Double dark matter vision: twice the number of compact-source lenses with narrow-line lensing and the WFC3 Grism;** Ni
  - *
- **Warm dark matter chills out: constraints on the halo mass function and the free-streaming length of dark matter with 8 q**
  - *
- **Constraints on the mass-concentration relation of cold dark matter halos with 11 strong gravitational lenses;** Gilman et al
  - *

- **Circumventing Lens Modeling to Detect Dark Matter Substructure in Strong Lens Images with Convolutional Neural Net**
    - *
- **Dark Matter Subhalos, Strong Lensing and Machine Learning;** Varma, Fairbairn, Figueroa
    - *
- **Quantifying the Line-of-Sight Halo Contribution to the Dark Matter Convergence Power Spectrum from Strong Gravitat**
    - *
- **Detecting Subhalos in Strong Gravitational Lens Images with Image Segmentation;** Ostdiek et al. 2020a
    - *
- **Extracting the Subhalo Mass Function from Strong Lens Images with Image Segmentation;** Ostdiek et al. 2020b
    - *
- **Strong lensing signatures of self-interacting dark matter in low-mass halos;** Gilman et al. 2021a
    - *
- **Substructure Detection Reanalyzed: Dark Perturber shown to be a Line-of-Sight Halo;** Sengul et al. 2021
  modeling a line-of-sight mini-halo
- **The primordial matter power spectrum on sub-galactic scales;** Gilman et al. 2021b *rendering sub- and line-of-sight halos*
- **From Images to Dark Matter: End-To-End Inference of Substructure From Hundreds of Strong Gravitational Lenses;** W
  rendering sub- and line-of-sight halos and generating realistic training sets of images for substructure quantifications
- **Interlopers speak out: Studying the dark universe using small-scale lensing anisotropies;** Dhanasingham et al. 2022
  rendering line of sight and subhalos with pyhalo on top of lenstronomy
- **Probing Dark Matter with Strong Gravitational Lensing through an Effective Density Slope;** Senguel & Dvorkin 2022
  measuring an effective slope of a subhalo in HST data and tests on mock data from N-body simulations
- **Quantum fluctuations masquerade as halos: Bounds on ultra-light dark matter from quadruply-imaged quasars;** Laroche
  using lenstronomy for flux ratio statistics calculation with pyHalo
- **Constraining resonant dark matter self-interactions with strong gravitational lenses;** Gilman et al. 2022
  using lenstronomy for flux ratio statistics calculation with pyHalo

Lens searches

- **Strong lens systems search in the Dark Energy Survey using Convolutional Neural Networks;** Rojas et al. 2021
  simulating training sets for lens searches
- **On machine learning search for gravitational lenses;** Khachatryan 2021 *simulating training sets for lens searches*
- **DeepZipper: A Novel Deep Learning Architecture for Lensed Supernovae Identification;** Morgan et al. 2021b
  Using deeplenstronomy to simulate lensed supernovae data sets
- **Detecting gravitational lenses using machine learning: exploring interpretability and sensitivity to rare lensing configurati**
  Simulating compound lenses

- **DeepZipper II: Searching for Lensed Supernovae in Dark Energy Survey Data with Deep Learning;** Morgan et al. 2022
  Using deeplenstronomy to simulate lensed supernovae training sets
- **DeepGraviLens: a Multi-Modal Architecture for Classifying Gravitational Lensing Data;** Oreste Pinciroli Vago et al. 2022
  Using deeplenstronomy to simulate lensed supernovae training sets

Galaxy formation and evolution

- **Massive elliptical galaxies at z0.2 are well described by stars and a Navarro-Frenk-White dark matter halo;** Shajib et al. 2
  Automatized modeling of 23 SLACS lenses with dolphin, a lenstronomy wrapper
- **High-resolution imaging follow-up of doubly imaged quasars;** Shajib et al. 2020b *Modeling of doubly lensed quasars from Keck Adaptive Optics data*
- **The evolution of the size-mass relation at z=1-3 derived from the complete Hubble Frontier Fields data set;** Yang et al. 202
  reconstructing the intrinsic size-mass relation of strongly lensed sources in clusters
- **PS J1721+8842: A gravitationally lensed dual AGN system at redshift 2.37 with two radio components;** Mangat et al. 2021
  Imaging modeling of a dual lensed AGN with point sources and extended surface brightness
- **RELICS: Small Lensed z5.5 Galaxies Selected as Potential Lyman Continuum Leakers;** Neufeld et al. 2021
  size measurements of high-z lensed galaxies
- **The size-luminosity relation of lensed galaxies at z=69 in the Hubble Frontier Fields;** Yang et al. 2022a
  size measurements of high-z lensed galaxies
- **The Near Infrared Imager and Slitless Spectrograph for the James Webb Space Telescope – II. Wide Field Slitless Spectro**
  lensing calculations in cluster environments
- **Inferences on relations between distant supermassive black holes and their hosts complemented by the galaxy fundamenta**
  galaxy size measurement with quasar decomposition
- **Concordance between observations and simulations in the evolution of the mass relation between supermassive black hole**
  galaxy size measurement with quasar decomposition
- **Early results from GLASS-JWST. V: the first rest-frame optical size-luminosity relation of galaxies at z>7;** Yang et al. 202
  galaxy size measurement from JWST data with Galight/lenstronomy
- A New Polar Ring Galaxy Discovered in the COSMOS Field; Nishimura et al. 2022
- Webb's PEARLS: dust attenuation and gravitational lensing in the backlit-galaxy system VV 191; Keel et al. 2022

Automatized Lens Modeling

- **Is every strong lens model unhappy in its own way? Uniform modelling of a sample of 12 quadruply+ imaged quasars;** Sh
  This work presents a uniform modelling framework to model 13 quadruply lensed quasars in three HST bands.
- **Hierarchical Inference With Bayesian Neural Networks: An Application to Strong Gravitational Lensing;** Wagner-Caren
  This work conducts hierarchical inference of strongly-lensed systems with Bayesian neural networks.
- **A search for galaxy-scale strong gravitational lenses in the Ultraviolet Near Infrared Optical Northern Survey (UNIONS);**
  Automated modeling of best candidates of ground based data.
- **GIGA-Lens: Fast Bayesian Inference for Strong Gravitational Lens Modeling;** Gu et al. 2022
  lenstronomy-inspired GPU lensing code with PEMD+shear and Sersic modeling, and tested against lenstronomy.
- **STRIDES: Automated uniform models for 30 quadruply imaged quasars;** Schmidt et al. 2022
  Automated and uniform modeling of 30 quadruply lensed quasars.

Quasar-host galaxy decomposition

- **The mass relations between supermassive black holes and their host galaxies at 1<z<2 with HST-WFC3;** Ding et al. 2019
  Quasar host galaxy decomposition at high redshift on HST imaging and marginalization over PSF uncertainties.
- **Testing the Evolution of the Correlations between Supermassive Black Holes and their Host Galaxies using Eight Strongly**
  Quasar host galaxy decomposition with lensed quasars.
- **A local baseline of the black hole mass scaling relations for active galaxies. IV. Correlations between MBH and host galaxy**
  Detailed measurement of galaxy morphology, decomposing in spheroid, disk and bar, and central AGN
- **The Sizes of Quasar Host Galaxies with the Hyper Suprime-Cam Subaru Strategic Program;** Li et al. 2021a
  Quasar-host decomposition of 5000 SDSS quasars
- **The eROSITA Final Equatorial-Depth Survey (eFEDS): A multiwavelength view of WISE mid-infrared galaxies/active ga**
  Quasar-host decomposition of HSC imaging
- **Synchronized Co-evolution between Supermassive Black Holes and Galaxies Over the Last Seven Billion Years as Reveale**
  Quasar-host decomposition of SDSS quasars with HSC data
- **Evidence for a milli-parsec separation Supermassive Black Hole Binary with quasar microlensing;** Millon et al. 2022
  Using lenstronomy to generate the microlensed images of the accretion disk

Lensing of Gravitational Waves

- **lensingGW: a Python package for lensing of gravitational waves;** Pagano et al. 2020 *A Python package designed to handle both strong and microlensing of compact binaries and the related gravitational-wave signals.*
- **Localizing merging black holes with sub-arcsecond precision using gravitational-wave lensing;** Hannuksela et al. 2020
  solving the lens equation with lenstronomy using lensingGW
- **Lensing magnification: gravitational wave from coalescing stellar-mass binary black holes;** Shan & Hu 2020
  lensing magnification calculations
- **Identifying Type-II Strongly-Lensed Gravitational-Wave Images in Third-Generation Gravitational-Wave Detectors;** Y. W
  solving the lens equation
- **Beyond the detector horizon: Forecasting gravitational-wave strong lensing;** Renske et al. 2021
  computing image positions, time delays and magnifications for gravitational wave forecasting
- **A lensing multi-messenger channel: Combining LIGO-Virgo-Kagra lensed gravitational-wave measurements with Euclid**
  simulating Euclid-like simulations using lenstronomy and presenting a fast method to cacluate caustics for a PEMD+Shear model

Theory papers

- **Line-of-sight effects in strong lensing: putting theory into practice;** Birrer et al. 2017a *This paper formulates an effective parameterization of line-of-sight structure for strong gravitational lens modelling and applies this technique to an Einstein ring in the COSMOS field*
- **Cosmic Shear with Einstein Rings;** Birrer et al. 2018a *Forecast paper to measure cosmic shear with Einstein ring lenses. The forecast is made based on lenstronomy simulations.*
- **Unified lensing and kinematic analysis for any elliptical mass profile;** Shajib 2019 *Provides a methodology to generalize the multi-Gaussian expansion to general elliptical mass and light profiles*

- **Gravitational lensing formalism in a curved arc basis: A continuous description of observables and degeneracies from the**
  Lensing formalism with curved arc distortion formalism. Link to code repository 'here <https://github.com/sibirrer/curved_arcs>'_.

Simulation products

- **The LSST DESC DC2 Simulated Sky Survey; LSST Dark Energy Science Collaboration et al. 2020**
  Strong lensing simulations produced by SLSprinkler utilizing lenstronomy functionalities
- **The impact of mass map truncation on strong lensing simulations; Van de Vyvere et al. 2020** *Uses numerical integration to compute lensing quantities from projected mass maps from simulations.*

Large scale structure

- **Combining strong and weak lensingestimates in the Cosmos field; Kuhn et al. 2020** *inferring cosmic shear with three strong lenses in the COSMOS field*

Others

- **Predicting future astronomical events using deep learning; Singh et al.** *simulating strongly lensed galaxy merger pairs in time sequence*
- **Role of the companion lensing galaxy in the CLASS gravitational lens B1152+199; Zhang et al. 2022**
  modeling of a double lensed quasar with HST and VLBI data

6.1.8 Affiliated packages

Here is an (incomplete) list of packages and wrappers that are using lenstronomy in various ways for specific scientific applications:

- baobab: Training data generator for hierarchically modeling of strong lenses with Bayesian neural networks.
- dolphin: Automated pipeline for lens modeling based on lenstronomy.
- hierArc: Hierarchical Bayesian time-delay cosmography to infer the Hubble constant and galaxy density profiles in conjunction with lenstronomy.
- lenstruction: Versatile tool for cluster source reconstruction and local perturbative lens modeling.
- SLITronomy: Updated and improved version of the Sparse Lens Inversion Technique (SLIT), developed within the framework of lenstronomy.
- LSSTDESC SLSprinkler: The DESC SL (Strong Lensing) Sprinkler adds strongly lensed AGN and SNe to simulated catalogs and generates postage stamps for these systems.
- lensingGW: A Python package designed to handle both strong and microlensing of compact binaries and the related gravitational-wave signals.
- ovejero: Conducts hierarchical inference of strongly-lensed systems with Bayesian neural networks.
- h0rton: H0 inferences with Bayesian neural network lens modeling.
- deeplenstronomy: Tool for simulating large datasets for applying deep learning to strong gravitational lensing.
- pyHalo: Tool for rendering full substructure mass distributions for gravitational lensing simulations.
- GaLight: Tool to perform two-dimensional model fitting of optical and near-infrared images to characterize surface brightness distributions.

- paltas: Package for conducting simulation-based inference on strong gravitational lensing images.
- LensingETC: A Python package to select an optimal observing strategy for multi-filter imaging campaigns of strong lensing systems. This package simulates imaging data corresponding to provided instrument specifications and extract lens model parameter uncertainties from the simulated images.
- PSF-r: Package for Point Spread Function (PSF) reconstruction for astronomical ground- and space-based imaging data. PSF-r makes use the PSF iteration functionality of lenstronomy in a re-packaged form.

These packages come with their own documentation and examples - so check them out!

Guidelines for affiliated packages

If you have a package/wrapper/analysis pipeline that is open source and you would like to have it advertised here, please let the developers know! Before you write your own wrapper and scripts in executing lenstronomy for your purpose check out the list of existing add-on packages. Affiliated packages should not duplicate the core routines of lenstronomy and whenever possible make use of the lenstronomy modules. The packages should be maintained to keep up with the development of lenstronomy. Please also make sure the citation guidelines are presented.

6.1.9 lenstronomy package

Subpackages

lenstronomy.Analysis package

Submodules

lenstronomy.Analysis.image_reconstruction module

class MultiBandImageReconstruction(*multi_band_list*, *kwargs_model*, *kwargs_params*, *multi_band_type='multi-linear'*, *kwargs_likelihood=None*, *verbose=True*)

Bases: `object`

this class manages the output/results of a fitting process and can conveniently access image reconstruction properties in multi-band fitting. In particular, the fitting result does not come with linear inversion parameters (which may or may not be joint or different for multiple bands) and this class performs the linear inversion for the surface brightness amplitudes and stores them for each individual band to be accessible by the user.

This class is the backbone of the ModelPlot routine that provides the interface of this class with plotting and illustration routines.

__init__(*multi_band_list*, *kwargs_model*, *kwargs_params*, *multi_band_type='multi-linear'*, *kwargs_likelihood=None*, *verbose=True*)

Parameters

- **multi_band_list** – list of imaging data configuration [[kwargs_data, kwargs_psf, kwargs_numerics], [. . .]]
- **kwargs_model** – model keyword argument list
- **kwargs_params** – keyword arguments of the model parameters, same as output of FittingSequence() 'kwargs_result'

- **multi_band_type** – string, option when having multiple imaging data sets modelled simultaneously. Options are: - 'multi-linear': linear amplitudes are inferred on single data set - 'linear-joint': linear amplitudes ae jointly inferred - 'single-band': single band
- **kwargs_likelihood** – likelihood keyword arguments as supported by the Likelihood() class
- **verbose** – if True (default), computes and prints the total log-likelihood. This option can be deactivated for speedup purposes (does not run linear inversion again), and reduces the number of prints.

band_setup (*band_index=0*)

ImageModel() instance and keyword arguments of the model components to execute all the options of the ImSim core modules.

Parameters **band_index** – integer (>=0) of imaging band in order of multi_band_list input to this class

Returns ImageModel() instance and keyword arguments of the model

class ModelBand (*multi_band_list*, *kwargs_model*, *model*, *error_map*, *cov_param*, *param*, *kwargs_params*, *image_likelihood_mask_list=None*, *band_index=0*, *verbose=True*)

Bases: `object`

class to plot a single band given the full modeling results This class has it's specific role when the linear inference is performed on the joint band level and/or when only a subset of model components get used for this specific band in the modeling.

__init__ (*multi_band_list*, *kwargs_model*, *model*, *error_map*, *cov_param*, *param*, *kwargs_params*, *image_likelihood_mask_list=None*, *band_index=0*, *verbose=True*)

Parameters

- **multi_band_list** – list of imaging data configuration [[kwargs_data, kwargs_psf, kwargs_numerics], [. . .]]
- **kwargs_model** – model keyword argument list for the full multi-band modeling
- **model** – 2d numpy array of modeled image for the specified band
- **error_map** – 2d numpy array of size of the image, additional error in the pixels coming from PSF uncertainties
- **cov_param** – covariance matrix of the linear inversion
- **param** – 1d numpy array of the linear coefficients of this imaging band
- **kwargs_params** – keyword argument of keyword argument lists of the different model components selected for the imaging band, NOT including linear amplitudes (not required as being overwritten by the param list)
- **image_likelihood_mask_list** – list of 2d numpy arrays of likelihood masks (for all bands)
- **band_index** – integer of the band to be considered in this class
- **verbose** – if True (default), prints the reduced chi2 value for the current band.

image_model_class

ImageModel() class instance of the single band with only the model components applied to this band

Returns SingleBandMultiModel() instance, which inherits the ImageModel instance

kwargs_model

Returns keyword argument of keyword argument lists of the different model components selected for the imaging band, including linear amplitudes. These format matches the image_model_class() return

model

Returns model, 2d numpy array

norm_residuals

Returns normalized residuals, 2d numpy array

check_solver_error(*image*)

Parameters **image** – numpy array of modelled image from linear inversion

Returns bool, True if solver could not find a unique solution, False if solver works

lenstronomy.Analysis.kinematics_api module

class KinematicsAPI(*z_lens, z_source, kwargs_model, kwargs_aperture, kwargs_seeing, anisotropy_model, cosmo=None, lens_model_kinematics_bool=None, light_model_kinematics_bool=None, multi_observations=False, kwargs_numerics_galkin=None, analytic_kinematics=False, Hernquist_approx=False, MGE_light=False, MGE_mass=False, kwargs_mge_light=None, kwargs_mge_mass=None, sampling_number=1000, num_kin_sampling=1000, num_psf_sampling=100*)

Bases: object

this class contains routines to compute time delays, magnification ratios, line of sight velocity dispersions etc for a given lens model

__init__(*z_lens, z_source, kwargs_model, kwargs_aperture, kwargs_seeing, anisotropy_model, cosmo=None, lens_model_kinematics_bool=None, light_model_kinematics_bool=None, multi_observations=False, kwargs_numerics_galkin=None, analytic_kinematics=False, Hernquist_approx=False, MGE_light=False, MGE_mass=False, kwargs_mge_light=None, kwargs_mge_mass=None, sampling_number=1000, num_kin_sampling=1000, num_psf_sampling=100*)

Parameters

- **z_lens** – redshift of lens
- **z_source** – redshift of source
- **kwargs_model** – model keyword arguments, needs 'lens_model_list', 'lens_light_model_list'
- **kwargs_aperture** – spectroscopic aperture keyword arguments, see lenstronomy.Galkin.aperture for options
- **kwargs_seeing** – seeing condition of spectroscopic observation, corresponds to kwargs_psf in the GalKin module specified in lenstronomy.GalKin.psf
- **cosmo** – astropy.cosmology instance, if None then will be set to the default cosmology
- **lens_model_kinematics_bool** – bool list of length of the lens model. Only takes a subset of all the models as part of the kinematics computation (can be used to ignore substructure, shear etc that do not describe the main deflector potential

- **light_model_kinematics_bool** – bool list of length of the light model. Only takes a subset of all the models as part of the kinematics computation (can be used to ignore light components that do not describe the main deflector
- **multi_observations** – bool, if True uses multi-observation to predict a set of different observations with the GalkinMultiObservation() class. kwargs_aperture and kwargs_seeing require to be lists of the individual observations.
- **anisotropy_model** – type of stellar anisotropy model. See details in MamonLokasAnisotropy() class of lenstronomy.GalKin.anisotropy
- **analytic_kinematics** – boolean, if True, used the analytic JAM modeling for a power-law profile on top of a Hernquist light profile ATTENTION: This may not be accurate for your specific problem!
- **Hernquist_approx** – bool, if True, uses a Hernquist light profile matched to the half light radius of the deflector light profile to compute the kinematics
- **MGE_light** – bool, if true performs the MGE for the light distribution
- **MGE_mass** – bool, if true performs the MGE for the mass distribution
- **kwargs_numerics_galkin** – numerical settings for the integrated line-of-sight velocity dispersion
- **kwargs_mge_mass** – keyword arguments that go into the MGE decomposition routine
- **kwargs_mge_light** – keyword arguments that go into the MGE decomposition routine
- **sampling_number** – int, number of spectral rendering to compute the light weighted integrated LOS dispersion within the aperture. This keyword should be chosen high enough to result in converged results within the tolerance.
- **num_kin_sampling** – number of kinematic renderings on a total IFU
- **num_psf_sampling** – number of PSF displacements for each kinematic rendering on the IFU

galkin_settings(*kwargs_lens*, *kwargs_lens_light*, *r_eff=None*, *theta_E=None*, *gamma=None*)

Parameters

- **kwargs_lens** – lens model keyword argument list
- **kwargs_lens_light** – deflector light keyword argument list
- **r_eff** – half-light radius (optional)
- **theta_E** – Einstein radius (optional)
- **gamma** – local power-law slope at the Einstein radius (optional)

Returns Galkin() instance and mass and light profiles configured for the Galkin module

kinematic_lens_profiles(*kwargs_lens*, *MGE_fit=False*, *model_kinematics_bool=None*, *theta_E=None*, *gamma=None*, *kwargs_mge=None*, *analytic_kinematics=False*)

translates the lenstronomy lens and mass profiles into a (sub) set of profiles that are compatible with the GalKin module to compute the kinematics thereof. The requirement is that the profiles are centered at (0, 0) and that for all profile types there exists a 3d de-projected analytical representation.

Parameters

- **kwargs_lens** – lens model parameters
- **MGE_fit** – bool, if true performs the MGE for the mass distribution

- **model_kinematics_bool** – bool list of length of the lens model. Only takes a subset of all the models as part of the kinematics computation (can be used to ignore substructure, shear etc that do not describe the main deflector potential
- **theta_E** – (optional float) estimate of the Einstein radius. If present, does not numerically compute this quantity in this routine numerically
- **gamma** – local power-law slope at the Einstein radius (optional)
- **kwargs_mge** – keyword arguments that go into the MGE decomposition routine
- **analytic_kinematics** – bool, if True, solves the Jeans equation analytically for the power-law mass profile with Hernquist light profile

Returns mass_profile_list, keyword argument list

kinematic_light_profile(*kwargs_lens_light*, *r_eff=None*, *MGE_fit=False*, *model_kinematics_bool=None*, *Hernquist_approx=False*, *kwargs_mge=None*, *analytic_kinematics=False*)

setting up of the light profile to compute the kinematics in the GalKin module. The requirement is that the profiles are centered at (0, 0) and that for all profile types there exists a 3d de-projected analytical representation.

Parameters

- **kwargs_lens_light** – deflector light model keyword argument list
- **r_eff** – (optional float, else=None) Pre-calculated projected half-light radius of the deflector profile. If not provided, numerical calculation is done in this routine if required.
- **MGE_fit** – boolean, if True performs a Multi-Gaussian expansion of the radial light profile and returns this solution.
- **model_kinematics_bool** – list of booleans to indicate a subset of light profiles to be part of the physical deflector light.
- **Hernquist_approx** – boolean, if True replaces the actual light profile(s) with a Hernquist model with matched half-light radius.
- **kwargs_mge** – keyword arguments that go into the MGE decomposition routine
- **analytic_kinematics** – bool, if True, solves the Jeans equation analytically for the power-law mass profile with Hernquist light profile and adjust the settings accordingly

Returns deflector type list, keyword arguments list

kinematics_modeling_settings(*anisotropy_model*, *kwargs_numerics_galkin*, *analytic_kinematics=False*, *Hernquist_approx=False*, *MGE_light=False*, *MGE_mass=False*, *kwargs_mge_light=None*, *kwargs_mge_mass=None*, *sampling_number=1000*, *num_kin_sampling=1000*, *num_psf_sampling=100*)

Parameters

- **anisotropy_model** – type of stellar anisotropy model. See details in MamonLokasAnisotropy() class of lenstronomy.GalKin.anisotropy
- **analytic_kinematics** – boolean, if True, used the analytic JAM modeling for a power-law profile on top of a Hernquist light profile ATTENTION: This may not be accurate for your specific problem!
- **Hernquist_approx** – bool, if True, uses a Hernquist light profile matched to the half light radius of the deflector light profile to compute the kinematics

- **MGE_light** – bool, if true performs the MGE for the light distribution
- **MGE_mass** – bool, if true performs the MGE for the mass distribution
- **kwargs_numerics_galkin** – numerical settings for the integrated line-of-sight velocity dispersion
- **kwargs_mge_mass** – keyword arguments that go into the MGE decomposition routine
- **kwargs_mge_light** – keyword arguments that go into the MGE decomposition routine
- **sampling_number** – number of spectral rendering on a single slit
- **num_kin_sampling** – number of kinematic renderings on a total IFU
- **num_psf_sampling** – number of PSF displacements for each kinematic rendering on the IFU

Returns

static transform_kappa_ext(*sigma_v*, *kappa_ext=0*)

Parameters

- **sigma_v** – velocity dispersion estimate of the lensing deflector without considering external convergence
- **kappa_ext** – external convergence to be used in the mass-sheet degeneracy

Returns transformed velocity dispersion

velocity_dispersion(*kwargs_lens*, *kwargs_lens_light*, *kwargs_anisotropy*, *r_eff=None*, *theta_E=None*, *gamma=None*, *kappa_ext=0*)

API for both, analytic and numerical JAM to compute the velocity dispersion [km/s] This routine uses the galkin_setting() routine for the Galkin configurations (see there what options and input is relevant.

Parameters

- **kwargs_lens** – lens model keyword arguments
- **kwargs_lens_light** – lens light model keyword arguments
- **kwargs_anisotropy** – stellar anisotropy keyword arguments
- **r_eff** – projected half-light radius of the stellar light associated with the deflector galaxy, optional, if set to None will be computed in this function with default settings that may not be accurate.
- **theta_E** – Einstein radius (optional)
- **gamma** – power-law slope (optional)
- **kappa_ext** – external convergence (optional)

Returns velocity dispersion [km/s]

velocity_dispersion_analytical(*theta_E*, *gamma*, *r_eff*, *r_ani*, *kappa_ext=0*)

computes the LOS velocity dispersion of the lens within a slit of size R_slit x dR_slit and seeing psf_fwhm. The assumptions are a Hernquist light profile and the spherical power-law lens model at the first position and an Osipkov and Merritt (‘OM’) stellar anisotropy distribution.

Further information can be found in the AnalyticKinematics() class.

Parameters

- **theta_E** – Einstein radius
- **gamma** – power-low slope of the mass profile (=2 corresponds to isothermal)

- **r_ani** – anisotropy radius in units of angles
- **r_eff** – projected half-light radius
- **kappa_ext** – external convergence not accounted in the lens models

Returns velocity dispersion in units [km/s]

velocity_dispersion_map(*kwargs_lens*, *kwargs_lens_light*, *kwargs_anisotropy*, *r_eff=None*, *theta_E=None*, *gamma=None*, *kappa_ext=0*)

API for both, analytic and numerical JAM to compute the velocity dispersion map with IFU data [km/s]

Parameters

- **kwargs_lens** – lens model keyword arguments
- **kwargs_lens_light** – lens light model keyword arguments
- **kwargs_anisotropy** – stellar anisotropy keyword arguments
- **r_eff** – projected half-light radius of the stellar light associated with the deflector galaxy, optional, if set to None will be computed in this function with default settings that may not be accurate.
- **theta_E** – circularized Einstein radius, optional, if not provided will either be computed in this function with default settings or not required
- **gamma** – power-law slope at the Einstein radius, optional
- **kappa_ext** – external convergence

Returns velocity dispersion [km/s]

lenstronomy.Analysis.lens_profile module

class LensProfileAnalysis(*lens_model*)

Bases: `object`

class with analysis routines to compute derived properties of the lens model

__init__(*lens_model*)

Parameters **lens_model** – LensModel instance

convergence_peak(*kwargs_lens*, *model_bool_list=None*, *grid_num=200*, *grid_spacing=0.01*, *center_x_init=0*, *center_y_init=0*)

computes the maximal convergence position on a grid and returns its coordinate

Parameters

- **kwargs_lens** – lens model keyword argument list
- **model_bool_list** – bool list (optional) to include certain models or not

Returns center_x, center_y

effective_einstein_radius(*kwargs_lens*, *center_x=None*, *center_y=None*, *model_bool_list=None*, *grid_num=200*, *grid_spacing=0.05*, *get_precision=False*, *verbose=True*)

computes the radius with mean convergence=1

Parameters

- **kwargs_lens** – list of lens model keyword arguments

- **center_x** – position of the center (if not set, is attempting to find it from the parameters kwargs_lens)
- **center_y** – position of the center (if not set, is attempting to find it from the parameters kwargs_lens)
- **model_bool_list** – list of booleans indicating the addition (=True) of a model component in computing the Einstein radius
- **grid_num** – integer, number of grid points to numerically evaluate the convergence and estimate the Einstein radius
- **grid_spacing** – spacing in angular units of the grid
- **get_precision** – If *True*, return the precision of estimated Einstein radius
- **verbose** – boolean, if True prints warning if indication of insufficient result

Returns estimate of the Einstein radius

local_lensing_effect(*kwargs_lens*, *ra_pos=0*, *dec_pos=0*, *model_list_bool=None*)
computes deflection, shear and convergence at (ra_pos,dec_pos) for those part of the lens model not included in the main deflector.

Parameters

- **kwargs_lens** – lens model keyword argument list
- **ra_pos** – RA position where to compute the external effect
- **dec_pos** – DEC position where to compute the external effect
- **model_list_bool** – boolean list indicating which models effect to be added to the estimate

Returns alpha_x, alpha_y, kappa, shear1, shear2

mass_fraction_within_radius(*kwargs_lens*, *center_x*, *center_y*, *theta_E*, *numPix=100*)
computes the mean convergence of all the different lens model components within a spherical aperture

Parameters

- **kwargs_lens** – lens model keyword argument list
- **center_x** – center of the aperture
- **center_y** – center of the aperture
- **theta_E** – radius of aperture

Returns list of average convergences for all the model components

mst_invariant_differential(*kwargs_lens*, *radius*, *center_x=None*, *center_y=None*, *model_list_bool=None*, *num_points=10*)
Average of the radial stretch differential in radial direction, divided by the radial stretch factor.

$$\xi = \frac{\partial \lambda_{\rm rad}}{\partial r} \frac{1}{\lambda_{\rm rad}}$$

This quantity is invariant under the MST. The specific definition is provided by Birrer 2021. Equivalent (proportional) definitions are provided by e.g. Kochanek 2020, Sonnenfeld 2018.

Parameters

- **kwargs_lens** – lens model keyword argument list
- **radius** – radius from the center where to compute the MST invariant differential

- **center_x** – center position
- **center_y** – center position
- **model_list_bool** – indicate which part of the model to consider
- **num_points** – number of estimates around the radius

Returns xi

multi_gaussian_lens (*kwargs_lens*, *center_x=None*, *center_y=None*, *model_bool_list=None*, *n_comp=20*)

multi-gaussian lens model in convergence space

Parameters

- **kwargs_lens** –
- **n_comp** –

Returns

profile_slope (*kwargs_lens*, *radius*, *center_x=None*, *center_y=None*, *model_list_bool=None*, *num_points=10*)

computes the logarithmic power-law slope of a profile. ATTENTION: this is not an observable!

Parameters

- **kwargs_lens** – lens model keyword argument list
- **radius** – radius from the center where to compute the logarithmic slope (angular units
- **center_x** – center of profile from where to compute the slope
- **center_y** – center of profile from where to compute the slope
- **model_list_bool** – bool list, indicate which part of the model to consider
- **num_points** – number of estimates around the Einstein radius

Returns logarithmic power-law slope

radial_lens_profile (*r_list*, *kwargs_lens*, *center_x=None*, *center_y=None*, *model_bool_list=None*)

Parameters

- **r_list** – list of radii to compute the spherically averaged lens light profile
- **center_x** – center of the profile
- **center_y** – center of the profile
- **kwargs_lens** – lens parameter keyword argument list
- **model_bool_list** – bool list or None, indicating which profiles to sum over

Returns flux amplitudes at r_list radii azimuthally averaged

lenstronomy.Analysis.light2mass module

light2mass_interpol (*lens_light_model_list*, *kwargs_lens_light*, *numPix=100*, *deltaPix=0.05*, *subgrid_res=5*, *center_x=0*, *center_y=0*)

takes a lens light model and turns it numerically in a lens model (with all lensmodel quantities computed on a grid). Then provides an interpolated grid for the quantities.

Parameters

- **kwargs_lens_light** – lens light keyword argument list
- **numPix** – number of pixels per axis for the return interpolation
- **deltaPix** – interpolation/pixel size
- **center_x** – center of the grid
- **center_y** – center of the grid
- **subgrid_res** – subgrid for the numerical integrals

Returns keyword arguments for 'INTERPOL' lens model

lenstronomy.Analysis.light_profile module

class LightProfileAnalysis(*light_model*)

Bases: object

class with analysis routines to compute derived properties of the lens model

__init__(*light_model*)

Parameters **light_model** – LightModel instance

ellipticity(*kwargs_light*, *grid_spacing*, *grid_num*, *center_x=None*, *center_y=None*, *model_bool_list=None*)

make sure that the window covers all the light, otherwise the moments may give a too low answers.

Parameters

- **kwargs_light** – keyword argument list of profiles
- **center_x** – center of profile, if None takes it from the first profile in kwargs_light
- **center_y** – center of profile, if None takes it from the first profile in kwargs_light
- **model_bool_list** – list of booleans to select subsets of the profile
- **grid_spacing** – grid spacing over which the moments are computed
- **grid_num** – grid size over which the moments are computed

Returns eccentricities e1, e2

flux_components(*kwargs_light*, *grid_num=400*, *grid_spacing=0.01*)

computes the total flux in each component of the model

Parameters

- **kwargs_light** –
- **grid_num** –
- **grid_spacing** –

Returns

half_light_radius(*kwargs_light*, *grid_spacing*, *grid_num*, *center_x=None*, *center_y=None*, *model_bool_list=None*)

computes numerically the half-light-radius of the deflector light and the total photon flux

Parameters

- **kwargs_light** – keyword argument list of profiles
- **center_x** – center of profile, if None takes it from the first profile in kwargs_light

- **center_y** – center of profile, if None takes it from the first profile in kwargs_light
- **model_bool_list** – list of booleans to select subsets of the profile
- **grid_spacing** – grid spacing over which the moments are computed
- **grid_num** – grid size over which the moments are computed

Returns half-light radius

multi_gaussian_decomposition(*kwargs_light*, *model_bool_list=None*, *n_comp=20*, *center_x=None*, *center_y=None*, *r_h=None*, *grid_spacing=0.02*, *grid_num=200*)

multi-gaussian decomposition of the lens light profile (in 1-dimension)

Parameters

- **kwargs_light** – keyword argument list of profiles
- **center_x** – center of profile, if None takes it from the first profile in kwargs_light
- **center_y** – center of profile, if None takes it from the first profile in kwargs_light
- **model_bool_list** – list of booleans to select subsets of the profile
- **grid_spacing** – grid spacing over which the moments are computed for the half-light radius
- **grid_num** – grid size over which the moments are computed
- **n_comp** – maximum number of Gaussian's in the MGE
- **r_h** – float, half light radius to be used for MGE (optional, otherwise using a numerical grid)

Returns amplitudes, sigmas, center_x, center_y

multi_gaussian_decomposition_ellipse(*kwargs_light*, *model_bool_list=None*, *center_x=None*, *center_y=None*, *grid_num=100*, *grid_spacing=0.05*, *n_comp=20*)

MGE with ellipticity estimate. Attention: numerical grid settings for ellipticity estimate and radial MGE may not necessarily be the same!

Parameters

- **kwargs_light** – keyword argument list of profiles
- **center_x** – center of profile, if None takes it from the first profile in kwargs_light
- **center_y** – center of profile, if None takes it from the first profile in kwargs_light
- **model_bool_list** – list of booleans to select subsets of the profile
- **grid_spacing** – grid spacing over which the moments are computed
- **grid_num** – grid size over which the moments are computed
- **n_comp** – maximum number of Gaussians in the MGE

Returns keyword arguments of the elliptical multi Gaussian profile in lenstronomy conventions

radial_light_profile(*r_list*, *kwargs_light*, *center_x=None*, *center_y=None*, *model_bool_list=None*)

Parameters

- **r_list** – list of radii to compute the spherically averaged lens light profile
- **center_x** – center of the profile

- **center_y** – center of the profile
- **kwargs_light** – lens light parameter keyword argument list
- **model_bool_list** – bool list or None, indicating which profiles to sum over

Returns flux amplitudes at r_list radii spherically averaged

lenstronomy.Analysis.multi_patch_reconstruction module

class MultiPatchReconstruction(*multi_band_list*, *kwargs_model*, *kwargs_params*, *multi_band_type='joint-linear'*, *kwargs_likelihood=None*, *kwargs_pixel_grid=None*, *verbose=True*)

Bases: *lenstronomy.Analysis.image_reconstruction.MultiBandImageReconstruction*

this class illustrates the model of disconnected multi-patch modeling with 'joint-linear' option in one single array.

__init__(*multi_band_list*, *kwargs_model*, *kwargs_params*, *multi_band_type='joint-linear'*, *kwargs_likelihood=None*, *kwargs_pixel_grid=None*, *verbose=True*)

Parameters

- **multi_band_list** – list of imaging data configuration [[kwargs_data, kwargs_psf, kwargs_numerics], [. . .]]
- **kwargs_model** – model keyword argument list
- **kwargs_params** – keyword arguments of the model parameters, same as output of FittingSequence() 'kwargs_result'
- **multi_band_type** – string, option when having multiple imaging data sets modelled simultaneously. Options are: - 'multi-linear': linear amplitudes are inferred on single data set - 'linear-joint': linear amplitudes ae jointly inferred - 'single-band': single band
- **kwargs_likelihood** – likelihood keyword arguments as supported by the Likelihood() class
- **kwargs_pixel_grid** – keyword argument of PixelGrid() class. This is optional and overwrites a minimal grid Attention for consistent pixel grid definitions!
- **verbose** – if True (default), computes and prints the total log-likelihood. This can deactivated for speedup purposes (does not run linear inversion again), and reduces the number of prints.

image_joint()

patch together the individual patches of data and models

Returns image_joint, model_joint, norm_residuals_joint

lens_model_joint()

patch together the individual patches of the lens model (can be discontinues)

Returns 2d numpy arrays of kappa_joint, magnification_joint, alpha_x_joint, alpha_y_joint

pixel_grid_joint

Returns PixelGrid() class instance covering the entire window of the sky including all individual patches

source(*num_pix*, *delta_pix*, *center=None*)

source in the same coordinate system as the image

Parameters

- **num_pix** – number of pixels per axes
- **delta_pix** – pixel size
- **center** – list with two entries [center_x, center_y] (optional)

Returns 2d surface brightness grid of the reconstructed source and PixelGrid() instance of source grid

lenstronomy.Analysis.td_cosmography module

class TDCosmography(*z_lens, z_source, kwargs_model, cosmo_fiducial=None, lens_model_kinematics_bool=None, light_model_kinematics_bool=None, kwargs_seeing=None, kwargs_aperture=None, anisotropy_model=None, **kwargs_kin_api*)

Bases: `lenstronomy.Analysis.kinematics_api.KinematicsAPI`

class equipped to perform a cosmographic analysis from a lens model with added measurements of time delays and kinematics. This class does not require any cosmological knowledge and can return angular diameter distance estimates self-consistently integrating the kinematics routines and time delay estimates in the lens modeling. This description follows Birrer et al. 2016, 2019.

__init__(*z_lens, z_source, kwargs_model, cosmo_fiducial=None, lens_model_kinematics_bool=None, light_model_kinematics_bool=None, kwargs_seeing=None, kwargs_aperture=None, anisotropy_model=None, **kwargs_kin_api*)

Parameters

- **z_lens** – redshift of deflector
- **z_source** – redshift of source
- **kwargs_model** – model configurations (according to FittingSequence)
- **cosmo_fiducial** – fiducial cosmology used to compute angular diameter distances where required
- **lens_model_kinematics_bool** – (optional) bool list, corresponding to lens models being included into the kinematics modeling
- **light_model_kinematics_bool** – (optional) bool list, corresponding to lens light models being included into the kinematics modeling
- **kwargs_seeing** – seeing conditions (see observation class in Galkin)
- **kwargs_aperture** – aperture keyword arguments (see aperture class in Galkin)
- **anisotropy_model** – string, anisotropy model type
- **kwargs_kin_api** – additional keyword arguments for KinematicsAPI class instance

ddt_dd_from_time_delay_and_kinematics(*d_fermat_model, dt_measured, sigma_v_measured, J, kappa_s=0, kappa_ds=0, kappa_d=0*)

Parameters

- **d_fermat_model** – relative Fermat potential in units arcsec^2
- **dt_measured** – measured relative time delay [days]
- **sigma_v_measured** – 1-sigma Gaussian uncertainty in the measured velocity dispersion

- **J** – modeled dimensionless kinematic estimate
- **kappa_s** – LOS convergence from observer to source
- **kappa_ds** – LOS convergence from deflector to source
- **kappa_d** – LOS convergence from observer to deflector

Returns D_dt, D_d

static ddt_from_time_delay(*d_fermat_model*, *dt_measured*, *kappa_s=0*, *kappa_ds=0*, *kappa_d=0*)

Time-delay distance in units of Mpc from the modeled Fermat potential and measured time delay from an image pair.

Parameters

- **d_fermat_model** – relative Fermat potential between two images from the same source in units arcsec^2
- **dt_measured** – measured time delay between the same image pair in units of days
- **kappa_s** – external convergence from observer to source
- **kappa_ds** – external convergence from lens to source
- **kappa_d** – external convergence form observer to lens

Returns D_dt, time-delay distance

static ds_dds_from_kinematics(*sigma_v*, *J*, *kappa_s=0*, *kappa_ds=0*)

computes the estimate of the ratio of angular diameter distances Ds/Dds from the kinematic estimate of the lens and the measured dispersion.

Parameters

- **sigma_v** – velocity dispersion [km/s]
- **J** – dimensionless kinematic constraint (see Birrer et al. 2016, 2019)

Returns Ds/Dds

fermat_potential(*kwargs_lens*, *kwargs_ps*, *original_ps_position=False*)

Fermat potential (negative sign means earlier arrival time)

Parameters

- **kwargs_lens** – lens model keyword argument list
- **kwargs_ps** – point source keyword argument list
- **original_ps_position** – boolean (only applies when first point source model is of type 'LENSED_POSITION'), uses the image positions in the model parameters and does not re-compute images (which might be differently ordered) in case of the lens equation solver

Returns Fermat potential of all the image positions in the first point source list entry

time_delays(*kwargs_lens*, *kwargs_ps*, *kappa_ext=0*, *original_ps_position=False*)

predicts the time delays of the image positions given the fiducial cosmology relative to a straight line without lensing. Negative values correspond to images arriving earlier, and positive signs correspond to images arriving later.

Parameters

- **kwargs_lens** – lens model parameters
- **kwargs_ps** – point source parameters

- **kappa_ext** – external convergence (optional)
- **original_ps_position** – boolean (only applies when first point source model is of type 'LENSED_POSITION'), uses the image positions in the model parameters and does not re-compute images (which might be differently ordered) in case of the lens equation solver

Returns time delays at image positions for the fixed cosmology in units of days

velocity_dispersion_dimension_less(*kwargs_lens*, *kwargs_lens_light*, *kwargs_anisotropy*, *r_eff=None*, *theta_E=None*, *gamma=None*)

sigma**2 = Dd/Dds * c**2 * J(kwargs_lens, kwargs_light, anisotropy) (Equation 4.11 in Birrer et al. 2016 or Equation 6 in Birrer et al. 2019) J() is a dimensionless and cosmological independent quantity only depending on angular units. This function returns J given the lens and light parameters and the anisotropy choice without an external mass sheet correction.

Parameters

- **kwargs_lens** – lens model keyword arguments
- **kwargs_lens_light** – lens light model keyword arguments
- **kwargs_anisotropy** – stellar anisotropy keyword arguments
- **r_eff** – projected half-light radius of the stellar light associated with the deflector galaxy, optional, if set to None will be computed in this function with default settings that may not be accurate.
- **theta_E** – pre-computed Einstein radius (optional)
- **gamma** – pre-computed power-law slope of mass profile

Returns dimensionless velocity dispersion (see e.g. Birrer et al. 2016, 2019)

velocity_dispersion_map_dimension_less(*kwargs_lens*, *kwargs_lens_light*, *kwargs_anisotropy*, *r_eff=None*, *theta_E=None*, *gamma=None*)

sigma**2 = Dd/Dds * c**2 * J(kwargs_lens, kwargs_light, anisotropy) (Equation 4.11 in Birrer et al. 2016 or Equation 6 in Birrer et al. 2019) J() is a dimensionless and cosmological independent quantity only depending on angular units. This function returns J given the lens and light parameters and the anisotropy choice without an external mass sheet correction. This routine computes the IFU map of the kinematic quantities.

Parameters

- **kwargs_lens** – lens model keyword arguments
- **kwargs_lens_light** – lens light model keyword arguments
- **kwargs_anisotropy** – stellar anisotropy keyword arguments
- **r_eff** – projected half-light radius of the stellar light associated with the deflector galaxy, optional, if set to None will be computed in this function with default settings that may not be accurate.

Returns dimensionless velocity dispersion (see e.g. Birrer et al. 2016, 2019)

Module contents

lenstronomy.Conf package

Submodules

lenstronomy.Conf.config_loader module

conventions_conf()

Returns convention keyword arguments

numba_conf()

Returns keyword arguments of numba configurations from yaml file

Module contents

lenstronomy.Cosmo package

Submodules

lenstronomy.Cosmo.background module

class Background(*cosmo=None*, *interp=False*, ***kwargs_interp*)

Bases: `object`

class to compute cosmological distances

T_xy(*z_observer*, *z_source*)

Parameters

- **z_observer** – observer
- **z_source** – source

Returns transverse comoving distance in units of Mpc

__init__(*cosmo=None*, *interp=False*, ***kwargs_interp*)

Parameters

- **cosmo** – instance of astropy.cosmology
- **interp** – boolean, if True, uses interpolated cosmology to evaluate specific redshifts
- **kwargs_interp** – keyword arguments of CosmoInterp specifying the interpolation interval and maximum redshift

Returns Background class with instance of astropy.cosmology

static a_z(*z*)

returns scale factor (a_0 = 1) for given redshift

Parameters z – redshift

Returns scale factor

d_xy(*z_observer*, *z_source*)

Parameters

- **z_observer** – observer redshift
- **z_source** – source redshift

Returns angular diameter distance in units of Mpc

ddt (*z_lens*, *z_source*)
time-delay distance

Parameters

- **z_lens** – redshift of lens
- **z_source** – redshift of source

Returns time-delay distance in units of proper Mpc

rho_crit
critical density

Returns value in M_sol/Mpc^3

lenstronomy.Cosmo.cosmo_solver module

cosmo2angular_diameter_distances (*H_0*, *omega_m*, *z_lens*, *z_source*)

Parameters

- **H_0** – Hubble constant [km/s/Mpc]
- **omega_m** – dimensionless matter density at z=0
- **z_lens** – deflector redshift
- **z_source** – source redshift

Returns angular diameter distances Dd and Ds/Dds

ddt2h0 (*ddt*, *z_lens*, *z_source*, *cosmo*)
converts time-delay distance to H0 for a given expansion history

Parameters

- **ddt** – time-delay distance in Mpc
- **z_lens** – deflector redshift
- **z_source** – source redshift
- **cosmo** – astropy.cosmology class instance

Returns h0 value which matches the cosmology class effectively replacing the h0 value used in the creation of this class

class SolverFlatLCDM (*z_d*, *z_s*)
Bases: `object`

class to solve multidimensional non-linear equations to determine the cosmological parameters H0 and omega_m given the angular diameter distance relations

F (*x*, *Dd*, *Ds_Dds*)

Parameters **x** – array of parameters (H_0, omega_m)

Returns

__init__ (*z_d*, *z_s*)
Initialize self. See help(type(self)) for accurate signature.

solve (*init*, *dd*, *ds_dds*)

class InvertCosmo(*z_d*, *z_s*, *H0_range=None*, *omega_m_range=None*)
Bases: object

class to do an interpolation and call the inverse of this interpolation to get H_0 and omega_m

__init__(*z_d*, *z_s*, *H0_range=None*, *omega_m_range=None*)
Initialize self. See help(type(self)) for accurate signature.

get_cosmo(*Dd*, *Ds_Dds*)
return the values of H0 and omega_m computed with an interpolation

Parameters

- **Dd** – flat
- **Ds_Dds** – float

Returns

lenstronomy.Cosmo.kde_likelihood module

class KDELikelihood(*D_d_sample*, *D_delta_t_sample*, *kde_type='scipy_gaussian'*, *bandwidth=1*)
Bases: object

class that samples the cosmographic likelihood given a distribution of points in the 2-dimensional distribution of D_d and D_delta_t

__init__(*D_d_sample*, *D_delta_t_sample*, *kde_type='scipy_gaussian'*, *bandwidth=1*)

Parameters

- **D_d_sample** – 1-d numpy array of angular diameter distances to the lens plane
- **D_delta_t_sample** – 1-d numpy array of time-delay distances
- **kde_type** (*string*) – The kernel to use. Valid kernels are 'scipy_gaussian' or ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine'] Default is 'gaussian'.
- **bandwidth** – width of kernel (in same units as the angular diameter quantities)

logLikelihood(*D_d*, *D_delta_t*)
likelihood of the data (represented in the distribution of this class) given a model with predicted angular diameter distances.

Parameters

- **D_d** – model predicted angular diameter distance
- **D_delta_t** – model predicted time-delay distance

Returns loglikelihood (log of KDE value)

lenstronomy.Cosmo.lcdm module

class LCDM(*z_lens*, *z_source*, *flat=True*)
Bases: object

Flat LCDM cosmology background with free Hubble parameter and Omega_m at fixed lens redshift configuration

D_d(*H_0*, *Om0*, *Ode0=None*)
angular diameter to deflector

Parameters

- **H_0** – Hubble parameter [km/s/Mpc]
- **Om0** – normalized matter density at present time

Returns float [Mpc]

D_ds (*H_0*, *Om0*, *Ode0=None*)
angular diameter from deflector to source

Parameters

- **H_0** – Hubble parameter [km/s/Mpc]
- **Om0** – normalized matter density at present time

Returns float [Mpc]

D_dt (*H_0*, *Om0*, *Ode0=None*)
time-delay distance

Parameters

- **H_0** – Hubble parameter [km/s/Mpc]
- **Om0** – normalized matter density at present time

Returns float [Mpc]

D_s (*H_0*, *Om0*, *Ode0=None*)
angular diameter to source

Parameters

- **H_0** – Hubble parameter [km/s/Mpc]
- **Om0** – normalized matter density at present time

Returns float [Mpc]

__init__ (*z_lens*, *z_source*, *flat=True*)

Parameters

- **z_lens** – redshift of lens
- **z_source** – redshift of source
- **flat** – bool, if True, flat universe is assumed

lenstronomy.Cosmo.lens_cosmo module

class LensCosmo (*z_lens*, *z_source*, *cosmo=None*)
Bases: `object`

class to manage the physical units and distances present in a single plane lens with fixed input cosmology

__init__ (*z_lens*, *z_source*, *cosmo=None*)

Parameters

- **z_lens** – redshift of lens
- **z_source** – redshift of source
- **cosmo** – astropy.cosmology instance

arcsec2phys_lens(*arcsec*)
convert angular to physical quantities for lens plane

Parameters **arcsec** – angular size at lens plane [arcsec]

Returns physical size at lens plane [Mpc]

arcsec2phys_source(*arcsec*)
convert angular to physical quantities for source plane

Parameters **arcsec** – angular size at source plane [arcsec]

Returns physical size at source plane [Mpc]

dd

Returns angular diameter distance to the deflector [Mpc]

dds

Returns angular diameter distance from deflector to source [Mpc]

ddt

Returns time delay distance [Mpc]

ds

Returns angular diameter distance to the source [Mpc]

h

kappa2proj_mass(*kappa*)
convert convergence to projected mass M_sun/Mpc^2

Parameters **kappa** – lensing convergence

Returns projected mass [M_sun/Mpc^2]

mass_in_coin(*theta_E*)

Parameters **theta_E** – Einstein radius [arcsec]

Returns mass in coin calculated in mean density of the universe

mass_in_theta_E(*theta_E*)
mass within Einstein radius (area * epsilon crit) [M_sun]

Parameters **theta_E** – Einstein radius [arcsec]

Returns mass within Einstein radius [M_sun]

nfwParam_physical(*M*, *c*)
returns the NFW parameters in physical units

Parameters

- **M** – physical mass in M_sun
- **c** – concentration

Returns rho0 [Msun/Mpc^3], Rs [Mpc], r200 [Mpc]

nfw_M_theta_r200(*M*)
returns r200 radius in angular units of arc seconds on the sky

Parameters **M** – physical mass in M_sun

Returns angle (in arc seconds) of the r200 radius

nfw_angle2physical (*Rs_angle*, *alpha_Rs*)
converts the angular parameters into the physical ones for an NFW profile

Parameters

- **alpha_Rs** – observed bending angle at the scale radius in units of arcsec
- **Rs_angle** – scale radius in units of arcsec

Returns rho0 [Msun/Mpc^3], Rs [Mpc], c, r200 [Mpc], M200 [Msun]

nfw_physical2angle (*M*, *c*)
converts the physical mass and concentration parameter of an NFW profile into the lensing quantities

Parameters

- **M** – mass enclosed 200 rho_crit in units of M_sun (physical units, meaning no little h)
- **c** – NFW concentration parameter (r200/r_s)

Returns Rs_angle (angle at scale radius) (in units of arcsec), alpha_Rs (observed bending angle at the scale radius

phys2arcsec_lens (*phys*)
convert physical Mpc into arc seconds

Parameters **phys** – physical distance [Mpc]

Returns angular diameter [arcsec]

sersic_k_eff2m_star (*k_eff*, *R_sersic*, *n_sersic*)
translates convergence at half-light radius to total integrated physical stellar mass for a Sersic profile

Parameters

- **k_eff** – lensing convergence at half-light radius
- **R_sersic** – half-light radius in arc seconds
- **n_sersic** – Sersic index

Returns stellar mass in physical Msun

sersic_m_star2k_eff (*m_star*, *R_sersic*, *n_sersic*)
translates a total stellar mass into 'k_eff', the convergence at 'R_sersic' (effective radius or half-light radius) for a Sersic profile

Parameters

- **m_star** – total stellar mass in physical Msun
- **R_sersic** – half-light radius in arc seconds
- **n_sersic** – Sersic index

Returns k_eff

sigma_crit
returns the critical projected lensing mass density in units of M_sun/Mpc^2

Returns critical projected lensing mass density

sigma_crit_angle
returns the critical surface density in units of M_sun/arcsec^2 (in physical solar mass units) when provided a physical mass per physical Mpc^2

Returns critical projected mass density

sis_sigma_v2theta_E (*v_sigma*)
converts the velocity dispersion into an Einstein radius for a SIS profile

Parameters **v_sigma** – velocity dispersion (km/s)

Returns theta_E (arcsec)

sis_theta_E2sigma_v (*theta_E*)
converts the lensing Einstein radius into a physical velocity dispersion

Parameters **theta_E** – Einstein radius (in arcsec)

Returns velocity dispersion in units (km/s)

time_delay2fermat_pot (*dt*)

Parameters **dt** – time delay in units of days

Returns Fermat potential in units arcsec**2 for a given cosmology

time_delay_units (*fermat_pot*, *kappa_ext=0*)

Parameters

- **fermat_pot** – in units of arcsec^2 (e.g. Fermat potential)
- **kappa_ext** – unit-less external shear not accounted for in the Fermat potential

Returns time delay in days

uldm_angular2phys (*kappa_0*, *theta_c*)
converts the anguar parameters entering the LensModel Uldm() (Ultra Light Dark Matter) class in physical masses, i.e. the total soliton mass and the mass of the particle

Parameters

- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcseconds)

Returns m_eV_log10, M_sol_log10, the log10 of the masses, m in eV and M in M_sun

uldm_mphys2angular (*m_log10*, *M_log10*)
converts physical ULDM mass in the ones, in angular units, that enter the LensModel Uldm() class

Parameters

- **m_log10** – exponent of ULDM mass in eV
- **M_log10** – exponent of soliton mass in M_sun

Returns kappa_0, theta_c, the central convergence and core radius (in arcseconds)

lenstronomy.Cosmo.nfw_param module

class NFWParam (*cosmo=None*)
Bases: `object`

class which contains a halo model parameters dependent on cosmology for NFW profile All distances are given in physical units. Mass definitions are relative to 200 crit including redshift evolution. The redshift evolution is cosmology dependent (dark energy). The H0 dependence is propagated into the input and return units.

static M200 (*rs*, *rho0*, *c*)
Calculation of the mass enclosed r_200 for NFW profile defined as

$$M_{200} = 4\pi\rho_0^3 * (\log(1+c) - c/(1+c)))$$

Parameters

- **rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density) in units mass/[distance unit of rs]^3
- **c** (*float [4,40]*) – concentration

Returns M(R_200) mass in units of rho0 * rs^3

M_r200 (*r200*, *z*)

Parameters

- **r200** – r200 in physical Mpc/h
- **z** – redshift

Returns M200 in M_sun/h

__init__ (*cosmo=None*)

Parameters **cosmo** – astropy.cosmology instance

static c_M_z (*M*, *z*)

fitting function of http://moriond.in2p3.fr/J08/proceedings/duffy.pdf for the mass and redshift dependence of the concentration parameter

Parameters

- **M** (*float or numpy array*) – halo mass in M_sun/h
- **z** (*float >0*) – redshift

Returns concentration parameter as float

c_rho0 (*rho0*, *z*)

computes the concentration given density normalization rho_0 in h^2/Mpc^3 (physical) (inverse of function rho0_c)

Parameters

- **rho0** – density normalization in h^2/Mpc^3 (physical)
- **z** – redshift

Returns concentration parameter c

nfw_Mz (*M*, *z*)

returns all needed parameter (in physical units modulo h) to draw the profile of the main halo r200 in physical Mpc/h rho_s in h^2/Mpc^3 (physical) Rs in Mpc/h physical c unit less

Parameters

- **M** – Mass in physical M_sun/h
- **z** – redshift

r200_M (*M*, *z*)

computes the radius R_200 crit of a halo of mass M in physical mass M/h

Parameters

- **M** (*float or numpy array*) – halo mass in M_sun/h
- **z** (*float*) – redshift

Returns radius R_200 in physical Mpc/h

rho0_c (*c*, *z*)
computes density normalization as a function of concentration parameter

Parameters

- **c** – concentration
- **z** – redshift

Returns density normalization in h^2/Mpc^3 (physical)

rhoc = 277536627000.0

rhoc_z (*z*)

Parameters **z** – redshift

Returns critical density of the universe at redshift z in physical units [h^2 M_sun Mpc^-3]

Module contents

lenstronomy.Data package

Submodules

lenstronomy.Data.coord_transforms module

class Coordinates (*transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*)
Bases: `object`

class to handle linear coordinate transformations of a square pixel image

__init__ (*transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*)
initialize the coordinate-to-pixel transform and their inverse

Parameters

- **transform_pix2angle** – 2x2 matrix, mapping of pixel to coordinate
- **ra_at_xy_0** – ra coordinate at pixel (0,0)
- **dec_at_xy_0** – dec coordinate at pixel (0,0)

coordinate_grid (*nx*, *ny*)

Parameters

- **nx** – number of pixels in x-direction
- **ny** – number of pixels in y-direction

Returns 2d arrays with coordinates in RA/DEC with ra_coord[y-axis, x-axis]

map_coord2pix (*ra*, *dec*)
maps the (ra,dec) coordinates of the system into the pixel coordinate of the image

Parameters

- **ra** – relative RA coordinate as defined by the coordinate frame
- **dec** – relative DEC coordinate as defined by the coordinate frame

Returns (x, y) pixel coordinates

map_pix2coord(*x*, *y*)
maps the (x,y) pixel coordinates of the image into the system coordinates

Parameters

- **x** – pixel coordinate (can be 1d numpy array), defined in the center of the pixel
- **y** – pixel coordinate (can be 1d numpy array), defined in the center of the pixel

Returns relative (RA, DEC) coordinates of the system

pixel_area
angular area of a pixel in the image

Returns area [arcsec^2]

pixel_width
size of pixel

Returns sqrt(pixel_area)

radec_at_xy_0

Returns RA, DEC coordinate at (0,0) pixel coordinate

shift_coordinate_system(*x_shift*, *y_shift*, *pixel_unit=False*)
shifts the coordinate system

Parameters

- **x_shift** – shift in x (or RA)
- **y_shift** – shift in y (or DEC)
- **pixel_unit** – bool, if True, units of pixels in input, otherwise RA/DEC

Returns updated data class with change in coordinate system

transform_angle2pix

Returns transformation matrix from angular to pixel coordinates

transform_pix2angle

Returns transformation matrix from pixel to angular coordinates

xy_at_radec_0

Returns pixel coordinate at angular (0,0) point

class Coordinates1D(*transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*)
Bases: *lenstronomy.Data.coord_transforms.Coordinates*

coordinate grid described in 1-d arrays

coordinate_grid(*nx*, *ny*)

Parameters

- **nx** – number of pixels in x-direction
- **ny** – number of pixels in y-direction

Returns 2d arrays with coordinates in RA/DEC with ra_coord[y-axis, x-axis]

lenstronomy.Data.image_noise module

class ImageNoise(*image_data*, *exposure_time=None*, *background_rms=None*, *noise_map=None*, *gradient_boost_factor=None*, *verbose=True*)

Bases: `object`

class that deals with noise properties of imaging data

C_D

Covariance matrix of all pixel values in 2d numpy array (only diagonal component) The covariance matrix is estimated from the data. WARNING: For low count statistics, the noise in the data may lead to biased estimates of the covariance matrix.

Returns covariance matrix of all pixel values in 2d numpy array (only diagonal component).

C_D_model(*model*)

Parameters **model** – model (same as data but without noise)

Returns estimate of the noise per pixel based on the model flux

__init__(*image_data*, *exposure_time=None*, *background_rms=None*, *noise_map=None*, *gradient_boost_factor=None*, *verbose=True*)

Parameters

- **image_data** – numpy array, pixel data values
- **exposure_time** – int or array of size the data; exposure time (common for all pixels or individually for each individual pixel)
- **background_rms** – root-mean-square value of Gaussian background noise
- **noise_map** – int or array of size the data; joint noise sqrt(variance) of each individual pixel. Overwrites meaning of background_rms and exposure_time.
- **gradient_boost_factor** – None or float, variance terms added in quadrature scaling with gradient^2 * gradient_boost_factor

background_rms

Returns rms value of background noise

exposure_map

Units of data and exposure map should result in: number of flux counts = data * exposure_map

Returns exposure map for each pixel

covariance_matrix(*data*, *background_rms*, *exposure_map*, *gradient_boost_factor=None*)

returns a diagonal matrix for the covariance estimation which describes the error

Notes:

- **the exposure map must be positive definite. Values that deviate too much from the mean exposure time will be** given a lower limit to not under-predict the Poisson component of the noise.
- **the data must be positive semi-definite for the Poisson noise estimate.** Values < 0 (Possible after mean subtraction) will not have a Poisson component in their noise estimate.

Parameters

- **data** – data array, eg in units of photons/second
- **background_rms** – background noise rms, eg. in units (photons/second)^2
- **exposure_map** – exposure time per pixel, e.g. in units of seconds

- **gradient_boost_factor** – None or float, variance terms added in quadrature scaling with gradient^2 * gradient_boost_factor

Returns len(d) x len(d) matrix that give the error of background and Poisson components; (photons/second)^2

lenstronomy.Data.imaging_data module

class ImageData(*image_data*, *exposure_time=None*, *background_rms=None*, *noise_map=None*, *gradient_boost_factor=None*, *ra_at_xy_0=0*, *dec_at_xy_0=0*, *transform_pix2angle=None*, *ra_shift=0*, *dec_shift=0*, *antenna_primary_beam=None*)

Bases: `lenstronomy.Data.pixel_grid.PixelGrid`, `lenstronomy.Data.image_noise.ImageNoise`

class to handle the data, coordinate system and masking, including convolution with various numerical precisions

The Data() class is initialized with keyword arguments:

- 'image_data': 2d numpy array of the image data
- 'transform_pix2angle' 2x2 transformation matrix (linear) to transform a pixel shift into a coordinate shift (x, y) -> (ra, dec)
- 'ra_at_xy_0' RA coordinate of pixel (0,0)
- 'dec_at_xy_0' DEC coordinate of pixel (0,0)

optional keywords for shifts in the coordinate system: - 'ra_shift': shifts the coordinate system with respect to 'ra_at_xy_0' - 'dec_shift': shifts the coordinate system with respect to 'dec_at_xy_0'

optional keywords for noise properties: - 'background_rms': rms value of the background noise - 'exp_time': float, exposure time to compute the Poisson noise contribution - 'exposure_map': 2d numpy array, effective exposure time for each pixel. If set, will replace 'exp_time' - 'noise_map': Gaussian noise (1-sigma) for each individual pixel. If this keyword is set, the other noise properties will be ignored.

optional keywords for interferometric quantities: - 'antenna_primary_beam': primary beam pattern of antennae (now treat each antenna with the same primary beam)

** notes ** the likelihood for the data given model P(data|model) is defined in the function below. Please make sure that your definitions and units of 'exposure_map', 'background_rms' and 'image_data' are in accordance with the likelihood function. In particular, make sure that the Poisson noise contribution is defined in the count rate.

__init__(*image_data*, *exposure_time=None*, *background_rms=None*, *noise_map=None*, *gradient_boost_factor=None*, *ra_at_xy_0=0*, *dec_at_xy_0=0*, *transform_pix2angle=None*, *ra_shift=0*, *dec_shift=0*, *antenna_primary_beam=None*)

Parameters

- **image_data** – 2d numpy array of the image data
- **exposure_time** – int or array of size the data; exposure time (common for all pixels or individually for each individual pixel)
- **background_rms** – root-mean-square value of Gaussian background noise in units counts per second
- **noise_map** – int or array of size the data; joint noise sqrt(variance) of each individual pixel.

- **gradient_boost_factor** – None or float, variance terms added in quadrature scaling with gradient^2 * gradient_boost_factor
- **transform_pix2angle** – 2x2 matrix, mapping of pixel to coordinate
- **ra_at_xy_0** – ra coordinate at pixel (0,0)
- **dec_at_xy_0** – dec coordinate at pixel (0,0)
- **ra_shift** – RA shift of pixel grid
- **dec_shift** – DEC shift of pixel grid
- **antenna_primary_beam** – 2d numpy array with the same size of imaga_data; more descriptions of the primary beam can be found in the AngularSensitivity class

data

Returns 2d numpy array of data

log_likelihood(*model*, *mask*, *additional_error_map=0*)

computes the likelihood of the data given the model p(data|model) The Gaussian errors are estimated with the covariance matrix, based on the model image. The errors include the background rms value and the exposure time to compute the Poisson noise level (in Gaussian approximation).

Parameters

- **model** – the model (same dimensions and units as data)
- **mask** – bool (1, 0) values per pixel. If =0, the pixel is ignored in the likelihood
- **additional_error_map** – additional error term (in same units as covariance matrix). This can e.g. come from model errors in the PSF estimation.

Returns the natural logarithm of the likelihood p(data|model)

update_data(*image_data*)

update the data as well as the error matrix estimated from it when done so using the data

Parameters **image_data** – 2d numpy array of same size as nx, ny

Returns None

lenstronomy.Data.pixel_grid module

class PixelGrid(*nx*, *ny*, *transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*, *antenna_primary_beam=None*)

Bases: `lenstronomy.Data.coord_transforms.Coordinates`, `lenstronomy.Data.angular_sensitivity.AngularSensitivity`

class that manages a specified pixel grid (rectangular at the moment) and its coordinates

__init__(*nx*, *ny*, *transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*, *antenna_primary_beam=None*)

Parameters

- **nx** – number of pixels in x-axis
- **ny** – number of pixels in y-axis
- **transform_pix2angle** – 2x2 matrix, mapping of pixel to coordinate
- **ra_at_xy_0** – ra coordinate at pixel (0,0)
- **dec_at_xy_0** – dec coordinate at pixel (0,0)

- **antenna_primary_beam** – 2d numpy array with the same size of imaga_data; more descriptions of the primary beam can be found in the AngularSensitivity class

center

Returns center_x, center_y of coordinate system

num_pixel

Returns number of pixels in the data

num_pixel_axes

Returns number of pixels per axis, nx ny

pixel_coordinates

Returns RA coords, DEC coords

shift_coordinate_system (*x_shift*, *y_shift*, *pixel_unit=False*)
shifts the coordinate system :param x_shift: shift in x (or RA) :param y_shift: shift in y (or DEC) :param pixel_unit: bool, if True, units of pixels in input, otherwise RA/DEC :return: updated data class with change in coordinate system

width

Returns width of data frame

lenstronomy.Data.psf module

class PSF (*psf_type='NONE'*, *fwhm=None*, *truncation=5*, *pixel_size=None*, *kernel_point_source=None*, *psf_error_map=None*, *point_source_supersampling_factor=1*, *kernel_point_source_init=None*, *kernel_point_source_normalisation=True*)
Bases: `object`

Point Spread Function class. This class describes and manages products used to perform the PSF modeling (convolution for extended surface brightness and painting of PSF's for point sources).

__init__ (*psf_type='NONE'*, *fwhm=None*, *truncation=5*, *pixel_size=None*, *kernel_point_source=None*, *psf_error_map=None*, *point_source_supersampling_factor=1*, *kernel_point_source_init=None*, *kernel_point_source_normalisation=True*)

Parameters

- **psf_type** – string, type of PSF: options are 'NONE', 'PIXEL', 'GAUSSIAN'
- **fwhm** – float, full width at half maximum, only required for 'GAUSSIAN' model
- **truncation** – float, Gaussian truncation (in units of sigma), only required for 'GAUSSIAN' model
- **pixel_size** – width of pixel (required for Gaussian model, not required when using in combination with ImageModel modules)
- **kernel_point_source** – 2d numpy array, odd length, centered PSF of a point source (if not normalized, will be normalized)
- **psf_error_map** – uncertainty in the PSF model per pixel (size of data, not super-sampled). 2d numpy array. Size can be larger or smaller than the pixel-sized PSF model and if so, will be matched. This error will be added to the pixel error around the position of point sources as follows: sigma^2_i += 'psf_error_map'_j * <point source amplitude>**2

- **point_source_supersampling_factor** – int, supersampling factor of kernel_point_source. This is the input PSF to this class and does not need to be the choice in the modeling (thought preferred if modeling choses supersampling)
- **kernel_point_source_init** – memory of an initial point source kernel that gets passed through the psf iteration
- **kernel_point_source_normalisation** – boolean, if False, the pixel PSF will not be normalized automatically.

fwhm

Returns full width at half maximum of kernel (in units of pixel)

kernel_pixel
returns the convolution kernel for a uniform surface brightness on a pixel size

Returns 2d numpy array

kernel_point_source

kernel_point_source_supersampled(*supersampling_factor*, *updata_cache=True*)
generates (if not already available) a supersampled PSF with ood numbers of pixels centered

Parameters

- **supersampling_factor** – int >=1, supersampling factor relative to pixel resolution
- **updata_cache** – boolean, if True, updates the cached supersampling PSF if generated. Attention, this will overwrite a previously used supersampled PSF if the resolution is changing.

Returns super-sampled PSF as 2d numpy array

psf_error_map
error variance of the normalized PSF. This error will be added to the pixel error around the position of point sources as follows: sigma^2_i += 'psf_error_map'_j * <point source amplitude>**2

Returns error variance of the normalized PSF. Variance of

Return type 2d numpy array of size of the PSF in pixel size (not supersampled)

set_pixel_size(*deltaPix*)
update pixel size

Parameters **deltaPix** – pixel size in angular units (arc seconds)

Returns None

Module contents

lenstronomy.GalKin package

Submodules

lenstronomy.GalKin.analytic_kinematics module

class AnalyticKinematics(*kwargs_cosmo*, *interpol_grid_num=100*, *log_integration=False*, *max_integrate=100*, *min_integrate=0.001*)
Bases: `lenstronomy.GalKin.anisotropy.Anisotropy`

class to compute eqn 20 in Suyu+2010 with a Monte-Carlo from rendering from the light profile distribution and displacing them with a Gaussian seeing convolution.

This class assumes spherical symmetry in light and mass distribution and

- a Hernquist light profile (parameterised by the half-light radius)
- a power-law mass profile (parameterized by the Einstein radius and logarithmic slop)

The analytic equations for the kinematics in this approximation are presented e.g. in Suyu et al. 2010 and the spectral rendering approach to compute the seeing convolved slit measurement is presented in Birrer et al. 2016. The stellar anisotropy is parameterised based on Osipkov 1979; Merritt 1985.

WARNING!!! Only supports Osipkov-Merritt anisotropy for now!

All units are meant to be in angular arc seconds. The physical units are fold in through the angular diameter distances

`__init__`(*kwargs_cosmo*, *interpol_grid_num=100*, *log_integration=False*, *max_integrate=100*, *min_integrate=0.001*)

Parameters

- **`kwargs_cosmo`** – keyword argument with angular diameter distances entering the Galkin.cosmo class
- **`interpol_grid_num`** – number of interpolations in radius to compute radial velocity dispersion
- **`log_integration`** – perform numerical integration in logarithmic space
- **`max_integrate`** – maximum radius of integration (in projected arc seconds)
- **`min_integrate`** – minimum drawing/calculation of velocity dispersion (in projected arc seconds)

`delete_cache`()

deletes temporary cache tight to a specific model

Returns

`static draw_light`(*kwargs_light*)

Parameters **`kwargs_light`** – keyword argument (list) of the light model

Returns 3d radius (if possible), 2d projected radius, x-projected coordinate, y-projected coordinate

`grav_potential`(*r*, *kwargs_mass*)

Gravitational potential in SI units

Parameters

- **`r`** – radius (arc seconds)
- **`kwargs_mass`** –

Returns gravitational potential

`sigma_r2`(*r*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)

equation (19) in Suyu+ 2010

Parameters

- **`r`** – 3d radius
- **`kwargs_mass`** – mass profile keyword arguments

- **kwargs_light** – light profile keyword arguments
- **kwargs_anisotropy** – anisotropy keyword arguments

Returns velocity dispersion in [m/s]

sigma_s2 (*r*, *R*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)
returns unweighted los velocity dispersion for a specified projected radius, with weight 1

Parameters

- **r** – 3d radius (not needed for this calculation)
- **R** – 2d projected radius (in angular units of arcsec)
- **kwargs_mass** – mass model parameters (following lenstronomy lens model conventions)
- **kwargs_light** – deflector light parameters (following lenstronomy light model conventions)
- **kwargs_anisotropy** – anisotropy parameters, may vary according to anisotropy type chosen. We refer to the Anisotropy() class for details on the parameters.

Returns line-of-sight projected velocity dispersion at projected radius R from 3d radius r

lenstronomy.GalKin.anisotropy module

class Anisotropy (*anisotropy_type*)
Bases: `object`

class that handles the kinematic anisotropy sources: Mamon & Lokas 2005 https://arxiv.org/pdf/astro-ph/0405491.pdf

Agnello et al. 2014 https://arxiv.org/pdf/1401.4462.pdf

K (*r*, *R*, ***kwargs*)
equation A16 im Mamon & Lokas for Osipkov&Merrit anisotropy

Parameters

- **r** – 3d radius
- **R** – projected 2d radius
- **kwargs** – parameters of the specified anisotropy model

Returns K(r, R)

__init__ (*anisotropy_type*)

Parameters **anisotropy_type** – string, anisotropy model type

anisotropy_solution (*r*, ***kwargs*)
the solution to d ln(f)/ d ln(r) = 2 beta(r)

Parameters

- **r** – 3d radius
- **kwargs** – parameters of the specified anisotropy model

Returns f(r)

beta_r (*r*, ***kwargs*)
returns the anisotropy parameter at a given radius

Parameters

- **r** – 3d radius
- **kwargs** – parameters of the specified anisotropy model

Returns beta(r)

delete_anisotropy_cache()
deletes cached interpolations for a fixed anisotropy model

Returns None

class Const
Bases: `object`

constant anisotropy model class See Mamon & Lokas 2005 for details

static K(*r*, *R*, *beta*)
equation A16 im Mamon & Lokas for constant anisotropy

Parameters

- **r** – 3d radius
- **R** – projected 2d radius
- **beta** – anisotropy, float >-0.5

Returns K(r, R, beta)

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

anisotropy_solution(*r*, ***kwargs*)
the solution to d ln(f)/ d ln(r) = 2 beta(r)

Parameters

- **r** – 3d radius
- **kwargs** – parameters of the specified anisotropy model

Returns f(r)

static beta_r(*r*, *beta*)
anisotropy as a function of radius

Parameters

- **r** – 3d radius
- **beta** – anisotropy

Returns beta

class Isotropic
Bases: `object`

class for isotropic (beta=0) stellar orbits See Mamon & Lokas 2005 for details

static K(*r*, *R*)
equation A16 im Mamon & Lokas for constant anisotropy

Parameters

- **r** – 3d radius
- **R** – projected 2d radius

Returns K(r, R)

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

static anisotropy_solution(*r*, ***kwargs*)
the solution to d ln(f)/ d ln(r) = 2 beta(r) See e.g. A3 in Mamon & Lokas

Parameters

- **r** – 3d radius
- **kwargs** – parameters of the specified anisotropy model

Returns f(r)

static beta_r(*r*)
anisotropy as a function of radius

Parameters **r** – 3d radius

Returns beta

class Radial
Bases: `object`

class for radial (beta=1) stellar orbits See Mamon & Lokas 2005 for details

static K(*r*, *R*)
equation A16 im Mamon & Lokas for constant anisotropy

Parameters

- **r** – 3d radius
- **R** – projected 2d radius

Returns K(r, R)

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

static anisotropy_solution(*r*)
the solution to d ln(f)/ d ln(r) = 2 beta(r) See e.g. A4 in Mamon & Lokas

Parameters **r** – 3d radius

Returns f(r)

static beta_r(*r*)
anisotropy as a function of radius

Parameters **r** – 3d radius

Returns beta

class OsipkovMerritt
Bases: `object`

class for Osipkov&Merrit stellar orbits See Mamon & Lokas 2005 for details

static K(*r*, *R*, *r_ani*)
equation A16 im Mamon & Lokas 2005 for Osipkov&Merrit anisotropy

Parameters

- **r** – 3d radius

- **R** – projected 2d radius
- **r_ani** – anisotropy radius

Returns K(r, R)

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

static anisotropy_solution(*r*, *r_ani*)
the solution to d ln(f)/ d ln(r) = 2 beta(r) See e.g. A5 in Mamon & Lokas

Parameters

- **r** – 3d radius
- **r_ani** – anisotropy radius

Returns f(r)

static beta_r(*r*, *r_ani*)
anisotropy as a function of radius

Parameters

- **r** – 3d radius
- **r_ani** – anisotropy radius

Returns beta

class GeneralizedOM
Bases: `object`

generalized Osipkov&Merrit profile see Agnello et al. 2014 https://arxiv.org/pdf/1401.4462.pdf b(r) = beta_inf * r^2 / (r^2 + r_ani^2)

K(*r*, *R*, *r_ani*, *beta_inf*)
equation19 in Agnello et al. 2014 for k_beta(R, r) such that K(R, r) = (sqrt(r^2 - R^2) + k_beta(R, r)) / r

Parameters

- **r** – 3d radius
- **R** – projected 2d radius
- **r_ani** – anisotropy radius
- **beta_inf** – anisotropy at infinity

Returns K(r, R)

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

static anisotropy_solution(*r*, *r_ani*, *beta_inf*)
the solution to d ln(f)/ d ln(r) = 2 beta(r) See e.g. A5 in Mamon & Lokas with a scaling (nominator of Agnello et al. 2014 Equation (12)

Parameters

- **r** – 3d radius
- **r_ani** – anisotropy radius
- **beta_inf** – anisotropy at infinity

Returns f(r)

static beta_r(*r*, *r_ani*, *beta_inf*)
anisotropy as a function of radius

Parameters

- **r** – 3d radius
- **r_ani** – anisotropy radius
- **beta_inf** – anisotropy at infinity

Returns beta

delete_cache()
deletes the interpolation function of the hypergeometic function for a specific beta_inf

Returns deleted self variables

class Colin
Bases: `object`

class for stellar orbits anisotropy parameter based on Colin et al. (2000) See Mamon & Lokas 2005 for details

static K(*r*, *R*, *r_ani*)
equation A16 im Mamon & Lokas for Osipkov&Merrit anisotropy

Parameters

- **r** – 3d radius
- **R** – projected 2d radius
- **r_ani** – anisotropy radius

Returns K(r, R)

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

static beta_r(*r*, *r_ani*)
anisotropy as a function of radius

Parameters

- **r** – 3d radius
- **r_ani** – anisotropy radius

Returns beta

lenstronomy.GalKin.aperture module

class Aperture(*aperture_type*, ***kwargs_aperture*)
Bases: `object`

defines mask(s) of spectra, can handle IFU and single slit/box type data.

__init__(*aperture_type*, ***kwargs_aperture*)

Parameters

- **aperture_type** – string
- **kwargs_aperture** – keyword arguments reflecting the aperture type chosen. We refer to the specific class instances for documentation.

aperture_select (*ra*, *dec*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray

Returns bool, True if photon/ray is within the slit, False otherwise, int of the segment of the IFU

num_segments

lenstronomy.GalKin.aperture_types module

class Slit (*length*, *width*, *center_ra=0*, *center_dec=0*, *angle=0*)

Bases: `object`

Slit aperture description

__init__ (*length*, *width*, *center_ra=0*, *center_dec=0*, *angle=0*)

Parameters

- **length** – length of slit
- **width** – width of slit
- **center_ra** – center of slit
- **center_dec** – center of slit
- **angle** – orientation angle of slit, angle=0 corresponds length in RA direction

aperture_select (*ra*, *dec*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray

Returns bool, True if photon/ray is within the slit, False otherwise

num_segments

number of segments with separate measurements of the velocity dispersion

Returns int

slit_select (*ra*, *dec*, *length*, *width*, *center_ra=0*, *center_dec=0*, *angle=0*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray
- **length** – length of slit
- **width** – width of slit
- **center_ra** – center of slit
- **center_dec** – center of slit
- **angle** – orientation angle of slit, angle=0 corresponds length in RA direction

Returns bool, True if photon/ray is within the slit, False otherwise

class Frame(*width_outer*, *width_inner*, *center_ra=0*, *center_dec=0*, *angle=0*)
Bases: `object`

rectangular box with a hole in the middle (also rectangular), effectively a frame

__init__(*width_outer*, *width_inner*, *center_ra=0*, *center_dec=0*, *angle=0*)

Parameters

- **width_outer** – width of box to the outer parts
- **width_inner** – width of inner removed box
- **center_ra** – center of slit
- **center_dec** – center of slit
- **angle** – orientation angle of slit, angle=0 corresponds length in RA direction

aperture_select(*ra*, *dec*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray

Returns bool, True if photon/ray is within the slit, False otherwise

num_segments
number of segments with separate measurements of the velocity dispersion

Returns int

frame_select(*ra*, *dec*, *width_outer*, *width_inner*, *center_ra=0*, *center_dec=0*, *angle=0*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray
- **width_outer** – width of box to the outer parts
- **width_inner** – width of inner removed box
- **center_ra** – center of slit
- **center_dec** – center of slit
- **angle** – orientation angle of slit, angle=0 corresponds length in RA direction

Returns bool, True if photon/ray is within the box with a hole, False otherwise

class Shell(*r_in*, *r_out*, *center_ra=0*, *center_dec=0*)
Bases: `object`

Shell aperture

__init__(*r_in*, *r_out*, *center_ra=0*, *center_dec=0*)

Parameters

- **r_in** – innermost radius to be selected
- **r_out** – outermost radius to be selected
- **center_ra** – center of the sphere
- **center_dec** – center of the sphere

aperture_select (*ra*, *dec*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray

Returns bool, True if photon/ray is within the slit, False otherwise

num_segments
number of segments with separate measurements of the velocity dispersion

Returns int

shell_select (*ra*, *dec*, *r_in*, *r_out*, *center_ra=0*, *center_dec=0*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray
- **r_in** – innermost radius to be selected
- **r_out** – outermost radius to be selected
- **center_ra** – center of the sphere
- **center_dec** – center of the sphere

Returns boolean, True if within the radial range, False otherwise

class IFUShells (*r_bins*, *center_ra=0*, *center_dec=0*)
Bases: `object`

class for an Integral Field Unit spectrograph with azimuthal shells where the kinematics are measured

__init__ (*r_bins*, *center_ra=0*, *center_dec=0*)

Parameters

- **r_bins** – array of radial bins to average the dispersion spectra in ascending order. It starts with the inner-most edge to the outermost edge.
- **center_ra** – center of the sphere
- **center_dec** – center of the sphere

aperture_select (*ra*, *dec*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray

Returns bool, True if photon/ray is within the slit, False otherwise, index of shell

num_segments
number of segments with separate measurements of the velocity dispersion :return: int

shell_ifu_select (*ra*, *dec*, *r_bin*, *center_ra=0*, *center_dec=0*)

Parameters

- **ra** – angular coordinate of photon/ray
- **dec** – angular coordinate of photon/ray

- **r_bin** – array of radial bins to average the dispersion spectra in ascending order. It starts with the inner-most edge to the outermost edge.
- **center_ra** – center of the sphere
- **center_dec** – center of the sphere

Returns boolean, True if within the radial range, False otherwise

lenstronomy.GalKin.cosmo module

class Cosmo(*d_d*, *d_s*, *d_ds*)

Bases: `object`

cosmological quantities

__init__(*d_d*, *d_s*, *d_ds*)

Parameters

- **d_d** – angular diameter distance to the deflector
- **d_s** – angular diameter distance to the source
- **d_ds** – angular diameter distance between deflector and source

arcsec2phys_lens(*theta*)

converts arc seconds to physical units on the deflector

Parameters theta – angle observed on the sky in units of arc seconds

Returns physical distance of the angle in units of Mpc

epsilon_crit

returns the critical projected mass density in units of M_sun/Mpc^2 (physical units)

lenstronomy.GalKin.galkin module

class Galkin(*kwargs_model*, *kwargs_aperture*, *kwargs_psf*, *kwargs_cosmo*, *kwargs_numerics=None*, *analytic_kinematics=False*)

Bases: `lenstronomy.GalKin.galkin_model.GalkinModel`, `lenstronomy.GalKin.observation.GalkinObservation`

Major class to compute velocity dispersion measurements given light and mass models

The class supports any mass and light distribution (and superposition thereof) that has a 3d correspondance in their 2d lens model distribution. For models that do not have this correspondance, you may want to apply a Multi-Gaussian Expansion (MGE) on their models and use the MGE to be de-projected to 3d.

The computation follows Mamon&Lokas 2005 and performs the spectral rendering of the seeing convolved apperture with the method introduced by Birrer et al. 2016.

The class supports various types of anisotropy models (see Anisotropy class) and aperture types (see Aperture class).

Solving the Jeans Equation requires a numerical integral over the 3d light and mass profile (see Mamon&Lokas 2005). This class (as well as the dedicated LightModel and MassModel classes) perform those integral numerically with an interpolated grid.

The seeing convolved integral over the aperture is computed by rendering spectra (light weighted LOS kinematics) from the light distribution.

The cosmology assumed to compute the physical mass and distances are set via the kwargs_cosmo keyword arguments.
d_d: Angular diameter distance to the deflector (in Mpc) d_s: Angular diameter distance to the source (in Mpc) d_ds: Angular diameter distance from the deflector to the source (in Mpc)

The numerical options can be chosen through the kwargs_numerics keywords

interpol_grid_num: number of interpolation points in the light and mass profile (radially). This number should be chosen high enough to accurately describe the true light profile underneath. log_integration: bool, if True, performs the interpolation and numerical integration in log-scale.

max_integrate: maximum 3d radius to where the numerical integration of the Jeans Equation solver is made. This value should be large enough to contain most of the light and to lead to a converged result. min_integrate: minimal integration value. This value should be very close to zero but some mass and light profiles are diverging and a numerically stabel value should be chosen.

These numerical options should be chosen to allow for a converged result (within your tolerance) but not too conservative to impact too much the computational cost. Reasonable values might depend on the specific problem.

`__init__`(*kwargs_model*, *kwargs_aperture*, *kwargs_psf*, *kwargs_cosmo*, *kwargs_numerics=None*, *analytic_kinematics=False*)

Parameters

- **`kwargs_model`** – keyword arguments describing the model components
- **`kwargs_aperture`** – keyword arguments describing the spectroscopic aperture, see Aperture() class
- **`kwargs_psf`** – keyword argument specifying the PSF of the observation
- **`kwargs_cosmo`** – keyword arguments that define the cosmology in terms of the angular diameter distances involved
- **`kwargs_numerics`** – numerics keyword arguments
- **`analytic_kinematics`** – bool, if True uses the analytic kinematic model

`dispersion`(*kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*, *sampling_number=1000*)
computes the averaged LOS velocity dispersion in the slit (convolved)

Parameters

- **`kwargs_mass`** – mass model parameters (following lenstronomy lens model conventions)
- **`kwargs_light`** – deflector light parameters (following lenstronomy light model conventions)
- **`kwargs_anisotropy`** – anisotropy parameters, may vary according to anisotropy type chosen. We refer to the Anisotropy() class for details on the parameters.
- **`sampling_number`** – int, number of spectral sampling of the light distribution

Returns integrated LOS velocity dispersion in units [km/s]

`dispersion_map`(*kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*, *num_kin_sampling=1000*, *num_psf_sampling=100*)
computes the velocity dispersion in each Integral Field Unit

Parameters

- **`kwargs_mass`** – keyword arguments of the mass model
- **`kwargs_light`** – keyword argument of the light model

- **kwargs_anisotropy** – anisotropy keyword arguments
- **num_kin_sampling** – int, number of draws from a kinematic prediction of a LOS
- **num_psf_sampling** – int, number of displacements/render from a spectra to be displaced on the IFU

Returns ordered array of velocity dispersions [km/s] for each unit

lenstronomy.GalKin.galkin_model module

class GalkinModel(*kwargs_model*, *kwargs_cosmo*, *kwargs_numerics=None*, *analytic_kinematics=False*)

Bases: object

this class handles all the kinematic modeling aspects of Galkin Excluded are observational conditions (seeing, aperture etc) Major class to compute velocity dispersion measurements given light and mass models

The class supports any mass and light distribution (and superposition thereof) that has a 3d correspondance in their 2d lens model distribution. For models that do not have this correspondence, you may want to apply a Multi-Gaussian Expansion (MGE) on their models and use the MGE to be de-projected to 3d.

The computation follows Mamon&Lokas 2005.

The class supports various types of anisotropy models (see Anisotropy class). Solving the Jeans Equation requires a numerical integral over the 3d light and mass profile (see Mamon&Lokas 2005). This class (as well as the dedicated LightModel and MassModel classes) perform those integral numerically with an interpolated grid.

The cosmology assumed to compute the physical mass and distances are set via the kwargs_cosmo keyword arguments.
d_d: Angular diameter distance to the deflector (in Mpc) d_s: Angular diameter distance to the source (in Mpc) d_ds: Angular diameter distance from the deflector to the source (in Mpc)

The numerical options can be chosen through the kwargs_numerics keywords

interpol_grid_num: number of interpolation points in the light and mass profile (radially). This number should be chosen high enough to accurately describe the true light profile underneath. log_integration: bool, if True, performs the interpolation and numerical integration in log-scale.

max_integrate: maximum 3d radius to where the numerical integration of the Jeans Equation solver is made. This value should be large enough to contain most of the light and to lead to a converged result. min_integrate: minimal integration value. This value should be very close to zero but some mass and light profiles are diverging and a numerically stable value should be chosen.

These numerical options should be chosen to allow for a converged result (within your tolerance) but not too conservative to impact too much the computational cost. Reasonable values might depend on the specific problem.

__init__(*kwargs_model*, *kwargs_cosmo*, *kwargs_numerics=None*, *analytic_kinematics=False*)

Parameters

- **kwargs_model** – keyword arguments describing the model components
- **kwargs_cosmo** – keyword arguments that define the cosmology in terms of the angular diameter distances involved
- **kwargs_numerics** – numerics keyword arguments
- **analytic_kinematics** – bool, if True uses the analytic kinematic model

check_df (*r*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)
checks whether the phase space distribution function of a given anisotropy model is positive. Currently this is implemented by the relation provided by Ciotti and Morganti 2010 equation (10) https://arxiv.org/pdf/1006.2344.pdf

Parameters

- **r** – 3d radius to check slope-anisotropy constraint
- **kwargs_mass** – keyword arguments for mass (lens) profile
- **kwargs_light** – keyword arguments for light profile
- **kwargs_anisotropy** – keyword arguments for stellar anisotropy distribution

Returns equation (10) >= 0 for physical interpretation

lenstronomy.GalKin.light_profile module

class LightProfile (*profile_list*, *interpol_grid_num=2000*, *max_interpolate=1000*, *min_interpolate=0.001*, *max_draw=None*)
Bases: `object`

class to deal with the light distribution for GalKin

In particular, this class allows for:

- (faster) interpolated calculation for a given profile (for a range that the Jeans equation is computed)
- drawing 3d and 2d distributions from a given (spherical) profile (within bounds where the Jeans equation is expected to be accurate)
- 2d projected profiles within the 3d integration range (truncated)

__init__ (*profile_list*, *interpol_grid_num=2000*, *max_interpolate=1000*, *min_interpolate=0.001*, *max_draw=None*)

Parameters

- **profile_list** – list of light profiles for LightModel module (must support light_3d() functionalities)
- **interpol_grid_num** – int; number of interpolation steps (logarithmically between min and max value)
- **max_interpolate** – float; maximum interpolation of 3d light profile
- **min_interpolate** – float; minimum interpolate (and also drawing of light profile)
- **max_draw** – float; (optional) if set, draws up to this radius, else uses max_interpolate value

delete_cache ()
deletes cached interpolation function of the CDF for a specific light profile

Returns None

draw_light_2d (*kwargs_list*, *n=1*, *new_compute=False*)
constructs the CDF and draws from it random realizations of projected radii R CDF is constructed in logarithmic projected radius spacing

Parameters

- **kwargs_list** – light model keyword argument list

- **n** – int, number of draws per functino call
- **new_compute** – re-computes the interpolated CDF

Returns realization of projected radius following the distribution of the light model

draw_light_2d_linear(*kwargs_list*, *n=1*, *new_compute=False*)
constructs the CDF and draws from it random realizations of projected radii R The interpolation of the CDF is done in linear projected radius space

Parameters

- **kwargs_list** – list of keyword arguments of light profiles (see LightModule)
- **n** – int; number of draws
- **new_compute** – boolean, if True, re-computes the interpolation (becomes valid with updated kwargs_list argument)

Returns draw of projected radius for the given light profile distribution

draw_light_3d(*kwargs_list*, *n=1*, *new_compute=False*)
constructs the CDF and draws from it random realizations of 3D radii r

Parameters

- **kwargs_list** – light model keyword argument list
- **n** – int, number of draws per function call
- **new_compute** – re-computes the interpolated CDF

Returns realization of projected radius following the distribution of the light model

light_2d(*R*, *kwargs_list*)
projected light profile (integrated to infinity in the projected axis)

Parameters

- **R** – projected 2d radius
- **kwargs_list** – list of keyword arguments of light profiles (see LightModule)

Returns projected surface brightness

light_2d_finite(*R*, *kwargs_list*)
projected light profile (integrated to FINITE 3d boundaries from the max_interpolate)

Parameters

- **R** – projected 2d radius (between min_interpolate and max_interpolate
- **kwargs_list** – list of keyword arguments of light profiles (see LightModule)

Returns projected surface brightness

light_3d(*r*, *kwargs_list*)
three-dimensional light profile

Parameters

- **r** – 3d radius
- **kwargs_list** – list of keyword arguments of light profiles (see LightModule)

Returns flux per 3d volume at radius r

light_3d_interp (*r*, *kwargs_list*, *new_compute=False*)
interpolated three-dimensional light profile within bounds [min_interpolate, max_interpolate] in logarithmic units with interpol_grid_num numbers of interpolation steps

Parameters

- **r** – 3d radius
- **kwargs_list** – list of keyword arguments of light profiles (see LightModule)
- **new_compute** – boolean, if True, re-computes the interpolation (becomes valid with updated kwargs_list argument)

Returns flux per 3d volume at radius r

lenstronomy.GalKin.numeric_kinematics module

class NumericKinematics (*kwargs_model*, *kwargs_cosmo*, *interpol_grid_num=1000*, *log_integration=True*, *max_integrate=1000*, *min_integrate=0.0001*, *max_light_draw=None*, *lum_weight_int_method=True*)
Bases: *lenstronomy.GalKin.anisotropy.Anisotropy*

__init__ (*kwargs_model*, *kwargs_cosmo*, *interpol_grid_num=1000*, *log_integration=True*, *max_integrate=1000*, *min_integrate=0.0001*, *max_light_draw=None*, *lum_weight_int_method=True*)
What we need: - max projected R to have ACCURATE I_R_sigma values - make sure everything outside cancels out (or is not rendered)

Parameters

- **interpol_grid_num** – number of interpolation bins for integrand and interpolated functions
- **log_integration** – bool, if True, performs the numerical integral in log space distance (adviced) (only applies for lum_weight_int_method=True)
- **max_integrate** – maximum radius (in arc seconds) of the Jeans equation integral (assumes zero tracer particles outside this radius)
- **max_light_draw** – float; (optional) if set, draws up to this radius, else uses max_interpolate value
- **lum_weight_int_method** – bool, luminosity weighted dispersion integral to calculate LOS projected Jean's solution. ATTENTION: currently less accurate than 3d solution
- **min_integrate** –

delete_cache ()
delete interpolation function for a specific mass and light profile as well as for a specific anisotropy model

Returns

draw_light (*kwargs_light*)

Parameters **kwargs_light** – keyword argument (list) of the light model

Returns 3d radius (if possible), 2d projected radius, x-projected coordinate, y-projected coordinate

grav_potential (*r*, *kwargs_mass*)
Gravitational potential in SI units

Parameters

- **r** – radius (arc seconds)
- **kwargs_mass** –

Returns gravitational potential

mass_3d (*r*, *kwargs*)
mass enclosed a 3d radius

Parameters

- **r** – in arc seconds
- **kwargs** – lens model parameters in arc seconds

Returns mass enclosed physical radius in kg

sigma_r2 (*r*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)
computes numerically the solution of the Jeans equation for a specific 3d radius E.g. Equation (A1) of Mamon & Lokas https://arxiv.org/pdf/astro-ph/0405491.pdf

$$l(r)\sigma_r(r)^2 = 1/f(r) \int_r^{\infty} f(s)l(s)GM(s)/s^2 ds$$

where l(r) is the 3d light profile M(s) is the enclosed 3d mass f is the solution to d ln(f)/ d ln(r) = 2 beta(r)

Parameters

- **r** – 3d radius
- **kwargs_mass** – mass model parameters (following lenstronomy lens model conventions)
- **kwargs_light** – deflector light parameters (following lenstronomy light model conventions)
- **kwargs_anisotropy** – anisotropy parameters, may vary according to anisotropy type chosen. We refer to the Anisotropy() class for details on the parameters.

Returns sigma_r**2

sigma_s2 (*r*, *R*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)
returns unweighted los velocity dispersion for a specified 3d and projected radius (if lum_weight_int_method=True then the 3d radius is not required and the function directly performs the luminosity weighted integral in projection at R)

Parameters

- **r** – 3d radius (not needed for this calculation)
- **R** – 2d projected radius (in angular units of arcsec)
- **kwargs_mass** – mass model parameters (following lenstronomy lens model conventions)
- **kwargs_light** – deflector light parameters (following lenstronomy light model conventions)
- **kwargs_anisotropy** – anisotropy parameters, may vary according to anisotropy type chosen. We refer to the Anisotropy() class for details on the parameters.

Returns weighted line-of-sight projected velocity dispersion at projected radius R with weights I

sigma_s2_project (*R*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)
returns luminosity-weighted los velocity dispersion for a specified projected radius R and weight

Parameters

- **R** – 2d projected radius (in angular units of arcsec)
- **kwargs_mass** – mass model parameters (following lenstronomy lens model conventions)
- **kwargs_light** – deflector light parameters (following lenstronomy light model conventions)
- **kwargs_anisotropy** – anisotropy parameters, may vary according to anisotropy type chosen. We refer to the Anisotropy() class for details on the parameters.

Returns line-of-sight projected velocity dispersion at projected radius R

sigma_s2_r(*r*, *R*, *kwargs_mass*, *kwargs_light*, *kwargs_anisotropy*)
returns unweighted los velocity dispersion for a specified 3d radius r at projected radius R

Parameters

- **r** – 3d radius (not needed for this calculation)
- **R** – 2d projected radius (in angular units of arcsec)
- **kwargs_mass** – mass model parameters (following lenstronomy lens model conventions)
- **kwargs_light** – deflector light parameters (following lenstronomy light model conventions)
- **kwargs_anisotropy** – anisotropy parameters, may vary according to anisotropy type chosen. We refer to the Anisotropy() class for details on the parameters.

Returns line-of-sight projected velocity dispersion at projected radius R from 3d radius r

lenstronomy.GalKin.observation module

class GalkinObservation(*kwargs_aperture*, *kwargs_psf*)
Bases: `lenstronomy.GalKin.psf.PSF`, `lenstronomy.GalKin.aperture.Aperture`

this class sets the base for the observational properties (aperture and seeing condition)

__init__(*kwargs_aperture*, *kwargs_psf*)

Parameters

- **psf_type** – string, point spread function type, current support for 'GAUSSIAN' and 'MOFFAT'
- **kwargs_psf** – keyword argument describing the relevant parameters of the PSF.

lenstronomy.GalKin.psf module

class PSF(*psf_type*, ***kwargs_psf*)
Bases: `object`

general class to handle the PSF in the GalKin module for rendering the displacement of photons/spectro

__init__(*psf_type*, ***kwargs_psf*)

Parameters

- **psf_type** – string, point spread function type, current support for 'GAUSSIAN' and 'MOFFAT'
- **kwargs_psf** – keyword argument describing the relevant parameters of the PSF.

displace_psf (*x*, *y*)

Parameters

- **x** – x-coordinate of light ray
- **y** – y-coordinate of light ray

Returns x', y' displaced by the two dimensional PSF distribution function

class PSFGaussian (*fwhm*)

Bases: object

Gaussian PSF

__init__ (*fwhm*)

Parameters **fwhm** – full width at half maximum seeing condition

displace_psf (*x*, *y*)

Parameters

- **x** – x-coordinate of light ray
- **y** – y-coordinate of light ray

Returns x', y' displaced by the two dimensional PSF distribution function

class PSFMoffat (*fwhm*, *moffat_beta*)

Bases: object

Moffat PSF

__init__ (*fwhm*, *moffat_beta*)

Parameters

- **fwhm** – full width at half maximum seeing condition
- **moffat_beta** – float, beta parameter of Moffat profile

displace_psf (*x*, *y*)

Parameters

- **x** – x-coordinate of light ray
- **y** – y-coordinate of light ray

Returns x', y' displaced by the two dimensional PSF distribution function

lenstronomy.GalKin.velocity_util module

hyp_2F1 (*a*, *b*, *c*, *z*)

http://docs.sympy.org/0.7.1/modules/mpmath/functions/hypergeometric.html

displace_PSF_gaussian (*x*, *y*, *FWHM*)

Parameters

- **x** – x-coord (arc sec)

- **y** – y-coord (arc sec)
- **FWHM** – psf size (arc sec)

Returns x', y' random displaced according to psf

moffat_r (*r*, *alpha*, *beta*)
Moffat profile

Parameters

- **r** – radial coordinate
- **alpha** – Moffat parameter
- **beta** – exponent

Returns Moffat profile

moffat_fwhm_alpha (*FWHM*, *beta*)
computes alpha parameter from FWHM and beta for a Moffat profile

Parameters

- **FWHM** – full width at half maximum
- **beta** – beta parameter of Moffat profile

Returns alpha parameter of Moffat profile

draw_moffat_r (*FWHM*, *beta*)

Parameters

- **FWHM** – full width at half maximum
- **beta** – Moffat beta parameter

Returns draw from radial Moffat distribution

displace_PSF_moffat (*x*, *y*, *FWHM*, *beta*)

Parameters

- **x** – x-coordinate of light ray
- **y** – y-coordinate of light ray
- **FWHM** – full width at half maximum
- **beta** – Moffat beta parameter

Returns displaced ray by PSF

draw_cdf_Y (*beta*)
Draw c.d.f for Moffat function according to Berge et al. Ufig paper, equation B2 cdf(Y) = 1-Y**(1-beta)

Returns

project2d_random (*r*)
draws a random projection from radius r in 2d and 1d :param r: 3d radius :return: R, x, y

draw_xy (*R*)

Parameters **R** – projected radius

Returns

draw_hernquist (*a*)

Parameters **a** – 0.551*r_eff

Returns realisation of radius of Hernquist luminosity weighting in 3d

Module contents

lenstronomy.ImSim package

Subpackages

lenstronomy.ImSim.MultiBand package

Submodules

lenstronomy.ImSim.MultiBand.joint_linear module

class JointLinear(*multi_band_list*, *kwargs_model*, *compute_bool=None*, *likelihood_mask_list=None*)

Bases: `lenstronomy.ImSim.MultiBand.multi_linear.MultiLinear`

class to model multiple exposures in the same band and makes a constraint fit to all bands simultaneously with joint constraints on the surface brightness of the model. This model setting require the same surface brightness models to be called in all available images/bands

__init__(*multi_band_list*, *kwargs_model*, *compute_bool=None*, *likelihood_mask_list=None*)

Parameters

- **multi_band_list** – list of imaging band configurations [[kwargs_data, kwargs_psf, kwargs_numerics],[. . .], . . .]
- **kwargs_model** – model option keyword arguments
- **likelihood_mask_list** – list of likelihood masks (booleans with size of the individual images)
- **compute_bool** – (optional), bool list to indicate which band to be included in the modeling
- **linear_solver** – bool, if True (default) fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

data_response

returns the 1d array of the data element that is fitted for (including masking)

Returns 1d numpy array

error_response(*kwargs_lens*, *kwargs_ps*, *kwargs_special=None*)

returns the 1d array of the error estimate corresponding to the data response

Returns 1d numpy array of response, 2d array of additonal errors (e.g. point source uncertainties)

image_linear_solve(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *inv_bool=False*)

computes the image (lens and source surface brightness with a given lens model). The linear parameters are computed with a weighted linear least square optimization (i.e. flux normalization of the brightness profiles)

Parameters

- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **kwargs_ps** – keyword arguments corresponding to "other" parameters, such as external shear and point source image positions
- **inv_bool** – if True, invert the full linear solver Matrix Ax = y for the purpose of the covariance matrix.

Returns 1d array of surface brightness pixels of the optimal solution of the linear parameters to match the data

likelihood_data_given_model(*kwargs_lens=None, kwargs_source=None, kwargs_lens_light=None, kwargs_ps=None, kwargs_extinction=None, kwargs_special=None, source_marg=False, linear_prior=None, check_positive_flux=False*)

computes the likelihood of the data given a model This is specified with the non-linear parameters and a linear inversion and prior marginalisation.

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –
- **check_positive_flux** – bool, if True, checks whether the linear inversion resulted in non-negative flux components and applies a punishment in the likelihood if so.

Returns log likelihood (natural logarithm) (sum of the log likelihoods of the individual images)

linear_response_matrix(*kwargs_lens=None, kwargs_source=None, kwargs_lens_light=None, kwargs_ps=None, kwargs_extinction=None, kwargs_special=None*)

computes the linear response matrix (m x n), with n being the data size and m being the coefficients

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –

Returns

lenstronomy.ImSim.MultiBand.multi_data_base module

class MultiDataBase(*image_model_list, compute_bool=None*)

Bases: `object`

Base class with definitions that are shared among all variations of modelling multiple data sets

__init__(*image_model_list*, *compute_bool=None*)

Parameters

- **image_model_list** – list of ImageModel instances (supporting linear inversions)
- **compute_bool** – list of booleans for each imaging band indicating whether to model it or not.

num_bands

num_data_evaluate

num_param_linear(*kwargs_lens*, *kwargs_source*, *kwargs_lens_light*, *kwargs_ps*)

Returns number of linear coefficients to be solved for in the linear inversion

num_response_list

list of number of data elements that are used in the minimization

Returns list of integers

reduced_residuals(*model_list*, *error_map_list=None*)

Parameters

- **model_list** – list of models
- **error_map_list** – list of error maps

Returns

reset_point_source_cache(*cache=True*)

deletes all the cache in the point source class and saves it from then on

Returns

lenstronomy.ImSim.MultiBand.multi_linear module

class MultiLinear(*multi_band_list*, *kwargs_model*, *likelihood_mask_list=None*, *compute_bool=None*, *kwargs_pixelbased=None*, *linear_solver=True*)

Bases: `lenstronomy.ImSim.MultiBand.multi_data_base.MultiDataBase`

class to simulate/reconstruct images in multi-band option. This class calls functions of image_model.py with different bands with joint non-linear parameters and decoupled linear parameters.

the class supports keyword arguments 'index_lens_model_list', 'index_source_light_model_list', 'index_lens_light_model_list', 'index_point_source_model_list', 'index_optical_depth_model_list' in kwargs_model These arguments should be lists of length the number of imaging bands available and each entry in the list is a list of integers specifying the model components being evaluated for the specific band.

E.g. there are two bands and you want to different light profiles being modeled. - you define two different light profiles lens_light_model_list = ['SERSIC', 'SERSIC'] - set index_lens_light_model_list = [[0], [1]] - (optional) for now all the parameters between the two light profiles are independent in the model. You have the possibility to join a subset of model parameters (e.g. joint centroid). See the Param() class for documentation.

__init__(*multi_band_list*, *kwargs_model*, *likelihood_mask_list=None*, *compute_bool=None*, *kwargs_pixelbased=None*, *linear_solver=True*)

Parameters

- **multi_band_list** – list of imaging band configurations [[kwargs_data, kwargs_psf, kwargs_numerics],[...], ...]

- **kwargs_model** – model option keyword arguments
- **likelihood_mask_list** – list of likelihood masks (booleans with size of the individual images)
- **compute_bool** – (optional), bool list to indicate which band to be included in the modeling
- **linear_solver** – bool, if True (default) fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

image_linear_solve(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *inv_bool=False*)

computes the image (lens and source surface brightness with a given lens model). The linear parameters are computed with a weighted linear least square optimization (i.e. flux normalization of the brightness profiles)

Parameters

- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **kwargs_ps** – keyword arguments corresponding to "other" parameters, such as external shear and point source image positions
- **inv_bool** – if True, invert the full linear solver Matrix Ax = y for the purpose of the covariance matrix.

Returns 1d array of surface brightness pixels of the optimal solution of the linear parameters to match the data

likelihood_data_given_model(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *source_marg=False*, *linear_prior=None*, *check_positive_flux=False*)

computes the likelihood of the data given a model This is specified with the non-linear parameters and a linear inversion and prior marginalisation.

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –
- **check_positive_flux** – bool, if True, checks whether the linear inversion resulted in non-negative flux components and applies a punishment in the likelihood if so.

Returns log likelihood (natural logarithm) (sum of the log likelihoods of the individual images)

lenstronomy.ImSim.MultiBand.single_band_multi_model module

class SingleBandMultiModel(*multi_band_list, kwargs_model, likelihood_mask_list=None, band_index=0, kwargs_pixelbased=None, linear_solver=True*)

Bases: `lenstronomy.ImSim.image_linear_solve.ImageLinearFit`

class to simulate/reconstruct images in multi-band option. This class calls functions of image_model.py with different bands with decoupled linear parameters and the option to pass/select different light models for the different bands

the class supports keyword arguments 'index_lens_model_list', 'index_source_light_model_list', 'index_lens_light_model_list', 'index_point_source_model_list', 'index_optical_depth_model_list' in kwargs_model These arguments should be lists of length the number of imaging bands available and each entry in the list is a list of integers specifying the model components being evaluated for the specific band.

E.g. there are two bands and you want to different light profiles being modeled. - you define two different light profiles lens_light_model_list = ['SERSIC', 'SERSIC'] - set index_lens_light_model_list = [[0], [1]] - (optional) for now all the parameters between the two light profiles are independent in the model. You have the possibility to join a subset of model parameters (e.g. joint centroid). See the Param() class for documentation.

__init__(*multi_band_list, kwargs_model, likelihood_mask_list=None, band_index=0, kwargs_pixelbased=None, linear_solver=True*)

Parameters

- **multi_band_list** – list of imaging band configurations [[kwargs_data, kwargs_psf, kwargs_numerics],[. . .], . . .]
- **kwargs_model** – model option keyword arguments
- **likelihood_mask_list** – list of likelihood masks (booleans with size of the individual images
- **band_index** – integer, index of the imaging band to model
- **kwargs_pixelbased** – keyword arguments with various settings related to the pixel-based solver (see SLITronomy documentation)
- **linear_solver** – bool, if True (default) fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

error_map_source(*kwargs_source, x_grid, y_grid, cov_param, model_index_select=True*)

variance of the linear source reconstruction in the source plane coordinates, computed by the diagonal elements of the covariance matrix of the source reconstruction as a sum of the errors of the basis set.

Parameters

- **kwargs_source** – keyword arguments of source model
- **x_grid** – x-axis of positions to compute error map
- **y_grid** – y-axis of positions to compute error map
- **cov_param** – covariance matrix of liner inversion parameters
- **model_index_select** – boolean, if True, selects the model components of this band (default). If False, assumes input kwargs_source is already selected list.

Returns diagonal covariance errors at the positions (x_grid, y_grid)

image_linear_solve(*kwargs_lens=None, kwargs_source=None, kwargs_lens_light=None, kwargs_ps=None, kwargs_extinction=None, kwargs_special=None, inv_bool=False*)

computes the image (lens and source surface brightness with a given lens model). The linear parameters are computed with a weighted linear least square optimization (i.e. flux normalization of the brightness profiles) :param kwargs_lens: list of keyword arguments corresponding to the superposition of different lens profiles :param kwargs_source: list of keyword arguments corresponding to the superposition of different source light profiles :param kwargs_lens_light: list of keyword arguments corresponding to different lens light surface brightness profiles :param kwargs_ps: keyword arguments corresponding to "other" parameters, such as external shear and point source image positions :param inv_bool: if True, invert the full linear solver Matrix Ax = y for the purpose of the covariance matrix. :return: 1d array of surface brightness pixels of the optimal solution of the linear parameters to match the data

likelihood_data_given_model(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *source_marg=False*, *linear_prior=None*, *check_positive_flux=False*)

computes the likelihood of the data given a model This is specified with the non-linear parameters and a linear inversion and prior marginalisation.

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –
- **check_positive_flux** – bool, if True, checks whether the linear inversion resulted in non-negative flux components and applies a punishment in the likelihood if so.

Returns log likelihood (natural logarithm) (sum of the log likelihoods of the individual images)

linear_response_matrix(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*)

computes the linear response matrix (m x n), with n beeing the data size and m being the coefficients

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –

Returns

num_param_linear(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*)

Returns number of linear coefficients to be solved for in the linear inversion

select_kwargs(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*)

select subset of kwargs lists referenced to this imaging band

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –

Returns

Module contents

lenstronomy.ImSim.Numerics package

Submodules

lenstronomy.ImSim.Numerics.adaptive_numerics module

class AdaptiveConvolution(*kernel_super*, *supersampling_factor*, *conv_supersample_pixels*, *supersampling_kernel_size=None*, *compute_pixels=None*, *nopython=True*, *cache=True*, *parallel=False*)

Bases: `object`

This class performs convolutions of a subset of pixels at higher supersampled resolution Goal: speed up relative to higher resolution FFT when only considering a (small) subset of pixels to be convolved on the higher resolution grid.

strategy: 1. lower resolution convolution over full image with FFT 2. subset of pixels with higher resolution Numba convolution (with smaller kernel) 3. the same subset of pixels with low resolution Numba convolution (with same kernel as step 2) adaptive solution is 1 + 2 - 3

__init__(*kernel_super*, *supersampling_factor*, *conv_supersample_pixels*, *supersampling_kernel_size=None*, *compute_pixels=None*, *nopython=True*, *cache=True*, *parallel=False*)

Parameters

- **kernel_super** – convolution kernel in units of super sampled pixels provided, odd length per axis
- **supersampling_factor** – factor of supersampling relative to pixel grid
- **conv_supersample_pixels** – bool array same size as data, pixels to be convolved and their light to be blurred
- **supersampling_kernel_size** – number of pixels (in units of the image pixels) that are convolved with the supersampled kernel
- **compute_pixels** – bool array of size of image, these pixels (if True) will get blurred light from other pixels
- **nopython** – bool, numba jit setting to use python or compiled.
- **cache** – bool, numba jit setting to use cache
- **parallel** – bool, numba jit setting to use parallel mode

convolve2d(*image_high_res*)

Parameters **image_high_res** – supersampled image/model to be convolved on a regular pixel grid

Returns convolved and re-sized image

re_size_convolve(*image_low_res*, *image_high_res*)

Parameters

- **image_low_res** – regular sampled image/model

- **image_high_res** – supersampled image/model to be convolved on a regular pixel grid

Returns convolved and re-sized image

lenstronomy.ImSim.Numerics.convolution module

class PixelKernelConvolution(*kernel*, *convolution_type='fft_static'*)

Bases: object

class to compute convolutions for a given pixelized kernel (fft, grid)

__init__(*kernel*, *convolution_type='fft_static'*)

Parameters

- **kernel** – 2d array, convolution kernel
- **convolution_type** – string, 'fft', 'grid', 'fft_static' mode of 2d convolution

convolution2d(*image*)

Parameters **image** – 2d array (image) to be convolved

Returns fft convolution

copy_transpose()

Returns copy of the class with kernel set to the transpose of original one

pixel_kernel(*num_pix=None*)

access pixelated kernel

Parameters **num_pix** – size of returned kernel (odd number per axis). If None, return the original kernel.

Returns pixel kernel centered

re_size_convolve(*image_low_res*, *image_high_res=None*)

Parameters

- **image_low_res** – regular sampled image/model
- **image_high_res** – supersampled image/model to be convolved on a regular pixel grid

Returns convolved and re-sized image

class SubgridKernelConvolution(*kernel_supersampled*, *supersampling_factor*, *supersampling_kernel_size=None*, *convolution_type='fft_static'*)

Bases: object

class to compute the convolution on a supersampled grid with partial convolution computed on the regular grid

__init__(*kernel_supersampled*, *supersampling_factor*, *supersampling_kernel_size=None*, *convolution_type='fft_static'*)

Parameters

- **kernel_supersampled** – kernel in supersampled pixels
- **supersampling_factor** – supersampling factor relative to the image pixel grid
- **supersampling_kernel_size** – number of pixels (in units of the image pixels) that are convolved with the supersampled kernel

convolution2d(*image*)

Parameters **image** – 2d array (high resoluton image) to be convolved and re-sized

Returns convolved image

re_size_convolve(*image_low_res*, *image_high_res*)

Parameters **image_high_res** – supersampled image/model to be convolved on a regular pixel grid

Returns convolved and re-sized image

class MultiGaussianConvolution(*sigma_list*, *fraction_list*, *pixel_scale*, *supersampling_factor=1*, *supersampling_convolution=False*, *truncation=2*)

Bases: `object`

class to perform a convolution consisting of multiple 2d Gaussians This is aimed to lead to a speed-up without significant loss of accuracy do to the simplified convolution kernel relative to a pixelized kernel.

__init__(*sigma_list*, *fraction_list*, *pixel_scale*, *supersampling_factor=1*, *supersampling_convolution=False*, *truncation=2*)

Parameters

- **sigma_list** – list of std value of Gaussian kernel
- **fraction_list** – fraction of flux to be convoled with each Gaussian kernel
- **pixel_scale** – scale of pixel width (to convert sigmas into units of pixels)
- **truncation** – float. Truncate the filter at this many standard deviations. Default is 4.0.

convolution2d(*image*)

2d convolution

Parameters **image** – 2d numpy array, image to be convolved

Returns convolved image, 2d numpy array

pixel_kernel(*num_pix*)

computes a pixelized kernel from the MGE parameters

Parameters **num_pix** – int, size of kernel (odd number per axis)

Returns pixel kernel centered

re_size_convolve(*image_low_res*, *image_high_res*)

Parameters **image_high_res** – supersampled image/model to be convolved on a regular pixel grid

Returns convolved and re-sized image

class FWHMGaussianConvolution(*kernel*, *truncation=4*)

Bases: `object`

uses a two-dimensional Gaussian function with same FWHM of given kernel as approximation

__init__(*kernel*, *truncation=4*)

Parameters

- **kernel** – 2d kernel
- **truncation** – sigma scaling of kernel truncation

convolution2d(*image*)

2d convolution

Parameters **image** – 2d numpy array, image to be convolved

Returns convolved image, 2d numpy array

class MGEConvolution(*kernel*, *pixel_scale*, *order=1*)

Bases: `object`

approximates a 2d kernel with an azimuthal Multi-Gaussian expansion

__init__(*kernel*, *pixel_scale*, *order=1*)

Parameters

- **kernel** – 2d convolution kernel (centered, odd axis number)
- **order** – order of Multi-Gaussian Expansion

convolution2d(*image*)

Parameters **image** –

Returns

kernel_difference()

Returns difference between true kernel and MGE approximation

lenstronomy.ImSim.Numerics.grid module

class AdaptiveGrid(*nx*, *ny*, *transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*, *supersampling_indexes*, *supersampling_factor*, *flux_evaluate_indexes=None*)

Bases: `lenstronomy.Data.coord_transforms.Coordinates1D`

manages a super-sampled grid on the partial image

__init__(*nx*, *ny*, *transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*, *supersampling_indexes*, *supersampling_factor*, *flux_evaluate_indexes=None*)

Parameters

- **nx** – number of pixels in x-axis
- **ny** – number of pixels in y-axis
- **transform_pix2angle** – 2x2 matrix, mapping of pixel to coordinate
- **ra_at_xy_0** – ra coordinate at pixel (0,0)
- **dec_at_xy_0** – dec coordinate at pixel (0,0)
- **supersampling_indexes** – bool array of shape nx x ny, corresponding to pixels being super_sampled
- **supersampling_factor** – int, factor (per axis) of super-sampling
- **flux_evaluate_indexes** – bool array of shape nx x ny, corresponding to pixels being evaluated (for both low and high res). Default is None, replaced by setting all pixels to being evaluated.

coordinates_evaluate

Returns 1d array of all coordinates being evaluated to perform the image computation

flux_array2image_low_high(*flux_array*, *high_res_return=True*)

Parameters

- **flux_array** – 1d array of low and high resolution flux values corresponding to the coordinates_evaluate order

- **high_res_return** – bool, if True also returns the high resolution image (needs more computation and is only needed when convolution is performed on the supersampling level)

Returns 2d array, 2d array, corresponding to (partial) images in low and high resolution (to be convolved)

class RegularGrid(*nx*, *ny*, *transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*, *supersampling_factor=1*, *flux_evaluate_indexes=None*)

Bases: `lenstronomy.Data.coord_transforms.Coordinates1D`

manages a super-sampled grid on the partial image

__init__(*nx*, *ny*, *transform_pix2angle*, *ra_at_xy_0*, *dec_at_xy_0*, *supersampling_factor=1*, *flux_evaluate_indexes=None*)

Parameters

- **nx** – number of pixels in x-axis
- **ny** – number of pixels in y-axis
- **transform_pix2angle** – 2x2 matrix, mapping of pixel to coordinate
- **ra_at_xy_0** – ra coordinate at pixel (0,0)
- **dec_at_xy_0** – dec coordinate at pixel (0,0)
- **supersampling_factor** – int, factor (per axis) of super-sampling
- **flux_evaluate_indexes** – bool array of shape nx x ny, corresponding to pixels being evaluated (for both low and high res). Default is None, replaced by setting all pixels to being evaluated.

coordinates_evaluate

Returns 1d array of all coordinates being evaluated to perform the image computation

flux_array2image_low_high(*flux_array*, ***kwargs*)

Parameters flux_array – 1d array of low and high resolution flux values corresponding to the coordinates_evaluate order

Returns 2d array, 2d array, corresponding to (partial) images in low and high resolution (to be convolved)

grid_points_spacing

effective spacing between coordinate points, after supersampling :return: sqrt(pixel_area)/supersampling_factor

num_grid_points_axes

effective number of points along each axes, after supersampling :return: number of pixels per axis, nx*supersampling_factor ny*supersampling_factor

supersampling_factor

Returns factor (per axis) of super-sampling relative to a pixel

lenstronomy.ImSim.Numerics.numba_convolution module

class NumbaConvolution(*kernel*, *conv_pixels*, *compute_pixels=None*, *nopython=True*, *cache=True*, *parallel=False*, *memory_raise=True*)

Bases: `object`

class to convolve explicit pixels only

the convolution is inspired by pyautolens: https://github.com/Jammy2211/PyAutoLens

__init__(*kernel*, *conv_pixels*, *compute_pixels=None*, *nopython=True*, *cache=True*, *parallel=False*, *memory_raise=True*)

Parameters

- **kernel** – convolution kernel in units of the image pixels provided, odd length per axis
- **conv_pixels** – bool array same size as data, pixels to be convolved and their light to be blurred
- **compute_pixels** – bool array of size of image, these pixels (if True) will get blurred light from other pixels
- **nopython** – bool, numba jit setting to use python or compiled.
- **cache** – bool, numba jit setting to use cache
- **parallel** – bool, numba jit setting to use parallel mode
- **memory_raise** – bool, if True, checks whether memory required to store the convolution kernel is within certain bounds

convolve2d(*image*)

2d convolution

Parameters **image** – 2d numpy array, image to be convolved

Returns convolved image, 2d numpy array

lenstronomy.ImSim.Numerics.numerics module

class Numerics(*pixel_grid*, *psf*, *supersampling_factor=1*, *compute_mode='regular'*, *supersampling_convolution=False*, *supersampling_kernel_size=5*, *flux_evaluate_indexes=None*, *supersampled_indexes=None*, *compute_indexes=None*, *point_source_supersampling_factor=1*, *convolution_kernel_size=None*, *convolution_type='fft_static'*, *truncation=4*)

Bases: lenstronomy.ImSim.Numerics.point_source_rendering.PointSourceRendering

this classes manages the numerical options and computations of an image. The class has two main functions, re_size_convolve() and coordinates_evaluate()

__init__(*pixel_grid*, *psf*, *supersampling_factor=1*, *compute_mode='regular'*, *supersampling_convolution=False*, *supersampling_kernel_size=5*, *flux_evaluate_indexes=None*, *supersampled_indexes=None*, *compute_indexes=None*, *point_source_supersampling_factor=1*, *convolution_kernel_size=None*, *convolution_type='fft_static'*, *truncation=4*)

Parameters

- **pixel_grid** – PixelGrid() class instance
- **psf** – PSF() class instance
- **compute_mode** – options are: 'regular', 'adaptive'
- **supersampling_factor** – int, factor of higher resolution sub-pixel sampling of surface brightness

- **supersampling_convolution** – bool, if True, performs (part of) the convolution on the super-sampled grid/pixels
- **supersampling_kernel_size** – int (odd number), size (in regular pixel units) of the super-sampled convolution
- **flux_evaluate_indexes** – boolean 2d array of size of image (or None, then initiated as gird of True's). Pixels indicated with True will be used to perform the surface brightness computation (and possible lensing ray-shooting). Pixels marked as False will be assigned a flux value of zero (or ignored in the adaptive convolution)
- **supersampled_indexes** – 2d boolean array (only used in mode='adaptive') of pixels to be supersampled (in surface brightness and if supersampling_convolution=True also in convolution). All other pixels not set to =True will not be super-sampled.
- **compute_indexes** – 2d boolean array (only used in compute_mode='adaptive'), marks pixel that the response after convolution is computed (all others =0). This can be set to likelihood_mask in the Likelihood module for consistency.
- **point_source_supersampling_factor** – super-sampling resolution of the point source placing
- **convolution_kernel_size** – int, odd number, size of convolution kernel. If None, takes size of point_source_kernel
- **convolution_type** – string, 'fft', 'grid', 'fft_static' mode of 2d convolution

convolution_class

Returns convolution class (can be SubgridKernelConvolution, PixelKernelConvolution, MultiGaussianConvolution, . . .)

coordinates_evaluate

Returns 1d array of all coordinates being evaluated to perform the image computation

grid_class

Returns grid class (can be RegularGrid, AdaptiveGrid)

grid_supersampling_factor

Returns supersampling factor set for higher resolution sub-pixel sampling of surface brightness

re_size_convolve (*flux_array*, *unconvolved=False*)

Parameters

- **flux_array** – 1d array, flux values corresponding to coordinates_evaluate
- **unconvolved** – boolean, if True, does not apply a convolution

Returns convolved image on regular pixel grid, 2d array

lenstronomy.ImSim.Numerics.partial_image module

class PartialImage (*partial_read_bools*)

Bases: `object`

class to deal with the use of partial slicing of a 2d data array, to be used for various computations where only a subset of pixels need to be know.

__init__ (*partial_read_bools*)

Parameters **partial_read_bools** – 2d numpy array of bools indicating which indexes to be processed

array_from_partial (*partial_array*)

Parameters **partial_array** – 1d array of the partial indexes

Returns full 1d array

image_from_partial (*partial_array*)

Parameters **partial_array** – 1d array corresponding to the indexes of the partial read

Returns full image with zeros elsewhere

index_array

Returns 2d array with indexes (integers) corresponding to the 1d array, -1 when masked

num_partial

Returns number of indexes handled in the partial section

partial_array (*image*)

Parameters **image** – 2d array

Returns 1d array of partial list

lenstronomy.ImSim.Numerics.point_source_rendering module

class PointSourceRendering (*pixel_grid, supersampling_factor, psf*)

Bases: `object`

numerics to compute the point source response on an image

__init__ (*pixel_grid, supersampling_factor, psf*)

Parameters

- **pixel_grid** – PixelGrid() instance
- **supersampling_factor** – int, factor of supersampling of point source
- **psf** – PSF() instance

point_source_rendering (*ra_pos, dec_pos, amp*)

Parameters

- **ra_pos** – list of RA positions of point source(s)
- **dec_pos** – list of DEC positions of point source(s)
- **amp** – list of amplitudes of point source(s)

Returns 2d numpy array of size of the image with the point source(s) rendered

psf_error_map (*ra_pos, dec_pos, amp, data, fix_psf_error_map=False*)

Parameters

- **ra_pos** – image positions of point sources
- **dec_pos** – image positions of point sources
- **amp** – amplitude of modeled point sources

- **data** – 2d numpy array of the data
- **fix_psf_error_map** – bool, if True, estimates the error based on the input (modeled) amplitude, else uses the data to do so.

Returns 2d array of size of the image with error terms (sigma**2) expected from inaccuracies in the PSF modeling

Module contents

Submodules

lenstronomy.ImSim.de_lens module

get_param_WLS (*A*, *C_D_inv*, *d*, *inv_bool=True*)
returns the parameter values given

Parameters

- **A** – response matrix Nd x Ns (Nd = # data points, Ns = # parameters)
- **C_D_inv** – inverse covariance matrix of the data, Nd x Nd, diagonal form
- **d** – data array, 1-d Nd
- **inv_bool** – boolean, whether returning also the inverse matrix or just solve the linear system

Returns 1-d array of parameter values

marginalisation_const (*M_inv*)
get marginalisation constant 1/2 log(M_beta) for flat priors

Parameters M_inv – 2D covariance matrix

Returns float

marginalization_new (*M_inv*, *d_prior=None*)

Parameters

- **M_inv** – 2D covariance matrix
- **d_prior** – maximum prior length of linear parameters

Returns log determinant with eigenvalues to be smaller or equal d_prior

lenstronomy.ImSim.image2source_mapping module

class Image2SourceMapping (*lensModel*, *sourceModel*)
Bases: `object`

this class handles multiple source planes and performs the computation of predicted surface brightness at given image positions. The class is enable to deal with an arbitrary number of different source planes. There are two different settings:

Single lens plane modelling: In case of a single deflector, lenstronomy models the reduced deflection angles (matched to the source plane in single source plane mode). Each source light model can be added a number (scale_factor) that rescales the reduced deflection angle to the specific source plane.

Multiple lens plane modelling: The multi-plane lens modelling requires the assumption of a cosmology and the redshifts of the multiple lens and source planes. The backwards ray-tracing is performed and stopped at the different source plane redshift to compute the mapping between source to image plane.

__init__(*lensModel*, *sourceModel*)

Parameters

- **lensModel** – LensModel() class instance
- **sourceModel** – LightModel() class instance.

  The lightModel includes:

  – source_scale_factor_list: list of floats corresponding to the rescaled deflection angles to the specific source components. None indicates that the list will be set to 1, meaning a single source plane model (in single lens plane mode).

  – source_redshift_list: list of redshifts of the light components (in multi lens plane mode)

image2source(*x*, *y*, *kwargs_lens*, *index_source*)

mapping of image plane to source plane coordinates WARNING: for multi lens plane computations and multi source planes, this computation can be slow and should be used as rarely as possible.

Parameters

- **x** – image plane coordinate (angle)
- **y** – image plane coordinate (angle)
- **kwargs_lens** – lens model kwargs list
- **index_source** – int, index of source model

Returns source plane coordinate corresponding to the source model of index idex_source

image_flux_joint(*x*, *y*, *kwargs_lens*, *kwargs_source*, *k=None*)

Parameters

- **x** – coordinate in image plane
- **y** – coordinate in image plane
- **kwargs_lens** – lens model kwargs list
- **kwargs_source** – source model kwargs list
- **k** – None or int or list of int for partial evaluation of light models

Returns surface brightness of all joint light components at image position (x, y)

image_flux_split(*x*, *y*, *kwargs_lens*, *kwargs_source*)

Parameters

- **x** – coordinate in image plane
- **y** – coordinate in image plane
- **kwargs_lens** – lens model kwargs list
- **kwargs_source** – source model kwargs list

Returns list of responses of every single basis component with default amplitude amp=1, in the same order as the light_model_list

lenstronomy.ImSim.image_linear_solve module

class ImageLinearFit(*data_class*, *psf_class=None*, *lens_model_class=None*, *source_model_class=None*, *lens_light_model_class=None*, *point_source_class=None*, *extinction_class=None*, *kwargs_numerics=None*, *likelihood_mask=None*, *psf_error_map_bool_list=None*, *kwargs_pixelbased=None*)

Bases: `lenstronomy.ImSim.image_model.ImageModel`

linear version class, inherits ImageModel.

When light models use pixel-based profile types, such as 'SLIT_STARLETS', the WLS linear inversion is replaced by the regularized inversion performed by an external solver. The current pixel-based solver is provided by the SLITronomy plug-in.

__init__(*data_class*, *psf_class=None*, *lens_model_class=None*, *source_model_class=None*, *lens_light_model_class=None*, *point_source_class=None*, *extinction_class=None*, *kwargs_numerics=None*, *likelihood_mask=None*, *psf_error_map_bool_list=None*, *kwargs_pixelbased=None*)

Parameters

- **data_class** – ImageData() instance
- **psf_class** – PSF() instance
- **lens_model_class** – LensModel() instance
- **source_model_class** – LightModel() instance
- **lens_light_model_class** – LightModel() instance
- **point_source_class** – PointSource() instance
- **kwargs_numerics** – keyword arguments passed to the Numerics module
- **likelihood_mask** – 2d boolean array of pixels to be counted in the likelihood calculation/linear optimization
- **psf_error_map_bool_list** – list of boolean of length of point source models. Indicates whether PSF error map is used for the point source model stated as the index.
- **kwargs_pixelbased** – keyword arguments with various settings related to the pixel-based solver (see SLITronomy documentation) being applied to the point sources.

array_masked2image(*array*)

Parameters **array** – 1d array of values not masked out (part of linear fitting)

Returns 2d array of full image

check_positive_flux(*kwargs_source*, *kwargs_lens_light*, *kwargs_ps*)

checks whether the surface brightness profiles contain positive fluxes and returns bool if True

Parameters

- **kwargs_source** – source surface brightness keyword argument list
- **kwargs_lens_light** – lens surface brightness keyword argument list
- **kwargs_ps** – point source keyword argument list

Returns boolean

data_response

returns the 1d array of the data element that is fitted for (including masking)

Returns 1d numpy array

error_map_source (*kwargs_source*, *x_grid*, *y_grid*, *cov_param*)
variance of the linear source reconstruction in the source plane coordinates, computed by the diagonal elements of the covariance matrix of the source reconstruction as a sum of the errors of the basis set.

Parameters

- **kwargs_source** – keyword arguments of source model
- **x_grid** – x-axis of positions to compute error map
- **y_grid** – y-axis of positions to compute error map
- **cov_param** – covariance matrix of liner inversion parameters

Returns diagonal covariance errors at the positions (x_grid, y_grid)

error_response (*kwargs_lens*, *kwargs_ps*, *kwargs_special*)
returns the 1d array of the error estimate corresponding to the data response

Returns 1d numpy array of response, 2d array of additonal errors (e.g. point source uncertainties)

image2array_masked (*image*)
returns 1d array of values in image that are not masked out for the likelihood computation/linear minimization :param image: 2d numpy array of full image :return: 1d array

image_linear_solve (*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *inv_bool=False*)
computes the image (lens and source surface brightness with a given lens model). The linear parameters are computed with a weighted linear least square optimization (i.e. flux normalization of the brightness profiles) However in case of pixel-based modelling, pixel values are constrained by an external solver (e.g. SLITronomy).

Parameters

- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **kwargs_ps** – keyword arguments corresponding to "other" parameters, such as external shear and point source image positions
- **inv_bool** – if True, invert the full linear solver Matrix Ax = y for the purpose of the covariance matrix.

Returns 2d array of surface brightness pixels of the optimal solution of the linear parameters to match the data

image_pixelbased_solve (*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *init_lens_light_model=None*)
computes the image (lens and source surface brightness with a given lens model) using the pixel-based solver.

Parameters

- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **kwargs_ps** – keyword arguments corresponding to point sources
- **kwargs_extinction** – keyword arguments corresponding to dust extinction
- **kwargs_special** – keyword arguments corresponding to "special" parameters
- **init_lens_light_model** – optional initial guess for the lens surface brightness

Returns 2d array of surface brightness pixels of the optimal solution of the linear parameters to match the data

likelihood_data_given_model(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *source_marg=False*, *linear_prior=None*, *check_positive_flux=False*)

computes the likelihood of the data given a model This is specified with the non-linear parameters and a linear inversion and prior marginalisation.

Parameters

- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **kwargs_ps** – keyword arguments corresponding to "other" parameters, such as external shear and point source image positions
- **kwargs_extinction** –
- **kwargs_special** –
- **source_marg** – bool, performs a marginalization over the linear parameters
- **linear_prior** – linear prior width in eigenvalues
- **check_positive_flux** – bool, if True, checks whether the linear inversion resulted in non-negative flux components and applies a punishment in the likelihood if so.

Returns log likelihood (natural logarithm)

linear_param_from_kwargs(*kwargs_source*, *kwargs_lens_light*, *kwargs_ps*)

inverse function of update_linear() returning the linear amplitude list for the keyword argument list

Parameters

- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –

Returns list of linear coefficients

linear_response_matrix(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*)
computes the linear response matrix (m x n), with n being the data size and m being the coefficients

Parameters

- **kwargs_lens** – lens model keyword argument list
- **kwargs_source** – extended source model keyword argument list
- **kwargs_lens_light** – lens light model keyword argument list
- **kwargs_ps** – point source model keyword argument list
- **kwargs_extinction** – extinction model keyword argument list
- **kwargs_special** – special keyword argument list

Returns linear response matrix

num_data_evaluate
number of data points to be used in the linear solver :return:

num_param_linear(*kwargs_lens*, *kwargs_source*, *kwargs_lens_light*, *kwargs_ps*)

Returns number of linear coefficients to be solved for in the linear inversion

point_source_linear_response_set(*kwargs_ps*, *kwargs_lens*, *kwargs_special*, *with_amp=True*)

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list
- **kwargs_special** – special keyword argument list, may include 'delta_x_image' and 'delta_y_image'
- **with_amp** – bool, if True, relative magnification between multiply imaged point sources are held fixed.

Returns list of positions and amplitudes split in different basis components with applied astrometric corrections

reduced_chi2(*model*, *error_map=0*)
returns reduced chi2 :param model: 2d numpy array of a model predicted image :param error_map: same format as model, additional error component (such as PSF errors) :return: reduced chi2

reduced_residuals(*model*, *error_map=0*)

Parameters

- **model** – 2d numpy array of the modeled image
- **error_map** – 2d numpy array of additional noise/error terms from model components (such as PSF model uncertainties)

Returns 2d numpy array of reduced residuals per pixel

update_data(*data_class*)

Parameters **data_class** – instance of Data() class

Returns no return. Class is updated.

update_linear_kwargs(*param*, *kwargs_lens*, *kwargs_source*, *kwargs_lens_light*, *kwargs_ps*)
links linear parameters to kwargs arguments

Parameters **param** – linear parameter vector corresponding to the response matrix

Returns updated list of kwargs with linear parameter values

update_pixel_kwargs(*kwargs_source*, *kwargs_lens_light*)

Update kwargs arguments for pixel-based profiles with fixed properties such as their number of pixels, scale, and center coordinates (fixed to the origin).

Parameters

- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to the superposition of different lens light profiles

Returns updated kwargs_source and kwargs_lens_light

lenstronomy.ImSim.image_model module

class ImageModel(*data_class*, *psf_class*, *lens_model_class=None*, *source_model_class=None*, *lens_light_model_class=None*, *point_source_class=None*, *extinction_class=None*, *kwargs_numerics=None*, *kwargs_pixelbased=None*)

Bases: `object`

this class uses functions of lens_model and source_model to make a lensed image

__init__(*data_class*, *psf_class*, *lens_model_class=None*, *source_model_class=None*, *lens_light_model_class=None*, *point_source_class=None*, *extinction_class=None*, *kwargs_numerics=None*, *kwargs_pixelbased=None*)

Parameters

- **data_class** – instance of ImageData() or PixelGrid() class
- **psf_class** – instance of PSF() class
- **lens_model_class** – instance of LensModel() class
- **source_model_class** – instance of LightModel() class describing the source parameters
- **lens_light_model_class** – instance of LightModel() class describing the lens light parameters
- **point_source_class** – instance of PointSource() class describing the point sources
- **kwargs_numerics** – keyword arguments with various numeric description (see ImageNumerics class for options)
- **kwargs_pixelbased** – keyword arguments with various settings related to the pixel-based solver (see SLITronomy documentation)

extinction_map(*kwargs_extinction=None*, *kwargs_special=None*)

differential extinction per pixel

Parameters

- **kwargs_extinction** – list of keyword arguments corresponding to the optical depth models tau, such that extinction is exp(-tau)
- **kwargs_special** – keyword arguments, additional parameter to the extinction

Returns 2d array of size of the image

image(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_extinction=None*, *kwargs_special=None*, *unconvolved=False*, *source_add=True*, *lens_light_add=True*, *point_source_add=True*)

make an image with a realisation of linear parameter values "param"

Parameters

- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **kwargs_ps** – keyword arguments corresponding to "other" parameters, such as external shear and point source image positions
- **unconvolved** – if True: returns the unconvolved light distribution (prefect seeing)
- **source_add** – if True, compute source, otherwise without
- **lens_light_add** – if True, compute lens light, otherwise without
- **point_source_add** – if True, add point sources, otherwise without

Returns 2d array of surface brightness pixels of the simulation

lens_surface_brightness(*kwargs_lens_light*, *unconvolved=False*, *k=None*)

computes the lens surface brightness distribution

Parameters

- **kwargs_lens_light** – list of keyword arguments corresponding to different lens light surface brightness profiles
- **unconvolved** – if True, returns unconvolved surface brightness (perfect seeing), otherwise convolved with PSF kernel

Returns 2d array of surface brightness pixels

point_source(*kwargs_ps*, *kwargs_lens=None*, *kwargs_special=None*, *unconvolved=False*, *k=None*)

computes the point source positions and paints PSF convolutions on them

Parameters

- **kwargs_ps** –
- **k** –

Returns

reset_point_source_cache(*cache=True*)

deletes all the cache in the point source class and saves it from then on

Parameters **cache** – boolean, if True, saves the next occuring point source positions in the cache

Returns None

source_surface_brightness(*kwargs_source*, *kwargs_lens=None*, *kwargs_extinction=None*, *kwargs_special=None*, *unconvolved=False*, *de_lensed=False*, *k=None*, *update_pixelbased_mapping=True*)

computes the source surface brightness distribution

Parameters

- **kwargs_source** – list of keyword arguments corresponding to the superposition of different source light profiles
- **kwargs_lens** – list of keyword arguments corresponding to the superposition of different lens profiles
- **kwargs_extinction** – list of keyword arguments of extinction model
- **unconvolved** – if True: returns the unconvolved light distribution (prefect seeing)
- **de_lensed** – if True: returns the un-lensed source surface brightness profile, otherwise the lensed.
- **k** – integer, if set, will only return the model of the specific index

Returns 2d array of surface brightness pixels

update_psf (*psf_class*)
update the instance of the class with a new instance of PSF() with a potentially different point spread function

Parameters psf_class –

Returns no return. Class is updated.

Module contents

lenstronomy.LensModel package

Subpackages

lenstronomy.LensModel.LightConeSim package

Submodules

lenstronomy.LensModel.LightConeSim.light_cone module

class LightCone (*mass_map_list*, *grid_spacing_list*, *redshift_list*)
Bases: `object`

class to perform multi-plane ray-tracing from convergence maps at different redshifts From the convergence maps the deflection angles and lensing potential are computed (from different settings) and then an interpolated grid of all those quantities generate an instance of the lenstronomy LensModel multi-plane instance. All features of the LensModel module are supported.

Improvements that can be made for accuracy and speed: 1. adaptive mesh integral for the convergence map 2. Interpolated deflection map on different scales than the mass map.

The design principles should allow those implementations 'under the hook' of this class.

__init__ (*mass_map_list*, *grid_spacing_list*, *redshift_list*)

Parameters

- **mass_map_list** – 2d numpy array of mass map (in units physical Solar masses enclosed in each pixel/gird point of the map)
- **grid_spacing_list** – list of grid spacing of the individual mass maps in units of physical Mpc

- **redshift_list** – list of redshifts of the mass maps

cone_instance (*z_source*, *cosmo*, *multi_plane=True*, *kwargs_interp=None*)

Parameters

- **z_source** – redshift to where lensing quantities are computed
- **cosmo** – astropy.cosmology class
- **multi_plane** – boolean, if True, computes multi-plane ray-tracing
- **kwargs_interp** – interpolation keyword arguments specifying the numerics. See description in the Interpolate() class. Only applicable for 'INTERPOL' and 'INTERPOL_SCALED' models.

Returns LensModel instance, keyword argument list of lens model

class MassSlice (*mass_map*, *grid_spacing*, *redshift*)

Bases: `object`

class to describe a single mass slice

__init__ (*mass_map*, *grid_spacing*, *redshift*)

Parameters

- **mass_map** – 2d numpy array of mass map (in units physical Msol)
- **grid_spacing** – grid spacing of the mass map (in units physical Mpc)
- **redshift** – redshift

interpol_instance (*z_source*, *cosmo*)

scales the mass map integrals (with units of mass not convergence) into a convergence map for the given cosmology and source redshift and returns the keyword arguments of the interpolated reduced deflection and lensing potential.

Parameters

- **z_source** – redshift of the source
- **cosmo** – astropy.cosmology instance

Returns keyword arguments of the interpolation instance with numerically computed deflection angles and lensing potential

Module contents

lenstronomy.LensModel.LineOfSight package

Subpackages

lenstronomy.LensModel.LineOfSight.LOSModels package

Submodules

lenstronomy.LensModel.LineOfSight.LOSModels.los module

class LOS (**args*, ***kwargs*)

Bases: `object`

Class allowing one to add tidal line-of-sight effects (convergence and shear) to single-plane lensing. Stricly speaking, this is not a profile, but when present in list of lens models, it is automatically recognised by ModelAPI(), which sets the flag los_effects to True, and thereby leads LensModel to use SinglePlaneLOS() instead of SinglePlane(). It is however incompatible with MultiPlane().

The key-word arguments are the three line-of-sight convergences, the two components of the three line-of-sight shears, and the three line-of-sight rotations, all defined with the convention of https://arxiv.org/abs/2104.08883: kappa_od, kappa_os, kappa_ds, gamma1_od, gamma2_od, gamma1_os, gamma2_os, gamma1_ds, gamma2_ds, omega_od, omega_os, omega_ds

Because LOS is not a profile, it does not contain the usual functions function(), derivatives(), and hessian(), but rather modifies the behaviour of those functions in the SinglePlaneLOS() class.

Instead, it contains the essential building blocks of this modification.

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

static distort_vector(*x*, *y*, *kappa=0*, *gamma1=0*, *gamma2=0*, *omega=0*)
This function applies a distortion matrix to a vector (x, y) and returns (x', y') as follows:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1-\kappa-\gamma_1 & -\gamma_2+\omega \\ -\gamma_2-\omega & 1-\kappa+\gamma_1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

Parameters

- **x** – x-component of the vector to which the distortion matrix is applied
- **y** – y-component of the vector to which the distortion matrix is applied
- **kappa** – the convergence
- **gamma1** – the first shear component
- **gamma2** – the second shear component
- **omega** – the rotation

Returns the distorted vector

static left_multiply(*f_xx*, *f_xy*, *f_yx*, *f_yy*, *kappa=0*, *gamma1=0*, *gamma2=0*, *omega=0*)
Left-multiplies the Hessian matrix of a lens with a distortion matrix with convergence kappa, shear gamma1, gamma2, and rotation omega:

$$\mathsf{H}' = \begin{pmatrix} 1-\kappa-\gamma_1 & -\gamma_2+\omega \\ -\gamma_2-\omega & 1-\kappa+\gamma_1 \end{pmatrix} \mathsf{H}$$

Parameters

- **f_xx** – the i, i element of the Hessian matrix
- **f_xy** – the i, j element of the Hessian matrix
- **f_yx** – the j, i element of the Hessian matrix
- **f_yy** – the j, j element of the Hessian matrix
- **kappa** – the convergence
- **gamma1** – the first shear component
- **gamma2** – the second shear component
- **omega** – the rotation

Returns the Hessian left-multiplied by the distortion matrix

lower_limit_default = {'gamma1_ds': -0.5, 'gamma1_od': -0.5, 'gamma1_os': -0.5, 'ga

param_names = ['kappa_od', 'kappa_os', 'kappa_ds', 'gamma1_od', 'gamma2_od', 'gamma1_o

static right_multiply(*f_xx*, *f_xy*, *f_yx*, *f_yy*, *kappa=0*, *gamma1=0*, *gamma2=0*, *omega=0*)

Right-multiplies the Hessian matrix of a lens with a distortion matrix with convergence kappa and shear gamma1, gamma2:

$$\mathsf{H}' = \mathsf{H} \begin{pmatrix} 1-\kappa-\gamma_1 & -\gamma_2+\omega \\ -\gamma_2-\omega & 1-\kappa+\gamma_1 \end{pmatrix}$$

Parameters

- **f_xx** – the i, i element of the Hessian matrix
- **f_xy** – the i, j element of the Hessian matrix
- **f_yx** – the j, i element of the Hessian matrix
- **f_yy** – the j, j element of the Hessian matrix
- **kappa** – the convergence
- **gamma1** – the first shear component
- **gamma2** – the second shear component
- **omega** – the rotation

Returns the Hessian right-multiplied by the distortion matrix

set_dynamic()

Returns no return, deletes pre-computed variables for certain lens models

set_static(***kwargs*)

pre-computes certain computations that do only relate to the lens model parameters and not to the specific position where to evaluate the lens model

Parameters kwargs – lens model parameters

Returns no return, for certain lens model some private self variables are initiated

upper_limit_default = {'gamma1_ds': 0.5, 'gamma1_od': 0.5, 'gamma1_os': 0.5, 'gamma

lenstronomy.LensModel.LineOfSight.LOSModels.los_minimal module

class LOSMinimal(**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.LineOfSight.LOSModels.los.LOS*

Class deriving from LOS containing the parameters for line-of-sight corrections within the "minimal model" defined in https://arxiv.org/abs/2104.08883 It is equivalent to LOS but with fewer parameters, namely: kappa_od, gamma1_od, gamma2_od, omega_od, kappa_los, gamma1_los, gamma2_los, omega_los.

lower_limit_default = {'gamma1_los': -0.5, 'gamma1_od': -0.5, 'gamma2_los': -0.5, '

param_names = ['kappa_od', 'gamma1_od', 'gamma2_od', 'omega_od', 'kappa_los', 'gamma1_

upper_limit_default = {'gamma1_los': 0.5, 'gamma1_od': 0.5, 'gamma2_los': 0.5, 'gam

Module contents

Submodules

lenstronomy.LensModel.LineOfSight.single_plane_los module

class SinglePlaneLOS(*lens_model_list*, *index_los*, *numerical_alpha_class=None*, *lens_redshift_list=None*, *z_source_convention=None*, *kwargs_interp=None*)

Bases: `lenstronomy.LensModel.single_plane.SinglePlane`

This class is based on the 'SinglePlane' class, modified to include line-of-sight effects as presented by Fleury et al. in 2104.08883.

Are modified: - init (to include a new attribute, self.los) - fermat potential - alpha - hessian

Are unchanged (inherited from SinglePlane): - ray_shooting, because it calls the modified alpha - mass_2d, mass_3d, density which refer to the main lens without LOS corrections.

__init__(*lens_model_list*, *index_los*, *numerical_alpha_class=None*, *lens_redshift_list=None*, *z_source_convention=None*, *kwargs_interp=None*)

Instance of SinglePlaneLOS() based on the SinglePlane(), except: - argument "index_los" indicating the position of the LOS model in the lens_model_list (for correct association with kwargs) - attribute "los" containing the LOS model.

alpha(*x*, *y*, *kwargs*, *k=None*)

Displacement angle including the line-of-sight corrections

Parameters

- **x** (`numpy array`) – x-position (preferentially arcsec)
- **y** (`numpy array`) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes, including line-of-sight corrections
- **k** – only evaluate the k-th lens model

Returns deflection angles in units of arcsec

density(*r*, *kwargs*, *bool_list=None*)

3d mass density at radius r *for the main lens only* The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius (in angular units)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **bool_list** – list of bools that are part of the output

Returns mass density at radius r (in angular units, modulo epsilon_crit)

fermat_potential(*x_image*, *y_image*, *kwargs_lens*, *x_source=None*, *y_source=None*, *k=None*)

Calculates the Fermat Potential with LOS corrections in the tidal regime

Parameters

- **x_image** – image position
- **y_image** – image position

- **x_source** – source position
- **y_source** – source position
- **kwargs_lens** – list of keyword arguments of lens model parameters matching the lens model classes

Returns fermat potential in arcsec**2 as a list

hessian (*x*, *y*, *kwargs*, *k=None*)
Hessian matrix

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model

Returns f_xx, f_xy, f_yx, f_yy components

mass_2d (*r*, *kwargs*, *bool_list=None*)
Computes the mass enclosed a projected (2d) radius r *for the main lens only*

The mass definition is such that:

$$\alpha = mass_2 d/r/\pi$$

with alpha is the deflection angle

Parameters

- **r** – radius (in angular units)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **bool_list** – list of bools that are part of the output

Returns projected mass (in angular units, modulo epsilon_crit)

mass_3d (*r*, *kwargs*, *bool_list=None*)
Computes the mass within a 3d sphere of radius r *for the main lens only*

Parameters

- **r** – radius (in angular units)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **bool_list** – list of bools that are part of the output

Returns mass (in angular units, modulo epsilon_crit)

potential (*x*, *y*, *kwargs*, *k=None*)
Lensing potential *of the main lens only* In the presence of LOS corrections, the system generally does not admit a potential, in the sense that the curl of alpha is generally non-zero

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)

- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model

Returns lensing potential in units of arcsec^2

split_lens_los(*kwargs*)

This function splits the list of key-word arguments given to the lens model into those that correspond to the lens itself (kwargs_main), and those that correspond to the line-of-sight corrections (kwargs_los).

Parameters **kwargs** – the list of key-word arguments passed to lenstronomy

Returns a list of kwargs corresponding to the lens and a list of kwargs corresponding to the LOS effects

Module contents

lenstronomy.LensModel.MultiPlane package

Submodules

lenstronomy.LensModel.MultiPlane.multi_plane module

class MultiPlane(*z_source*, *lens_model_list*, *lens_redshift_list*, *cosmo=None*, *numerical_alpha_class=None*, *observed_convention_index=None*, *ignore_observed_positions=False*, *z_source_convention=None*, *cosmo_interp=False*, *z_interp_stop=None*, *num_z_interp=100*, *kwargs_interp=None*)

Bases: `object`

Multi-plane lensing class with option to assign positions of a selected set of lens models in the observed plane.

The lens model deflection angles are in units of reduced deflections from the specified redshift of the lens to the source redshift of the class instance.

__init__(*z_source*, *lens_model_list*, *lens_redshift_list*, *cosmo=None*, *numerical_alpha_class=None*, *observed_convention_index=None*, *ignore_observed_positions=False*, *z_source_convention=None*, *cosmo_interp=False*, *z_interp_stop=None*, *num_z_interp=100*, *kwargs_interp=None*)

Parameters

- **z_source** – source redshift for default computation of reduced lensing quantities
- **lens_model_list** – list of lens model strings
- **lens_redshift_list** – list of floats with redshifts of the lens models indicated in lens_model_list
- **cosmo** – instance of astropy.cosmology
- **numerical_alpha_class** – an instance of a custom class for use in NumericalAlpha() lens model (see documentation in Profiles/numerical_alpha)
- **kwargs_interp** – interpolation keyword arguments specifying the numerics. See description in the Interpolate() class. Only applicable for 'INTERPOL' and 'INTERPOL_SCALED' models.
- **observed_convention_index** – a list of indices, corresponding to the lens_model_list element with same index, where the 'center_x' and 'center_y' kwargs

correspond to observed (lensed) positions, not physical positions. The code will compute the physical locations when performing computations

- **ignore_observed_positions** – bool, if True, will ignore the conversion between observed to physical position of deflectors
- **z_source_convention** – float, redshift of a source to define the reduced deflection angles of the lens models. If None, 'z_source' is used.

alpha (*theta_x*, *theta_y*, *kwargs_lens*, *check_convention=True*, *k=None*)
reduced deflection angle

Parameters

- **theta_x** – angle in x-direction
- **theta_y** – angle in y-direction
- **kwargs_lens** – lens model kwargs
- **check_convention** – flag to check the image position convention (leave this alone)

Returns deflection angles in x and y directions

arrival_time (*theta_x*, *theta_y*, *kwargs_lens*, *check_convention=True*)
light travel time relative to a straight path through the coordinate (0,0) Negative sign means earlier arrival time

Parameters

- **theta_x** – angle in x-direction on the image
- **theta_y** – angle in y-direction on the image
- **kwargs_lens** – lens model keyword argument list

Returns travel time in unit of days

co_moving2angle_source (*x*, *y*)
special case of the co_moving2angle definition at the source redshift

Parameters

- **x** – co-moving distance
- **y** – co-moving distance

Returns angles on the sky at the nominal source plane

geo_shapiro_delay (*theta_x*, *theta_y*, *kwargs_lens*, *check_convention=True*)
geometric and Shapiro (gravitational) light travel time relative to a straight path through the coordinate (0,0) Negative sign means earlier arrival time

Parameters

- **theta_x** – angle in x-direction on the image
- **theta_y** – angle in y-direction on the image
- **kwargs_lens** – lens model keyword argument list
- **check_convention** – boolean, if True goes through the lens model list and checks whether the positional conventions are satisfied.

Returns geometric delay, gravitational delay [days]

hessian (*theta_x*, *theta_y*, *kwargs_lens*, *k=None*, *diff=1e-08*, *check_convention=True*)
computes the hessian components f_xx, f_yy, f_xy from f_x and f_y with numerical differentiation

Parameters

- **theta_x** (*numpy array*) – x-position (preferentially arcsec)
- **theta_y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs_lens** – list of keyword arguments of lens model parameters matching the lens model classes
- **diff** – numerical differential step (float)
- **check_convention** – boolean, if True goes through the lens model list and checks whether the positional conventions are satisfied.

Returns f_xx, f_xy, f_yx, f_yy

observed2flat_convention (*kwargs_lens*)

Parameters **kwargs_lens** – keyword argument list of lens model parameters in the observed convention

Returns kwargs_lens positions mapped into angular position without lensing along its LOS

ray_shooting (*theta_x*, *theta_y*, *kwargs_lens*, *check_convention=True*, *k=None*)

ray-tracing (backwards light cone) to the default z_source redshift

Parameters

- **theta_x** – angle in x-direction on the image (usually arc seconds, in the same convention as lensing deflection angles)
- **theta_y** – angle in y-direction on the image (usually arc seconds, in the same convention as lensing deflection angles)
- **kwargs_lens** – lens model keyword argument list
- **check_convention** – flag to check the image position convention (leave this alone)

Returns angles in the source plane

ray_shooting_partial (*x*, *y*, *alpha_x*, *alpha_y*, *z_start*, *z_stop*, *kwargs_lens*, *include_z_start=False*, *check_convention=True*, *T_ij_start=None*, *T_ij_end=None*)

ray-tracing through parts of the coin, starting with (x,y) co-moving distances and angles (alpha_x, alpha_y) at redshift z_start and then backwards to redshift z_stop

Parameters

- **x** – co-moving position [Mpc] / angle definition
- **y** – co-moving position [Mpc] / angle definition
- **alpha_x** – ray angle at z_start [arcsec]
- **alpha_y** – ray angle at z_start [arcsec]
- **z_start** – redshift of start of computation
- **z_stop** – redshift where output is computed
- **kwargs_lens** – lens model keyword argument list
- **include_z_start** – bool, if True, includes the computation of the deflection angle at the same redshift as the start of the ray-tracing. ATTENTION: deflection angles at the same redshift as z_stop will be computed! This can lead to duplications in the computation of deflection angles.
- **check_convention** – flag to check the image position convention (leave this alone)

- **T_ij_start** – transverse angular distance between the starting redshift to the first lens plane to follow. If not set, will compute the distance each time this function gets executed.
- **T_ij_end** – transverse angular distance between the last lens plane being computed and z_end. If not set, will compute the distance each time this function gets executed.

Returns co-moving position (modulo angle definition) and angles at redshift z_stop

set_dynamic()

Returns

set_static(*kwargs*)

Parameters **kwargs** – lens model keyword argument list

Returns lens model keyword argument list with positional parameters all in flat sky coordinates

transverse_distance_start_stop(*z_start*, *z_stop*, *include_z_start=False*)

computes the transverse distance (T_ij) that is required by the ray-tracing between the starting redshift and the first deflector afterwards and the last deflector before the end of the ray-tracing.

Parameters

- **z_start** – redshift of the start of the ray-tracing
- **z_stop** – stop of ray-tracing
- **include_z_start** – bool, i

Returns T_ij_start, T_ij_end

update_source_redshift(*z_source*)

update instance of this class to compute reduced lensing quantities and time delays to a specific source redshift

Parameters **z_source** – float; source redshift

Returns self variables update to new redshift

class PhysicalLocation

Bases: object

center_x and center_y kwargs correspond to angular location of deflectors without lensing along the LOS

class LensedLocation(*multiplane_instance*, *observed_convention_index*)

Bases: object

center_x and center_y kwargs correspond to observed (lensed) locations of deflectors given a model for the line of sight structure, compute the angular position of the deflector without lensing contribution along the LOS

__init__(*multiplane_instance*, *observed_convention_index*)

Parameters

- **multiplane_instance** – instance of the MultiPlane class
- **observed_convention_index** – list of lens model indexes to be modelled in the observed plane

lenstronomy.LensModel.MultiPlane.multi_plane_base module

class MultiPlaneBase(*lens_model_list*, *lens_redshift_list*, *z_source_convention*, *cosmo=None*, *numerical_alpha_class=None*, *cosmo_interp=False*, *z_interp_stop=None*, *num_z_interp=100*, *kwargs_interp=None*)

Bases: `lenstronomy.LensModel.profile_list_base.ProfileListBase`

Multi-plane lensing class

The lens model deflection angles are in units of reduced deflections from the specified redshift of the lens to the source redshift of the class instance.

__init__(*lens_model_list*, *lens_redshift_list*, *z_source_convention*, *cosmo=None*, *numerical_alpha_class=None*, *cosmo_interp=False*, *z_interp_stop=None*, *num_z_interp=100*, *kwargs_interp=None*)

A description of the recursive multi-plane formalism can be found e.g. here: https://arxiv.org/abs/1312.1536

Parameters

- **lens_model_list** – list of lens model strings
- **lens_redshift_list** – list of floats with redshifts of the lens models indicated in lens_model_list
- **z_source_convention** – float, redshift of a source to define the reduced deflection angles of the lens models. If None, 'z_source' is used.
- **cosmo** – instance of astropy.cosmology
- **numerical_alpha_class** – an instance of a custom class for use in NumericalAlpha() lens model (see documentation in Profiles/numerical_alpha)
- **kwargs_interp** – interpolation keyword arguments specifying the numerics. See description in the Interpolate() class. Only applicable for 'INTERPOL' and 'INTERPOL_SCALED' models.

geo_shapiro_delay(*theta_x*, *theta_y*, *kwargs_lens*, *z_stop*, *T_z_stop=None*, *T_ij_end=None*)

geometric and Shapiro (gravitational) light travel time relative to a straight path through the coordinate (0,0) Negative sign means earlier arrival time

Parameters

- **theta_x** – angle in x-direction on the image
- **theta_y** – angle in y-direction on the image
- **kwargs_lens** – lens model keyword argument list
- **z_stop** – redshift of the source to stop the backwards ray-tracing
- **T_z_stop** – optional, transversal angular distance from z=0 to z_stop
- **T_ij_end** – optional, transversal angular distance between the last lensing plane and the source plane

Returns dt_geo, dt_shapiro, [days]

ray_shooting_partial(*x*, *y*, *alpha_x*, *alpha_y*, *z_start*, *z_stop*, *kwargs_lens*, *include_z_start=False*, *T_ij_start=None*, *T_ij_end=None*)

ray-tracing through parts of the coin, starting with (x,y) co-moving distances and angles (alpha_x, alpha_y) at redshift z_start and then backwards to redshift z_stop

Parameters

- **x** – co-moving position [Mpc]
- **y** – co-moving position [Mpc]
- **alpha_x** – ray angle at z_start [arcsec]
- **alpha_y** – ray angle at z_start [arcsec]
- **z_start** – redshift of start of computation
- **z_stop** – redshift where output is computed
- **kwargs_lens** – lens model keyword argument list
- **include_z_start** – bool, if True, includes the computation of the deflection angle at the same redshift as the start of the ray-tracing. ATTENTION: deflection angles at the same redshift as z_stop will be computed always! This can lead to duplications in the computation of deflection angles.
- **T_ij_start** – transverse angular distance between the starting redshift to the first lens plane to follow. If not set, will compute the distance each time this function gets executed.
- **T_ij_end** – transverse angular distance between the last lens plane being computed and z_end. If not set, will compute the distance each time this function gets executed.

Returns co-moving position and angles at redshift z_stop

transverse_distance_start_stop (*z_start*, *z_stop*, *include_z_start=False*)
computes the transverse distance (T_ij) that is required by the ray-tracing between the starting redshift and the first deflector afterwards and the last deflector before the end of the ray-tracing.

Parameters

- **z_start** – redshift of the start of the ray-tracing
- **z_stop** – stop of ray-tracing
- **include_z_start** – boolean, if True includes the computation of the starting position if the first deflector is at z_start

Returns T_ij_start, T_ij_end

Module contents

lenstronomy.LensModel.Profiles package

Submodules

lenstronomy.LensModel.Profiles.arc_perturbations module

class ArcPerturbations
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

uses radial and tangential fourier modes within a specific range in both directions to perturb a lensing potential

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

derivatives (*x*, *y*, *coeff*, *d_r*, *d_phi*, *center_x*, *center_y*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **coeff** – float, amplitude of basis
- **d_r** – period of radial sinusoidal in units of angle
- **d_phi** – period of tangential sinusoidal in radian
- **center_x** – center of rotation for tangential basis
- **center_y** – center of rotation for tangential basis

Returns f_x, f_y

function (*x*, *y*, *coeff*, *d_r*, *d_phi*, *center_x*, *center_y*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **coeff** – float, amplitude of basis
- **d_r** – period of radial sinusoidal in units of angle
- **d_phi** – period of tangential sinusoidal in radian
- **center_x** – center of rotation for tangential basis
- **center_y** – center of rotation for tangential basis

Returns

hessian (*x*, *y*, *coeff*, *d_r*, *d_phi*, *center_x*, *center_y*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **coeff** – float, amplitude of basis
- **d_r** – period of radial sinusoidal in units of angle
- **d_phi** – period of tangential sinusoidal in radian
- **center_x** – center of rotation for tangential basis
- **center_y** – center of rotation for tangential basis

Returns f_xx, f_yy, f_xy

lenstronomy.LensModel.Profiles.base_profile module

class LensProfileBase (**args*, ***kwargs*)

Bases: `object`

this class acts as the base class of all lens model functions and indicates raise statements and default outputs if these functions are not defined in the specific lens model class

__init__ (**args*, ***kwargs*)

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

density_lens(**args*, ***kwargs*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity. (optional definition)

$$\kappa(x, y) = \int_{-\infty}^{\infty} \rho(x, y, z) dz$$

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

derivatives(**args*, ***kwargs*)
deflection angles

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

function(**args*, ***kwargs*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian(**args*, ***kwargs*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

mass_2d_lens(**args*, ***kwargs*)
two-dimensional enclosed mass at radius r (optional definition)

$$M_{2d}(R) = \int_{0}^{R} \rho_{2d}(r) 2\pi r dr$$

with $\rho_{2d}(r)$ is the density_2d_lens() definition

The mass definition is such that:

$$\alpha = mass_2 d/r/\pi$$

with alpha is the deflection angle

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

mass_3d_lens(**args*, ***kwargs*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units The input parameter are identical as for the derivatives definition. (optional definition)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

set_dynamic()

Returns no return, deletes pre-computed variables for certain lens models

set_static(***kwargs*)
pre-computes certain computations that do only relate to the lens model parameters and not to the specific position where to evaluate the lens model

Parameters **kwargs** – lens model parameters

Returns no return, for certain lens model some private self variables are initiated

lenstronomy.LensModel.Profiles.chameleon module

class Chameleon(*static=False*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class of the Chameleon model (See Suyu+2014) an elliptical truncated double isothermal profile

__init__(*static=False*)

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

density_lens(*r, alpha_1, w_c, w_t, e1=0, e2=0, center_x=0, center_y=0*)

spherical average density as a function of 3d radius

Parameters

- **r** – 3d radius
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **w_c** – see Suyu+2014
- **w_t** – see Suyu+2014
- **e1** – ellipticity parameter
- **e2** – ellipticity parameter
- **center_x** – ra center
- **center_y** – dec center

Returns matter density at 3d radius r

derivatives(*x, y, alpha_1, w_c, w_t, e1, e2, center_x=0, center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **w_c** – see Suyu+2014
- **w_t** – see Suyu+2014
- **e1** – ellipticity parameter
- **e2** – ellipticity parameter
- **center_x** – ra center
- **center_y** – dec center

Returns deflection angles (RA, DEC)

function(*x, y, alpha_1, w_c, w_t, e1, e2, center_x=0, center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate

- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **w_c** – see Suyu+2014
- **w_t** – see Suyu+2014
- **e1** – ellipticity parameter
- **e2** – ellipticity parameter
- **center_x** – ra center
- **center_y** – dec center

Returns lensing potential

hessian (*x*, *y*, *alpha_1*, *w_c*, *w_t*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **w_c** – see Suyu+2014
- **w_t** – see Suyu+2014
- **e1** – ellipticity parameter
- **e2** – ellipticity parameter
- **center_x** – ra center
- **center_y** – dec center

Returns second derivatives of the lensing potential (Hessian: f_xx, f_xy, f_yx, f_yy)

lower_limit_default = {'alpha_1': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.

mass_3d_lens (*r*, *alpha_1*, *w_c*, *w_t*, *e1=0*, *e2=0*, *center_x=0*, *center_y=0*)
mass enclosed 3d radius

Parameters

- **r** – 3d radius
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **w_c** – see Suyu+2014
- **w_t** – see Suyu+2014
- **e1** – ellipticity parameter
- **e2** – ellipticity parameter
- **center_x** – ra center
- **center_y** – dec center

Returns mass enclosed 3d radius r

param_convert (*alpha_1*, *w_c*, *w_t*, *e1*, *e2*)
convert the parameter alpha_1 (deflection angle one arcsecond from the center) into the "Einstein radius" scale parameter of the two NIE profiles

Parameters

- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **w_c** – see Suyu+2014
- **w_t** – see Suyu+2014
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus

Returns

param_names = ['alpha_1', 'w_c', 'w_t', 'e1', 'e2', 'center_x', 'center_y']

set_dynamic()

Returns

set_static(*alpha_1*, *w_c*, *w_t*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **alpha_1** –
- **w_c** –
- **w_t** –
- **e1** –
- **e2** –
- **center_x** –
- **center_y** –

Returns

upper_limit_default = {'alpha_1': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.8

class DoubleChameleon

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class of the Chameleon model (See Suyu+2014) an elliptical truncated double isothermal profile

__init__()

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

density_lens(*r*, *alpha_1*, *ratio*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *center_x=0*, *center_y=0*)

Parameters

- **r** – 3d radius
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **ratio** – ratio of deflection amplitude at radius = 1 of the first to second Chameleon profile
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile

- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile
- **center_x** – ra center
- **center_y** – dec center

Returns 3d density at radius r

derivatives (*x*, *y*, *alpha_1*, *ratio*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *center_x=0*, *center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **ratio** – ratio of deflection amplitude at radius = 1 of the first to second Chameleon profile
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile
- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile^V
- **center_x** – ra center
- **center_y** – dec center

Returns deflection angles (RA, DEC)

function (*x*, *y*, *alpha_1*, *ratio*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *center_x=0*, *center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **ratio** – ratio of deflection amplitude at radius = 1 of the first to second Chameleon profile
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile
- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile

- **center_x** – ra center
- **center_y** – dec center

Returns lensing potential

hessian (*x, y, alpha_1, ratio, w_c1, w_t1, e11, e21, w_c2, w_t2, e12, e22, center_x=0, center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **ratio** – ratio of deflection amplitude at radius = 1 of the first to second Chameleon profile
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile
- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile
- **center_x** – ra center
- **center_y** – dec center

Returns second derivatives of the lensing potential (Hessian: f_xx, f_yy, f_xy)

lower_limit_default = {'alpha_1': 0, 'center_x': -100, 'center_y': -100, 'e11': -0

mass_3d_lens (*r, alpha_1, ratio, w_c1, w_t1, e11, e21, w_c2, w_t2, e12, e22, center_x=0, center_y=0*)

Parameters

- **r** – 3d radius
- **alpha_1** – deflection angle at 1 (arcseconds) from the center
- **ratio** – ratio of deflection amplitude at radius = 1 of the first to second Chameleon profile
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile
- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile
- **center_x** – ra center
- **center_y** – dec center

Returns mass enclosed 3d radius

param_names = ['alpha_1', 'ratio', 'w_c1', 'w_t1', 'e11', 'e21', 'w_c2', 'w_t2', 'e12'

set_dynamic()

Returns no return, deletes pre-computed variables for certain lens models

set_static(*alpha_1*, *ratio*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *center_x=0*, *center_y=0*)
pre-computes certain computations that do only relate to the lens model parameters and not to the specific position where to evaluate the lens model

Parameters kwargs – lens model parameters

Returns no return, for certain lens model some private self variables are initiated

upper_limit_default = {'alpha_1': 100, 'center_x': 100, 'center_y': 100, 'e11': 0.

class TripleChameleon
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class of the Chameleon model (See Suyu+2014) an elliptical truncated double isothermal profile

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

density_lens(*r*, *alpha_1*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

Parameters

- **r** – 3d radius
- **alpha_1** –
- **ratio12** – ratio of first to second amplitude
- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns density at radius r (spherical average)

derivatives(*x*, *y*, *alpha_1*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

Parameters

- **alpha_1** –
- **ratio12** – ratio of first to second amplitude

- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns

function (*x*, *y*, *alpha_1*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

Parameters

- **alpha_1** –
- **ratio12** – ratio of first to second amplitude
- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns

hessian (*x*, *y*, *alpha_1*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

Parameters

- **alpha_1** –
- **ratio12** – ratio of first to second amplitude
- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –

- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns

lower_limit_default = {'alpha_1': 0, 'center_x': -100, 'center_y': -100, 'e11': -0

mass_3d_lens (*r*, *alpha_1*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

Parameters

- **r** – 3d radius
- **alpha_1** –
- **ratio12** – ratio of first to second amplitude
- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns mass enclosed 3d radius

param_names = ['alpha_1', 'ratio12', 'ratio13', 'w_c1', 'w_t1', 'e11', 'e21', 'w_c2',

set_dynamic ()

Returns no return, deletes pre-computed variables for certain lens models

set_static (*alpha_1*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

pre-computes certain computations that do only relate to the lens model parameters and not to the specific position where to evaluate the lens model

Parameters **kwargs** – lens model parameters

Returns no return, for certain lens model some private self variables are initiated

upper_limit_default = {'alpha_1': 100, 'center_x': 100, 'center_y': 100, 'e11': 0.

class DoubleChameleonPointMass

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class of the Chameleon model (See Suyu+2014) an elliptical truncated double isothermal profile

__init__()

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

derivatives(*x, y, alpha_1, ratio_pointmass, ratio_chameleon, w_c1, w_t1, e11, e21, w_c2, w_t2, e12, e22, center_x=0, center_y=0*)

Parameters

- **x** –
- **y** –
- **alpha_1** –
- **ratio_pointmass** – ratio of point source Einstein radius to combined Chameleon deflection angle at r=1
- **ratio_chameleon** – ratio in deflection angles at r=1 for the two Chameleon profiles
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile
- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile
- **center_x** – ra center
- **center_y** – dec center

Returns

function(*x, y, alpha_1, ratio_pointmass, ratio_chameleon, w_c1, w_t1, e11, e21, w_c2, w_t2, e12, e22, center_x=0, center_y=0*)

#TODO chose better parameterization for combining point mass and Chameleon profiles :param x: ra-coordinate :param y: dec-coordinate :param alpha_1: deflection angle at 1 (arcseconds) from the center :param ratio_pointmass: ratio of point source Einstein radius to combined Chameleon deflection angle at r=1 :param ratio_chameleon: ratio in deflection angles at r=1 for the two Chameleon profiles :param w_c1: Suyu+2014 for first profile :param w_t1: Suyu+2014 for first profile :param e11: ellipticity parameter for first profile :param e21: ellipticity parameter for first profile :param w_c2: Suyu+2014 for second profile :param w_t2: Suyu+2014 for second profile :param e12: ellipticity parameter for second profile :param e22: ellipticity parameter for second profile :param center_x: ra center :param center_y: dec center :return:

hessian(*x, y, alpha_1, ratio_pointmass, ratio_chameleon, w_c1, w_t1, e11, e21, w_c2, w_t2, e12, e22, center_x=0, center_y=0*)

Parameters

- **x** –
- **y** –
- **alpha_1** –

- **ratio_pointmass** – ratio of point source Einstein radius to combined Chameleon deflection angle at r=1
- **ratio_chameleon** – ratio in deflection angles at r=1 for the two Chameleon profiles
- **w_c1** – Suyu+2014 for first profile
- **w_t1** – Suyu+2014 for first profile
- **e11** – ellipticity parameter for first profile
- **e21** – ellipticity parameter for first profile
- **w_c2** – Suyu+2014 for second profile
- **w_t2** – Suyu+2014 for second profile
- **e12** – ellipticity parameter for second profile
- **e22** – ellipticity parameter for second profile
- **center_x** – ra center
- **center_y** – dec center

Returns

`lower_limit_default = {'alpha_1': 0, 'center_x': -100, 'center_y': -100, 'e11': -0`

`param_names = ['alpha_1', 'ratio_chameleon', 'ratio_pointmass', 'w_c1', 'w_t1', 'e11',`

`upper_limit_default = {'alpha_1': 100, 'center_x': 100, 'center_y': 100, 'e11': 0.`

lenstronomy.LensModel.Profiles.cnfw module

class CNFW

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class computes the lensing quantities of a cored NFW profile: rho = rho0 * (r + r_core)^-1 * (r + rs)^-2 alpha_Rs is the normalization equivalent to the deflection angle at rs in the absence of a core

__init__()

alpha_r(*R*, *Rs*, *rho0*, *r_core*)

deflection angel of NFW profile along the radial direction

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius

Returns Epsilon(R) projected density at radius R

cnfwGamma(*R*, *Rs*, *rho0*, *r_core*, *ax_x*, *ax_y*)

shear gamma of NFW profile (times Sigma_crit) along the projection to coordinate 'axis'

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)

Returns Epsilon(R) projected density at radius R

density (*R, Rs, rho0, r_core*)
three dimensional truncated NFW profile

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (central core density)

Returns rho(R) density

density_2d (*x, y, Rs, rho0, r_core, center_x=0, center_y=0*)
projected two dimenstional NFW profile (kappa*Sigma_crit)

Parameters

- **x** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)

Returns Epsilon(R) projected density at radius R

density_lens (*R, Rs, alpha_Rs, r_core*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

derivatives (*x, y, Rs, alpha_Rs, r_core, center_x=0, center_y=0*)
deflection angles

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

function (*x, y, Rs, alpha_Rs, r_core, center_x=0, center_y=0*)

Parameters

- **x** – angular position
- **y** – angular position
- **Rs** – angular turn over point
- **alpha_Rs** – deflection at Rs (in the absence of a core
- **r_core** – core radius
- **center_x** – center of halo
- **center_y** – center of halo

Returns

hessian (*x, y, Rs, alpha_Rs, r_core, center_x=0, center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^f/dy^2, d^2/dxdy

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_2d (*R, Rs, rho0, r_core*)
analytic solution of the projection integral (convergence)

mass_3d (*R, Rs, rho0, r_core*)
mass enclosed a 3d sphere or radius r

Parameters

- **R** –
- **Rs** –
- **rho0** –
- **r_core** –

Returns

mass_3d_lens (*R, Rs, alpha_Rs, r_core*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units

Returns

model_name = 'CNFW'

param_names = ['Rs', 'alpha_Rs', 'r_core', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.cnfw_ellipse module

class CNFW_ELLIPSE
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

this class contains functions concerning the NFW profile

relation are: R_200 = c * Rs

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

density_lens (*R, Rs, alpha_Rs, r_core, e1=0, e2=0*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

derivatives (*x, y, Rs, alpha_Rs, r_core, e1, e2, center_x=0, center_y=0*)
returns df/dx and df/dy of the function (integral of NFW)

function (*x, y, Rs, alpha_Rs, r_core, e1, e2, center_x=0, center_y=0*)
returns double integral of NFW profile

hessian (*x, y, Rs, alpha_Rs, r_core, e1, e2, center_x=0, center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^f/dy^2, d^2/dxdy

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_3d_lens (*R, Rs, alpha_Rs, r_core, e1=0, e2=0*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units

Returns

param_names = ['Rs', 'alpha_Rs', 'r_core', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.const_mag module

class ConstMag (**args, **kwargs*)
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

this class implements the macromodel potential of Diego et al. Convergence and shear are computed according to Diego2018

derivatives (*x*, *y*, *mu_r*, *mu_t*, *parity*, *phi_G*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **mu_r** – radial magnification
- **mu_t** – tangential magnification
- **parity** – parity of the side of the macromodel. Either +1 (positive parity) or -1 (negative parity)
- **phi_G** – shear orientation angle (relative to the x-axis)

Returns deflection angle (in angles)

function (*x*, *y*, *mu_r*, *mu_t*, *parity*, *phi_G*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **mu_r** – radial magnification
- **mu_t** – tangential magnification
- **parity** – parity side of the macromodel. Either +1 (positive parity) or -1 (negative parity)
- **phi_G** – shear orientation angle (relative to the x-axis)

Returns lensing potential

hessian (*x*, *y*, *mu_r*, *mu_t*, *parity*, *phi_G*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **mu_r** – radial magnification
- **mu_t** – tangential magnification
- **parity** – parity of the side of the macromodel. Either +1 (positive parity) or -1 (negative parity)
- **phi_G** – shear orientation angle (relative to the x-axis)

Returns hessian matrix (in angles)

lower_limit_default = {'center_x': -100, 'center_y': -100, 'mu_r': 1, 'mu_t': 1000

param_names = ['center_x', 'center_y', 'mu_r', 'mu_t', 'parity', 'phi_G']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'mu_r': 1, 'mu_t': 1000,

lenstronomy.LensModel.Profiles.constant_shift module

class Shift (**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

Lens model with a constant shift of the deflection field

derivatives (*x*, *y*, *alpha_x*, *alpha_y*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **alpha_x** – shift in x-direction (angle)
- **alpha_y** – shift in y-direction (angle)

Returns deflection in x- and y-direction

function (*x*, *y*, *alpha_x*, *alpha_y*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **alpha_x** – shift in x-direction (angle)
- **alpha_y** – shift in y-direction (angle)

Returns lensing potential

hessian (*x*, *y*, *alpha_x*, *alpha_y*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **alpha_x** – shift in x-direction (angle)
- **alpha_y** – shift in y-direction (angle)

Returns hessian elements f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'alpha_x': -1000, 'alpha_y': -1000}

param_names = ['alpha_x', 'alpha_y']

upper_limit_default = {'alpha_x': 1000, 'alpha_y': 1000}

lenstronomy.LensModel.Profiles.convergence module

class Convergence (**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

a single mass sheet (external convergence)

derivatives (*x*, *y*, *kappa*, *ra_0=0*, *dec_0=0*)

deflection angle

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **kappa** – (external) convergence

Returns deflection angles (first order derivatives)

function (*x*, *y*, *kappa*, *ra_0=0*, *dec_0=0*)
lensing potential

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **kappa** – (external) convergence

Returns lensing potential

hessian (*x*, *y*, *kappa*, *ra_0=0*, *dec_0=0*)
Hessian matrix

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **kappa** – external convergence
- **ra_0** – zero point of polynomial expansion (no deflection added)
- **dec_0** – zero point of polynomial expansion (no deflection added)

Returns second order derivatives f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'dec_0': -100, 'kappa': -10, 'ra_0': -100}

model_name = 'CONVERGENCE'

param_names = ['kappa', 'ra_0', 'dec_0']

upper_limit_default = {'dec_0': 100, 'kappa': 10, 'ra_0': 100}

lenstronomy.LensModel.Profiles.coreBurkert module

class CoreBurkert (**args*, ***kwargs*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

lensing properties of a modified Burkert profile with variable core size normalized by rho0, the central core density

cBurkGamma (*R*, *Rs*, *rho0*, *r_core*, *ax_x*, *ax_y*)

Parameters

- **R** – projected distance
- **Rs** – scale radius
- **rho0** – central core density
- **r_core** – core radius
- **ax_x** – x coordinate

- **ax_y** – y coordinate

Returns

cBurkPot (*R*, *Rs*, *rho0*, *r_core*)

Parameters

- **R** – projected distance
- **Rs** – scale radius
- **rho0** – central core density
- **r_core** – core radius

coreBurkAlpha (*R*, *Rs*, *rho0*, *r_core*, *ax_x*, *ax_y*)
deflection angle

Parameters

- **R** –
- **Rs** –
- **rho0** –
- **r_core** –
- **ax_x** –
- **ax_y** –

Returns

density (*R*, *Rs*, *rho0*, *r_core*)
three dimensional cored Burkert profile

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – characteristic density

Returns rho(R) density

density_2d (*x*, *y*, *Rs*, *rho0*, *r_core*, *center_x=0*, *center_y=0*)
projected two dimenstional core Burkert profile (kappa*Sigma_crit)

Parameters

- **x** – x coordinate
- **y** – y coordinate
- **Rs** – scale radius
- **rho0** – central core density
- **r_core** – core radius

derivatives (*x*, *y*, *Rs*, *alpha_Rs*, *r_core*, *center_x=0*, *center_y=0*)
deflection angles :param x: x coordinate :param y: y coordinate :param Rs: scale radius :param alpha_Rs: deflection angle at Rs :param r_core: core radius :param center_x: :param center_y: :return:

function (*x*, *y*, *Rs*, *alpha_Rs*, *r_core*, *center_x=0*, *center_y=0*)

Parameters

- **x** – angular position
- **y** – angular position
- **Rs** – angular turn over point
- **alpha_Rs** – deflection angle at Rs
- **center_x** – center of halo
- **center_y** – center of halo

Returns

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *r_core*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x coordinate
- **y** – y coordinate
- **Rs** – scale radius
- **alpha_Rs** – deflection angle at Rs
- **r_core** – core radius
- **center_x** –
- **center_y** –

Returns

lower_limit_default = {'Rs': 1, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_2d (*R*, *Rs*, *rho0*, *r_core*)

analytic solution of the projection integral (convergence)

Parameters

- **R** – projected distance
- **Rs** – scale radius
- **rho0** – central core density
- **r_core** – core radius

mass_3d (*R*, *Rs*, *rho0*, *r_core*)

Parameters

- **R** – projected distance
- **Rs** – scale radius
- **rho0** – central core density
- **r_core** – core radius

param_names = ['Rs', 'alpha_Rs', 'r_core', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'alpha_Rs': 100, 'center_x': 100, 'center_y': 10

lenstronomy.LensModel.Profiles.cored_density module

class CoredDensity(**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class for a uniform cored density dropping steep in the outskirts This profile is e.g. featured in Blum et al. 2020 https://arxiv.org/abs/2001.07182v1

..math:: rho_c(r) = frac{2}{pi} Sigma_{c} R_c^3 left(R_c^2 + r^2 right)^{-2}

with the convergence profile as

..math:: kappa_c(theta) = left(1 + frac{theta^2}{theta_c^2} right)^{-3/2}.

An approximate mass-sheet degeneracy can then be written as

..math:: kappa_{lambda_c}(theta) = lambda_c kappa(theta) + (1-lambda_c) kappa_c(theta).

static alpha_r(*r*, *sigma0*, *r_core*)

radial deflection angle of the cored density profile

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns deflection angle

static d_alpha_dr(*r*, *sigma0*, *r_core*)

radial derivatives of the radial deflection angle

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns dalpha/dr

static density(*r*, *sigma0*, *r_core*)

rho(r) = 2/pi * Sigma_crit R_c**3 * (R_c**2 + r**2)**(-2)

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns density at radius r

density_2d(*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)

projected density at projected radius r

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius

- **center_x** – center of the profile
- **center_y** – center of the profile

Returns projected density

density_lens (*r*, *sigma0*, *r_core*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns desnity at radius r

derivatives (*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)
deflection angle of cored density profile

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius
- **center_x** – center of the profile
- **center_y** – center of the profile

Returns alpha_x, alpha_y at (x, y)

function (*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)
potential of cored density profile

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius
- **center_x** – center of the profile
- **center_y** – center of the profile

Returns lensing potential at (x, y)

hessian (*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius

- **center_x** – center of the profile
- **center_y** – center of the profile

Returns Hessian df/dxdx, df/dxdy, df/dydx, df/dydy at position (x, y)

static kappa_r(*r*, *sigma0*, *r_core*)
convergence of the cored density profile. This routine is also for testing

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns convergence at r

lower_limit_default = {'center_x': -100, 'center_y': -100, 'r_core': 0, 'sigma0':

mass_2d(*r*, *sigma0*, *r_core*)
mass enclosed in cylinder of radius r

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns mass enclosed in cylinder of radius r

static mass_3d(*r*, *sigma0*, *r_core*)
mass enclosed 3d radius

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns mass enclosed 3d radius

mass_3d_lens(*r*, *sigma0*, *r_core*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units For this profile those are identical.

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns mass enclosed 3d radius

param_names = ['sigma0', 'r_core', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'r_core': 100, 'sigma0':

lenstronomy.LensModel.Profiles.cored_density_2 module

class CoredDensity2(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class for a uniform cored density dropping steep in the outskirts credits for suggesting this profile goes to Kfir Blum

$$\rho(r) = 2/\pi * \Sigma_{\rm crit} R_c^2 * (R_c^2 + r^2)^{-3/2}$$

This profile drops like an NFW profile as math:*rho(r)^{-3}*.

static alpha_r(*r*, *sigma0*, *r_core*)

radial deflection angle of the cored density profile

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns deflection angle

static d_alpha_dr(*r*, *sigma0*, *r_core*)

radial derivatives of the radial deflection angle

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns dalpha/dr

static density(*r*, *sigma0*, *r_core*)

rho(r) = 2/pi * Sigma_crit R_c**3 * (R_c**2 + r**2)**(-3/2)

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns density at radius r

density_2d(*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)

projected density at projected radius r

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius
- **center_x** – center of the profile
- **center_y** – center of the profile

Returns projected density

density_lens (*r*, *sigma0*, *r_core*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns density at radius r

derivatives (*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)
deflection angle of cored density profile

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius
- **center_x** – center of the profile
- **center_y** – center of the profile

Returns alpha_x, alpha_y at (x, y)

function (*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)
potential of cored density profile

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius
- **center_x** – center of the profile
- **center_y** – center of the profile

Returns lensing potential at (x, y)

hessian (*x*, *y*, *sigma0*, *r_core*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in angular units
- **y** – y-coordinate in angular units
- **sigma0** – convergence in the core
- **r_core** – core radius
- **center_x** – center of the profile
- **center_y** – center of the profile

Returns Hessian df/dxdx, df/dxdy, df/dydx, df/dydy at position (x, y)

static kappa_r(*r, sigma0, r_core*)
convergence of the cored density profile. This routine is also for testing

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns convergence at r

lower_limit_default = {'center_x': -100, 'center_y': -100, 'r_core': 0, 'sigma0':

static mass_2d(*r, sigma0, r_core*)
mass enclosed in cylinder of radius r

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns mass enclosed in cylinder of radius r

static mass_3d(*r, sigma0, r_core*)
mass enclosed 3d radius

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns mass enclosed 3d radius

mass_3d_lens(*r, sigma0, r_core*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units For this profile those are identical.

Parameters

- **r** – radius (angular scale)
- **sigma0** – convergence in the core
- **r_core** – core radius

Returns mass enclosed 3d radius

model_name = 'CORED_DENSITY_2'

param_names = ['sigma0', 'r_core', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'r_core': 100, 'sigma0':

lenstronomy.LensModel.Profiles.cored_density_exp module

class CoredDensityExp(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

this class contains functions concerning an exponential cored density profile, namely

..math:: rho(r) = rho_0 exp(- (theta / theta_c)^2)

static alpha_radial(*r*, *kappa_0*, *theta_c*)

returns the radial part of the deflection angle :param r: angular position (normally in units of arc seconds) :param kappa_0: central convergence of profile :param theta_c: core radius (in arcsec) :return: radial deflection angle

density(*R*, *kappa_0*, *theta_c*)

three dimensional density profile in angular units (rho0_physical = rho0_angular Sigma_crit / D_lens)

Parameters

- **R** – projected angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)

Returns rho(R) density

density_2d(*x*, *y*, *kappa_0*, *theta_c*, *center_x=0*, *center_y=0*)

projected two dimensional ULDM profile (convergence * Sigma_crit), but given our units convention for rho0, it is basically the convergence

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)

Returns Epsilon(R) projected density at radius R

density_lens(*r*, *kappa_0*, *theta_c*)

computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)

Returns density rho(r)

derivatives(*x*, *y*, *kappa_0*, *theta_c*, *center_x=0*, *center_y=0*)

returns df/dx and df/dy of the function (lensing potential), which are the deflection angles

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile

- **theta_c** – core radius (in arcsec)
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection angle in x, deflection angle in y

function (*x, y, kappa_0, theta_c, center_x=0, center_y=0*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns lensing potential (in arcsec^2)

hessian (*x, y, kappa_0, theta_c, center_x=0, center_y=0*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

static kappa_r (*R, kappa_0, theta_c*)

convergence of the cored density profile. This routine is also for testing

Parameters

- **R** – radius (angular scale)
- **kappa_0** – convergence in the core
- **theta_c** – core radius

Returns convergence at r

lower_limit_default = {'center_x': -100, 'center_y': -100, 'kappa_0': 0, 'theta_c':

mass_2d (*R, kappa_0, theta_c*)

mass enclosed a 2d sphere of radius r returns

$$M_{2D} = 2\pi \int_0^r dr' r' \int dz \rho(\sqrt(r'^2 + z^2))$$

Parameters

- **kappa_0** – central convergence of soliton
- **theta_c** – core radius (in arcsec)

Returns M_2D (ULDM only)

static mass_3d(*R*, *kappa_0*, *theta_c*)
mass enclosed a 3d sphere or radius r :param kappa_0: central convergence of profile :param theta_c: core radius (in arcsec) :param R: radius in arcseconds :return: mass of soliton in angular units

mass_3d_lens(*r*, *kappa_0*, *theta_c*)
mass enclosed a 3d sphere or radius r :param kappa_0: central convergence of profile :param theta_c: core radius (in arcsec) :return: mass

param_names = ['kappa_0', 'theta_c', 'center_x', 'center_y']

static rhotilde(*kappa_0*, *theta_c*)
Computes the central density in angular units :param kappa_0: central convergence of profile :param theta_c: core radius (in arcsec) :return: central density in 1/arcsec

upper_limit_default = {'center_x': 100, 'center_y': 100, 'kappa_0': 10, 'theta_c':

lenstronomy.LensModel.Profiles.cored_density_mst module

class CoredDensityMST(*profile_type='CORED_DENSITY'*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

approximate mass-sheet transform of a density core. This routine takes the parameters of the density core and subtracts a mass-sheet that approximates the cored profile in it's center to counter-act (in approximation) this model. This allows for better sampling of the mass-sheet transformed quantities that do not have strong covariances. The subtraction of the mass-sheet is done such that the sampler returns the real central convergence of the original model (but be careful, the output of quantities like the Einstein angle of the main deflector are still the not-scaled one). Attention!!! The interpretation of the result is that the mass sheet as 'CONVERGENCE' that is present needs to be subtracted in post-processing.

__init__(*profile_type='CORED_DENSITY'*)
Initialize self. See help(type(self)) for accurate signature.

derivatives(*x*, *y*, *lambda_approx*, *r_core*, *center_x=0*, *center_y=0*)
deflection angles of approximate mass-sheet correction

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **lambda_approx** – approximate mass sheet transform
- **r_core** – core radius of the cored density profile
- **center_x** – x-center of the profile
- **center_y** – y-center of the profile

Returns alpha_x, alpha_y

function(*x*, *y*, *lambda_approx*, *r_core*, *center_x=0*, *center_y=0*)
lensing potential of approximate mass-sheet correction

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **lambda_approx** – approximate mass sheet transform

- **r_core** – core radius of the cored density profile
- **center_x** – x-center of the profile
- **center_y** – y-center of the profile

Returns lensing potential correction

hessian (*x*, *y*, *lambda_approx*, *r_core*, *center_x=0*, *center_y=0*)
Hessian terms of approximate mass-sheet correction

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **lambda_approx** – approximate mass sheet transform
- **r_core** – core radius of the cored density profile
- **center_x** – x-center of the profile
- **center_y** – y-center of the profile

Returns df/dxx, df/dxy, df/dyx, df/dyy

lower_limit_default = {'center_x': -100, 'center_y': -100, 'lambda_approx': -1, 'r_c

param_names = ['lambda_approx', 'r_core', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'lambda_approx': 10, 'r_co

lenstronomy.LensModel.Profiles.cored_steep_ellipsoid module

class CSE (*axis='product_avg'*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Cored steep ellipsoid (CSE) :param axis: 'major' or 'product_avg' ; whether to evaluate corresponding to r= major axis or r= sqrt(ab) source: Keeton and Kochanek (1998) Oguri 2021: https://arxiv.org/pdf/2106.11464.pdf

$$\kappa(u;s) = \frac{A}{2(s^2 + \xi^2)^{3/2}}$$

with

$$\xi(x,y) = \sqrt{x^2 + \frac{y^2}{q^2}}$$

__init__ (*axis='product_avg'*)
Initialize self. See help(type(self)) for accurate signature.

derivatives (*x*, *y*, *a*, *s*, *e1*, *e2*, *center_x*, *center_y*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius

- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – center of profile
- **center_y** – center of profile

Returns deflection in x- and y-direction

function (*x*, *y*, *a*, *s*, *e1*, *e2*, *center_x*, *center_y*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – center of profile
- **center_y** – center of profile

Returns lensing potential

hessian (*x*, *y*, *a*, *s*, *e1*, *e2*, *center_x*, *center_y*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – center of profile
- **center_y** – center of profile

Returns hessian elements f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'A': -1000, 'center_x': -100, 'center_y': -100, 'e1': -0.5,

param_names = ['A', 's', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'A': 1000, 'center_x': -100, 'center_y': -100, 'e1': 0.5, 'e

class CSEMajorAxis (**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Cored steep ellipsoid (CSE) along the major axis source: Keeton and Kochanek (1998) Oguri 2021: https://arxiv.org/pdf/2106.11464.pdf

$$\kappa(u;s) = \frac{A}{2(s^2 + \xi^2)^{3/2}}$$

with

$$\xi(x,y) = \sqrt{x^2 + \frac{y^2}{q^2}}$$

derivatives (*x*, *y*, *a*, *s*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **q** – axis ratio

Returns deflection in x- and y-direction

function (*x*, *y*, *a*, *s*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **q** – axis ratio

Returns lensing potential

hessian (*x*, *y*, *a*, *s*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **q** – axis ratio

Returns hessian elements f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'A': -1000, 'center_x': -100, 'center_y': -100, 'q': 0.001,

param_names = ['A', 's', 'q', 'center_x', 'center_y']

upper_limit_default = {'A': 1000, 'center_x': -100, 'center_y': -100, 'e2': 0.5, 'q

class CSEMajorAxisSet

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

a set of CSE profiles along a joint center and axis

__init__ ()

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

derivatives (*x*, *y*, *a_list*, *s_list*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a_list** – list of lensing strength
- **s_list** – list of core radius
- **q** – axis ratio

Returns deflection in x- and y-direction

function (*x*, *y*, *a_list*, *s_list*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a_list** – list of lensing strength
- **s_list** – list of core radius
- **q** – axis ratio

Returns lensing potential

hessian (*x*, *y*, *a_list*, *s_list*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a_list** – list of lensing strength
- **s_list** – list of core radius
- **q** – axis ratio

Returns hessian elements f_xx, f_xy, f_yx, f_yy

class CSEProductAvg

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Cored steep ellipsoid (CSE) evaluated at the product-averaged radius sqrt(ab), such that mass is not changed when increasing ellipticity

Same as CSEMajorAxis but evaluated at r=sqrt(q)*r_original

Keeton and Kochanek (1998) Oguri 2021: https://arxiv.org/pdf/2106.11464.pdf

$$\kappa(u;s) = \frac{A}{2(s^2 + \xi^2)^{3/2}}$$

with

$$\xi(x,y) = \sqrt{qx^2 + \frac{y^2}{q}}$$

__init__ ()

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

derivatives (*x*, *y*, *a*, *s*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **q** – axis ratio

Returns deflection in x- and y-direction

function (*x*, *y*, *a*, *s*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **q** – axis ratio

Returns lensing potential

hessian (*x*, *y*, *a*, *s*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a** – lensing strength
- **s** – core radius
- **q** – axis ratio

Returns hessian elements f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'A': -1000, 'center_x': -100, 'center_y': -100, 'q': 0.001,

param_names = ['A', 's', 'q', 'center_x', 'center_y']

upper_limit_default = {'A': 1000, 'center_x': -100, 'center_y': -100, 'e2': 0.5, 'q

class CSEProductAvgSet

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

a set of CSE profiles along a joint center and axis

__init__ ()

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

derivatives (*x*, *y*, *a_list*, *s_list*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a_list** – list of lensing strength

- **s_list** – list of core radius
- **q** – axis ratio

Returns deflection in x- and y-direction

function (*x*, *y*, *a_list*, *s_list*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a_list** – list of lensing strength
- **s_list** – list of core radius
- **q** – axis ratio

Returns lensing potential

hessian (*x*, *y*, *a_list*, *s_list*, *q*)

Parameters

- **x** – coordinate in image plane (angle)
- **y** – coordinate in image plane (angle)
- **a_list** – list of lensing strength
- **s_list** – list of core radius
- **q** – axis ratio

Returns hessian elements f_xx, f_xy, f_yx, f_yy

lenstronomy.LensModel.Profiles.curved_arc_const module

class CurvedArcConstMST

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

lens model that describes a section of a highly magnified deflector region. The parameterization is chosen to describe local observables efficient.

Observables are: - curvature radius (basically bending relative to the center of the profile) - radial stretch (plus sign) thickness of arc with parity (more generalized than the power-law slope) - tangential stretch (plus sign). Infinity means at critical curve - direction of curvature - position of arc

Requirements: - Should work with other perturbative models without breaking its meaning (say when adding additional shear terms) - Must best reflect the observables in lensing - minimal covariances between the parameters, intuitive parameterization.

__init__ ()

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

derivatives (*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction

- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

function (*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)
ATTENTION: there may not be a global lensing potential!

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

hessian (*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'curvature': 1e-06, 'dire

param_names = ['tangential_stretch', 'radial_stretch', 'curvature', 'direction', 'cent

upper_limit_default = {'center_x': 100, 'center_y': 100, 'curvature': 100, 'directi

class CurvedArcConst (**args*, ***kwargs*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

curved arc lensing with orientation of curvature perpendicular to the x-axis with unity radial stretch

derivatives (*x*, *y*, *tangential_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

function (*x*, *y*, *tangential_stretch*, *curvature*, *direction*, *center_x*, *center_y*)
ATTENTION: there may not be a global lensing potential!

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

hessian (*x*, *y*, *tangential_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'curvature': 1e-06, 'dir

param_names = ['tangential_stretch', 'curvature', 'direction', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'curvature': 100, 'directi

lenstronomy.LensModel.Profiles.curved_arc_sis_mst module

class CurvedArcSISMST
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

lens model that describes a section of a highly magnified deflector region. The parameterization is chosen to describe local observables efficient.

Observables are: - curvature radius (basically bending relative to the center of the profile) - radial stretch (plus sign) thickness of arc with parity (more generalized than the power-law slope) - tangential stretch (plus sign). Infinity means at critical curve - direction of curvature - position of arc

Requirements: - Should work with other perturbative models without breaking its meaning (say when adding additional shear terms) - Must best reflect the observables in lensing - minimal covariances between the parameters, intuitive parameterization.

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

`derivatives`(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

`function`(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)
ATTENTION: there may not be a global lensing potential!

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

`hessian`(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction

- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'curvature': 1e-06, 'dir

param_names = ['tangential_stretch', 'radial_stretch', 'curvature', 'direction', 'cent

static sis_mst2stretch (*theta_E*, *kappa_ext*, *center_x_sis*, *center_y_sis*, *center_x*, *center_y*)

turn Singular power-law lens model into stretch parameterization at position (center_x, center_y) This is the inverse function of stretch2spp()

Parameters

- **theta_E** – Einstein radius of SIS profile
- **kappa_ext** – external convergence (MST factor 1 - kappa_ext)
- **center_x_sis** – center of SPP model
- **center_y_sis** – center of SPP model
- **center_x** – center of curved model definition
- **center_y** – center of curved model definition

Returns tangential_stretch, radial_stretch, curvature, direction

Returns

static stretch2sis_mst (*tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns parameters in terms of a spherical SIS + MST resulting in the same observables

upper_limit_default = {'center_x': 100, 'center_y': 100, 'curvature': 100, 'directi

lenstronomy.LensModel.Profiles.curved_arc_spp module

class CurvedArcSPP

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

lens model that describes a section of a highly magnified deflector region. The parameterization is chosen to describe local observables efficient.

Observables are: - curvature radius (basically bending relative to the center of the profile) - radial stretch (plus sign) thickness of arc with parity (more generalized than the power-law slope) - tangential stretch (plus sign). Infinity means at critical curve - direction of curvature - position of arc

Requirements: - Should work with other perturbative models without breaking its meaning (say when adding additional shear terms) - Must best reflect the observables in lensing - minimal covariances between the parameters, intuitive parameterization.

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

derivatives(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

function(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)
ATTENTION: there may not be a global lensing potential!

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

hessian(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius

- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'curvature': 1e-06, 'dir

param_names = ['tangential_stretch', 'radial_stretch', 'curvature', 'direction', 'cent

static spp2stretch (*theta_E*, *gamma*, *center_x_spp*, *center_y_spp*, *center_x*, *center_y*)
turn Singular power-law lens model into stretch parameterization at position (center_x, center_y) This is the inverse function of stretch2spp()

Parameters

- **theta_E** – Einstein radius of SPP model
- **gamma** – power-law slope
- **center_x_spp** – center of SPP model
- **center_y_spp** – center of SPP model
- **center_x** – center of curved model definition
- **center_y** – center of curved model definition

Returns tangential_stretch, radial_stretch, curvature, direction

static stretch2spp (*tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns parameters in terms of a spherical power-law profile resulting in the same observables

upper_limit_default = {'center_x': 100, 'center_y': 100, 'curvature': 100, 'directi

center_deflector (*curvature*, *direction*, *center_x*, *center_y*)

Parameters

- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns center_spp_x, center_spp_y

lenstronomy.LensModel.Profiles.curved_arc_spt module

class CurvedArcSPT

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Curved arc model based on SIS+MST with an additional non-linear shear distortions applied on the source coordinates around the center. This profile is effectively a Source Position Transform of a curved arc and a shear distortion.

__init__()

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

derivatives(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *gamma1*, *gamma2*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **gamma1** – non-linear reduced shear distortion in the source plane
- **gamma2** – non-linear reduced shear distortion in the source plane
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

function(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *gamma1*, *gamma2*, *center_x*, *center_y*)

ATTENTION: there may not be a global lensing potential!

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **gamma1** – non-linear reduced shear distortion in the source plane
- **gamma2** – non-linear reduced shear distortion in the source plane
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

hessian(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *gamma1*, *gamma2*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **gamma1** – non-linear reduced shear distortion in the source plane
- **gamma2** – non-linear reduced shear distortion in the source plane
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'curvature': 1e-06, 'dire

param_names = ['tangential_stretch', 'radial_stretch', 'curvature', 'direction', 'gamma

upper_limit_default = {'center_x': 100, 'center_y': 100, 'curvature': 100, 'directi

lenstronomy.LensModel.Profiles.curved_arc_tan_diff module

class CurvedArcTanDiff

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Curved arc model with an additional non-zero tangential stretch differential in tangential direction component

Observables are: - curvature radius (basically bending relative to the center of the profile) - radial stretch (plus sign) thickness of arc with parity (more generalized than the power-law slope) - tangential stretch (plus sign). Infinity means at critical curve - direction of curvature - position of arc

Requirements: - Should work with other perturbative models without breaking its meaning (say when adding additional shear terms) - Must best reflect the observables in lensing - minimal covariances between the parameters, intuitive parameterization.

__init__()

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

derivatives(*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *dtan_dtan*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **dtan_dtan** – d(tangential_stretch) / d(tangential direction) / tangential stretch

- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

function (*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *dtan_dtan*, *direction*, *center_x*, *center_y*)
ATTENTION: there may not be a global lensing potential!

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **dtan_dtan** – d(tangential_stretch) / d(tangential direction) / tangential stretch
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

hessian (*x*, *y*, *tangential_stretch*, *radial_stretch*, *curvature*, *dtan_dtan*, *direction*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **dtan_dtan** – d(tangential_stretch) / d(tangential direction) / tangential stretch
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'curvature': 1e-06, 'dire

param_names = ['tangential_stretch', 'radial_stretch', 'curvature', 'dtan_dtan', 'dire

static stretch2sie_mst (*tangential_stretch*, *radial_stretch*, *curvature*, *dtan_dtan*, *direction*, *center_x*, *center_y*)

Parameters

- **tangential_stretch** – float, stretch of intrinsic source in tangential direction
- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **dtan_dtan** – d(tangential_stretch) / d(tangential direction) / tangential stretch

- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane

Returns parameters in terms of a spherical SIS + MST resulting in the same observables

upper_limit_default = {'center_x': 100, 'center_y': 100, 'curvature': 100, 'directi

lenstronomy.LensModel.Profiles.dipole module

class Dipole(**args*, ***kwargs*)
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class for dipole response of two massive bodies (experimental)

derivatives(*x*, *y*, *com_x*, *com_y*, *phi_dipole*, *coupling*)
deflection angles

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

function(*x*, *y*, *com_x*, *com_y*, *phi_dipole*, *coupling*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian(*x*, *y*, *com_x*, *com_y*, *phi_dipole*, *coupling*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

lower_limit_default = {'com_x': -100, 'com_y': -100, 'coupling': -10, 'phi_dipole':

param_names = ['com_x', 'com_y', 'phi_dipole', 'coupling']

upper_limit_default = {'com_x': 100, 'com_y': 100, 'coupling': 10, 'phi_dipole': 1

class DipoleUtil
Bases: `object`

pre-calculation of dipole properties

static angle(*center1_x*, *center1_y*, *center2_x*, *center2_y*)
compute the rotation angle of the dipole :return:

static com(*center1_x*, *center1_y*, *center2_x*, *center2_y*, *Fm*)

Returns center of mass

static mass_ratio(*theta_E*, *theta_E_sub*)
computes mass ration of the two clumps with given Einstein radius and power law slope (clump1/sub-clump) :param theta_E: :param theta_E_sub: :return:

lenstronomy.LensModel.Profiles.elliptical_density_slice module

class EllISLICE(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

This class computes the lensing quantities for an elliptical slice of constant density. Based on Schramm 1994 https://ui.adsabs.harvard.edu/abs/1994A%26A...284...44S/abstract

Computes the lensing quantities of an elliptical slice with semi major axis 'a' and semi minor axis 'b', centered on 'center_x' and 'center_y', oriented with an angle 'psi' in radian, and with constant surface mass density 'sigma_0'. In other words, this lens model is characterized by the surface mass density :

..math::

kappa(x,y) = left{

begin{array}{ll} sigma_0 & mbox{if } frac{x_{rot}^2}{a^2} + frac{y_{rot}^2}{b^2} leq 1 0 & mbox{else}

end{array}

right}.

with

..math:: x_{rot} = x_c cos psi + y_c sin psi y_{rot} = - x_c sin psi + y_c cos psi x_c = x - center_x y_c = y - center_y

alpha_ext(*x*, *y*, *kwargs_slice*)

deflection angle for (x,y) outside the elliptical slice

Parameters kwargs_slice – dict, dictionary with the slice definition (a,b,psi,sigma_0)

alpha_in(*x*, *y*, *kwargs_slice*)

deflection angle for (x,y) inside the elliptical slice

Parameters kwargs_slice – dict, dictionary with the slice definition (a,b,psi,sigma_0)

derivatives(*x*, *y*, *a*, *b*, *psi*, *sigma_0*, *center_x=0.0*, *center_y=0.0*)

lensing deflection angle

Parameters

- **a** – float, semi-major axis, must be positive
- **b** – float, semi-minor axis, must be positive
- **psi** – float, orientation in radian
- **sigma_0** – float, surface mass density, must be positive
- **center_x** – float, center on the x axis
- **center_y** – float, center on the y axis

function(*x*, *y*, *a*, *b*, *psi*, *sigma_0*, *center_x=0.0*, *center_y=0.0*)

lensing potential

Parameters

- **a** – float, semi-major axis, must be positive
- **b** – float, semi-minor axis, must be positive
- **psi** – float, orientation in radian
- **sigma_0** – float, surface mass density, must be positive

- **center_x** – float, center on the x axis
- **center_y** – float, center on the y axis

hessian (*x*, *y*, *a*, *b*, *psi*, *sigma_0*, *center_x=0.0*, *center_y=0.0*)
lensing second derivatives

Parameters

- **a** – float, semi-major axis, must be positive
- **b** – float, semi-minor axis, must be positive
- **psi** – float, orientation in radian
- **sigma_0** – float, surface mass density, must be positive
- **center_x** – float, center on the x axis
- **center_y** – float, center on the y axis

lower_limit_default = {'a': 0.0, 'b': 0.0, 'center_x': -100.0, 'center_y': -100.0,

param_names = ['a', 'b', 'psi', 'sigma_0', 'center_x', 'center_y']

pot_ext (*x*, *y*, *kwargs_slice*)
lensing potential for (x,y) outside the elliptical slice

Parameters **kwargs_slice** – dict, dictionary with the slice definition (a,b,psi,sigma_0)

static pot_in (*x*, *y*, *kwargs_slice*)
lensing potential for (x,y) inside the elliptical slice

Parameters **kwargs_slice** – dict, dictionary with the slice definition (a,b,psi,sigma_0)

static sign (*z*)
sign function

Parameters **z** – complex

upper_limit_default = {'a': 100.0, 'b': 100.0, 'center_x': 100.0, 'center_y': 100.

lenstronomy.LensModel.Profiles.epl module

class EPL
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Elliptical Power Law mass profile

$$\kappa(x,y) = \frac{3-\gamma}{2}\left(\frac{\theta_E}{\sqrt{qx^2+y^2/q}}\right)^{\gamma-1}$$

with θ_E is the (circularized) Einstein radius, γ is the negative power-law slope of the 3D mass distributions, q is the minor/major axis ratio, and x and y are defined in a coordinate sys- tem aligned with the major and minor axis of the lens.

In terms of eccentricities, this profile is defined as

$$\kappa(r) = \frac{3-\gamma}{2}\left(\frac{\theta'_E}{r\sqrt{1-e*\cos(2*\phi)}}\right)^{\gamma-1}$$

with ϵ is the ellipticity defined as

$$\epsilon = \frac{1-q^2}{1+q^2}$$

And an Einstein radius $\theta'_{\rm E}$ related to the definition used is

$$\left(\frac{\theta'_{\rm E}}{\theta_{\rm E}}\right)^2 = \frac{2q}{1+q^2}.$$

The mathematical form of the calculation is presented by Tessore & Metcalf (2015), https://arxiv.org/abs/1507.01819. The current implementation is using hyperbolic functions. The paper presents an iterative calculation scheme, converging in few iterations to high precision and accuracy.

A (faster) implementation of the same model using numba is accessible as 'EPL_NUMBA' with the iterative calculation scheme.

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

density_lens(*r*, *theta_E*, *gamma*, *e1=None*, *e2=None*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius within the mass is computed
- **theta_E** – Einstein radius
- **gamma** – power-law slope
- **e1** – eccentricity component (not used)
- **e2** – eccentricity component (not used)

Returns mass enclosed a 3D radius r

derivatives(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane
- **theta_E** – Einstein radius
- **gamma** – power law slope
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – profile center
- **center_y** – profile center

Returns alpha_x, alpha_y

function(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane

- **theta_E** – Einstein radius
- **gamma** – power law slope
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – profile center
- **center_y** – profile center

Returns lensing potential

hessian (*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane
- **theta_E** – Einstein radius
- **gamma** – power law slope
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – profile center
- **center_y** – profile center

Returns f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

mass_3d_lens (*r*, *theta_E*, *gamma*, *e1=None*, *e2=None*)
computes the spherical power-law mass enclosed (with SPP routine) :param r: radius within the mass is computed :param theta_E: Einstein radius :param gamma: power-law slope :param e1: eccentricity component (not used) :param e2: eccentricity component (not used) :return: mass enclosed a 3D radius r

param_conv (*theta_E*, *gamma*, *e1*, *e2*)
converts parameters as defined in this class to the parameters used in the EPLMajorAxis() class

Parameters

- **theta_E** – Einstein radius as defined in the profile class
- **gamma** – negative power-law slope
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus

Returns b, t, q, phi_G

param_names = ['theta_E', 'gamma', 'e1', 'e2', 'center_x', 'center_y']

set_dynamic ()

Returns

set_static (*theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **theta_E** – Einstein radius

- **gamma** – power law slope
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – profile center
- **center_y** – profile center

Returns self variables set

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 'ga

class EPLMajorAxis

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

This class contains the function and the derivatives of the elliptical power law.

$$\kappa = (2-t)/2 * \left[\frac{b}{\sqrt{q^2 x^2 + y^2}}\right]^t$$

where with $t = \gamma - 1$ (from EPL class) being the projected power-law slope of the convergence profile, critical radius b, axis ratio q.

Tessore & Metcalf (2015), https://arxiv.org/abs/1507.01819

__init__()

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

derivatives(*x, y, b, t, q*)

returns the deflection angles

Parameters

- **x** – x-coordinate in image plane relative to center (major axis)
- **y** – y-coordinate in image plane relative to center (minor axis)
- **b** – critical radius
- **t** – projected power-law slope
- **q** – axis ratio

Returns f_x, f_y

function(*x, y, b, t, q*)

returns the lensing potential

Parameters

- **x** – x-coordinate in image plane relative to center (major axis)
- **y** – y-coordinate in image plane relative to center (minor axis)
- **b** – critical radius
- **t** – projected power-law slope
- **q** – axis ratio

Returns lensing potential

hessian(*x, y, b, t, q*)

Hessian matrix of the lensing potential

Parameters

- **x** – x-coordinate in image plane relative to center (major axis)
- **y** – y-coordinate in image plane relative to center (minor axis)
- **b** – critical radius
- **t** – projected power-law slope
- **q** – axis ratio

Returns f_xx, f_yy, f_xy

param_names = ['b', 't', 'q', 'center_x', 'center_y']

lenstronomy.LensModel.Profiles.epl_numba module

class EPL_numba

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

" Elliptical Power Law mass profile - computation accelerated with numba

$$\kappa(x,y) = \frac{3-\gamma}{2}\left(\frac{\theta_E}{\sqrt{qx^2+y^2/q}}\right)^{\gamma-1}$$

with θ_E is the (circularized) Einstein radius, γ is the negative power-law slope of the 3D mass distributions, q is the minor/major axis ratio, and x and y are defined in a coordinate system aligned with the major and minor axis of the lens.

In terms of eccentricities, this profile is defined as

$$\kappa(r) = \frac{3-\gamma}{2}\left(\frac{\theta'_E}{r\sqrt{1e*\cos(2*\phi)}}\right)^{\gamma-1}$$

with ϵ is the ellipticity defined as

$$\epsilon = \frac{1-q^2}{1+q^2}$$

And an Einstein radius $\theta'_{\rm E}$ related to the definition used is

$$\left(\frac{\theta'_{\rm E}}{\theta_{\rm E}}\right)^2 = \frac{2q}{1+q^2}.$$

The mathematical form of the calculation is presented by Tessore & Metcalf (2015), https://arxiv.org/abs/1507.01819. The current implementation is using hyperbolic functions. The paper presents an iterative calculation scheme, converging in few iterations to high precision and accuracy.

A (slower) implementation of the same model using hyperbolic functions without the iterative calculation is accessible as 'EPL' not requiring numba.

__init__()

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

derivatives

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)

- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns deflection angles alpha_x, alpha_y

function

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns lensing potential

hessian

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns Hessian components f_xx, f_yy, f_xy

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

param_names = ['theta_E', 'gamma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 'ga

lenstronomy.LensModel.Profiles.flexion module

class Flexion(**args*, ***kwargs*)
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class for flexion

derivatives(*x*, *y*, *g1*, *g2*, *g3*, *g4*, *ra_0=0*, *dec_0=0*)
deflection angles

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

function(*x*, *y*, *g1*, *g2*, *g3*, *g4*, *ra_0=0*, *dec_0=0*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian(*x*, *y*, *g1*, *g2*, *g3*, *g4*, *ra_0=0*, *dec_0=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

lower_limit_default = {'dec_0': -100, 'g1': -0.1, 'g2': -0.1, 'g3': -0.1, 'g4': -

param_names = ['g1', 'g2', 'g3', 'g4', 'ra_0', 'dec_0']

upper_limit_default = {'dec_0': 100, 'g1': 0.1, 'g2': 0.1, 'g3': 0.1, 'g4': 0.1,

lenstronomy.LensModel.Profiles.flexionfg module

class Flexionfg
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

Flexion consist of basis F flexion and G flexion (F1,F2,G1,G2), see formulas 2.54, 2.55 in Massimo Meneghetti 2017 - "Introduction to Gravitational Lensing".

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

derivatives(*x*, *y*, *F1*, *F2*, *G1*, *G2*, *ra_0=0*, *dec_0=0*)
deflection angle :param x: x-coordinate :param y: y-coordinate :param F1: F1 flexion, derivative of kappa in x direction :param F2: F2 flexion, derivative of kappa in y direction :param G1: G1 flexion :param G2: G2 flexion :param ra_0: center x-coordinate :param dec_0: center x-coordinate :return: deflection angle

function(*x*, *y*, *F1*, *F2*, *G1*, *G2*, *ra_0=0*, *dec_0=0*)
lensing potential

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **F1** – F1 flexion, derivative of kappa in x direction
- **F2** – F2 flexion, derivative of kappa in y direction
- **G1** – G1 flexion

- **G2** – G2 flexion
- **ra_0** – center x-coordinate
- **dec_0** – center y-coordinate

Returns lensing potential

hessian (*x, y, F1, F2, G1, G2, ra_0=0, dec_0=0*)
Hessian matrix :param x: x-coordinate :param y: y-coordinate :param F1: F1 flexion, derivative of kappa in x direction :param F2: F2 flexion, derivative of kappa in y direction :param G1: G1 flexion :param G2: G2 flexion :param ra_0: center x-coordinate :param dec_0: center y-coordinate :return: second order derivatives f_xx, f_yy, f_xy

lower_limit_default = {'F1': -0.1, 'F2': -0.1, 'G1': -0.1, 'G2': -0.1, 'dec_0': -

param_names = ['F1', 'F2', 'G1', 'G2', 'ra_0', 'dec_0']

static transform_fg (*F1, F2, G1, G2*)
basis transform from (F1,F2,G1,G2) to (g1,g2,g3,g4) :param F1: F1 flexion, derivative of kappa in x direction :param F2: F2 flexion, derivative of kappa in y direction :param G1: G1 flexion :param G2: G2 flexion :return: g1,g2,g3,g4 (phi_xxx, phi_xxy, phi_xyy, phi_yyy)

upper_limit_default = {'F1': 0.1, 'F2': 0.1, 'G1': 0.1, 'G2': 0.1, 'dec_0': 100,

lenstronomy.LensModel.Profiles.gauss_decomposition module

This module contains the class to compute lensing properties of any elliptical profile using Shajib (2019)'s Gauss decomposition.

class GaussianEllipseKappaSet (*use_scipy_wofz=True, min_ellipticity=1e-05*)
Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

This class computes the lensing properties of a set of concentric elliptical Gaussian convergences.

__init__ (*use_scipy_wofz=True, min_ellipticity=1e-05*)

Parameters

- **use_scipy_wofz** (`bool`) – To initiate `class GaussianEllipseKappa`. If `True`, Gaussian lensing will use `scipy.special.wofz` function. Set `False` for lower precision, but faster speed.
- **min_ellipticity** (`float`) – To be passed to `class GaussianEllipseKappa`. Minimum ellipticity for Gaussian elliptical lensing calculation. For lower ellipticity than min_ellipticity the equations for the spherical case will be used.

density_2d (*x, y, amp, sigma, e1, e2, center_x=0, center_y=0*)
Compute the density of a set of concentric elliptical Gaussian convergence profiles $\sum A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **amp** (`numpy.array` with `dtype=float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`numpy.array` with `dtype=float`) – Standard deviation of Gaussian
- **e1** (`float`) – Ellipticity parameter 1

- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordianate of centroid

Returns Density κ for elliptical Gaussian convergence

Return type `float`, or `numpy.array` with shape equal to `x.shape`

derivatives (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute the derivatives of function angles $\partial f/\partial x$, $\partial f/\partial y$ at $x,\ y$ for a set of concentric elliptic Gaussian convergence profiles.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **amp** (`numpy.array` with `dtype=float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`numpy.array` with `dtype=float`) – Standard deviation of Gaussian
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordianate of centroid

Returns Deflection angle $\partial f/\partial x$, $\partial f/\partial y$ for elliptical Gaussian convergence

Return type tuple `(float, float)` or `(numpy.array, numpy.array)` with each `numpy` array's shape equal to `x.shape`

function (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute the potential function for a set of concentric elliptical Gaussian convergence profiles.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **amp** (`numpy.array` with `dtype=float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`numpy.array` with `dtype=float`) – Standard deviation of Gaussian
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordianate of centroid

Returns Potential for elliptical Gaussian convergence

Return type `float`, or `numpy.array` with `shape = x.shape`

hessian (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute Hessian matrix of function $\partial^2 f/\partial x^2$, $\partial^2 f/\partial y^2$, $\partial^2 f/\partial x\partial y$ for a set of concentric elliptic Gaussian convergence profiles.

Parameters

- **x** (float or numpy.array) – x coordinate
- **y** (float or numpy.array) – y coordinate
- **amp** (numpy.array with dtype=float) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (numpy.array with dtype=float) – Standard deviation of Gaussian
- **e1** (float) – Ellipticity parameter 1
- **e2** (float) – Ellipticity parameter 2
- **center_x** (float) – x coordinate of centroid
- **center_y** (float) – y coordianate of centroid

Returns Hessian $\partial^2 f/\partial x^2$, $\partial^2/\partial x\partial y$, $\partial^2/\partial y\partial x$, $\partial^2 f/\partial y^2$ for elliptical Gaussian convergence.

Return type tuple (float, float, float) , or (numpy.array, numpy.array, numpy.array) with each numpy array's shape equal to x.shape

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'sigma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

class GaussDecompositionAbstract(*n_sigma=15*, *sigma_start_mult=0.02*, *sigma_end_mult=15.0*, *precision=10*, *use_scipy_wofz=True*, *min_ellipticity=1e-05*)

Bases: object

This abstract class sets up a template for computing lensing properties of an elliptical convergence through Shajib (2019)'s Gauss decomposition.

__init__(*n_sigma=15*, *sigma_start_mult=0.02*, *sigma_end_mult=15.0*, *precision=10*, *use_scipy_wofz=True*, *min_ellipticity=1e-05*)

Set up settings for the Gaussian decomposition. For more details about the decomposition parameters, see Shajib (2019).

Parameters

- **n_sigma** (int) – Number of Gaussian components
- **sigma_start_mult** (float) – Lower range of logarithmically spaced sigmas
- **sigma_end_mult** (float) – Upper range of logarithmically spaced sigmas
- **precision** (int) – Numerical precision of Gaussian decomposition
- **use_scipy_wofz** (bool) – To be passed to class GaussianEllipseKappa. If True, Gaussian lensing will use scipy.special.wofz function. Set False for lower precision, but faster speed.
- **min_ellipticity** (float) – To be passed to class GaussianEllipseKappa. Minimum ellipticity for Gaussian elliptical lensing calculation. For lower ellipticity than min_ellipticity the equations for the spherical case will be used.

density_2d(*x*, *y*, *e1=0.0*, *e2=0.0*, *center_x=0.0*, *center_y=0.0*, ***kwargs*)

Compute the convergence profile for Gauss-decomposed elliptic Sersic profile.

Parameters

- **x** (float or numpy.array) – x coordinate
- **y** (float or numpy.array) – y coordinate

- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordinate of centroid
- **kwargs** – Keyword arguments that are defined by the child class that are particular for the convergence profile in the child class.

Returns Convergence profile

Return type `type(x)`

derivatives(*x*, *y*, *e1=0.0*, *e2=0.0*, *center_x=0.0*, *center_y=0.0*, ***kwargs*)
Compute the derivatives of the deflection potential $\partial f/\partial x$, $\partial f/\partial y$ for a Gauss-decomposed elliptic convergence.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordinate of centroid
- **kwargs** – Keyword arguments that are defined by the child class that are particular for the convergence profile

Returns Derivatives of deflection potential

Return type tuple `(type(x), type(x))`

function(*x*, *y*, *e1=0.0*, *e2=0.0*, *center_x=0.0*, *center_y=0.0*, ***kwargs*)
Compute the deflection potential of a Gauss-decomposed elliptic convergence.

Parameters

- **x** (`float`) – x coordinate
- **y** (`float`) – y coordinate
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordinate of centroid
- **kwargs** – Keyword arguments that are defined by the child class that are particular for the convergence profile

Returns Deflection potential

Return type `float`

gauss_decompose(***kwargs*)
Compute the amplitudes and sigmas of Gaussian components using the integral transform with Gaussian kernel from Shajib (2019). The returned values are in the convention of eq. (2.13).

Parameters **kwargs** – Keyword arguments to send to `func`

Returns Amplitudes and standard deviations of the Gaussian components

Return type tuple (numpy.array, numpy.array)

get_kappa_1d(*y*, ***kwargs*)
Abstract method to compute the spherical Sersic profile at y. The concrete method has to defined by the child class.

Parameters

- **y** (float or numpy.array) – y coordinate
- **kwargs** – Keyword arguments that are defined by the child class that are particular for the convergence profile

get_scale(***kwargs*)
Abstract method to identify the keyword argument for the scale size among the profile parameters of the child class' convergence profile.

Parameters **kwargs** – Keyword arguments

Returns Scale size

Return type float

hessian(*x*, *y*, *e1=0.0*, *e2=0.0*, *center_x=0.0*, *center_y=0.0*, ***kwargs*)
Compute the Hessian of the deflection potential $\partial^2 f/\partial x^2$, $\partial^2 f/\partial y^2$, $\partial^2 f/\partial x\partial y$ of a Gauss-decomposed elliptic Sersic convergence.

Parameters

- **x** (float or numpy.array) – x coordinate
- **y** (float or numpy.array) – y coordinate
- **e1** (float) – Ellipticity parameter 1
- **e2** (float) – Ellipticity parameter 2
- **center_x** (float) – x coordinate of centroid
- **center_y** (float) – y coordinate of centroid
- **kwargs** – Keyword arguments that are defined by the child class that are particular for the convergence profile

Returns Hessian of deflection potential

Return type tuple (type(x), type(x), type(x))

class SersicEllipseGaussDec(*n_sigma=15*, *sigma_start_mult=0.02*, *sigma_end_mult=15.0*, *precision=10*, *use_scipy_wofz=True*, *min_ellipticity=1e-05*)
Bases: lenstronomy.LensModel.Profiles.gauss_decomposition.GaussDecompositionAbstract

This class computes the lensing properties of an elliptical Sersic profile using the Shajib (2019)'s Gauss decomposition method.

get_kappa_1d(*y*, ***kwargs*)
Compute the spherical Sersic profile at y.

Parameters

- **y** (float) – y coordinate
- **kwargs** – Keyword arguments

Keyword Arguments

- **n_sersic** (`float`) – Sersic index
- **R_sersic** (`float`) – Sersic scale radius
- **k_eff** (`float`) – Sersic convergence at R_sersic

Returns Sersic function at y

Return type `type(y)`

get_scale(***kwargs*)
Identify the scale size from the keyword arguments.

Parameters **kwargs** – Keyword arguments

Keyword Arguments

- **n_sersic** (`float`) – Sersic index
- **R_sersic** (`float`) – Sersic scale radius
- **k_eff** (`float`) – Sersic convergence at R_sersic

Returns Sersic radius

Return type `float`

lower_limit_default = {'R_sersic': 0.0, 'center_x': -100.0, 'center_y': -100.0, 'e1

param_names = ['k_eff', 'R_sersic', 'n_sersic', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'R_sersic': 100.0, 'center_x': 100.0, 'center_y': 100.0, 'e1

class NFWEllipseGaussDec(*n_sigma=15, sigma_start_mult=0.005, sigma_end_mult=50.0, precision=10, use_scipy_wofz=True, min_ellipticity=1e-05*)
Bases: `lenstronomy.LensModel.Profiles.gauss_decomposition.GaussDecompositionAbstract`

This class computes the lensing properties of an elliptical, projected NFW profile using Shajib (2019)'s Gauss decomposition method.

__init__(*n_sigma=15, sigma_start_mult=0.005, sigma_end_mult=50.0, precision=10, use_scipy_wofz=True, min_ellipticity=1e-05*)
Set up settings for the Gaussian decomposition. For more details about the decomposition parameters, see Shajib (2019).

Parameters

- **n_sigma** (`int`) – Number of Gaussian components
- **sigma_start_mult** (`float`) – Lower range of logarithmically spaced sigmas
- **sigma_end_mult** (`float`) – Upper range of logarithmically spaced sigmas
- **precision** (`int`) – Numerical precision of Gaussian decomposition
- **use_scipy_wofz** (`bool`) – To be passed to `class GaussianEllipseKappa`. If `True`, Gaussian lensing will use `scipy.special.wofz` function. Set `False` for lower precision, but faster speed.
- **min_ellipticity** (`float`) – To be passed to `class GaussianEllipseKappa`. Minimum ellipticity for Gaussian elliptical lensing calculation. For lower ellipticity than min_ellipticity the equations for the spherical case will be used.

get_kappa_1d(*y, **kwargs*)
Compute the spherical projected NFW profile at y.

Parameters

- **y** (`float`) – y coordinate
- **kwargs** – Keyword arguments

Keyword Arguments

- **alpha_Rs** (`float`) – Deflection angle at `Rs`
- **R_s** (`float`) – NFW scale radius

Returns projected NFW profile at y

Return type `type(y)`

get_scale(***kwargs*)

Identify the scale size from the keyword arguments.

Parameters **kwargs** – Keyword arguments

Keyword Arguments

- **alpha_Rs** (`float`) – Deflection angle at `Rs`
- **R_s** (`float`) – NFW scale radius

Returns NFW scale radius

Return type `float`

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

param_names = ['Rs', 'alpha_Rs', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

class GaussDecompositionAbstract3D(*n_sigma=15, sigma_start_mult=0.02, sigma_end_mult=15.0, precision=10, use_scipy_wofz=True, min_ellipticity=1e-05*)

Bases: `lenstronomy.LensModel.Profiles.gauss_decomposition.GaussDecompositionAbstract`

This abstract class sets up a template for computing lensing properties of a convergence from 3D spherical profile through Shajib (2019)'s Gauss decomposition.

gauss_decompose(***kwargs*)

Compute the amplitudes and sigmas of Gaussian components using the integral transform with Gaussian kernel from Shajib (2019). The returned values are in the convention of eq. (2.13).

Parameters **kwargs** – Keyword arguments to send to `func`

Returns Amplitudes and standard deviations of the Gaussian components

Return type tuple `(numpy.array, numpy.array)`

class CTNFWGaussDec(*n_sigma=15, sigma_start_mult=0.01, sigma_end_mult=20.0, precision=10, use_scipy_wofz=True*)

Bases: `lenstronomy.LensModel.Profiles.gauss_decomposition.GaussDecompositionAbstract3D`

This class computes the lensing properties of an projection from a spherical cored-truncated NFW profile using Shajib (2019)'s Gauss decomposition method.

__init__(*n_sigma=15, sigma_start_mult=0.01, sigma_end_mult=20.0, precision=10, use_scipy_wofz=True*)

Set up settings for the Gaussian decomposition. For more details about the decomposition parameters, see Shajib (2019).

Parameters

- **n_sigma** (`int`) – Number of Gaussian components
- **sigma_start_mult** (`float`) – Lower range of logarithmically spaced sigmas
- **sigma_end_mult** (`float`) – Upper range of logarithmically spaced sigmas
- **precision** (`int`) – Numerical precision of Gaussian decomposition
- **use_scipy_wofz** (`bool`) – To be passed to `class GaussianEllipseKappa`. If `True`, Gaussian lensing will use `scipy.special.wofz` function. Set `False` for lower precision, but faster speed.

get_kappa_1d(*y*, ***kwargs*)

Compute the spherical cored-truncated NFW profile at y.

Parameters

- **y** (`float`) – y coordinate
- **kwargs** – Keyword arguments

Keyword Arguments

- **r_s** (`float`) – Scale radius
- **r_trunc** (`float`) – Truncation radius
- **r_core** (`float`) – Core radius
- **rho_s** (`float`) – Density normalization
- **a** (`float`) – Core regularization parameter

Returns projected NFW profile at y

Return type `type(y)`

get_scale(***kwargs*)

Identify the scale size from the keyword arguments.

Parameters **kwargs** – Keyword arguments

Keyword Arguments

- **r_s** (`float`) – Scale radius
- **r_trunc** (`float`) – Truncation radius
- **r_core** (`float`) – Core radius
- **rho_s** (`float`) – Density normalization
- **a** (`float`) – Core regularization parameter

Returns NFW scale radius

Return type `float`

lower_limit_default = {'a': 0.0, 'center_x': -100, 'center_y': -100, 'r_core': 0,

param_names = ['r_s', 'r_core', 'r_trunc', 'a', 'rho_s', 'center_xcenter_y']

upper_limit_default = {'a': 10.0, 'center_x': 100, 'center_y': 100, 'r_core': 100,

lenstronomy.LensModel.Profiles.gaussian_ellipse_kappa module

This module defines `class GaussianEllipseKappa` to compute the lensing properties of an elliptical Gaussian profile with ellipticity in the convergence using the formulae from Shajib (2019).

class GaussianEllipseKappa(*use_scipy_wofz=True*, *min_ellipticity=1e-05*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

This class contains functions to evaluate the derivative and hessian matrix of the deflection potential for an elliptical Gaussian convergence.

The formulae are from Shajib (2019).

__init__(*use_scipy_wofz=True*, *min_ellipticity=1e-05*)

Setup which method to use the Faddeeva function and the ellipticity limit for spherical approximation.

Parameters

- **use_scipy_wofz** (`bool`) – If `True`, use `scipy.special.wofz`.
- **min_ellipticity** (`float`) – Minimum allowed ellipticity. For `q > 1 - min_ellipticity`, values for spherical case will be returned.

density_2d(*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute the density of elliptical Gaussian $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate.
- **y** (`float` or `numpy.array`) – y coordinate.
- **amp** (`float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`float`) – Standard deviation of Gaussian.
- **e1** (`float`) – Ellipticity parameter 1.
- **e2** (`float`) – Ellipticity parameter 2.
- **center_x** (`float`) – x coordinate of centroid.
- **center_y** (`float`) – y coordianate of centroid.

Returns Density κ for elliptical Gaussian convergence.

Return type `float`, or `numpy.array` with shape = `x.shape`.

derivatives(*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute the derivatives of function angles $\partial f/\partial x$, $\partial f/\partial y$ at $x,\ y$.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **amp** (`float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`float`) – Standard deviation of Gaussian
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordianate of centroid

Returns Deflection angle $\partial f/\partial x$, $\partial f/\partial y$ for elliptical Gaussian convergence.

Return type tuple `(float, float)` or `(numpy.array, numpy.array)` with each `numpy.array`'s shape equal to `x.shape`.

function (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute the potential function for elliptical Gaussian convergence.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **amp** (`float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`float`) – Standard deviation of Gaussian
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordianate of centroid

Returns Potential for elliptical Gaussian convergence

Return type `float`, or `numpy.array` with shape equal to `x.shape`

hessian (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Compute Hessian matrix of function $\partial^2 f/\partial x^2$, $\partial^2 f/\partial y^2$, $\partial^2/\partial x\partial y$.

Parameters

- **x** (`float` or `numpy.array`) – x coordinate
- **y** (`float` or `numpy.array`) – y coordinate
- **amp** (`float`) – Amplitude of Gaussian, convention: $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$
- **sigma** (`float`) – Standard deviation of Gaussian
- **e1** (`float`) – Ellipticity parameter 1
- **e2** (`float`) – Ellipticity parameter 2
- **center_x** (`float`) – x coordinate of centroid
- **center_y** (`float`) – y coordianate of centroid

Returns Hessian $A/(2\pi\sigma^2)\exp(-(x^2+y^2/q^2)/2\sigma^2)$ for elliptical Gaussian convergence.

Return type tuple `(float, float, float)` , or `(numpy.array, numpy.array, numpy.array)` with each `numpy.array`'s shape equal to `x.shape`.

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'sigma', 'e1', 'e2', 'center_x', 'center_y']

static sgn (*z*)

Compute the sign function $\mathrm{sgn}(z)$ factor for deflection as sugggested by Bray (1984). For current implementation, returning 1 is sufficient.

Parameters **z** (`complex`) – Complex variable $z = x + \mathrm{i}y$

Returns $\mathrm{sgn}(z)$

Return type `float`

sigma_function (*x*, *y*, *q*)

Compute the function $\varsigma(z; q)$ from equation (4.12) of Shajib (2019).

Parameters

- **x** (float or numpy.array) – Real part of complex variable, $x = \mathrm{Re}(z)$
- **y** (float or numpy.array) – Imaginary part of complex variable, $y = \mathrm{Im}(z)$
- **q** (float) – Axis ratio

Returns real and imaginary part of $\varsigma(z; q)$ function

Return type tuple (type(x), type(x))

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

static w_f_approx (*z*)

Compute the Faddeeva function $w_{\mathrm{F}}(z)$ using the approximation given in Zaghloul (2017).

Parameters **z** (complex or numpy.array(dtype=complex)) – complex number

Returns $w_{\mathrm{F}}(z)$

Return type complex

lenstronomy.LensModel.Profiles.gaussian_ellipse_potential module

class GaussianEllipsePotential

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions to evaluate a Gaussian function and calculates its derivative and hessian matrix with ellipticity in the convergence

the calculation follows Glenn van de Ven et al. 2009

__init__ ()

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

density (*r*, *amp*, *sigma*, *e1*, *e2*)

Parameters

- **r** –
- **amp** –
- **sigma** –

Returns

density_2d (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **e1** –
- **e2** –

- **center_x** –
- **center_y** –

Returns

derivatives (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function

function (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Gaussian

hessian (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

mass_2d (*R*, *amp*, *sigma*, *e1*, *e2*)

Parameters

- **R** –
- **amp** –
- **sigma** –
- **e1** –
- **e2** –

Returns

mass_2d_lens (*R*, *amp*, *sigma*, *e1*, *e2*)

Parameters

- **R** –
- **amp** –
- **sigma** –
- **e1** –
- **e2** –

Returns

mass_3d (*R*, *amp*, *sigma*, *e1*, *e2*)

Parameters

- **R** –
- **amp** –
- **sigma** –
- **e1** –
- **e2** –

Returns

mass_3d_lens (*R*, *amp*, *sigma*, *e1*, *e2*)

Parameters

- **R** –

- **amp** –
- **sigma** –
- **e1** –
- **e2** –

Returns

param_names = ['amp', 'sigma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

lenstronomy.LensModel.Profiles.gaussian_kappa module

class GaussianKappa

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions to evaluate a Gaussian function and calculates its derivative and hessian matrix

__init__()

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

alpha_abs(*R*, *amp*, *sigma*)

absolute value of the deflection :param R: :param amp: :param sigma: :return:

d_alpha_dr(*R*, *amp*, *sigma_x*, *sigma_y*)

Parameters

- **R** –
- **amp** –
- **sigma_x** –
- **sigma_y** –

Returns

density(*r*, *amp*, *sigma*)

Parameters

- **r** –
- **amp** –
- **sigma** –

Returns

density_2d(*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

derivatives (*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function

function (*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)
returns Gaussian

hessian (*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'sigma': 0}

mass_2d (*R*, *amp*, *sigma*)

Parameters

- **R** –
- **amp** –
- **sigma** –

Returns

mass_2d_lens (*R*, *amp*, *sigma*)

Parameters

- **R** –
- **amp** –
- **sigma** –

Returns

mass_3d (*R*, *amp*, *sigma*)

Parameters

- **R** –
- **amp** –
- **sigma** –

Returns

mass_3d_lens (*R*, *amp*, *sigma*)

Parameters

- **R** –
- **amp** –
- **sigma** –

Returns

param_names = ['amp', 'sigma', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'sigma': 100}

lenstronomy.LensModel.Profiles.gaussian_potential module

class Gaussian(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

this class contains functions to evaluate a Gaussian function and calculates its derivative and hessian matrix

derivatives(*x*, *y*, *amp*, *sigma_x*, *sigma_y*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function

function(*x*, *y*, *amp*, *sigma_x*, *sigma_y*, *center_x=0*, *center_y=0*)
returns Gaussian

hessian(*x*, *y*, *amp*, *sigma_x*, *sigma_y*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'sigma': 0}

param_names = ['amp', 'sigma_x', 'sigma_y', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'sigma': 100}

lenstronomy.LensModel.Profiles.general_nfw module

class GNFW(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

This class contains a double power law profile with flexible inner and outer logarithmic slopes g and n

$$\rho(r) = \frac{\rho_0}{r^{\gamma}} \frac{Rs^n}{\left(r^2 + Rs^2\right)^{(n - \gamma)/2}}$$

For g = 1.0 and n=3, it is approximately the same as an NFW profile The original reference is[1].

TODO: implement the gravitational potential for this profile

alpha2rho0(*alpha_Rs*, *Rs*, *gamma_inner*, *gamma_outer*)
convert angle at Rs into rho0

Parameters

- **alpha_Rs** – deflection angle at RS
- **Rs** – scale radius
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs

Returns density normalization (characteristic density)

static density(*R*, *Rs*, *rho0*, *gamma_inner*, *gamma_outer*)
three dimensional NFW profile

Parameters

- **R** – radius of interest
- **rho0** – central density normalization
- **gamma_inner** – logarithmic profile slope interior to Rs

[1] Munoz, Kochanek and Keeton, (2001), astro-ph/0103009, doi:10.1086/322314

- **gamma_outer** – logarithmic profile slope outside Rs

Returns rho(R) density

density_2d (*x*, *y*, *Rs*, *rho0*, *gamma_inner*, *gamma_outer*, *center_x=0*, *center_y=0*)
projected two dimensional profile

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **rho0** – density normalization at Rs
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs
- **center_x** – profile center (same units as x)
- **center_y** – profile center (same units as x)

Returns Epsilon(R) projected density at radius R

density_lens (*r*, *Rs*, *alpha_Rs*, *gamma_inner*, *gamma_outer*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – 3d radios
- **Rs** – scale radius
- **alpha_Rs** – deflection at Rs
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs

Returns density rho(r)

derivatives (*x*, *y*, *Rs*, *alpha_Rs*, *gamma_inner*, *gamma_outer*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function which are the deflection angles

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection angle in x, deflection angle in y

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *gamma_inner*, *gamma_outer*, *center_x=0*, *center_y=0*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_2d(*R*, *Rs*, *rho0*, *gamma_inner*, *gamma_outer*)
mass enclosed a 2d cylinder or projected radius R

Parameters

- **R** – 3d radius
- **Rs** – scale radius
- **rho0** – central density normalization
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs

Returns mass in cylinder

static mass_3d(*r*, *Rs*, *rho0*, *gamma_inner*, *gamma_outer*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – 3d radius
- **Rs** – scale radius
- **rho0** – density normalization
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs

Returns M(<r)

mass_3d_lens(*r*, *Rs*, *alpha_Rs*, *gamma_inner*, *gamma_outer*)
mass enclosed a 3d sphere or radius r. This function takes as input the lensing parameterization.

Parameters

- **r** – 3d radius
- **Rs** – scale radius
- **alpha_Rs** – deflection angle at Rs
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs

Returns M(<r)

nfwAlpha (*R*, *Rs*, *rho0*, *gamma_inner*, *gamma_outer*, *ax_x*, *ax_y*)
deflection angel of NFW profile (times Sigma_crit D_OL) along the projection to coordinate 'axis'

Parameters

- **R** – 3d radius
- **Rs** – scale radius
- **rho0** – central density normalization
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs
- **ax_x** – x coordinate relative to center
- **ax_y** – y coordinate relative to center

Returns Epsilon(R) projected density at radius R

nfwGamma (*R*, *Rs*, *rho0*, *gamma_inner*, *gamma_outer*, *ax_x*, *ax_y*)
shear gamma of NFW profile (times Sigma_crit) along the projection to coordinate 'axis'

Parameters

- **R** – 3d radius
- **Rs** – scale radius
- **rho0** – central density normalization
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs
- **ax_x** – x coordinate relative to center
- **ax_y** – y coordinate relative to center

Returns Epsilon(R) projected density at radius R

param_names = ['Rs', 'alpha_Rs', 'center_x', 'center_y', 'gamma_inner', 'gamma_outer']

profile_name = 'GNFW'

rho02alpha (*rho0*, *Rs*, *gamma_inner*, *gamma_outer*)
convert rho0 to angle at Rs

Parameters

- **rho0** – density normalization (characteristic density)
- **Rs** – scale radius
- **gamma_inner** – logarithmic profile slope interior to Rs
- **gamma_outer** – logarithmic profile slope outside Rs

Returns deflection angle at RS

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.hernquist module

class Hernquist(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class to compute the Hernquist 1990 model, which is in 3d: rho(r) = rho0 / (r/Rs * (1 + (r/Rs))**3)

in lensing terms, the normalization parameter 'sigma0' is defined such that the deflection at projected RS leads to alpha = 2./3 * Rs * sigma0

density(*r*, *rho0*, *Rs*)

computes the 3-d density

Parameters

- **r** – 3-d radius
- **rho0** – density normalization
- **Rs** – Hernquist radius

Returns density at radius r

density_2d(*x*, *y*, *rho0*, *Rs*, *center_x=0*, *center_y=0*)

projected density along the line of sight at coordinate (x, y)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **rho0** – density normalization
- **Rs** – Hernquist radius
- **center_x** – x-center of the profile
- **center_y** – y-center of the profile

Returns projected density

density_lens(*r*, *sigma0*, *Rs*)

Density as a function of 3d radius in lensing parameters This function converts the lensing definition sigma0 into the 3d density

Parameters

- **r** – 3d radius
- **sigma0** – rho0 * Rs (units of projected density)
- **Rs** – Hernquist radius

Returns enclosed mass in 3d

derivatives(*x*, *y*, *sigma0*, *Rs*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate position (units of angle)
- **y** – y-coordinate position (units of angle)
- **sigma0** – normalization parameter defined such that the deflection at projected RS leads to alpha = 2./3 * Rs * sigma0
- **Rs** – Hernquist radius in units of angle

- **center_x** – x-center of the profile (units of angle)
- **center_y** – y-center of the profile (units of angle)

Returns derivative of function (deflection angles in x- and y-direction)

function (*x*, *y*, *sigma0*, *Rs*, *center_x=0*, *center_y=0*)
lensing potential

Parameters

- **x** – x-coordinate position (units of angle)
- **y** – y-coordinate position (units of angle)
- **sigma0** – normalization parameter defined such that the deflection at projected RS leads to alpha = 2./3 * Rs * sigma0
- **Rs** – Hernquist radius in units of angle
- **center_x** – x-center of the profile (units of angle)
- **center_y** – y-center of the profile (units of angle)

Returns lensing potential at (x,y)

grav_pot (*x*, *y*, *rho0*, *Rs*, *center_x=0*, *center_y=0*)
#TODO decide whether these functions are needed or not

gravitational potential (modulo 4 pi G and rho0 in appropriate units) :param x: x-coordinate position (units of angle) :param y: y-coordinate position (units of angle) :param rho0: density normalization parameter of Hernquist profile :param Rs: Hernquist radius in units of angle :param center_x: x-center of the profile (units of angle) :param center_y: y-center of the profile (units of angle) :return: gravitational potential at projected radius

hessian (*x*, *y*, *sigma0*, *Rs*, *center_x=0*, *center_y=0*)
Hessian terms of the function

Parameters

- **x** – x-coordinate position (units of angle)
- **y** – y-coordinate position (units of angle)
- **sigma0** – normalization parameter defined such that the deflection at projected RS leads to alpha = 2./3 * Rs * sigma0
- **Rs** – Hernquist radius in units of angle
- **center_x** – x-center of the profile (units of angle)
- **center_y** – y-center of the profile (units of angle)

Returns df/dxdx, df/dxdy, df/dydx, df/dydy

lower_limit_default = {'Rs': 0, 'center_x': -100, 'center_y': -100, 'sigma0': 0}

mass_2d (*r*, *rho0*, *Rs*)
mass enclosed projected 2d sphere of radius r

Parameters

- **r** – projected radius
- **rho0** – density normalization
- **Rs** – Hernquist radius

Returns mass enclosed 2d projected radius

mass_2d_lens (*r*, *sigma0*, *Rs*)
mass enclosed projected 2d sphere of radius r Same as mass_2d but with input normalization in units of projected density :param r: projected radius :param sigma0: rho0 * Rs (units of projected density) :param Rs: Hernquist radius :return: mass enclosed 2d projected radius

mass_3d (*r*, *rho0*, *Rs*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – 3-d radius within the mass is integrated (same distance units as density definition)
- **rho0** – density normalization
- **Rs** – Hernquist radius

Returns enclosed mass

mass_3d_lens (*r*, *sigma0*, *Rs*)
mass enclosed a 3d sphere or radius r for lens parameterisation This function converts the lensing definition sigma0 into the 3d density

Parameters

- **sigma0** – rho0 * Rs (units of projected density)
- **Rs** – Hernquist radius

Returns enclosed mass in 3d

mass_tot (*rho0*, *Rs*)
total mass within the profile :param rho0: density normalization :param Rs: Hernquist radius :return: total mass within profile

param_names = ['sigma0', 'Rs', 'center_x', 'center_y']

rho2sigma (*rho0*, *Rs*)
converts 3d density into 2d projected density parameter :param rho0: 3d density normalization of Hernquist model :param Rs: Hernquist radius :return: sigma0 defined quantity in projected units

sigma2rho (*sigma0*, *Rs*)
converts projected density parameter (in units of deflection) into 3d density parameter :param sigma0: density defined quantity in projected units :param Rs: Hernquist radius :return: rho0 the 3d density normalization of Hernquist model

upper_limit_default = {'Rs': 100, 'center_x': 100, 'center_y': 100, 'sigma0': 100}

lenstronomy.LensModel.Profiles.hernquist_ellipse module

class Hernquist_Ellipse
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions for the elliptical Hernquist profile. Ellipticity is defined in the potential.

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

density (*r*, *rho0*, *Rs*, *e1=0*, *e2=0*)
computes the 3-d density

Parameters

- **r** – 3-d radius
- **rho0** – density normalization
- **Rs** – Hernquist radius

Returns density at radius r

density_2d (*x*, *y*, *rho0*, *Rs*, *e1=0*, *e2=0*, *center_x=0*, *center_y=0*)
projected density along the line of sight at coordinate (x, y)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **rho0** – density normalization
- **Rs** – Hernquist radius
- **center_x** – x-center of the profile
- **center_y** – y-center of the profile

Returns projected density

density_lens (*r*, *sigma0*, *Rs*, *e1=0*, *e2=0*)
Density as a function of 3d radius in lensing parameters This function converts the lensing definition sigma0 into the 3d density

Parameters

- **r** – 3d radius
- **sigma0** – rho0 * Rs (units of projected density)
- **Rs** – Hernquist radius

Returns enclosed mass in 3d

derivatives (*x*, *y*, *sigma0*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function (integral of NFW)

function (*x*, *y*, *sigma0*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns double integral of NFW profile

hessian (*x*, *y*, *sigma0*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e

mass_2d (*r*, *rho0*, *Rs*, *e1=0*, *e2=0*)
mass enclosed projected 2d sphere of radius r

Parameters

- **r** – projected radius
- **rho0** – density normalization
- **Rs** – Hernquist radius

Returns mass enclosed 2d projected radius

mass_2d_lens (*r*, *sigma0*, *Rs*, *e1=0*, *e2=0*)
mass enclosed projected 2d sphere of radius r Same as mass_2d but with input normalization in units of

projected density :param r: projected radius :param sigma0: rho0 * Rs (units of projected density) :param Rs: Hernquist radius :return: mass enclosed 2d projected radius

mass_3d (*r*, *rho0*, *Rs*, *e1=0*, *e2=0*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – 3-d radius within the mass is integrated (same distance units as density definition)
- **rho0** – density normalization
- **Rs** – Hernquist radius

Returns enclosed mass

mass_3d_lens (*r*, *sigma0*, *Rs*, *e1=0*, *e2=0*)
mass enclosed a 3d sphere or radius r in lensing parameterization

Parameters

- **r** – 3-d radius within the mass is integrated (same distance units as density definition)
- **sigma0** – rho0 * Rs (units of projected density)
- **Rs** – Hernquist radius

Returns enclosed mass

param_names = ['sigma0', 'Rs', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2

lenstronomy.LensModel.Profiles.hernquist_ellipse_cse module

class HernquistEllipseCSE
Bases: *lenstronomy.LensModel.Profiles.hernquist_ellipse.Hernquist_Ellipse*

this class contains functions for the elliptical Hernquist profile. Ellipticity is defined in the convergence. Approximation with CSE profile introduced by Oguri 2021: https://arxiv.org/pdf/2106.11464.pdf

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

derivatives (*x*, *y*, *sigma0*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function (integral of NFW)

function (*x*, *y*, *sigma0*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns double integral of NFW profile

hessian (*x*, *y*, *sigma0*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e

param_names = ['sigma0', 'Rs', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2

lenstronomy.LensModel.Profiles.hessian module

class Hessian(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class for constant Hessian distortion (second order) The input is in the same convention as the LensModel.hessian() output.

derivatives(*x*, *y*, *f_xx*, *f_yy*, *f_xy*, *f_yx*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **f_xx** – dalpha_x/dx
- **f_yy** – dalpha_y/dy
- **f_xy** – dalpha_x/dy
- **f_yx** – dalpha_y/dx
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns deflection angles

function(*x*, *y*, *f_xx*, *f_yy*, *f_xy*, *f_yx*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **f_xx** – dalpha_x/dx
- **f_yy** – dalpha_y/dy
- **f_xy** – dalpha_x/dy
- **f_yx** – dalpha_y/dx
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns lensing potential

hessian(*x*, *y*, *f_xx*, *f_yy*, *f_xy*, *f_yx*, *ra_0=0*, *dec_0=0*)

Hessian. Attention: If f_xy != f_yx then this function is not accurate!

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **f_xx** – dalpha_x/dx
- **f_yy** – dalpha_y/dy
- **f_xy** – dalpha_x/dy
- **f_yx** – dalpha_y/dx
- **ra_0** – x/ra position where shear deflection is 0

- **dec_0** – y/dec position where shear deflection is 0

Returns f_xx, f_yy, f_xy

`lower_limit_default = {'dec_0': -100, 'f_xx': -100, 'f_xy': -100, 'f_yx': -100, 'f`

`param_names = ['f_xx', 'f_yy', 'f_xy', 'f_yx', 'ra_0', 'dec_0']`

`upper_limit_default = {'dec_0': 100, 'f_xx': 100, 'f_xy': 100, 'f_yx': 100, 'f_yy'`

lenstronomy.LensModel.Profiles.interpol module

class Interpol(*grid=False*, *min_grid_number=100*, *kwargs_spline=None*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class which uses an interpolation of a lens model and its first and second order derivatives

See also the tests in lenstronomy.test.test_LensModel.test_Profiles.test_interpol.py for example use cases as checks against known analytic models.

The deflection angle is in the same convention as the one in the LensModel module, meaning that: source position = image position - deflection angle

__init__(*grid=False*, *min_grid_number=100*, *kwargs_spline=None*)

Parameters

- **grid** – bool, if True, computes the calculation on a grid
- **min_grid_number** – minimum numbers of positions to compute the interpolation on a grid, otherwise in a loop
- **kwargs_spline** – keyword arguments for the scipy.interpolate.RectBivariateSpline() interpolation (optional) if =None, a default linear interpolation is chosen.

derivatives(*x*, *y*, *grid_interp_x=None*, *grid_interp_y=None*, *f_=None*, *f_x=None*, *f_y=None*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

returns df/dx and df/dy of the function

Parameters

- **x** – x-coordinate (angular position), float or numpy array
- **y** – y-coordinate (angular position), float or numpy array
- **grid_interp_x** – numpy array (ascending) to mark the x-direction of the interpolation grid
- **grid_interp_y** – numpy array (ascending) to mark the y-direction of the interpolation grid
- **f** – 2d numpy array of lensing potential, matching the grids in grid_interp_x and grid_interp_y
- **f_x** – 2d numpy array of deflection in x-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_y** – 2d numpy array of deflection in y-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_xx** – 2d numpy array of df/dxx, matching the grids in grid_interp_x and grid_interp_y
- **f_yy** – 2d numpy array of df/dyy, matching the grids in grid_interp_x and grid_interp_y
- **f_xy** – 2d numpy array of df/dxy, matching the grids in grid_interp_x and grid_interp_y

Returns f_x, f_y at interpolated positions (x, y)

do_interp (*x_grid*, *y_grid*, *f_*, *f_x*, *f_y*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

f_interp (*x*, *y*, *x_grid=None*, *y_grid=None*, *f_=None*, *grid=False*)

f_x_interp (*x*, *y*, *x_grid=None*, *y_grid=None*, *f_x=None*, *grid=False*)

f_xx_interp (*x*, *y*, *x_grid=None*, *y_grid=None*, *f_xx=None*, *grid=False*)

f_xy_interp (*x*, *y*, *x_grid=None*, *y_grid=None*, *f_xy=None*, *grid=False*)

f_y_interp (*x*, *y*, *x_grid=None*, *y_grid=None*, *f_y=None*, *grid=False*)

f_yy_interp (*x*, *y*, *x_grid=None*, *y_grid=None*, *f_yy=None*, *grid=False*)

function (*x*, *y*, *grid_interp_x=None*, *grid_interp_y=None*, *f_=None*, *f_x=None*, *f_y=None*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

Parameters

- **x** – x-coordinate (angular position), float or numpy array
- **y** – y-coordinate (angular position), float or numpy array
- **grid_interp_x** – numpy array (ascending) to mark the x-direction of the interpolation grid
- **grid_interp_y** – numpy array (ascending) to mark the y-direction of the interpolation grid
- **f** – 2d numpy array of lensing potential, matching the grids in grid_interp_x and grid_interp_y
- **f_x** – 2d numpy array of deflection in x-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_y** – 2d numpy array of deflection in y-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_xx** – 2d numpy array of df/dxx, matching the grids in grid_interp_x and grid_interp_y
- **f_yy** – 2d numpy array of df/dyy, matching the grids in grid_interp_x and grid_interp_y
- **f_xy** – 2d numpy array of df/dxy, matching the grids in grid_interp_x and grid_interp_y

Returns potential at interpolated positions (x, y)

hessian (*x*, *y*, *grid_interp_x=None*, *grid_interp_y=None*, *f_=None*, *f_x=None*, *f_y=None*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters

- **x** – x-coordinate (angular position), float or numpy array
- **y** – y-coordinate (angular position), float or numpy array
- **grid_interp_x** – numpy array (ascending) to mark the x-direction of the interpolation grid
- **grid_interp_y** – numpy array (ascending) to mark the y-direction of the interpolation grid
- **f** – 2d numpy array of lensing potential, matching the grids in grid_interp_x and grid_interp_y

- **f_x** – 2d numpy array of deflection in x-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_y** – 2d numpy array of deflection in y-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_xx** – 2d numpy array of df/dxx, matching the grids in grid_interp_x and grid_interp_y
- **f_yy** – 2d numpy array of df/dyy, matching the grids in grid_interp_x and grid_interp_y
- **f_xy** – 2d numpy array of df/dxy, matching the grids in grid_interp_x and grid_interp_y

Returns f_xx, f_xy, f_yx, f_yy at interpolated positions (x, y)

lower_limit_default = {}

param_names = ['grid_interp_x', 'grid_interp_y', 'f_', 'f_x', 'f_y', 'f_xx', 'f_yy', '

upper_limit_default = {}

class InterpolScaled(*grid=True*, *min_grid_number=100*, *kwargs_spline=None*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class for handling an interpolated lensing map and has the freedom to scale its lensing effect. Applications are e.g. mass to light ratio.

__init__(*grid=True*, *min_grid_number=100*, *kwargs_spline=None*)

Parameters

- **grid** – bool, if True, computes the calculation on a grid
- **min_grid_number** – minimum numbers of positions to compute the interpolation on a grid
- **kwargs_spline** – keyword arguments for the scipy.interpolate.RectBivariateSpline() interpolation (optional) if =None, a default linear interpolation is chosen.

derivatives(*x*, *y*, *scale_factor=1*, *grid_interp_x=None*, *grid_interp_y=None*, *f_=None*, *f_x=None*, *f_y=None*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

Parameters

- **x** – x-coordinate (angular position), float or numpy array
- **y** – y-coordinate (angular position), float or numpy array
- **scale_factor** – float, overall scaling of the lens model relative to the input interpolation grid
- **grid_interp_x** – numpy array (ascending) to mark the x-direction of the interpolation grid
- **grid_interp_y** – numpy array (ascending) to mark the y-direction of the interpolation grid
- **f** – 2d numpy array of lensing potential, matching the grids in grid_interp_x and grid_interp_y
- **f_x** – 2d numpy array of deflection in x-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_y** – 2d numpy array of deflection in y-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_xx** – 2d numpy array of df/dxx, matching the grids in grid_interp_x and grid_interp_y
- **f_yy** – 2d numpy array of df/dyy, matching the grids in grid_interp_x and grid_interp_y

- **f_xy** – 2d numpy array of df/dxy, matching the grids in grid_interp_x and grid_interp_y

Returns deflection angles in x- and y-direction at position (x, y)

function(*x*, *y*, *scale_factor=1*, *grid_interp_x=None*, *grid_interp_y=None*, *f_=None*, *f_x=None*, *f_y=None*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

Parameters

- **x** – x-coordinate (angular position), float or numpy array
- **y** – y-coordinate (angular position), float or numpy array
- **scale_factor** – float, overall scaling of the lens model relative to the input interpolation grid
- **grid_interp_x** – numpy array (ascending) to mark the x-direction of the interpolation grid
- **grid_interp_y** – numpy array (ascending) to mark the y-direction of the interpolation grid
- **f** – 2d numpy array of lensing potential, matching the grids in grid_interp_x and grid_interp_y
- **f_x** – 2d numpy array of deflection in x-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_y** – 2d numpy array of deflection in y-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_xx** – 2d numpy array of df/dxx, matching the grids in grid_interp_x and grid_interp_y
- **f_yy** – 2d numpy array of df/dyy, matching the grids in grid_interp_x and grid_interp_y
- **f_xy** – 2d numpy array of df/dxy, matching the grids in grid_interp_x and grid_interp_y

Returns potential at interpolated positions (x, y)

hessian(*x*, *y*, *scale_factor=1*, *grid_interp_x=None*, *grid_interp_y=None*, *f_=None*, *f_x=None*, *f_y=None*, *f_xx=None*, *f_yy=None*, *f_xy=None*)

Parameters

- **x** – x-coordinate (angular position), float or numpy array
- **y** – y-coordinate (angular position), float or numpy array
- **scale_factor** – float, overall scaling of the lens model relative to the input interpolation grid
- **grid_interp_x** – numpy array (ascending) to mark the x-direction of the interpolation grid
- **grid_interp_y** – numpy array (ascending) to mark the y-direction of the interpolation grid
- **f** – 2d numpy array of lensing potential, matching the grids in grid_interp_x and grid_interp_y
- **f_x** – 2d numpy array of deflection in x-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_y** – 2d numpy array of deflection in y-direction, matching the grids in grid_interp_x and grid_interp_y
- **f_xx** – 2d numpy array of df/dxx, matching the grids in grid_interp_x and grid_interp_y

- **f_yy** – 2d numpy array of df/dyy, matching the grids in grid_interp_x and grid_interp_y
- **f_xy** – 2d numpy array of df/dxy, matching the grids in grid_interp_x and grid_interp_y

Returns second derivatives of the lensing potential f_xx, f_yy, f_xy at position (x, y)

lower_limit_default = {'scale_factor': 0}

param_names = ['scale_factor', 'grid_interp_x', 'grid_interp_y', 'f_', 'f_x', 'f_y', '

upper_limit_default = {'scale_factor': 100}

lenstronomy.LensModel.Profiles.multi_gaussian_kappa module

class MultiGaussianKappa

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

__init__()

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

density(*r, amp, sigma, scale_factor=1*)

Parameters

- **r** –
- **amp** –
- **sigma** –

Returns

density_2d(*x, y, amp, sigma, center_x=0, center_y=0, scale_factor=1*)

Parameters

- **R** –
- **am** –
- **sigma_x** –
- **sigma_y** –

Returns

derivatives(*x, y, amp, sigma, center_x=0, center_y=0, scale_factor=1*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

function(*x, y, amp, sigma, center_x=0, center_y=0, scale_factor=1*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

hessian (*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*, *scale_factor=1*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'sigma': 0}

mass_3d_lens (*R*, *amp*, *sigma*, *scale_factor=1*)

Parameters

- **R** –
- **amp** –
- **sigma** –

Returns

param_names = ['amp', 'sigma', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'sigma': 100}

class MultiGaussianKappaEllipse

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

__init__ ()

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

density (*r*, *amp*, *sigma*, *e1*, *e2*, *scale_factor=1*)

Parameters

- **r** –
- **amp** –
- **sigma** –

Returns

density_2d (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*, *scale_factor=1*)

Parameters

- **R** –
- **am** –
- **sigma_x** –
- **sigma_y** –

Returns

derivatives (*x, y, amp, sigma, e1, e2, center_x=0, center_y=0, scale_factor=1*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

function (*x, y, amp, sigma, e1, e2, center_x=0, center_y=0, scale_factor=1*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

hessian (*x, y, amp, sigma, e1, e2, center_x=0, center_y=0, scale_factor=1*)

Parameters

- **x** –
- **y** –
- **amp** –
- **sigma** –
- **center_x** –
- **center_y** –

Returns

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

mass_3d_lens (*R, amp, sigma, e1, e2, scale_factor=1*)

Parameters

- **R** –

- **amp** –
- **sigma** –

Returns

param_names = ['amp', 'sigma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

lenstronomy.LensModel.Profiles.multipole module

class Multipole(**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

This class contains a multipole contribution (for 1 component with m>=2) This uses the same definitions as Xu et al.(2013) in Appendix B3 https://arxiv.org/pdf/1307.4220.pdf m : int, multipole order, m>=2 a_m : float, multipole strength phi_m : float, multipole orientation in radian

derivatives(*x*, *y*, *m*, *a_m*, *phi_m*, *center_x=0*, *center_y=0*)

Deflection of a multipole contribution (for 1 component with m>=2) This uses the same definitions as Xu et al.(2013) in Appendix B3 https://arxiv.org/pdf/1307.4220.pdf

Parameters

- **m** – int, multipole order, m>=2
- **a_m** – float, multipole strength
- **phi_m** – float, multipole orientation in radian
- **center_x** – x-position
- **center_y** – x-position

Returns deflection angles alpha_x, alpha_y

function(*x*, *y*, *m*, *a_m*, *phi_m*, *center_x=0*, *center_y=0*)

Lensing potential of multipole contribution (for 1 component with m>=2) This uses the same definitions as Xu et al.(2013) in Appendix B3 https://arxiv.org/pdf/1307.4220.pdf

Parameters

- **m** – int, multipole order, m>=2
- **a_m** – float, multipole strength
- **phi_m** – float, multipole orientation in radian
- **center_x** – x-position
- **center_y** – x-position

Returns lensing potential

hessian(*x*, *y*, *m*, *a_m*, *phi_m*, *center_x=0*, *center_y=0*)

Hessian of a multipole contribution (for 1 component with m>=2) This uses the same definitions as Xu et al.(2013) in Appendix B3 https://arxiv.org/pdf/1307.4220.pdf

Parameters

- **m** – int, multipole order, m>=2
- **a_m** – float, multipole strength
- **phi_m** – float, multipole orientation in radian

- **center_x** – x-position
- **center_y** – x-position

Returns f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'a_m': 0, 'center_x': -100, 'center_y': -100, 'm': 2, 'phi_m

param_names = ['m', 'a_m', 'phi_m', 'center_x', 'center_y']

upper_limit_default = {'a_m': 100, 'center_x': 100, 'center_y': 100, 'm': 100, 'ph

lenstronomy.LensModel.Profiles.nfw module

class NFW(*interpol=False*, *num_interp_X=1000*, *max_interp_X=10*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions concerning the NFW profile

relation are: R_200 = c * Rs The definition of 'Rs' is in angular (arc second) units and the normalization is put in in regards to a deflection angle at 'Rs' - 'alpha_Rs'. To convert a physical mass and concentration definition into those lensing quantities for a specific redshift configuration and cosmological model, you can find routines in lenstronomy.Cosmo.lens_cosmo.py

```
>>> from lenstronomy.Cosmo.lens_cosmo import LensCosmo
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Ob0=0.05)
>>> lens_cosmo = LensCosmo(z_lens=0.5, z_source=1.5, cosmo=cosmo)
```

Here we compute the angular scale of Rs on the sky (in arc seconds) and the deflection angle at Rs (in arc seconds):

```
>>> Rs_angle, alpha_Rs = lens_cosmo.nfw_physical2angle(M=10**13, c=6)
```

And here we perform the inverse calculation given Rs_angle and alpha_Rs to return the physical halo properties.

```
>>> rho0, Rs, c, r200, M200 = lens_cosmo.nfw_angle2physical(Rs_angle=Rs_angle,
↪alpha_Rs=alpha_Rs)
```

The lens model calculation uses angular units as arguments! So to execute a deflection angle calculation one uses

```
>>> from lenstronomy.LensModel.Profiles.nfw import NFW
>>> nfw = NFW()
>>> alpha_x, alpha_y = nfw.derivatives(x=1, y=1, Rs=Rs_angle, alpha_Rs=alpha_Rs,
↪center_x=0, center_y=0)
```

F_(*X*)

computes h()

Parameters X –

Returns

__init__(*interpol=False*, *num_interp_X=1000*, *max_interp_X=10*)

Parameters

- **interpol** – bool, if True, interpolates the functions F(), g() and h()

- **num_interp_X** – int (only considered if interpol=True), number of interpolation elements in units of r/r_s
- **max_interp_X** – float (only considered if interpol=True), maximum r/r_s value to be interpolated (returning zeros outside)

static alpha2rho0(*alpha_Rs*, *Rs*)
convert angle at Rs into rho0

Parameters

- **alpha_Rs** – deflection angle at RS
- **Rs** – scale radius

Returns density normalization (characteristic density)

density(*R*, *Rs*, *rho0*)
three dimensional NFW profile

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)

Returns rho(R) density

density_2d(*x*, *y*, *Rs*, *rho0*, *center_x=0*, *center_y=0*)
projected two dimensional NFW profile (kappa*Sigma_crit)

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r200** (*float>0*) – radius of (sub)halo

Returns Epsilon(R) projected density at radius R

density_lens(*r*, *Rs*, *alpha_Rs*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – 3d radios
- **Rs** – turn-over radius of NFW profile
- **alpha_Rs** – deflection at Rs

Returns density rho(r)

derivatives(*x*, *y*, *Rs*, *alpha_Rs*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function (integral of NFW), which are the deflection angles

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit

- **alpha_Rs** – deflection (angular units) at projected Rs
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection angle in x, deflection angle in y

function (*x*, *y*, *Rs*, *alpha_Rs*, *center_x=0*, *center_y=0*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns lensing potential

g_ (*X*)

computes h()

Parameters X (*float >0*) – R/Rs

Returns

h_ (*X*)

computes h()

Parameters X (*float >0*) – R/Rs

Returns h(X)

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *center_x=0*, *center_y=0*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100}

mass_2d (*R*, *Rs*, *rho0*)

mass enclosed a 2d cylinder or projected radius R :param R: projected radius :param Rs: scale radius :param rho0: density normalization (characteristic density) :return: mass in cylinder

mass_2d_lens (*R*, *Rs*, *alpha_Rs*)

Parameters

- **R** – projected radius

- **Rs** – scale radius
- **alpha_Rs** – deflection (angular units) at projected Rs

Returns mass enclosed 2d cylinder <R

mass_3d (*r*, *Rs*, *rho0*)
mass enclosed a 3d sphere or radius r

Parameters
- **r** – 3d radius
- **Rs** – scale radius
- **rho0** – density normalization (characteristic density)

Returns M(<r)

mass_3d_lens (*r*, *Rs*, *alpha_Rs*)
mass enclosed a 3d sphere or radius r. This function takes as input the lensing parameterization.

Parameters
- **r** – 3d radius
- **Rs** – scale radius
- **alpha_Rs** – deflection (angular units) at projected Rs

Returns M(<r)

nfwAlpha (*R*, *Rs*, *rho0*, *ax_x*, *ax_y*)
deflection angel of NFW profile (times Sigma_crit D_OL) along the projection to coordinate 'axis'

Parameters
- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r200** (*float>0*) – radius of (sub)halo
- **axis** (*same as R*) – projection to either x- or y-axis

Returns Epsilon(R) projected density at radius R

nfwGamma (*R*, *Rs*, *rho0*, *ax_x*, *ax_y*)
shear gamma of NFW profile (times Sigma_crit) along the projection to coordinate 'axis'

Parameters
- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r200** (*float>0*) – radius of (sub)halo
- **axis** (*same as R*) – projection to either x- or y-axis

Returns Epsilon(R) projected density at radius R

nfwPot (*R*, *Rs*, *rho0*)
lensing potential of NFW profile (Sigma_crit D_OL**2)

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r200** (*float>0*) – radius of (sub)halo

Returns Epsilon(R) projected density at radius R

param_names = ['Rs', 'alpha_Rs', 'center_x', 'center_y']

profile_name = 'NFW'

static rho02alpha(*rho0*, *Rs*)
convert rho0 to angle at Rs

Parameters

- **rho0** – density normalization (characteristic density)
- **Rs** – scale radius

Returns deflection angle at RS

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.nfw_ellipse module

class NFW_ELLIPSE(*interpol=False*, *num_interp_X=1000*, *max_interp_X=10*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions concerning the NFW profile with an ellipticity defined in the potential parameterization of alpha_Rs and Rs is the same as for the spherical NFW profile

from Glose & Kneib: https://cds.cern.ch/record/529584/files/0112138.pdf

relation are: R_200 = c * Rs

__init__(*interpol=False*, *num_interp_X=1000*, *max_interp_X=10*)

Parameters

- **interpol** – bool, if True, interpolates the functions F(), g() and h()
- **num_interp_X** – int (only considered if interpol=True), number of interpolation elements in units of r/r_s
- **max_interp_X** – float (only considered if interpol=True), maximum r/r_s value to be interpolated (returning zeros outside)

density_lens(*r*, *Rs*, *alpha_Rs*, *e1=1*, *e2=0*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – 3d radios
- **Rs** – turn-over radius of NFW profile
- **alpha_Rs** – deflection at Rs

Returns density rho(r)

derivatives (*x*, *y*, *Rs*, *alpha_Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function, calculated as an elliptically distorted deflection angle of the spherical NFW profile

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection in x-direction, deflection in y-direction

function (*x*, *y*, *Rs*, *alpha_Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns elliptically distorted NFW lensing potential

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns lensing potential

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^f/dy^2, d^2/dxdy the calculation is performed as a numerical differential from the deflection field. Analytical relations are possible

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_3d_lens (*r*, *Rs*, *alpha_Rs*, *e1=1*, *e2=0*)

Parameters

- **r** – radius (in angular units)
- **Rs** –
- **alpha_Rs** –
- **e1** –
- **e2** –

Returns

param_names = ['Rs', 'alpha_Rs', 'e1', 'e2', 'center_x', 'center_y']

profile_name = 'NFW_ELLIPSE'

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.nfw_ellipse_cse module

class NFW_ELLIPSE_CSE (*high_accuracy=True*)

Bases: `lenstronomy.LensModel.Profiles.nfw_ellipse.NFW_ELLIPSE`

this class contains functions concerning the NFW profile with an ellipticity defined in the convergence parameterization of alpha_Rs and Rs is the same as for the spherical NFW profile Approximation with CSE profile introduced by Oguri 2021: https://arxiv.org/pdf/2106.11464.pdf Match to NFW using CSEs is approximate: kappa matches to ~1-2%

relation are: R_200 = c * Rs

__init__ (*high_accuracy=True*)

Parameters high_accuracy (*boolean*) – if True uses a more accurate larger set of CSE profiles (see Oguri 2021)

derivatives (*x*, *y*, *Rs*, *alpha_Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)

returns df/dx and df/dy of the function, calculated as an elliptically distorted deflection angle of the spherical NFW profile

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection in x-direction, deflection in y-direction

function (*x*, *y*, *Rs*, *alpha_Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns elliptically distorted NFW lensing potential

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns lensing potential

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^f/dy^2, d^2/dxdy the calculation is performed as a numerical differential from the deflection field. Analytical relations are possible.

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

param_names = ['Rs', 'alpha_Rs', 'e1', 'e2', 'center_x', 'center_y']

profile_name = 'NFW_ELLIPSE_CSE'

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.nfw_mass_concentration module

class NFWMC (*z_lens*, *z_source*, *cosmo=None*, *static=False*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions parameterises the NFW profile with log10 M200 and the concentration rs/r200 relation are: R_200 = c * Rs

ATTENTION: the parameterization is cosmology and redshift dependent! The cosmology to connect mass and deflection relations is fixed to default H0=70km/s Omega_m=0.3 flat LCDM. It is recommended to keep a given cosmology definition in the lens modeling as the observable reduced deflection angles are sensitive in this parameterization. If you do not want to impose a mass-concentration relation, it is recommended to use the default NFW lensing profile parameterized in reduced deflection angles.

__init__(*z_lens*, *z_source*, *cosmo=None*, *static=False*)

Parameters

- **z_lens** – redshift of lens
- **z_source** – redshift of source
- **cosmo** – astropy cosmology instance
- **static** – boolean, if True, only operates with fixed parameter values

derivatives(*x*, *y*, *logM*, *concentration*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function (integral of NFW)

function(*x*, *y*, *logM*, *concentration*, *center_x=0*, *center_y=0*)

Parameters

- **x** – angular position
- **y** – angular position
- **Rs** – angular turn over point
- **alpha_Rs** – deflection at Rs
- **center_x** – center of halo
- **center_y** – center of halo

Returns

hessian(*x*, *y*, *logM*, *concentration*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'center_x': -100, 'center_y': -100, 'concentration': 0.01, '

param_names = ['logM', 'concentration', 'center_x', 'center_y']

set_dynamic()

Returns

set_static(*logM*, *concentration*, *center_x=0*, *center_y=0*)

Parameters

- **logM** –
- **concentration** –
- **center_x** –
- **center_y** –

Returns

upper_limit_default = {'center_x': 100, 'center_y': 100, 'concentration': 1000, 'lo

lenstronomy.LensModel.Profiles.nfw_vir_trunc module

class NFWVirTrunc(*z_lens*, *z_source*, *cosmo=None*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions concerning the NFW profile that is sharply truncated at the virial radius https://arxiv.org/pdf/astro-ph/0304034.pdf

relation are: R_200 = c * Rs

__init__(*z_lens*, *z_source*, *cosmo=None*)

Parameters

- **z_lens** – redshift of lens
- **z_source** – redshift of source
- **cosmo** – astropy cosmology instance

kappa(*theta*, *logM*, *c*)

projected surface brightness

Parameters

- **theta** – radial angle from the center of the profile
- **logM** – log_10 halo mass in physical units of M_sun
- **c** – concentration of the halo; r_200 = c * r_s

Returns convergence at theta

lenstronomy.LensModel.Profiles.nie module

class NIE

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

Non-singular isothermal ellipsoid (NIE)

$$\kappa = \theta_E/2\left[s_{scale}^2 + qx^2 + y^2/q\right]1/2$$

__init__()

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

density_lens(*r*, *theta_E*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

3d mass density at 3d radius r. This function assumes spherical symmetry/ignoring the eccentricity.

Parameters

- **r** – 3d radius
- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale
- **center_x** – profile center
- **center_y** – profile center

Returns 3d mass density at 3d radius r

derivatives (*x*, *y*, *theta_E*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane
- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale
- **center_x** – profile center
- **center_y** – profile center

Returns alpha_x, alpha_y

function (*x*, *y*, *theta_E*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane
- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale
- **center_x** – profile center
- **center_y** – profile center

Returns lensing potential

hessian (*x*, *y*, *theta_E*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane
- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale
- **center_x** – profile center
- **center_y** – profile center

Returns f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

mass_3d_lens (*r*, *theta_E*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)
mass enclosed a 3d radius r. This function assumes spherical symmetry/ignoring the eccentricity.

Parameters

- **r** – 3d radius
- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale
- **center_x** – profile center
- **center_y** – profile center

Returns 3d mass density at 3d radius r

param_conv (*theta_E*, *e1*, *e2*, *s_scale*)

param_names = ['theta_E', 'e1', 'e2', 's_scale', 'center_x', 'center_y']

set_dynamic ()

Returns

set_static (*theta_E*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale
- **center_x** – profile center
- **center_y** – profile center

Returns self variables set

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 's_

class NIEMajorAxis (*diff=1e-10*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

This class contains the function and the derivatives of the non-singular isothermal ellipse. See Keeton and Kochanek 1998, https://arxiv.org/pdf/astro-ph/9705194.pdf

$$\kappa = b * (q2(s2 + x2) + y2)^{1/2}$$

__init__ (*diff=1e-10*)
Initialize self. See help(type(self)) for accurate signature.

derivatives (*x*, *y*, *b*, *s*, *q*)
returns df/dx and df/dy of the function

function (*x*, *y*, *b*, *s*, *q*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian (*x*, *y*, *b*, *s*, *q*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

static kappa (*x*, *y*, *b*, *s*, *q*)
convergence

Parameters

- **x** – major axis coordinate
- **y** – minor axis coordinate
- **b** – normalization
- **s** – smoothing scale
- **q** – axis ratio

Returns convergence

param_names = ['b', 's', 'q', 'center_x', 'center_y']

lenstronomy.LensModel.Profiles.nie_potential module

class NIE_POTENTIAL
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class implements the elliptical potential of Eq. (67) of LECTURES ON GRAVITATIONAL LENSING and Eq. (1) of Blandford & Kochanek 1987, mapped to Eq. (8) of Barnaka1998 to find the ellipticity bounds

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

derivatives (*x*, *y*, *theta_E*, *theta_c*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **theta_E** – Einstein radius (in angles)
- **theta_c** – core radius (in angles)
- **e1** – eccentricity component, x direction(dimensionless)
- **e2** – eccentricity component, y direction (dimensionless)

Returns deflection angle (in angles)

function (*x*, *y*, *theta_E*, *theta_c*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **theta_E** – Einstein radius (in angles)
- **theta_c** – core radius (in angles)
- **e1** – eccentricity component, x direction(dimensionless)

- **e2** – eccentricity component, y direction (dimensionless)

Returns lensing potential

hessian (*x*, *y*, *theta_E*, *theta_c*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **theta_E** – Einstein radius (in angles)
- **theta_c** – core radius (in angles)
- **e1** – eccentricity component, x direction(dimensionless)
- **e2** – eccentricity component, y direction (dimensionless)

Returns hessian matrix (in angles)

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': 0, 'e2': 0, 'thet

param_conv (*theta_E*, *theta_c*, *e1*, *e2*)

param_names = ['center_x', 'center_y', 'theta_E', 'theta_c', 'e1', 'e2']

set_dynamic ()

Returns

set_static (*theta_E*, *theta_c*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate in image plane
- **y** – y-coordinate in image plane
- **theta_E** – Einstein radius
- **theta_c** – core radius
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – profile center
- **center_y** – profile center

Returns self variables set

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.2, 'e2': 0.2, 'th

class NIEPotentialMajorAxis (*diff=1e-10*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

this class implements the elliptical potential of Eq. (67) of LECTURES ON GRAVITATIONAL LENSING and Eq. (1) of Blandford & Kochanek 1987, mapped to Eq. (8) of Barnaka1998 to find the ellipticity bounds

__init__ (*diff=1e-10*)
Initialize self. See help(type(self)) for accurate signature.

derivatives (*x*, *y*, *theta_E*, *theta_c*, *eps*)
returns df/dx and df/dy of the function

function (*x*, *y*, *theta_E*, *theta_c*, *eps*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian (*x, y, theta_E, theta_c, eps*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

param_names = ['theta_E', 'theta_c', 'eps', 'center_x', 'center_y']

lenstronomy.LensModel.Profiles.numerical_deflections module

class NumericalAlpha (*custom_class*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

This class allows one to incorporate any lens profile into the usage framework of lenstronomy. When creating the instance of LensModel with this lens profile, you must pass in numerical_alpha_class = CustomClass(), where CustomClass is a class with a call method that returns the x/y deflection angles. This allows one to numerically compute and interpolate deflection angles for potentially very complex mass profiles, and then use the results with lenstronomy without having to heavily modify the existing structure of the software.

__init__ (*custom_class*)

Parameters **custom_class** – a user-defined class that has a __call___ method that returns deflection angles

Code example:

```
>>> custom_class = CustomLensingClass()
>>> alpha_x, alpha_y = custom_class(x, y, **kwargs)
```

or equivalently:

```
>>> from lenstronomy.LensModel.lens_model import LensModel
>>> lens_model_list = ['NumericalAlpha']
>>> lens_model = LensModel(lens_model_list, numerical_alpha_class=custom_
↪class)
>>>> alpha_x, alpha_y = lens_model.alpha(x, y, **kwargs)
```

derivatives (*x, y, center_x=0, center_y=0, **kwargs*)

Parameters

- **x** – x coordinate [arcsec]
- **y** – x coordinate [arcsec]
- **center_x** – deflector x center [arcsec]
- **center_y** – deflector y center [arcsec]
- **kwargs** – keyword arguments for the custom profile

Returns

function (*x, y, center_x=0, center_y=0, **kwargs*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian (*x*, *y*, *center_x=0*, *center_y=0*, ***kwargs*)
Returns the components of the hessian matrix :param x: x coordinate [arcsec] :param y: y coordinate [arcsec] :param center_x: the deflector x coordinate :param center_y: the deflector y coordinate :param kwargs: keyword arguments for the profile :return: the derivatives of the deflection angles that make up the hessian matrix

lenstronomy.LensModel.Profiles.p_jaffe module

class PJaffe
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class to compute the DUAL PSEUDO ISOTHERMAL ELLIPTICAL MASS DISTRIBUTION based on Eliasdottir (2007) https://arxiv.org/pdf/0710.5636.pdf Appendix A

Module name: 'PJAFFE';

An alternative name is dPIED.

The 3D density distribution is

$$\rho(r) = \frac{\rho_0}{(1+r^2/Ra^2)(1+r^2/Rs^2)}$$

with $Rs > Ra$.

The projected density is

$$\Sigma(R) = \Sigma_0 \frac{RaRs}{Rs-Ra}\left(\frac{1}{\sqrt{Ra^2+R^2}} - \frac{1}{\sqrt{Rs^2+R^2}}\right)$$

with

$$\Sigma_0 = \pi \rho_0 \frac{RaRs}{Rs+Ra}$$

In the lensing parameterization,

$$\sigma_0 = \frac{\Sigma_0}{\Sigma_{\rm crit}}$$

__init__ ()

density (*r*, *rho0*, *Ra*, *Rs*)
computes the density

Parameters

- **r** – radial distance from the center (in 3D)
- **rho0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4

Returns density at r

density_2d (*x*, *y*, *rho0*, *Ra*, *Rs*, *center_x=0*, *center_y=0*)
projected density

Parameters

- **x** – projected coordinate on the sky
- **y** – projected coordinate on the sky
- **rho0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4
- **center_x** – center of profile
- **center_y** – center of profile

Returns projected density

derivatives (*x*, *y*, *sigma0*, *Ra*, *Rs*, *center_x=0*, *center_y=0*)
deflection angles

Parameters

- **x** – projected coordinate on the sky
- **y** – projected coordinate on the sky
- **sigma0** – sigma0/sigma_crit (see class documentation above)
- **Ra** – core radius (see class documentation above)
- **Rs** – transition radius from logarithmic slope -2 to -4 (see class documentation above)
- **center_x** – center of profile
- **center_y** – center of profile

Returns f_x, f_y

function (*x*, *y*, *sigma0*, *Ra*, *Rs*, *center_x=0*, *center_y=0*)
lensing potential

Parameters

- **x** – projected coordinate on the sky
- **y** – projected coordinate on the sky
- **sigma0** – sigma0/sigma_crit (see class documentation above)
- **Ra** – core radius (see class documentation above)
- **Rs** – transition radius from logarithmic slope -2 to -4 (see class documentation above)
- **center_x** – center of profile
- **center_y** – center of profile

Returns lensing potential

grav_pot (*r*, *rho0*, *Ra*, *Rs*)
gravitational potential (modulo 4 pi G and rho0 in appropriate units)

Parameters

- **r** – radial distance from the center (in 3D)
- **rho0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4

Returns gravitational potential (modulo 4 pi G and rho0 in appropriate units)

hessian (*x*, *y*, *sigma0*, *Ra*, *Rs*, *center_x=0*, *center_y=0*)
Hessian of lensing potential

Parameters

- **x** – projected coordinate on the sky
- **y** – projected coordinate on the sky
- **sigma0** – sigma0/sigma_crit (see class documentation above)
- **Ra** – core radius (see class documentation above)
- **Rs** – transition radius from logarithmic slope -2 to -4 (see class documentation above)
- **center_x** – center of profile
- **center_y** – center of profile

Returns f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'Ra': 0, 'Rs': 0, 'center_x': -100, 'center_y': -100, 'sigm

mass_2d (*r*, *rho0*, *Ra*, *Rs*)
mass enclosed projected 2d sphere of radius r

Parameters

- **r** – radial distance from the center in projection
- **rho0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4

Returns Sigma(<r)

mass_3d (*r*, *rho0*, *Ra*, *Rs*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – radial distance from the center (in 3D)
- **rho0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4

Returns M(<r)

mass_3d_lens (*r*, *sigma0*, *Ra*, *Rs*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units

Parameters

- **r** – radial distance from the center (in 3D)
- **sigma0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4

Returns M(<r) in angular units (modulo critical mass density)

mass_tot (*rho0*, *Ra*, *Rs*)
total mass within the profile

Parameters

- **rho0** – density normalization (see class documentation above)
- **Ra** – core radius
- **Rs** – transition radius from logarithmic slope -2 to -4

Returns total mass

param_names = ['sigma0', 'Ra', 'Rs', 'center_x', 'center_y']

rho2sigma (*rho0*, *Ra*, *Rs*)
converts 3d density into 2d projected density parameter, Equation A4 in Eliasdottir (2007)

Parameters

- **rho0** – density normalization
- **Ra** – core radius (see class documentation above)
- **Rs** – transition radius from logarithmic slope -2 to -4 (see class documentation above)

Returns projected density normalization

sigma2rho (*sigma0*, *Ra*, *Rs*)
inverse of rho2sigma()

Parameters

- **sigma0** – projected density normalization
- **Ra** – core radius (see class documentation above)
- **Rs** – transition radius from logarithmic slope -2 to -4 (see class documentation above)

Returns 3D density normalization

upper_limit_default = {'Ra': 100, 'Rs': 100, 'center_x': 100, 'center_y': 100, 'si

lenstronomy.LensModel.Profiles.p_jaffe_ellipse module

class PJaffe_Ellipse
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class to compute the DUAL PSEUDO ISOTHERMAL ELLIPTICAL MASS DISTRIBUTION based on Eliasdottir (2007) https://arxiv.org/pdf/0710.5636.pdf Appendix A with the ellipticity implemented in the potential

Module name: 'PJAFFE_ELLIPSE';

An alternative name is dPIED.

The 3D density distribution is

$$\rho(r) = \frac{\rho_0}{(1 + r^2/Ra^2)(1 + r^2/Rs^2)}$$

with $Rs > Ra$.

The projected density is

$$\Sigma(R) = \Sigma_0 \frac{RaRs}{Rs - Ra} \left(\frac{1}{\sqrt{Ra^2 + R^2}} - \frac{1}{\sqrt{Rs^2 + R^2}} \right)$$

with

$$\Sigma_0 = \pi\rho_0 \frac{RaRs}{Rs + Ra}$$

In the lensing parameterization,

$$\sigma_0 = \frac{\Sigma_0}{\Sigma_{\rm crit}}$$

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

derivatives(*x*, *y*, *sigma0*, *Ra*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function (integral of NFW)

function(*x*, *y*, *sigma0*, *Ra*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns double integral of NFW profile

hessian(*x*, *y*, *sigma0*, *Ra*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Ra': 0, 'Rs': 0, 'center_x': -100, 'center_y': -100, 'e1':

mass_3d_lens(*r*, *sigma0*, *Ra*, *Rs*, *e1=0*, *e2=0*)

Parameters

- **r** –
- **sigma0** –
- **Ra** –
- **Rs** –
- **e1** –
- **e2** –

Returns

param_names = ['sigma0', 'Ra', 'Rs', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Ra': 100, 'Rs': 100, 'center_x': 100, 'center_y': 100, 'e1

lenstronomy.LensModel.Profiles.pemd module

class PEMD(*suppress_fastell=False*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class for power law ellipse mass density profile. This class effectively calls the class SPEMD_SMOOTH with a fixed and very small central smoothing scale to perform the numerical integral using the FASTELL code by Renan Barkana.

$$\kappa(x, y) = \frac{3-\gamma}{2} \left(\frac{\theta_E}{\sqrt{qx^2 + y^2/q}}\right)^{\gamma-1}$$

with θ_E is the (circularized) Einstein radius, γ is the negative power-law slope of the 3D mass distributions, q is the minor/major axis ratio, and x and y are defined in a coordinate system aligned with the major and minor axis of the lens.

In terms of eccentricities, this profile is defined as

$$\kappa(r) = \frac{3-\gamma}{2} \left(\frac{\theta'_E}{r \sqrt{1 e*\cos(2*\phi)}}\right)^{\gamma-1}$$

with ϵ is the ellipticity defined as

$$\epsilon = \frac{1-q^2}{1+q^2}$$

And an Einstein radius $\theta'_{\rm E}$ related to the definition used is

$$\left(\frac{\theta'_{\rm E}}{\theta_{\rm E}}\right)^{2} = \frac{2q}{1+q^2}.$$

`__init__`(*suppress_fastell=False*)

Parameters **`suppress_fastell`** – bool, if True, does not raise if fastell4py is not installed

`density_lens`(*r*, *theta_E*, *gamma*, *e1=None*, *e2=None*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **`r`** – radius within the mass is computed
- **`theta_E`** – Einstein radius
- **`gamma`** – power-law slope
- **`e1`** – eccentricity component (not used)
- **`e2`** – eccentricity component (not used)

Returns mass enclosed a 3D radius r

`derivatives`(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **`x`** – x-coordinate (angle)
- **`y`** – y-coordinate (angle)
- **`theta_E`** – Einstein radius (angle), pay attention to specific definition!
- **`gamma`** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **`e1`** – eccentricity component
- **`e2`** – eccentricity component
- **`center_x`** – x-position of lens center
- **`center_y`** – y-position of lens center

Returns deflection angles alpha_x, alpha_y

`function`(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **`x`** – x-coordinate (angle)
- **`y`** – y-coordinate (angle)

- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns lensing potential

hessian (*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns Hessian components f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

mass_3d_lens (*r*, *theta_E*, *gamma*, *e1=None*, *e2=None*)
computes the spherical power-law mass enclosed (with SPP routine) :param r: radius within the mass is computed :param theta_E: Einstein radius :param gamma: power-law slope :param e1: eccentricity component (not used) :param e2: eccentricity component (not used) :return: mass enclosed a 3D radius r

param_names = ['theta_E', 'gamma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 'ga

lenstronomy.LensModel.Profiles.point_mass module

class PointMass
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class to compute the physical deflection angle of a point mass, given as an Einstein radius

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

derivatives (*x*, *y*, *theta_E*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **theta_E** – Einstein radius (in angles)

Returns deflection angle (in angles)

function (*x, y, theta_E, center_x=0, center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **theta_E** – Einstein radius (in angles)

Returns lensing potential

hessian (*x, y, theta_E, center_x=0, center_y=0*)

Parameters

- **x** – x-coord (in angles)
- **y** – y-coord (in angles)
- **theta_E** – Einstein radius (in angles)

Returns hessian matrix (in angles)

lower_limit_default = {'center_x': -100, 'center_y': -100, 'theta_E': 0}

param_names = ['theta_E', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'theta_E': 100}

lenstronomy.LensModel.Profiles.sersic module

class Sersic (*smoothing=1e-05, sersic_major_axis=False*)

Bases: *lenstronomy.LensModel.Profiles.sersic_utils.SersicUtil*, *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions to evaluate a Sersic mass profile: https://arxiv.org/pdf/astro-ph/0311559.pdf

$$\kappa(R) = \kappa_{\rm eff} \exp\left[-b_n (R/R_{\rm Sersic})^{\frac{1}{n}}\right]$$

with $b_n \approx 1.999n - 0.327$

Example for converting physical mass units into convergence units used in the definition of this profile.

We first define an AstroPy cosmology instance and a LensCosmo class instance with a lens and source redshift.

```
>>> from lenstronomy.Cosmo.lens_cosmo import LensCosmo
>>> from astropy.cosmology import FlatLambdaCDM
>>> cosmo = FlatLambdaCDM(H0=70, Om0=0.3, Ob0=0.05)
>>> lens_cosmo = LensCosmo(z_lens=0.5, z_source=1.5, cosmo=cosmo)
```

We define the half-light radius R_sersic (arc seconds on the sky) and Sersic index n_sersic

```
>>> R_sersic = 2
>>> n_sersic = 4
```

Here we compute k_eff, the convergence at the half-light radius R_sersic for a stellar mass in Msun

```
>>> k_eff = lens_cosmo.sersic_m_star2k_eff(m_star=10**11.5, R_sersic=R_sersic, n_
↪sersic=n_sersic)
```

And here we perform the inverse calculation given k_eff to return the physical stellar mass.

```
>>> m_star = lens_cosmo.sersic_k_eff2m_star(k_eff=k_eff, R_sersic=R_sersic, n_
↪sersic=n_sersic)
```

The lens model calculation uses angular units as arguments! So to execute a deflection angle calculation one uses

```
>>> from lenstronomy.LensModel.Profiles.sersic import Sersic
>>> sersic = Sersic()
>>> alpha_x, alpha_y = sersic.derivatives(x=1, y=1, k_eff=k_eff, R_sersic=R_
↪sersic, center_x=0, center_y=0)
```

derivatives (*x, y, n_sersic, R_sersic, k_eff, center_x=0, center_y=0*)
returns df/dx and df/dy of the function

function (*x, y, n_sersic, R_sersic, k_eff, center_x=0, center_y=0*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **n_sersic** – Sersic index
- **R_sersic** – half light radius
- **k_eff** – convergence at half light radius
- **center_x** – x-center
- **center_y** – y-center

Returns

hessian (*x, y, n_sersic, R_sersic, k_eff, center_x=0, center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'R_sersic': 0, 'center_x': -100, 'center_y': -100, 'k_eff':

param_names = ['k_eff', 'R_sersic', 'n_sersic', 'center_x', 'center_y']

upper_limit_default = {'R_sersic': 100, 'center_x': 100, 'center_y': 100, 'k_eff':

lenstronomy.LensModel.Profiles.sersic_ellipse_kappa module

class SersicEllipseKappa
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains the function and the derivatives of an elliptical sersic profile with the ellipticity introduced in the convergence (not the potential).

This requires the use of numerical integrals (Keeton 2004)

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

derivatives (*x, y, n_sersic, R_sersic, k_eff, e1, e2, center_x=0, center_y=0*)
deflection angles

Parameters kwargs – keywords of the profile

Returns raise as definition is not defined

function (*x, y, n_sersic, R_sersic, k_eff, e1, e2, center_x=0, center_y=0*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters kwargs – keywords of the profile

Returns raise as definition is not defined

hessian (*x, y, n_sersic, R_sersic, k_eff, e1, e2, center_x=0, center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'R_sersic': 0, 'center_x': -100, 'center_y': -100, 'e1': -0

param_names = ['k_eff', 'R_sersic', 'n_sersic', 'e1', 'e2', 'center_x', 'center_y']

projected_mass (*x, y, q, n_sersic, R_sersic, k_eff, u=1, power=1*)

upper_limit_default = {'R_sersic': 100, 'center_x': 100, 'center_y': 100, 'e1': 0..

lenstronomy.LensModel.Profiles.sersic_ellipse_potential module

class SersicEllipse
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions to evaluate a Sersic mass profile: https://arxiv.org/pdf/astro-ph/0311559.pdf

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

derivatives (*x, y, n_sersic, R_sersic, k_eff, e1, e2, center_x=0, center_y=0*)
returns df/dx and df/dy of the function

function (*x, y, n_sersic, R_sersic, k_eff, e1, e2, center_x=0, center_y=0*)
returns Gaussian

hessian (*x, y, n_sersic, R_sersic, k_eff, e1, e2, center_x=0, center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'R_sersic': 0, 'center_x': -100, 'center_y': -100, 'e1': -0

param_names = ['k_eff', 'R_sersic', 'n_sersic', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'R_sersic': 100, 'center_x': 100, 'center_y': 100, 'e1': 0..

lenstronomy.LensModel.Profiles.sersic_utils module

class SersicUtil (*smoothing=1e-05, sersic_major_axis=False*)
Bases: object

__init__ (*smoothing=1e-05, sersic_major_axis=False*)

Parameters

- **smoothing** – smoothing scale of the innermost part of the profile (for numerical reasons)
- **sersic_major_axis** – boolean; if True, defines the half-light radius of the Sersic light profile along the semi-major axis (which is the Galfit convention) if False, uses the product average of semi-major and semi-minor axis as the convention (default definition for all light profiles in lenstronomy other than the Sersic profile)

alpha_abs (*x, y, n_sersic, r_eff, k_eff, center_x=0, center_y=0*)

Parameters

- **x** –
- **y** –
- **n_sersic** –
- **r_eff** –
- **k_eff** –
- **center_x** –
- **center_y** –

Returns

static b_n(*n*)

b(n) computation. This is the approximation of the exact solution to the relation,
2*incomplete_gamma_function(2n; b_n) = Gamma_function(2*n).

Parameters **n** – the sersic index

Returns b(n)

d_alpha_dr(*x, y, n_sersic, r_eff, k_eff, center_x=0, center_y=0*)

Parameters

- **x** –
- **y** –
- **n_sersic** –
- **r_eff** –
- **k_eff** –
- **center_x** –
- **center_y** –

Returns

density(*x, y, n_sersic, r_eff, k_eff, center_x=0, center_y=0*)
de-projection of the Sersic profile based on Prugniel & Simien (1997) :return:

get_distance_from_center(*x, y, e1, e2, center_x, center_y*)
Get the distance from the center of Sersic, accounting for orientation and axis ratio :param x: :param y: :param e1: eccentricity :param e2: eccentricity :param center_x: center x of sersic :param center_y: center y of sersic

k_Re(*n, k*)

k_bn(*n, Re*)
returns normalisation of the sersic profile such that Re is the half light radius given n_sersic slope

total_flux(*amp, R_sersic, n_sersic, e1=0, e2=0, **kwargs*)
computes analytical integral to compute total flux of the Sersic profile

Parameters

- **amp** – amplitude parameter in Sersic function (surface brightness at R_sersic
- **R_sersic** – half-light radius in semi-major axis
- **n_sersic** – Sersic index

- **e1** – eccentricity
- **e2** – eccentricity

Returns Analytic integral of the total flux of the Sersic profile

lenstronomy.LensModel.Profiles.shapelet_pot_cartesian module

class CartShapelets(**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains the function and the derivatives of the cartesian shapelets

H_n(*n*, *x*)

constructs the Hermite polynomial of order n at position x (dimensionless)

Parameters

- **n** – The n'the basis function.
- **x** – 1-dim position (dimensionless)

Returns array– H_n(x).

Raises AttributeError, KeyError

derivatives(*x*, *y*, *coeffs*, *beta*, *center_x=0*, *center_y=0*)

returns df/dx and df/dy of the function

function(*x*, *y*, *coeffs*, *beta*, *center_x=0*, *center_y=0*)

lensing potential (only needed for specific calculations, such as time delays)

Parameters kwargs – keywords of the profile

Returns raise as definition is not defined

hessian(*x*, *y*, *coeffs*, *beta*, *center_x=0*, *center_y=0*)

returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'beta': 0, 'center_x': -100, 'center_y': -100, 'coeffs': [0

param_names = ['coeffs', 'beta', 'center_x', 'center_y']

phi_n(*n*, *x*)

constructs the 1-dim basis function (formula (1) in Refregier et al. 2001)

Parameters

- **n** – The n'the basis function.
- **x** – 1-dim position (dimensionless)

Returns array– phi_n(x).

Raises AttributeError, KeyError

pre_calc(*x*, *y*, *beta*, *n_order*, *center_x*, *center_y*)

calculates the H_n(x) and H_n(y) for a given x-array and y-array :param x: :param y: :param amp: :param beta: :param n_order: :param center_x: :param center_y: :return: list of H_n(x) and H_n(y)

upper_limit_default = {'beta': 100, 'center_x': 100, 'center_y': 100, 'coeffs': [1

lenstronomy.LensModel.Profiles.shapelet_pot_polar module

class PolarShapelets

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains the function and the derivatives of the Singular Isothermal Sphere

__init__()

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

derivatives(*x, y, coeffs, beta, center_x=0, center_y=0*)

returns df/dx and df/dy of the function

function(*x, y, coeffs, beta, center_x=0, center_y=0*)

lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian(*x, y, coeffs, beta, center_x=0, center_y=0*)

returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'beta': 0, 'center_x': -100, 'center_y': -100, 'coeffs': [0

param_names = ['coeffs', 'beta', 'center_x', 'center_y']

upper_limit_default = {'beta': 100, 'center_x': 100, 'center_y': 100, 'coeffs': [1

lenstronomy.LensModel.Profiles.shear module

class Shear(**args, **kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class for external shear gamma1, gamma2 expression

derivatives(*x, y, gamma1, gamma2, ra_0=0, dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **gamma1** – shear component
- **gamma2** – shear component
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns deflection angles

function(*x, y, gamma1, gamma2, ra_0=0, dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **gamma1** – shear component
- **gamma2** – shear component

- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns lensing potential

hessian (*x*, *y*, *gamma1*, *gamma2*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **gamma1** – shear component
- **gamma2** – shear component
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns f_xx, f_xy, f_yx, f_yy

`lower_limit_default = {'dec_0': -100, 'gamma1': -0.5, 'gamma2': -0.5, 'ra_0': -100`

`param_names = ['gamma1', 'gamma2', 'ra_0', 'dec_0']`

`upper_limit_default = {'dec_0': 100, 'gamma1': 0.5, 'gamma2': 0.5, 'ra_0': 100}`

class ShearGammaPsi

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class to model a shear field with shear strength and direction. The translation ot the cartesian shear distortions is as follow:

$$\gamma_1 = \gamma_{ext} \cos(2\phi_{ext}) \gamma_2 = \gamma_{ext} \sin(2\phi_{ext})$$

__init__ ()

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

derivatives (*x*, *y*, *gamma_ext*, *psi_ext*, *ra_0=0*, *dec_0=0*)

deflection angles

Parameters kwargs – keywords of the profile

Returns raise as definition is not defined

static function (*x*, *y*, *gamma_ext*, *psi_ext*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **gamma_ext** – shear strength
- **psi_ext** – shear angle (radian)
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns

hessian (*x*, *y*, *gamma_ext*, *psi_ext*, *ra_0=0*, *dec_0=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

lower_limit_default = {'dec_0': -100, 'gamma_ext': 0, 'psi_ext': -3.141592653589793

param_names = ['gamma_ext', 'psi_ext', 'ra_0', 'dec_0']

upper_limit_default = {'dec_0': 100, 'gamma_ext': 1, 'psi_ext': 3.141592653589793,

class ShearReduced
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

reduced shear distortions $\gamma' = \gamma/(1-\kappa)$. This distortion keeps the magnification as unity and, thus, does not change the size of apparent objects. To keep the magnification at unity, it requires

$$(1-\kappa)^2) - \gamma_1^2 - \gamma_2^= 1$$

Thus, for given pair of reduced shear (γ'_1, γ'_2), an additional convergence term is calculated and added to the lensing distortions.

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

derivatives (*x*, *y*, *gamma1*, *gamma2*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **gamma1** – shear component
- **gamma2** – shear component
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns deflection angles

function (*x*, *y*, *gamma1*, *gamma2*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y0-coordinate (angle)
- **gamma1** – shear component
- **gamma2** – shear component
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns lensing potential

hessian (*x*, *y*, *gamma1*, *gamma2*, *ra_0=0*, *dec_0=0*)

Parameters

- **x** – x-coordinate (angle)

- **y** – y0-coordinate (angle)
- **gamma1** – shear component
- **gamma2** – shear component
- **ra_0** – x/ra position where shear deflection is 0
- **dec_0** – y/dec position where shear deflection is 0

Returns f_xx, f_xy, f_yx, f_yy

lower_limit_default = {'dec_0': -100, 'gamma1': -0.5, 'gamma2': -0.5, 'ra_0': -100

param_names = ['gamma1', 'gamma2', 'ra_0', 'dec_0']

upper_limit_default = {'dec_0': 100, 'gamma1': 0.5, 'gamma2': 0.5, 'ra_0': 100}

lenstronomy.LensModel.Profiles.sie module

class SIE(*NIE=True*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class for singular isothermal ellipsoid (SIS with ellipticity)

$$\kappa(x, y) = \frac{1}{2} \left(\frac{\theta_{E}}{\sqrt{q x^2 + y^2/q}}\right)$$

with θ_{E} is the (circularized) Einstein radius, q is the minor/major axis ratio, and x and y are defined in a coordinate sys- tem aligned with the major and minor axis of the lens.

In terms of eccentricities, this profile is defined as

$$\kappa(r) = \frac{1}{2} \left(\frac{\theta'_{E}}{r \sqrt{1 e*\cos(2*\phi)}}\right)$$

with ϵ is the ellipticity defined as

$$\epsilon = \frac{1-q^2}{1+q^2}$$

And an Einstein radius $\theta'_{\rm E}$ related to the definition used is

$$\left(\frac{\theta'_{\rm E}}{\theta_{\rm E}}\right)^{2} = \frac{2q}{1+q^2}.$$

__init__(*NIE=True*)

Parameters NIE – bool, if True, is using the NIE analytic model. Otherwise it uses PEMD with gamma=2 from fastell4py

static density(*r, rho0, e1=0, e2=0*)

computes the density

Parameters

- **r** – radius in angles
- **rho0** – density at angle=1

Returns density at r

static density_2d(*x*, *y*, *rho0*, *e1=0*, *e2=0*, *center_x=0*, *center_y=0*)

projected density

Parameters

- **x** –
- **y** –
- **rho0** –
- **e1** –
- **e2** –
- **center_x** –
- **center_y** –

Returns

density_lens(*r*, *theta_E*, *e1=0*, *e2=0*)

computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius in angles
- **theta_E** – Einstein radius
- **e1** – eccentricity component
- **e2** – eccentricity component

Returns density

derivatives(*x*, *y*, *theta_E*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate (angular coordinates)
- **y** – y-coordinate (angular coordinates)
- **theta_E** – Einstein radius
- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – centroid
- **center_y** – centroid

Returns

function(*x*, *y*, *theta_E*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate (angular coordinates)
- **y** – y-coordinate (angular coordinates)
- **theta_E** – Einstein radius
- **e1** – eccentricity
- **e2** – eccentricity

- **center_x** – centroid
- **center_y** – centroid

Returns

grav_pot (*x*, *y*, *rho0*, *e1=0*, *e2=0*, *center_x=0*, *center_y=0*)
gravitational potential (modulo 4 pi G and rho0 in appropriate units)

Parameters

- **x** –
- **y** –
- **rho0** –
- **e1** –
- **e2** –
- **center_x** –
- **center_y** –

Returns

hessian (*x*, *y*, *theta_E*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate (angular coordinates)
- **y** – y-coordinate (angular coordinates)
- **theta_E** – Einstein radius
- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – centroid
- **center_y** – centroid

Returns

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

mass_2d (*r*, *rho0*, *e1=0*, *e2=0*)
mass enclosed projected 2d sphere of radius r

Parameters

- **r** –
- **rho0** –
- **e1** –
- **e2** –

Returns

mass_2d_lens (*r*, *theta_E*, *e1=0*, *e2=0*)

Parameters

- **r** –
- **theta_E** –

- **e1** –
- **e2** –

Returns

static mass_3d(*r*, *rho0*, *e1=0*, *e2=0*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – radius in angular units
- **rho0** – density at angle=1

Returns mass in angular units

mass_3d_lens(*r*, *theta_E*, *e1=0*, *e2=0*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units

Parameters

- **r** – radius in angular units
- **theta_E** – Einstein radius

Returns mass in angular units

param_names = ['theta_E', 'e1', 'e2', 'center_x', 'center_y']

static theta2rho(*theta_E*)
converts projected density parameter (in units of deflection) into 3d density parameter

Parameters theta_E –

Returns

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 'th

lenstronomy.LensModel.Profiles.sis module

class SIS(**args*, ***kwargs*)
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains the function and the derivatives of the Singular Isothermal Sphere

$$\kappa(x, y) = \frac{1}{2}\left(\frac{\theta_E}{\sqrt{x^2 + y^2}}\right)$$

with θ_E is the Einstein radius,

static density(*r*, *rho0*)
computes the density :param r: radius in angles :param rho0: density at angle=1 :return: density at r

static density_2d(*x*, *y*, *rho0*, *center_x=0*, *center_y=0*)
projected density :param x: :param y: :param rho0: :param center_x: :param center_y: :return:

density_lens(*r*, *theta_E*)
computes the density at 3d radius r given lens model parameterization. The integral in projected in units of angles (i.e. arc seconds) results in the convergence quantity.

Parameters

- **r** – 3d radius

- **theta_E** – Einstein radius

Returns density(r)

derivatives (*x*, *y*, *theta_E*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function

function (*x*, *y*, *theta_E*, *center_x=0*, *center_y=0*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

grav_pot (*x*, *y*, *rho0*, *center_x=0*, *center_y=0*)
gravitational potential (modulo 4 pi G and rho0 in appropriate units) :param x: :param y: :param rho0: :param center_x: :param center_y: :return:

hessian (*x*, *y*, *theta_E*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'center_x': -100, 'center_y': -100, 'theta_E': 0}

static mass_2d (*r*, *rho0*)
mass enclosed projected 2d sphere of radius r :param r: :param rho0: :return:

mass_2d_lens (*r*, *theta_E*)

Parameters

- **r** – radius
- **theta_E** – Einstein radius

Returns mass within a radius in projection

static mass_3d (*r*, *rho0*)
mass enclosed a 3d sphere or radius r :param r: radius in angular units :param rho0: density at angle=1 :return: mass in angular units

mass_3d_lens (*r*, *theta_E*)
mass enclosed a 3d sphere or radius r given a lens parameterization with angular units

Parameters

- **r** – radius in angular units
- **theta_E** – Einstein radius

Returns mass in angular units

param_names = ['theta_E', 'center_x', 'center_y']

static rho2theta (*rho0*)
converts 3d density into 2d projected density parameter :param rho0: :return:

static theta2rho (*theta_E*)
converts projected density parameter (in units of deflection) into 3d density parameter :param theta_E: Einstein radius :return:

upper_limit_default = {'center_x': 100, 'center_y': 100, 'theta_E': 100}

lenstronomy.LensModel.Profiles.sis_truncate module

class SIS_truncate(**args*, ***kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

this class contains the function and the derivatives of the Singular Isothermal Sphere

derivatives(*x*, *y*, *theta_E*, *r_trunc*, *center_x=0*, *center_y=0*)

returns df/dx and df/dy of the function

function(*x*, *y*, *theta_E*, *r_trunc*, *center_x=0*, *center_y=0*)

lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian(*x*, *y*, *theta_E*, *r_trunc*, *center_x=0*, *center_y=0*)

returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'center_x': -100, 'center_y': -100, 'r_trunc': 0, 'theta_E':

param_names = ['theta_E', 'r_trunc', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'r_trunc': 100, 'theta_E':

lenstronomy.LensModel.Profiles.spemd module

class SPEMD(*suppress_fastell=False*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

class for smooth power law ellipse mass density profile (SPEMD). This class effectively performs the FASTELL calculations by Renan Barkana. The parameters are changed and represent a spherically averaged Einstein radius an a logarithmic 3D mass profile slope.

The SPEMD mass profile is defined as follow:

$$\kappa(x, y) = \frac{3-\gamma}{2} \left(\frac{\theta_E}{\sqrt{qx^2 + y^2/q + s^2}}\right)^{\gamma-1}$$

with θ_E is the (circularized) Einstein radius, γ is the negative power-law slope of the 3D mass distributions, q is the minor/major axis ratio, and x and y are defined in a coordinate system aligned with the major and minor axis of the lens.

the FASTELL definitions are as follows:

The parameters are position $(x1, x2)$, overall factor (b), power (gam), axis ratio (arat) which is <=1, core radius squared $(s2)$, and the output potential (ϕ). The projected mass density distribution, in units of the critical density, is

$$\kappa(x1, x2) = b_{fastell} \left[u2 + s2\right]^{-gam},$$

with $u2 = \left[x1^2 + x2^2/(arat^2)\right]$.

The conversion from lenstronomy definitions of this class to FASTELL are:

$$q_{fastell} \equiv q_{lenstronomy}$$

$$gam \equiv (\gamma - 1)/2$$

$$b_{fastell} \equiv (3-\gamma)/2. * \left(\theta_E^2/q\right)^{gam}$$

$$s2_{fastell} = s_{lenstronomy}^2 * q$$

__init__(*suppress_fastell=False*)

static convert_params(*theta_E*, *gamma*, *q*, *s_scale*)
converts parameter definitions into quantities used by the FASTELL fortran library

Parameters

- **theta_E** – Einstein radius
- **gamma** – 3D power-law slope of mass profile
- **q** – axis ratio minor/major
- **s_scale** – float, smoothing scale in the core

Returns pre-factors to SPEMP profile for FASTELL

derivatives(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale in the center of the profile
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns deflection angles alpha_x, alpha_y

function(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale in the center of the profile (angle)
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns lensing potential

hessian (*x, y, theta_E, gamma, e1, e2, s_scale, center_x=0, center_y=0*)

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale in the center of the profile
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns Hessian components f_xx, f_xy, f_yx, f_yy

static is_not_empty (*x1, x2*)
Check if float or not an empty array

Returns True if x1 and x2 are either floats/ints or an non-empty array, False if e.g. objects are []

Return type bool

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

param_names = ['theta_E', 'gamma', 'e1', 'e2', 's_scale', 'center_x', 'center_y']

param_transform (*x, y, theta_E, gamma, e1, e2, s_scale, center_x=0, center_y=0*)
transforms parameters in the format of fastell4py

Parameters

- **x** – x-coordinate (angle)
- **y** – y-coordinate (angle)
- **theta_E** – Einstein radius (angle), pay attention to specific definition!
- **gamma** – logarithmic slope of the power-law profile. gamma=2 corresponds to isothermal
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale in the center of the profile
- **center_x** – x-position of lens center
- **center_y** – y-position of lens center

Returns x-rotated, y-rotated, q_fastell, gam, s2, q, phi_G

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 'ga

lenstronomy.LensModel.Profiles.spep module

class SPEP
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class for Softened power-law elliptical potential (SPEP)

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

density_lens(*r*, *theta_E*, *gamma*, *e1=None*, *e2=None*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius within the mass is computed
- **theta_E** – Einstein radius
- **gamma** – power-law slope
- **e1** – eccentricity component (not used)
- **e2** – eccentricity component (not used)

Returns mass enclosed a 3D radius r

derivatives(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
deflection angles

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

function(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** (*array of size (n)*) – set of x-coordinates
- **theta_E** (*float.*) – Einstein radius of lense
- **gamma** (*<2 float*) – power law slope of mass profifle
- **e1** (*-1<e1<1*) – eccentricity
- **e2** (*-1<e1<1*) – eccentricity

Returns function

Raises AttributeError, KeyError

hessian(*x*, *y*, *theta_E*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

lower_limit_default = {'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e2': -0.5,

mass_3d_lens(*r*, *theta_E*, *gamma*, *e1=None*, *e2=None*)
computes the spherical power-law mass enclosed (with SPP routine)

Parameters

- **r** – radius within the mass is computed
- **theta_E** – Einstein radius
- **gamma** – power-law slope
- **e1** – eccentricity component (not used)
- **e2** – eccentricity component (not used)

Returns mass enclosed a 3D radius r

param_names = ['theta_E', 'gamma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e2': 0.5, 'ga

lenstronomy.LensModel.Profiles.splcore module

class SPLCORE(**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

This lens profile corresponds to a spherical power law (SPL) mass distribution with logarithmic slope gamma and a 3D core radius r_core

$$\rho\left(r, \rho_0, r_c, \gamma\right) = \rho_0 \frac{{r_c}^{\gamma}}{\left(r^2 + r_c^2\right)^{\frac{\gamma}{2}}}$$

The difference between this and EPL is that this model contains a core radius, is circular, and is also defined for gamma=3.

With respect to SPEMD, this model is different in that it is also defined for gamma = 3, is circular, and is defined in terms of a physical density parameter rho0, or the central density at r=0 divided by the critical density for lensing such that rho0 has units 1/arcsec.

This class is defined for all gamma > 1

alpha(*r*, *sigma0*, *r_core*, *gamma*)

Returns the deflection angle at r

Parameters

- **r** – radius [arcsec]
- **sigma0** – convergence at r=0
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns deflection angle at r

static density(*r*, *rho0*, *r_core*, *gamma*)

Returns the 3D density at r

Parameters

- **r** – radius [arcsec]
- **rho0** – convergence at r=0
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns density at r

density_2d(*x*, *y*, *rho0*, *r_core*, *gamma*)

Returns the convergence at radius r

Parameters

- **x** – x position [arcsec]
- **y** – y position [arcsec]
- **rho0** – convergence at r=0

- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns convergence at r

density_lens (*r*, *sigma0*, *r_core*, *gamma*)
Returns the 3D density at r

Parameters

- **r** – radius [arcsec]
- **sigma0** – convergence at r=0
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns density at r

derivatives (*x*, *y*, *sigma0*, *r_core*, *gamma*, *center_x=0*, *center_y=0*)

Parameters

- **x** – projected x position at which to evaluate function [arcsec]
- **y** – projected y position at which to evaluate function [arcsec]
- **sigma0** – convergence at r = 0
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity
- **center_x** – x coordinate center of lens model [arcsec]
- **center_y** – y coordinate center of lens model [arcsec]

Returns deflection angle alpha in x and y directions

function (*x*, *y*, *sigma0*, *r_core*, *gamma*, *center_x=0*, *center_y=0*)
lensing potential (only needed for specific calculations, such as time delays)

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

hessian (*x*, *y*, *sigma0*, *r_core*, *gamma*, *center_x=0*, *center_y=0*)

Parameters

- **x** – projected x position at which to evaluate function [arcsec]
- **y** – projected y position at which to evaluate function [arcsec]
- **sigma0** – convergence at r = 0
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity
- **center_x** – x coordinate center of lens model [arcsec]
- **center_y** – y coordinate center of lens model [arcsec]

Returns hessian elements

alpha_(x/y) = alpha_r * cos/sin(x/y / r)

lower_limit_default = {'center_x': -100, 'center_y': -100, 'gamma': 1.000001, 'r_co

mass_2d(*r*, *rho0*, *r_core*, *gamma*)
mass enclosed projected 2d disk of radius r

Parameters

- **r** – radius [arcsec]
- **rho0** – density at r = 0 in units [rho_0_physical / sigma_crit] (which should be equal to [1/arcsec]) where rho_0_physical is a physical density normalization and sigma_crit is the critical density for lensing
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns projected mass inside disk of radius r

mass_2d_lens(*r*, *sigma0*, *r_core*, *gamma*)
mass enclosed projected 2d disk of radius r

Parameters

- **r** – radius [arcsec]
- **sigma0** – convergence at r = 0 where rho_0_physical is a physical density normalization and sigma_crit is the critical density for lensing
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns projected mass inside disk of radius r

mass_3d(*r*, *rho0*, *r_core*, *gamma*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – radius [arcsec]
- **rho0** – density at r = 0 in units [rho_0_physical / sigma_crit] (which should be equal to [arcsec]) where rho_0_physical is a physical density normalization and sigma_crit is the critical density for lensing
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns mass inside radius r

mass_3d_lens(*r*, *sigma0*, *r_core*, *gamma*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – radius [arcsec]
- **sigma0** – convergence at r = 0
- **r_core** – core radius [arcsec]
- **gamma** – logarithmic slope at r -> infinity

Returns mass inside radius r

param_names = ['sigma0', 'center_x', 'center_y', 'r_core', 'gamma']

upper_limit_default = {'center_x': 100, 'center_y': 100, 'gamma': 5.0, 'r_core': 1

lenstronomy.LensModel.Profiles.spp module

class SPP(**args*, ***kwargs*)

Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

class for circular power-law mass distribution

static density(*r*, *rho0*, *gamma*)

computes the density

Parameters

- **r** –
- **rho0** –
- **gamma** –

Returns

static density_2d(*x*, *y*, *rho0*, *gamma*, *center_x=0*, *center_y=0*)

projected density

Parameters

- **x** –
- **y** –
- **rho0** –
- **gamma** –
- **center_x** –
- **center_y** –

Returns

density_lens(*r*, *theta_E*, *gamma*)

computes the density at 3d radius r given lens model parameterization. The integral in projected in units of angles (i.e. arc seconds) results in the convergence quantity.

derivatives(*x*, *y*, *theta_E*, *gamma*, *center_x=0.0*, *center_y=0.0*)

deflection angles

Parameters kwargs – keywords of the profile

Returns raise as definition is not defined

function(*x*, *y*, *theta_E*, *gamma*, *center_x=0*, *center_y=0*)

Parameters

- **x** (*array of size (n)*) – set of x-coordinates
- **y** (*array of size (n)*) – set of y-coordinates
- **theta_E** (*float.*) – Einstein radius of lens
- **gamma** (*<2 float*) – power law slope of mass profile

Returns function

Raises AttributeError, KeyError

grav_pot(*x*, *y*, *rho0*, *gamma*, *center_x=0*, *center_y=0*)

gravitational potential (modulo 4 pi G and rho0 in appropriate units)

Parameters

- **x** –
- **y** –
- **rho0** –
- **gamma** –
- **center_x** –
- **center_y** –

Returns

hessian (*x*, *y*, *theta_E*, *gamma*, *center_x=0.0*, *center_y=0.0*)
returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

Parameters **kwargs** – keywords of the profile

Returns raise as definition is not defined

lower_limit_default = {'center_x': -100, 'center_y': -100, 'gamma': 1.5, 'theta_E':

mass_2d (*r*, *rho0*, *gamma*)
mass enclosed projected 2d sphere of radius r

Parameters

- **r** –
- **rho0** –
- **gamma** –

Returns

mass_2d_lens (*r*, *theta_E*, *gamma*)

Parameters

- **r** – projected radius
- **theta_E** – Einstein radius
- **gamma** – power-law slope

Returns 2d projected radius enclosed

static mass_3d (*r*, *rho0*, *gamma*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** –
- **rho0** –
- **gamma** –

Returns

mass_3d_lens (*r*, *theta_E*, *gamma*)

Parameters

- **r** –
- **theta_E** –

- **gamma** –

Returns

param_names = ['theta_E', 'gamma', 'center_x', 'center_y']

static rho2theta (*rho0*, *gamma*)
converts 3d density into 2d projected density parameter

Parameters

- **rho0** –
- **gamma** –

Returns

static theta2rho (*theta_E*, *gamma*)
converts projected density parameter (in units of deflection) into 3d density parameter

Parameters

- **theta_E** –
- **gamma** –

Returns

upper_limit_default = {'center_x': 100, 'center_y': 100, 'gamma': 2.5, 'theta_E': 1

lenstronomy.LensModel.Profiles.tnfw module

class TNFW
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions concerning the truncated NFW profile with a truncation function (r_trunc^2)*(r^2+r_trunc^2)

density equation is:

$$\rho(r) = \frac{r_\text{trunc}^2}{r^2+r_\text{trunc}^2}\frac{\rho_0(\alpha_{R_s})}{r/R_s(1+r/R_s)^2}$$

relation are: R_200 = c * Rs

F (*x*)
Classic NFW function in terms of arctanh and arctan

Parameters x – r/Rs

Returns

__init__ ()

static alpha2rho0 (*alpha_Rs*, *Rs*)
convert angle at Rs into rho0; neglects the truncation

Parameters

- **alpha_Rs** – deflection angle at RS
- **Rs** – scale radius

Returns density normalization (characteristic density)

static density(*r*, *Rs*, *rho0*, *r_trunc*)
three dimensional truncated NFW profile

Parameters

- **r** (*float/numpy array*) – radius of interest
- **Rs** (*float > 0*) – scale radius
- **r_trunc** (*float > 0*) – truncation radius (angular units)

Returns rho(r) density

density_2d(*x*, *y*, *Rs*, *rho0*, *r_trunc*, *center_x=0*, *center_y=0*)
projected two dimensional NFW profile (kappa*Sigma_crit)

Parameters

- **R** (*float/numpy array*) – projected radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r_trunc** (*float > 0*) – truncation radius (angular units)

Returns Epsilon(R) projected density at radius R

derivatives(*x*, *y*, *Rs*, *alpha_Rs*, *r_trunc*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function (integral of TNFW), which are the deflection angles

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **r_trunc** – truncation radius (angular units)
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection angle in x, deflection angle in y

function(*x*, *y*, *Rs*, *alpha_Rs*, *r_trunc*, *center_x=0*, *center_y=0*)

Parameters

- **x** – angular position
- **y** – angular position
- **Rs** – angular turn over point
- **alpha_Rs** – deflection at Rs
- **r_trunc** – truncation radius
- **center_x** – center of halo
- **center_y** – center of halo

Returns lensing potential

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *r_trunc*, *center_x=0*, *center_y=0*)
returns d^2f/dx^2, d^2f/dxdy, d^2f/dydx, d^2f/dy^2 of the TNFW potential f

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **r_trunc** – truncation radius (angular units)
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns Hessian matrix of function d^2f/dx^2, d^f/dy^2, d^2/dxdy

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_2d (*R*, *Rs*, *rho0*, *r_trunc*)
analytic solution of the projection integral (convergence)

Parameters

- **R** – projected radius
- **Rs** – scale radius
- **rho0** – density normalization (characteristic density)
- **r_trunc** – truncation radius (angular units)

Returns mass enclosed 2d projected cylinder

mass_3d (*r*, *Rs*, *rho0*, *r_trunc*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – 3d radius
- **Rs** – scale radius
- **rho0** – density normalization (characteristic density)
- **r_trunc** – truncation radius (angular units)

Returns M(<r)

nfwAlpha (*R*, *Rs*, *rho0*, *r_trunc*, *ax_x*, *ax_y*)
deflection angel of NFW profile along the projection to coordinate axis

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r_trunc** (*float > 0*) – truncation radius (angular units)
- **axis** (*same as R*) – projection to either x- or y-axis

Returns

nfwGamma (*R, Rs, rho0, r_trunc, ax_x, ax_y*)
shear gamma of NFW profile (times Sigma_crit) along the projection to coordinate 'axis'

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r_trunc** (*float > 0*) – truncation radius (angular units)
- **axis** (*same as R*) – projection to either x- or y-axis

Returns

nfwPot (*R, Rs, rho0, r_trunc*)
lensing potential of truncated NFW profile

Parameters

- **R** (*float/numpy array*) – radius of interest
- **Rs** (*float*) – scale radius
- **rho0** (*float*) – density normalization (characteristic density)
- **r_trunc** (*float > 0*) – truncation radius (angular units)

Returns lensing potential

param_names = ['Rs', 'alpha_Rs', 'r_trunc', 'center_x', 'center_y']

profile_name = 'TNFW'

static rho02alpha (*rho0, Rs*)
convert rho0 to angle at Rs; neglects the truncation

Parameters

- **rho0** – density normalization (characteristic density)
- **Rs** – scale radius

Returns deflection angle at RS

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.tnfw_ellipse module

class TNFW_ELLIPSE
Bases: *lenstronomy.LensModel.Profiles.base_profile.LensProfileBase*

this class contains functions concerning the truncated NFW profile with an ellipticity defined in the potential parameterization of alpha_Rs, Rs and r_trunc is the same as for the spherical NFW profile

from Glose & Kneib: https://cds.cern.ch/record/529584/files/0112138.pdf

relation are: R_200 = c * Rs

__init__ ()

density_lens (*r, Rs, alpha_Rs, r_trunc, e1=1, e2=0*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – 3d radios
- **Rs** – turn-over radius of NFW profile
- **alpha_Rs** – deflection at Rs
- **r_trunc** – truncation radius
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction

Returns density rho(r)

derivatives (*x*, *y*, *Rs*, *alpha_Rs*, *r_trunc*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns df/dx and df/dy of the function, calculated as an elliptically distorted deflection angle of the spherical NFW profile

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **r_trunc** – truncation radius
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection in x-direction, deflection in y-direction

function (*x*, *y*, *Rs*, *alpha_Rs*, *r_trunc*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns elliptically distorted NFW lensing potential

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **r_trunc** – truncation radius
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns lensing potential

hessian (*x*, *y*, *Rs*, *alpha_Rs*, *r_trunc*, *e1*, *e2*, *center_x=0*, *center_y=0*)
returns Hessian matrix of function d^2f/dx^2, d^f/dy^2, d^2/dxdy the calculation is performed as a numerical differential from the deflection field. Analytical relations are possible

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **Rs** – turn over point in the slope of the NFW profile in angular unit
- **alpha_Rs** – deflection (angular units) at projected Rs
- **r_trunc** – truncation radius
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

lower_limit_default = {'Rs': 0, 'alpha_Rs': 0, 'center_x': -100, 'center_y': -100,

mass_3d_lens (*r, Rs, alpha_Rs, r_trunc, e1=1, e2=0*)

Parameters

- **r** – radius (in angular units)
- **Rs** – turn-over radius of NFW profile
- **alpha_Rs** – deflection at Rs
- **r_trunc** – truncation radius
- **e1** – eccentricity component in x-direction
- **e2** – eccentricity component in y-direction

Returns

param_names = ['Rs', 'alpha_Rs', 'r_trunc', 'e1', 'e2', 'center_x', 'center_y']

profile_name = 'TNFW_ELLIPSE'

upper_limit_default = {'Rs': 100, 'alpha_Rs': 10, 'center_x': 100, 'center_y': 100

lenstronomy.LensModel.Profiles.uldm module

class Uldm (**args, **kwargs*)

Bases: `lenstronomy.LensModel.Profiles.base_profile.LensProfileBase`

This class contains functions concerning the ULDM soliton density profile, whose good approximation is (see for example https://arxiv.org/pdf/1406.6586.pdf)

$$\rho = \rho_0 (1 + a(\theta/\theta_c)^2)^{-\beta}$$

where θ_c is the core radius, corresponding to the radius where the density drops by half its central value, :math: *beta* is the slope (called just slope in the parameters of this model), :math: *rho_0 = kappa_0 Sigma_c/D_lens*, and :math: *a* is a parameter, dependent on :math: *beta*, chosen such that :math: *theta_c* indeed corresponds to the radius where the density drops by half (simple math gives :math: *a = 0.5^{-1/beta} - 1*). For an ULDM soliton profile without contributions to background potential, it turns out that :math: *beta = 8, a = 0.091*. We allow :math: *beta* to be different from 8 to model solitons which feel the influence of background potential (see 2105.10873) The profile has, as parameters:

- kappa_0: central convergence
- theta_c: core radius (in arcseconds)
- slope: exponent entering the profile, default value is 8

static alpha_radial(*r*, *kappa_0*, *theta_c*, *slope=8*)
returns the radial part of the deflection angle

Parameters

- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile
- **r** – radius where the deflection angle is computed

Returns radial deflection angle

density(*R*, *kappa_0*, *theta_c*, *slope=8*)
three dimensional ULDM profile in angular units (rho0_physical = rho0_angular Sigma_crit / D_lens)

Parameters

- **R** – radius of interest
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns rho(R) density in angular units

density_2d(*x*, *y*, *kappa_0*, *theta_c*, *center_x=0*, *center_y=0*, *slope=8*)
projected two dimensional ULDM profile (convergence * Sigma_crit), but given our units convention for rho0, it is basically the convergence

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns Epsilon(R) projected density at radius R

density_lens(*r*, *kappa_0*, *theta_c*, *slope=8*)
computes the density at 3d radius r given lens model parameterization. The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – 3d radius
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns density rho(r)

derivatives (*x*, *y*, *kappa_0*, *theta_c*, *center_x=0*, *center_y=0*, *slope=8*)
returns df/dx and df/dy of the function (lensing potential), which are the deflection angles

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns deflection angle in x, deflection angle in y

function (*x*, *y*, *kappa_0*, *theta_c*, *center_x=0*, *center_y=0*, *slope=8*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns lensing potential (in arcsec^2)

hessian (*x*, *y*, *kappa_0*, *theta_c*, *center_x=0*, *center_y=0*, *slope=8*)

Parameters

- **x** – angular position (normally in units of arc seconds)
- **y** – angular position (normally in units of arc seconds)
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile
- **center_x** – center of halo (in angular units)
- **center_y** – center of halo (in angular units)

Returns Hessian matrix of function d^2f/dx^2, d^2/dxdy, d^2/dydx, d^f/dy^2

static kappa_r (*R*, *kappa_0*, *theta_c*, *slope=8*)
convergence of the cored density profile. This routine is also for testing

Parameters

- **R** – radius (angular scale)
- **kappa_0** – convergence in the core
- **theta_c** – core radius

- **slope** – exponent entering the profile

Returns convergence at r

lower_limit_default = {'center_x': -100, 'center_y': -100, 'kappa_0': 0, 'slope':

mass_2d (*R*, *kappa_0*, *theta_c*, *slope=8*)
mass enclosed a 2d sphere or radius r

Parameters

- **R** – radius over which the mass is computed
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns mass enclosed in 2d sphere

mass_3d (*R*, *kappa_0*, *theta_c*, *slope=8*)
mass enclosed a 3d sphere or radius r

Parameters

- **R** – radius in arcseconds
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns mass of soliton in angular units

mass_3d_lens (*r*, *kappa_0*, *theta_c*, *slope=8*)
mass enclosed a 3d sphere or radius r

Parameters

- **r** – radius over which the mass is computed
- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns mass enclosed in 3D ball

param_names = ['kappa_0', 'theta_c', 'slope', 'center_x', 'center_y']

static rhotilde (*kappa_0*, *theta_c*, *slope=8*)
Computes the central density in angular units

Parameters

- **kappa_0** – central convergence of profile
- **theta_c** – core radius (in arcsec)
- **slope** – exponent entering the profile

Returns central density in 1/arcsec

upper_limit_default = {'center_x': 100, 'center_y': 100, 'kappa_0': 1.0, 'slope':

Module contents

lenstronomy.LensModel.QuadOptimizer package

Submodules

lenstronomy.LensModel.QuadOptimizer.multi_plane_fast module

class MultiplaneFast(*x_image*, *y_image*, *z_lens*, *z_source*, *lens_model_list*, *redshift_list*, *astropy_instance*, *param_class*, *foreground_rays*, *tol_source=1e-05*, *numerical_alpha_class=None*)

Bases: `object`

This class accelerates ray tracing computations in multi plane lensing for quadruple image lenses by only computing the deflection from objects in front of the main deflector at z_lens one time. The first ray tracing computation through the foreground is saved and re-used, but it will always have the same shape as the initial x_image, y_image arrays.

__init__(*x_image*, *y_image*, *z_lens*, *z_source*, *lens_model_list*, *redshift_list*, *astropy_instance*, *param_class*, *foreground_rays*, *tol_source=1e-05*, *numerical_alpha_class=None*)

Parameters

- **x_image** – x_image to fit
- **y_image** – y_image to fit
- **z_lens** – lens redshift
- **z_source** – source redshift
- **lens_model_list** – list of lens models
- **redshift_list** – list of lens redshifts
- **astropy_instance** – instance of astropy to pass to lens model
- **param_class** – an instance of ParamClass (see documentation in QuadOptimizer.param_manager)
- **foreground_rays** – (optional) pre-computed foreground rays from a previous iteration, if they are not specified they will be re-computed
- **tol_source** – source plane chi^2 sigma
- **numerical_alpha_class** – class for computing numerically tabulated deflection angles

chi_square(*args_lens*, **args*, ***kwargs*)

Parameters args_lens – array of lens model parameters being optimized, computed from kwargs_lens in a specified param_class, see documentation in QuadOptimizer.param_manager

Returns total chi^2 penalty (source chi^2 + param chi^2), where param chi^2 is computed by the specified param_class

logL(*args_lens*, **args*, ***kwargs*)

Parameters args_lens – array of lens model parameters being optimized, computed from kwargs_lens in a specified param_class, see documentation in QuadOptimizer.param_manager

Returns the log likelihood corresponding to the given chi^2

ray_shooting_fast (*args_lens*)
Performs a ray tracing computation through observed coordinates on the sky (self._x_image, self._y_image) to the source plane, returning the final coordinates of each ray on the source plane

Parameters **args_lens** – An array of parameters being optimized. The array is computed from a set of key word arguments by an instance of ParamClass (see documentation in QuadOptimizer.param_manager)

Returns the xy coordinate of each ray traced back to the source plane

source_plane_chi_square (*args_lens*, **args*, ***kwargs*)

Parameters **args_lens** – array of lens model parameters being optimized, computed from kwargs_lens in a specified param_class, see documentation in QuadOptimizer.param_manager

Returns chi2 penalty for the source position (all images must map to the same source coordinate)

lenstronomy.LensModel.QuadOptimizer.optimizer module

class Optimizer (*x_image*, *y_image*, *lens_model_list*, *redshift_list*, *z_lens*, *z_source*, *parameter_class*, *astropy_instance=None*, *numerical_alpha_class=None*, *particle_swarm=True*, *re_optimize=False*, *re_optimize_scale=1.0*, *pso_convergence_mean=50000*, *foreground_rays=None*, *tol_source=1e-05*, *tol_simplex_func=0.001*, *simplex_n_iterations=400*)
Bases: `object`

class which executes the optimization routines. Currently implemented as a particle swarm optimization followed by a downhill simplex routine.

Particle swarm optimizer is modified from the CosmoHammer particle swarm routine with different convergence criteria implemented.

__init__ (*x_image*, *y_image*, *lens_model_list*, *redshift_list*, *z_lens*, *z_source*, *parameter_class*, *astropy_instance=None*, *numerical_alpha_class=None*, *particle_swarm=True*, *re_optimize=False*, *re_optimize_scale=1.0*, *pso_convergence_mean=50000*, *foreground_rays=None*, *tol_source=1e-05*, *tol_simplex_func=0.001*, *simplex_n_iterations=400*)

Parameters

- **x_image** – x_image to fit (should be length 4)
- **y_image** – y_image to fit (should be length 4)
- **lens_model_list** – list of lens models for the system
- **redshift_list** – list of lens redshifts for the system
- **z_lens** – the main deflector redshift, the lens models being optimizer must be at this redshift
- **z_source** – the source redshift
- **parameter_class** – an instance of ParamClass (see documentation in QuadOptimizer.param_manager)
- **astropy_instance** – an instance of astropy to pass to the lens model
- **numerical_alpha_class** – a class to compute numerical deflection angles to pass to the lens model

- **particle_swarm** – bool, whether or not to use a PSO fit first
- **re_optimize** – bool, if True the initial spread of particles will be very tight
- **re_optimize_scale** – float, controls how tight the initial spread of particles is
- **pso_convergence_mean** – when to terminate the PSO fit
- **foreground_rays** – (optional) can pass in pre-computed foreground light rays from a previous fit so as to not waste time recomputing them
- **tol_source** – sigma in the source plane chi^2
- **tol_simplex_func** – tolerance for the downhill simplex optimization
- **simplex_n_iterations** – number of iterations per dimension for the downhill simplex optimization

optimize(*n_particles=50*, *n_iterations=250*, *verbose=False*, *threadCount=1*)

Parameters

- **n_particles** – number of PSO particles, will be ignored if self._particle_swarm is False
- **n_iterations** – number of PSO iterations, will be ignored if self._particle_swarm is False
- **verbose** – whether to print stuff
- **threadCount** – integer; number of threads in multi-threading mode

Returns keyword arguments that map (x_image, y_image) to the same source coordinate (source_x, source_y)

lenstronomy.LensModel.QuadOptimizer.param_manager module

class PowerLawFixedShear(*kwargs_lens_init*, *shear_strength*)

Bases: *lenstronomy.LensModel.QuadOptimizer.param_manager.PowerLawParamManager*

This class implements a fit of EPL + external shear with every parameter except the power law slope AND the shear strength allowed to vary. The user should specify shear_strengh in the args_param_class keyword when creating the Optimizer class

__init__(*kwargs_lens_init*, *shear_strength*)

Parameters

- **kwargs_lens_init** – the initial kwargs_lens before optimizing
- **shear_strength** – the strenght of the external shear to be kept fixed

args_to_kwargs(*args*)

Parameters **args** – array of lens model parameters

Returns dictionary of lens model parameters with fixed shear = shear_strength

class PowerLawFixedShearMultipole(*kwargs_lens_init*, *shear_strength*)

Bases: *lenstronomy.LensModel.QuadOptimizer.param_manager.PowerLawFixedShear*

This class implements a fit of EPL + external shear + a multipole term with every parameter except the power law slope, shear strength, and multipole moment free to vary. The mass centroid and orientation of the multipole term are fixed to that of the EPL profile

args_to_kwargs (*args*)

Parameters **args** – array of lens model parameters

Returns dictionary of lens model parameters with fixed shear = shear_strength

to_vary_index

The number of lens models being varied in this routine. This is set to 3 because the first three lens models are EPL, SHEAR, and MULTIPOLE, and their parameters are being optimized.

The kwargs_list is split at to to_vary_index with indicies < to_vary_index accessed in this class, and lens models with indicies > to_vary_index kept fixed.

Note that this requires a specific ordering of lens_model_list :return:

class PowerLawFreeShear (*kwargs_lens_init*)

Bases: *lenstronomy.LensModel.QuadOptimizer.param_manager.PowerLawParamManager*

This class implements a fit of EPL + external shear with every parameter except the power law slope allowed to vary

args_to_kwargs (*args*)

Parameters **args** – array of lens model parameters

Returns dictionary of lens model parameters

class PowerLawFreeShearMultipole (*kwargs_lens_init*)

Bases: *lenstronomy.LensModel.QuadOptimizer.param_manager.PowerLawParamManager*

This class implements a fit of EPL + external shear + a multipole term with every parameter except the power law slope and multipole moment free to vary. The mass centroid and orientation of the multipole term are fixed to that of the EPL profile

args_to_kwargs (*args*)

to_vary_index

The number of lens models being varied in this routine. This is set to 3 because the first three lens models are EPL, SHEAR, and MULTIPOLE, and their parameters are being optimized.

The kwargs_list is split at to to_vary_index with indicies < to_vary_index accessed in this class, and lens models with indicies > to_vary_index kept fixed.

Note that this requires a specific ordering of lens_model_list :return:

class PowerLawParamManager (*kwargs_lens_init*)

Bases: `object`

Base class for handling the translation between key word arguments and parameter arrays for EPL mass models. This class is intended for use in modeling galaxy-scale lenses

__init__ (*kwargs_lens_init*)

Parameters **kwargs_lens_init** – the initial kwargs_lens before optimizing

bounds (*re_optimize*, *scale=1.0*)

Sets the low/high parameter bounds for the particle swarm optimization

NOTE: The low/high values specified here are intended for galaxy-scale lenses. If you want to use this for a different size system you should create a new ParamClass with different settings

Parameters

- **re_optimize** – keep a narrow window around each parameter

- **scale** – scales the size of the uncertainty window

Returns

static kwargs_to_args (*kwargs*)

Parameters kwargs – keyword arguments corresponding to the lens model parameters being optimized

Returns array of lens model parameters

param_chi_square_penalty (*args*)

to_vary_index

The number of lens models being varied in this routine. This is set to 2 because the first three lens models are EPL and SHEAR, and their parameters are being optimized.

The kwargs_list is split at to to_vary_index with indicies < to_vary_index accessed in this class, and lens models with indicies > to_vary_index kept fixed.

Note that this requires a specific ordering of lens_model_list :return:

Module contents

lenstronomy.LensModel.Solver package

Submodules

lenstronomy.LensModel.Solver.lens_equation_solver module

class LensEquationSolver (*lensModel*)

Bases: object

class to solve for image positions given lens model and source position

__init__ (*lensModel*)

This class must contain the following definitions (with same syntax as the standard LensModel() class: def ray_shooting() def hessian() def magnification()

Parameters lensModel – instance of a class according to lenstronomy.LensModel.lens_model

candidate_solutions (*sourcePos_x, sourcePos_y, kwargs_lens, min_distance=0.1, search_window=10, verbose=False, x_center=0, y_center=0*)

finds pixels in the image plane possibly hosting a solution of the lens equation, for the given source position and lens model

Parameters

- **sourcePos_x** – source position in units of angle
- **sourcePos_y** – source position in units of angle
- **kwargs_lens** – lens model parameters as keyword arguments
- **min_distance** – minimum separation to consider for two images in units of angle
- **search_window** – window size to be considered by the solver. Will not find image position outside this window
- **verbose** – bool, if True, prints some useful information for the user

- **x_center** – float, center of the window to search for point sources
- **y_center** – float, center of the window to search for point sources

Returns (approximate) angular position of (multiple) images ra_pos, dec_pos in units of angles, related ray-traced source displacements and pixel width

Raises AttributeError, KeyError

findBrightImage(*sourcePos_x*, *sourcePos_y*, *kwargs_lens*, *numImages=4*, *min_distance=0.01*, *search_window=5*, *precision_limit=1e-10*, *num_iter_max=10*, *arrival_time_sort=True*, *x_center=0*, *y_center=0*, *num_random=0*, *non_linear=False*, *magnification_limit=None*, *initial_guess_cut=True*, *verbose=False*)

Parameters

- **sourcePos_x** – source position in units of angle
- **sourcePos_y** – source position in units of angle
- **kwargs_lens** – lens model parameters as keyword arguments
- **min_distance** – minimum separation to consider for two images in units of angle
- **search_window** – window size to be considered by the solver. Will not find image position outside this window
- **precision_limit** – required precision in the lens equation solver (in units of angle in the source plane).
- **num_iter_max** – maximum iteration of lens-source mapping conducted by solver to match the required precision
- **arrival_time_sort** – bool, if True, sorts image position in arrival time (first arrival photon first listed)
- **initial_guess_cut** – bool, if True, cuts initial local minima selected by the grid search based on distance criteria from the source position
- **verbose** – bool, if True, prints some useful information for the user
- **x_center** – float, center of the window to search for point sources
- **y_center** – float, center of the window to search for point sources
- **num_random** – int, number of random positions within the search window to be added to be starting positions for the gradient decent solver
- **non_linear** – bool, if True applies a non-linear solver not dependent on Hessian computation
- **magnification_limit** – None or float, if set will only return image positions that have an abs(magnification) larger than this number

Returns (exact) angular position of (multiple) images ra_pos, dec_pos in units of angle

image_position_analytical(*x*, *y*, *kwargs_lens*, *arrival_time_sort=True*, *magnification_limit=None*, ***kwargs_solver*)

Solves the lens equation. Only supports EPL-like (plus shear) models. Uses a specialized recipe that solves a one-dimensional lens equation that is easier and more reliable to solve than the usual two-dimensional lens equation.

Parameters

- **x** – source position in units of angle, an array of positions is also supported.
- **y** – source position in units of angle, an array of positions is also supported.
- **kwargs_lens** – lens model parameters as keyword arguments
- **arrival_time_sort** – bool, if True, sorts image position in arrival time (first arrival photon first listed)
- **magnification_limit** – None or float, if set will only return image positions that have an abs(magnification) larger than this number
- **kwargs_solver** – additional kwargs to be supplied to the solver. Particularly relevant are Nmeas and Nmeas_extra

Returns (exact) angular position of (multiple) images ra_pos, dec_pos in units of angle Note: in contrast to the other solvers, generally the (heavily demagnified) central image will also be included, so setting a a proper magnification_limit is more important. To get similar behaviour, a limit of 1e-1 is acceptable

image_position_from_source(*sourcePos_x*, *sourcePos_y*, *kwargs_lens*, *solver='lenstronomy'*, ***kwargs*)

Solves the lens equation, i.e. finds the image positions in the lens plane that are mapped to a given source position.

Parameters

- **sourcePos_x** – source position in units of angle
- **sourcePos_y** – source position in units of angle
- **kwargs_lens** – lens model parameters as keyword arguments
- **solver** – which solver to use, can be 'lenstronomy' (default), 'analytical' or 'stochastic'.
- **kwargs** – Any additional kwargs are passed to the chosen solver, see the documentation of image_position_lenstronomy, image_position_analytical and image_position_stochastic

Returns (exact) angular position of (multiple) images ra_pos, dec_pos in units of angle

image_position_lenstronomy(*sourcePos_x*, *sourcePos_y*, *kwargs_lens*, *min_distance=0.1*, *search_window=10*, *precision_limit=1e-10*, *num_iter_max=100*, *arrival_time_sort=True*, *initial_guess_cut=True*, *verbose=False*, *x_center=0*, *y_center=0*, *num_random=0*, *non_linear=False*, *magnification_limit=None*)

Finds image position given source position and lens model. The solver first samples does a grid search in the lens plane, and the grid points that are closest to the supplied source position are fed to a specialized gradient-based root finder that finds the exact solutions. Works with all lens models.

Parameters

- **sourcePos_x** – source position in units of angle
- **sourcePos_y** – source position in units of angle
- **kwargs_lens** – lens model parameters as keyword arguments
- **min_distance** – minimum separation to consider for two images in units of angle
- **search_window** – window size to be considered by the solver. Will not find image position outside this window
- **precision_limit** – required precision in the lens equation solver (in units of angle in the source plane).

- **num_iter_max** – maximum iteration of lens-source mapping conducted by solver to match the required precision
- **arrival_time_sort** – bool, if True, sorts image position in arrival time (first arrival photon first listed)
- **initial_guess_cut** – bool, if True, cuts initial local minima selected by the grid search based on distance criteria from the source position
- **verbose** – bool, if True, prints some useful information for the user
- **x_center** – float, center of the window to search for point sources
- **y_center** – float, center of the window to search for point sources
- **num_random** – int, number of random positions within the search window to be added to be starting positions for the gradient decent solver
- **non_linear** – bool, if True applies a non-linear solver not dependent on Hessian computation
- **magnification_limit** – None or float, if set will only return image positions that have an abs(magnification) larger than this number

Returns (exact) angular position of (multiple) images ra_pos, dec_pos in units of angle

Raises AttributeError, KeyError

image_position_stochastic(*source_x*, *source_y*, *kwargs_lens*, *search_window=10*, *precision_limit=1e-10*, *arrival_time_sort=True*, *x_center=0*, *y_center=0*, *num_random=1000*)

Solves the lens equation stochastic with the scipy minimization routine on the quadratic distance between the backwards ray-shooted proposed image position and the source position. Credits to Giulia Pagano

Parameters

- **source_x** – source position
- **source_y** – source position
- **kwargs_lens** – lens model list of keyword arguments
- **search_window** – angular size of search window
- **precision_limit** – limit required on the precision in the source plane
- **arrival_time_sort** – bool, if True sorts according to arrival time
- **x_center** – center of search window
- **y_center** – center of search window
- **num_random** – number of random starting points of the non-linear solver in the search window

Returns x_image, y_image

sort_arrival_times(*x_mins*, *y_mins*, *kwargs_lens*)

sort arrival times (fermat potential) of image positions in increasing order of light travel time

Parameters

- **x_mins** – ra position of images
- **y_mins** – dec position of images
- **kwargs_lens** – keyword arguments of lens model

Returns sorted lists of x_mins and y_mins

lenstronomy.LensModel.Solver.solver module

class Solver(*solver_type*, *lensModel*, *num_images*)

Bases: object

joint solve class to manage with type of solver to be executed and checks whether the requirements are fulfilled.

__init__(*solver_type*, *lensModel*, *num_images*)

Parameters

- **solver_type** – string, option for specific solver type see detailed instruction of the Solver4Point and Solver2Point classes
- **lensModel** – instance of a LensModel() class
- **num_images** – int, number of images to be solved for

add_fixed_lens(*kwargs_fixed_lens*, *kwargs_lens_init*)

returns kwargs that are kept fixed during run, depending on options

Parameters

- **kwargs_fixed_lens** – keyword argument list of fixed parameters (indicated by fitting argument of the user)
- **kwargs_lens_init** – Initial values of the full lens model keyword arguments

Returns updated kwargs_fixed_lens, added fixed parameters being added (and replaced later on) by the non-linear solver.

check_solver(*image_x*, *image_y*, *kwargs_lens*)

returns the precision of the solver to match the image position

Parameters

- **kwargs_lens** – full lens model (including solved parameters)
- **image_x** – point source in image
- **image_y** – point source in image

Returns precision of Euclidean distances between the different rays arriving at the image positions

constraint_lensmodel(*x_pos*, *y_pos*, *kwargs_list*, *xtol=1.49012e-12*)

Parameters

- **x_pos** –
- **y_pos** –
- **kwargs_list** –
- **xtol** –

Returns

update_solver(*kwargs_lens*, *x_pos*, *y_pos*)

Parameters

- **kwargs_lens** –
- **x_pos** –
- **y_pos** –

Returns

lenstronomy.LensModel.Solver.solver2point module

class Solver2Point(*lensModel*, *solver_type='CENTER'*, *decoupling=True*)

Bases: object

class to solve a constraint lens model with two point source positions

options are: 'CENTER': solves for 'center_x', 'center_y' parameters of the first lens model 'ELLIPSE': solves for 'e1', 'e2' of the first lens (can also be shear) 'SHAPELETS': solves for shapelet coefficients c01, c10 'THETA_E_PHI: solves for Einstein radius of first lens model and shear angle of second model

__init__(*lensModel*, *solver_type='CENTER'*, *decoupling=True*)

Parameters

- **lensModel** – instance of LensModel class
- **solver_type** – string
- **decoupling** – bool

add_fixed_lens(*kwargs_fixed_lens_list*, *kwargs_lens_init*)

Parameters

- **kwargs_fixed_lens_list** –
- **kwargs_lens_init** –

Returns

constraint_lensmodel(*x_pos*, *y_pos*, *kwargs_list*, *xtol=1.49012e-12*)

constrains lens model parameters by demanding the solution to match the image positions to a single source position

Parameters

- **x_pos** – list of image positions (x-axis)
- **y_pos** – list of image position (y-axis)
- **kwargs_list** – list of lens model kwargs
- **xtol** – tolerance level of solution when to stop the non-linear solver

Returns updated lens model that satisfies the lens equation for the point sources

solve(*x_pos*, *y_pos*, *init*, *kwargs_list*, *a*, *xtol=1.49012e-12*)

lenstronomy.LensModel.Solver.solver4point module

class Solver4Point(*lensModel*, *solver_type='PROFILE'*)

Bases: object

class to make the constraints for the solver

`__init__`(*lensModel*, *solver_type='PROFILE'*)
Initialize self. See help(type(self)) for accurate signature.

`add_fixed_lens`(*kwargs_fixed_lens_list*, *kwargs_lens_init*)

Parameters

- **kwargs_fixed_lens_list** –
- **kwargs_lens_init** –

Returns

`constraint_lensmodel`(*x_pos*, *y_pos*, *kwargs_list*, *xtol=1.49012e-12*)

Parameters

- **x_pos** – list of image positions (x-axis)
- **y_pos** – list of image position (y-axis)
- **xtol** – numerical tolerance level
- **kwargs_list** – list of lens model kwargs

Returns updated lens model that satisfies the lens equation for the point sources

`solve`(*x_pos*, *y_pos*, *init*, *kwargs_list*, *a*, *xtol=1.49012e-10*)

Module contents

lenstronomy.LensModel.Util package

Submodules

lenstronomy.LensModel.Util.epl_util module

`brentq_inline`
A numba-compatible implementation of brentq (largely copied from scipy.optimize.brentq). Unfortunately, the scipy verison is not compatible with numba, hence this reimplementation :(:param f: function to optimize :param xa: left bound :param xb: right bound :param xtol: x-coord root tolerance :param rtol: x-coord relative tolerance :param maxiter: maximum num of iterations :param args: additional arguments to pass to function in the form f(x, args) :return:

`brentq_nojit`(*f*, *xa*, *xb*, *xtol=2e-14*, *rtol=3.552713678800501e-15*, *maxiter=100*, *args=()*)
A numba-compatible implementation of brentq (largely copied from scipy.optimize.brentq). Unfortunately, the scipy verison is not compatible with numba, hence this reimplementation :(:param f: function to optimize :param xa: left bound :param xb: right bound :param xtol: x-coord root tolerance :param rtol: x-coord relative tolerance :param maxiter: maximum num of iterations :param args: additional arguments to pass to function in the form f(x, args) :return:

`cart_to_pol`
Convert from cartesian to polar :param x: x-coordinate :param y: y-coordinate :return: tuple of (r, theta)

`cdot`
Calculates some complex dot-product that simplifies the math :param a: complex number :param b: complex number :return: dot-product

`ell_to_pol`
Converts from elliptical to polar coordinates

geomlinspace (*a*, *b*, *N*)
Constructs a geomspace from a to b, with a linspace prepended to it from 0 to a, with the same spacing as the geomspace would have at a

min_approx
Get the x-value of the minimum of the parabola through the points (x1,y1), ... :param x1: x-coordinate point 1 :param x2: x-coordinate point 2 :param x3: x-coordinate point 3 :param y1: y-coordinate point 1 :param y2: y-coordinate point 2 :param y3: y-coordinate point 3 :return: x-location of the minimum

pol_to_cart
Convert from polar to cartesian :param r: r-coordinate :param th: theta-coordinate :return: tuple of (x,y)

pol_to_ell
Converts from polar to elliptical coordinates

ps
A regularized power-law that gets rid of singularities, abs(x)**p*sign(x) :param x: x :param p: p :return:

rotmat
Calculates the rotation matrix :param th: angle :return: rotation matrix

solvequadeq
Solves a quadratic equation. Care is taken for the numerics, see also https://en.wikipedia.org/wiki/Loss_of_significance :param a: a :param b: b :param c: c :return: tuple of two solutions

Module contents

Submodules

lenstronomy.LensModel.convergence_integrals module

potential_from_kappa_grid (*kappa*, *grid_spacing*)
Lensing potential $\psi(\vec{\theta})$ on the convergence grid κ.

$$\psi(\vec{\theta}) = \frac{1}{\pi}\int d^2\theta' \kappa(\vec{\theta'}) \ln|\vec{\theta} - \vec{\theta'}|$$

The computation is performed as a convolution of the Green's function with the convergence map using FFT.

Parameters

- **kappa** – 2d grid of convergence values
- **grid_spacing** – scale of an individual pixel (per axis) of grid

Returns lensing potential in a 2d grid at positions x_grid, y_grid

potential_from_kappa_grid_adaptive (*kappa_high_res*, *grid_spacing*, *low_res_factor*, *high_res_kernel_size*)
lensing potential on the convergence grid the computation is performed as a convolution of the Green's function with the convergence map using FFT

Parameters

- **kappa_high_res** – 2d grid of convergence values
- **grid_spacing** – scale of an individual pixel (per axis) of grid
- **low_res_factor** – lower resolution factor of larger scale kernel.
- **high_res_kernel_size** – int, size of high resolution kernel in units of degraded pixels

Returns lensing potential in a 2d grid at positions x_grid, y_grid

deflection_from_kappa_grid(*kappa*, *grid_spacing*)

Deflection angle $\vec{\alpha}$ from a convergence grid κ.

$$\vec{\alpha}(\vec{\theta}) = \frac{1}{\pi}\int d^2\theta' \frac{(\vec{\theta}-\vec{\theta'})\kappa(\vec{\theta'})}{|\vec{\theta}-\vec{\theta'}|^2}$$

The computation is performed as a convolution of the Green's function with the convergence map using FFT.

Parameters

- **kappa** – convergence values for each pixel (2-d array)
- **grid_spacing** – scale of an individual pixel (per axis) of grid

Returns numerical deflection angles in x- and y- direction over the convergence grid points

deflection_from_kappa_grid_adaptive(*kappa_high_res*, *grid_spacing*, *low_res_factor*, *high_res_kernel_size*)

deflection angles on the convergence grid with adaptive FFT the computation is performed as a convolution of the Green's function with the convergence map using FFT The grid is returned in the lower resolution grid

Parameters

- **kappa_high_res** – convergence values for each pixel (2-d array)
- **grid_spacing** – pixel size of high resolution grid
- **low_res_factor** – lower resolution factor of larger scale kernel.
- **high_res_kernel_size** – int, size of high resolution kernel in units of degraded pixels

Returns numerical deflection angles in x- and y- direction

potential_kernel(*num_pix*, *delta_pix*)

numerical gridded integration kernel for convergence to lensing kernel with given pixel size

Parameters

- **num_pix** – integer; number of pixels of kernel per axis
- **delta_pix** – pixel size (per dimension in units of angle)

Returns kernel for lensing potential

deflection_kernel(*num_pix*, *delta_pix*)

numerical gridded integration kernel for convergence to deflection angle with given pixel size

Parameters

- **num_pix** – integer; number of pixels of kernel per axis, should be odd number to have a defined center
- **delta_pix** – pixel size (per dimension in units of angle)

Returns kernel for x-direction and kernel of y-direction deflection angles

lenstronomy.LensModel.lens_model module

class LensModel(*lens_model_list*, *z_lens=None*, *z_source=None*, *lens_redshift_list=None*, *cosmo=None*, *multi_plane=False*, *numerical_alpha_class=None*, *observed_convention_index=None*, *z_source_convention=None*, *cosmo_interp=False*, *z_interp_stop=None*, *num_z_interp=100*, *kwargs_interp=None*)

Bases: `object`

class to handle an arbitrary list of lens models. This is the main lenstronomy LensModel API for all other modules.

__init__(*lens_model_list*, *z_lens=None*, *z_source=None*, *lens_redshift_list=None*, *cosmo=None*, *multi_plane=False*, *numerical_alpha_class=None*, *observed_convention_index=None*, *z_source_convention=None*, *cosmo_interp=False*, *z_interp_stop=None*, *num_z_interp=100*, *kwargs_interp=None*)

Parameters

- **lens_model_list** – list of strings with lens model names
- **z_lens** – redshift of the deflector (only considered when operating in single plane mode). Is only needed for specific functions that require a cosmology.
- **z_source** – redshift of the source: Needed in multi_plane option only, not required for the core functionalities in the single plane mode.
- **lens_redshift_list** – list of deflector redshift (corresponding to the lens model list), only applicable in multi_plane mode.
- **cosmo** – instance of the astropy cosmology class. If not specified, uses the default cosmology.
- **multi_plane** – bool, if True, uses multi-plane mode. Default is False.
- **numerical_alpha_class** – an instance of a custom class for use in NumericalAlpha() lens model (see documentation in Profiles/numerical_alpha)
- **kwargs_interp** – interpolation keyword arguments specifying the numerics. See description in the Interpolate() class. Only applicable for 'INTERPOL' and 'INTERPOL_SCALED' models.
- **observed_convention_index** – a list of indices, corresponding to the lens_model_list element with same index, where the 'center_x' and 'center_y' kwargs correspond to observed (lensed) positions, not physical positions. The code will compute the physical locations when performing computations
- **z_source_convention** – float, redshift of a source to define the reduced deflection angles of the lens models. If None, 'z_source' is used.
- **cosmo_interp** – boolean (only employed in multi-plane mode), interpolates astropy.cosmology distances for faster calls when accessing several lensing planes
- **z_interp_stop** – (only in multi-plane with cosmo_interp=True); maximum redshift for distance interpolation This number should be higher or equal the maximum of the source redshift and/or the z_source_convention
- **num_z_interp** – (only in multi-plane with cosmo_interp=True); number of redshift bins for interpolating distances

alpha(*x*, *y*, *kwargs*, *k=None*, *diff=None*)
deflection angles

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model

- **diff** – None or float. If set, computes the deflection as a finite numerical differential of the lensing potential. This differential is only applicable in the single lensing plane where the form of the lensing potential is analytically known

Returns deflection angles in units of arcsec

arrival_time (*x_image*, *y_image*, *kwargs_lens*, *kappa_ext=0*, *x_source=None*, *y_source=None*)
Arrival time of images relative to a straight line without lensing. Negative values correspond to images arriving earlier, and positive signs correspond to images arriving later.

Parameters

- **x_image** – image position
- **y_image** – image position
- **kwargs_lens** – lens model parameter keyword argument list
- **kappa_ext** – external convergence contribution not accounted in the lens model that leads to the same observables in position and relative fluxes but rescales the time delays
- **x_source** – source position (optional), otherwise computed with ray-tracing
- **y_source** – source position (optional), otherwise computed with ray-tracing

Returns arrival time of image positions in units of days

curl (*x*, *y*, *kwargs*, *k=None*, *diff=None*, *diff_method='square'*)
curl computation F_xy - F_yx

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model
- **diff** – float, scale over which the finite numerical differential is computed. If None, then using the exact (if available) differentials.
- **diff_method** – string, 'square' or 'cross', indicating whether finite differentials are computed from a cross or a square of points around (x, y)

Returns curl at position (x, y)

fermat_potential (*x_image*, *y_image*, *kwargs_lens*, *x_source=None*, *y_source=None*)
Fermat potential (negative sign means earlier arrival time) for Multi-plane lensing, it computes the effective Fermat potential (derived from the arrival time and subtracted off the time-delay distance for the given cosmology). The units are given in arcsecond square.

Parameters

- **x_image** – image position
- **y_image** – image position
- **x_source** – source position
- **y_source** – source position
- **kwargs_lens** – list of keyword arguments of lens model parameters matching the lens model classes

Returns fermat potential in arcsec**2 without geometry term (second part of Eqn 1 in Suyu et al. 2013) as a list

flexion (*x*, *y*, *kwargs*, *k=None*, *diff=1e-06*, *hessian_diff=True*)
third derivatives (flexion)

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – int or None, if set, only evaluates the differential from one model component
- **diff** – numerical differential length of Flexion
- **hessian_diff** – boolean, if true also computes the numerical differential length of Hessian (optional)

Returns f_xxx, f_xxy, f_xyy, f_yyy

gamma (*x*, *y*, *kwargs*, *k=None*, *diff=None*, *diff_method='square'*)
shear computation g1 = 1/2(d^2phi/dx^2 - d^2phi/dy^2) g2 = d^2phi/dxdy

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model
- **diff** – float, scale over which the finite numerical differential is computed. If None, then using the exact (if available) differentials.
- **diff_method** – string, 'square' or 'cross', indicating whether finite differentials are computed from a cross or a square of points around (x, y)

Returns gamma1, gamma2

hessian (*x*, *y*, *kwargs*, *k=None*, *diff=None*, *diff_method='square'*)
hessian matrix

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model
- **diff** – float, scale over which the finite numerical differential is computed. If None, then using the exact (if available) differentials.
- **diff_method** – string, 'square' or 'cross', indicating whether finite differentials are computed from a cross or a square of points around (x, y)

Returns f_xx, f_xy, f_yx, f_yy components

kappa (*x*, *y*, *kwargs*, *k=None*, *diff=None*, *diff_method='square'*)
lensing convergence k = 1/2 laplacian(phi)

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model
- **diff** – float, scale over which the finite numerical differential is computed. If None, then using the exact (if available) differentials.
- **diff_method** – string, 'square' or 'cross', indicating whether finite differentials are computed from a cross or a square of points around (x, y)

Returns lensing convergence

magnification (*x*, *y*, *kwargs*, *k=None*, *diff=None*, *diff_method='square'*)
mag = 1/det(A) A = 1 - d^2phi/d_ij

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model
- **diff** – float, scale over which the finite numerical differential is computed. If None, then using the exact (if available) differentials.
- **diff_method** – string, 'square' or 'cross', indicating whether finite differentials are computed from a cross or a square of points around (x, y)

Returns magnification

potential (*x*, *y*, *kwargs*, *k=None*)
lensing potential

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model

Returns lensing potential in units of arcsec^2

ray_shooting (*x*, *y*, *kwargs*, *k=None*)
maps image to source position (inverse deflection)

Parameters

- **x** (*numpy array*) – x-position (preferentially arcsec)
- **y** (*numpy array*) – y-position (preferentially arcsec)

- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** – only evaluate the k-th lens model

Returns source plane positions corresponding to (x, y) in the image plane

set_dynamic()
deletes cache for static setting and makes sure the observed convention in the position of lensing profiles in the multi-plane setting is enabled. Dynamic is the default setting of this class enabling an accurate computation of lensing quantities with different parameters in the lensing profiles.

Returns None

set_static(*kwargs*)
set this instance to a static lens model. This can improve the speed in evaluating lensing quantities at different positions but must not be used with different lens model parameters!

Parameters **kwargs** – lens model keyword argument list

Returns kwargs_updated (in case of image position convention in multiplane lensing this is changed)

lenstronomy.LensModel.lens_model_extensions module

class LensModelExtensions(*lensModel*)
Bases: object

class with extension routines not part of the LensModel core routines

__init__(*lensModel*)

Parameters **lensModel** – instance of the LensModel() class, or with same functionalities. In particular, the following definitions are required to execute all functionalities presented in this class: def ray_shooting() def magnification() def kappa() def alpha() def hessian()

caustic_area(*kwargs_lens*, *kwargs_caustic_num*, *index_vertices=0*)
computes the area inside a connected caustic curve

Parameters

- **kwargs_lens** – lens model keyword argument list
- **kwargs_caustic_num** – keyword arguments for the numerical calculation of the caustics, as input of self.critical_curve_caustics()
- **index_vertices** – integer, index of connected vortex from the output of self.critical_curve_caustics() of disconnected curves.

Returns area within the caustic curve selected

critical_curve_caustics(*kwargs_lens*, *compute_window=5*, *grid_scale=0.01*, *center_x=0*, *center_y=0*)

Parameters

- **kwargs_lens** – lens model kwargs
- **compute_window** – window size in arcsec where the critical curve is computed
- **grid_scale** – numerical grid spacing of the computation of the critical curves
- **center_x** – float, center of the window to compute critical curves and caustics
- **center_y** – float, center of the window to compute critical curves and caustics

Returns lists of ra and dec arrays corresponding to different disconnected critical curves and their caustic counterparts

critical_curve_tiling(*kwargs_lens*, *compute_window=5*, *start_scale=0.5*, *max_order=10*, *center_x=0*, *center_y=0*)

Parameters

- **kwargs_lens** – lens model keyword argument list
- **compute_window** – total window in the image plane where to search for critical curves
- **start_scale** – float, angular scale on which to start the tiling from (if there are two distinct curves in a region, it might only find one.
- **max_order** – int, maximum order in the tiling to compute critical curve triangles
- **center_x** – float, center of the window to compute critical curves and caustics
- **center_y** – float, center of the window to compute critical curves and caustics

Returns list of positions representing coordinates of the critical curve (in RA and DEC)

curved_arc_estimate(*x*, *y*, *kwargs_lens*, *smoothing=None*, *smoothing_3rd=0.001*, *tan_diff=False*)

performs the estimation of the curved arc description at a particular position of an arbitrary lens profile

Parameters

- **x** – float, x-position where the estimate is provided
- **y** – float, y-position where the estimate is provided
- **kwargs_lens** – lens model keyword arguments
- **smoothing** – (optional) finite differential of second derivative (radial and tangential stretches)
- **smoothing_3rd** – differential scale for third derivative to estimate the tangential curvature
- **tan_diff** – boolean, if True, also returns the relative tangential stretch differential in tangential direction

Returns keyword argument list corresponding to a CURVED_ARC profile at (x, y) given the initial lens model

curved_arc_finite_area(*x*, *y*, *kwargs_lens*, *dr*)

computes an estimated curved arc over a finite extent mimicking the appearance of a finite source with radius dr

Parameters

- **x** – x-position (float)
- **y** – y-position (float)
- **kwargs_lens** – lens model keyword argument list
- **dr** – radius of finite source

Returns keyword arguments of curved arc

hessian_eigenvectors(*x*, *y*, *kwargs_lens*, *diff=None*)

computes magnification eigenvectors at position (x, y)

Parameters

- **x** – x-position
- **y** – y-position
- **kwargs_lens** – lens model keyword arguments

Returns radial stretch, tangential stretch

magnification_finite(*x_pos*, *y_pos*, *kwargs_lens*, *source_sigma=0.003*, *window_size=0.1*, *grid_number=100*, *polar_grid=False*, *aspect_ratio=0.5*)

returns the magnification of an extended source with Gaussian light profile :param x_pos: x-axis positons of point sources :param y_pos: y-axis position of point sources :param kwargs_lens: lens model kwargs :param source_sigma: Gaussian sigma in arc sec in source :param window_size: size of window to compute the finite flux :param grid_number: number of grid cells per axis in the window to numerically compute the flux :return: numerically computed brightness of the sources

magnification_finite_adaptive(*x_image*, *y_image*, *source_x*, *source_y*, *kwargs_lens*, *source_fwhm_parsec*, *z_source*, *cosmo=None*, *grid_resolution=None*, *grid_radius_arcsec=None*, *axis_ratio=0.5*, *tol=0.001*, *step_size=0.05*, *use_largest_eigenvalue=True*, *source_light_model='SINGLE_GAUSSIAN'*, *dx=None*, *dy=None*, *size_scale=None*, *amp_scale=None*, *fixed_aperture_size=False*)

This method computes image magnifications with a finite-size background source assuming a Gaussian or a double Gaussian source light profile. It can be much faster that magnification_finite for lens models with many deflectors and a compact source. This is because most pixels in a rectangular window around a lensed image of a compact source do not map onto the source, and therefore don't contribute to the integrated flux in the image plane.

Rather than ray tracing through a rectangular grid, this routine accelerates the computation of image magnifications with finite-size sources by ray tracing through an elliptical region oriented such that tracks the surface brightness of the lensed image. The aperture size is initially quite small, and increases in size until the flux inside of it (and hence the magnification) converges. The orientation of the elliptical aperture is computed from the magnification tensor evaluated at the image coordinate.

If for whatever reason you prefer a circular aperture to the elliptical approximation using the hessian eigenvectors, you can just set axis_ratio = 1.

To use the eigenvalues of the hessian matrix to estimate the optimum axis ratio, set axis_ratio = 0.

The default settings for the grid resolution and ray tracing window size work well for sources with fwhm between 0.5 - 100 pc.

Parameters

- **x_image** – a list or array of x coordinates [units arcsec]
- **y_image** – a list or array of y coordinates [units arcsec]
- **source_x** – float, source position
- **source_y** – float, source position
- **kwargs_lens** – keyword arguments for the lens model
- **source_fwhm_parsec** – the size of the background source [units parsec]
- **z_source** – the source redshift
- **cosmo** – (optional) an instance of astropy.cosmology; if not specified, a default cosmology will be used

- **grid_resolution** – the grid resolution in units arcsec/pixel; if not specified, an appropriate value will be estimated from the source size
- **grid_radius_arcsec** – (optional) the size of the ray tracing region in arcsec; if not specified, an appropriate value will be estimated from the source size
- **axis_ratio** – the axis ratio of the ellipse used for ray tracing; if axis_ratio = 0, then the eigenvalues the hessian matrix will be used to estimate an appropriate axis ratio. Be warned: if the image is highly magnified it will tend to curve out of the resulting ellipse
- **tol** – tolerance for convergence in the magnification
- **step_size** – sets the increment for the successively larger ray tracing windows
- **use_largest_eigenvalue** – bool; if True, then the major axis of the ray tracing ellipse region will be aligned with the eigenvector corresponding to the largest eigenvalue of the hessian matrix
- **source_light_model** – the model for backgourn source light; currently implemented are 'SINGLE_GAUSSIAN' and 'DOUBLE_GAUSSIAN'.
- **dx** – used with source model 'DOUBLE_GAUSSIAN', the offset of the second source light profile from the first [arcsec]
- **dy** – used with source model 'DOUBLE_GAUSSIAN', the offset of the second source light profile from the first [arcsec]
- **size_scale** – used with source model 'DOUBLE_GAUSSIAN', the size of the second source light profile relative to the first
- **amp_scale** – used with source model 'DOUBLE_GAUSSIAN', the peak brightness of the second source light profile relative to the first
- **fixed_aperture_size** – bool, if True the flux is computed inside a fixed aperture size with radius grid_radius_arcsec

Returns an array of image magnifications

radial_tangential_differentials(*x*, *y*, *kwargs_lens*, *center_x=0*, *center_y=0*, *smoothing_3rd=0.001*, *smoothing_2nd=None*)

computes the differentials in stretches and directions

Parameters

- **x** – x-position
- **y** – y-position
- **kwargs_lens** – lens model keyword arguments
- **center_x** – x-coord of center towards which the rotation direction is defined
- **center_y** – x-coord of center towards which the rotation direction is defined
- **smoothing_3rd** – finite differential length of third order in units of angle
- **smoothing_2nd** – float or None, finite average differential scale of Hessian

Returns

radial_tangential_stretch(*x*, *y*, *kwargs_lens*, *diff=None*, *ra_0=0*, *dec_0=0*, *coordinate_frame_definitions=False*)

computes the radial and tangential stretches at a given position

Parameters

- **x** – x-position

- **y** – y-position
- **kwargs_lens** – lens model keyword arguments
- **diff** – float or None, finite average differential scale

Returns radial stretch, tangential stretch

tangential_average (*x*, *y*, *kwargs_lens*, *dr*, *smoothing=None*, *num_average=9*)
computes average tangential stretch around position (x, y) within dr in radial direction

Parameters

- **x** – x-position (float)
- **y** – y-position (float)
- **kwargs_lens** – lens model keyword argument list
- **dr** – averaging scale in radial direction
- **smoothing** – smoothing scale of derivative
- **num_average** – integer, number of points averaged over within dr in the radial direction

Returns

zoom_source (*x_pos*, *y_pos*, *kwargs_lens*, *source_sigma=0.003*, *window_size=0.1*, *grid_number=100*, *shape='GAUSSIAN'*)
computes the surface brightness on an image with a zoomed window

Parameters

- **x_pos** – angular coordinate of center of image
- **y_pos** – angular coordinate of center of image
- **kwargs_lens** – lens model parameter list
- **source_sigma** – source size (in angular units)
- **window_size** – window size in angular units
- **grid_number** – number of grid points per axis
- **shape** – string, shape of source, supports 'GAUSSIAN' and 'TORUS

Returns 2d numpy array

lenstronomy.LensModel.lens_param module

class LensParam (*lens_model_list*, *kwargs_fixed*, *kwargs_lower=None*, *kwargs_upper=None*, *kwargs_logsampling=None*, *num_images=0*, *solver_type='NONE'*, *num_shapelet_lens=0*)
Bases: `object`

class to handle the lens model parameter

__init__ (*lens_model_list*, *kwargs_fixed*, *kwargs_lower=None*, *kwargs_upper=None*, *kwargs_logsampling=None*, *num_images=0*, *solver_type='NONE'*, *num_shapelet_lens=0*)

Parameters

- **lens_model_list** – list of strings of lens model names
- **kwargs_fixed** – list of keyword arguments for model parameters to be held fixed

- **kwargs_lower** – list of keyword arguments of the lower bounds of the model parameters
- **kwargs_upper** – list of keyword arguments of the upper bounds of the model parameters
- **kwargs_logsampling** – list of keyword arguments of parameters to be sampled in log10 space
- **num_images** – number of images to be constrained by a non-linear solver (only relevant when shapelet potential functions are used)
- **solver_type** – string, type of non-linear solver (only relevant in this class when 'SHAPELETS' is the solver type)
- **num_shapelet_lens** – integer, number of shapelets in the lensing potential (only relevant when 'SHAPELET' lens model is used)

get_params (*args*, *i*)

Parameters

- **args** – tuple of individual floats of sampling argument
- **i** – integer, index at the beginning of the tuple for read out to keyword argument convention

Returns kwargs_list, index at the end of read out of this model component

num_param ()

Returns integer, number of free parameters being sampled from the lens model components

set_params (*kwargs_list*)

Parameters **kwargs_list** – keyword argument list of lens model components

Returns tuple of arguments (floats) that are being sampled

lenstronomy.LensModel.profile_integrals module

class ProfileIntegrals (*profile_class*)

Bases: `object`

class to perform integrals of spherical profiles to compute: - projected densities - enclosed densities - projected enclosed densities

__init__ (*profile_class*)

Parameters **profile_class** – list of lens models

density_2d (*r*, *kwargs_profile*, *lens_param=False*)

computes the projected density along the line-of-sight

Parameters

- **r** – radius (arcsec)
- **kwargs_profile** – keyword argument list with lens model parameters
- **lens_param** – boolean, if True uses the lens model parameterization in computing the 3d density convention and the return is the convergence

Returns 2d projected density at projected radius r

mass_enclosed_2d(*r*, *kwargs_profile*)
computes the mass enclosed the projected line-of-sight :param r: radius (arcsec) :param kwargs_profile: keyword argument list with lens model parameters :return: projected mass enclosed radius r

mass_enclosed_3d(*r*, *kwargs_profile*, *lens_param=False*)
computes the mass enclosed within a sphere of radius r

Parameters

- **r** – radius (arcsec)
- **kwargs_profile** – keyword argument list with lens model parameters
- **lens_param** – boolean, if True uses the lens model parameterization in computing the 3d density convention and the return is the convergence

Returns 3d mass enclosed of r

lenstronomy.LensModel.profile_list_base module

class ProfileListBase(*lens_model_list*, *numerical_alpha_class=None*, *lens_redshift_list=None*, *z_source_convention=None*, *kwargs_interp=None*)
Bases: `object`

class that manages the list of lens model class instances. This class is applicable for single plane and multi plane lensing

__init__(*lens_model_list*, *numerical_alpha_class=None*, *lens_redshift_list=None*, *z_source_convention=None*, *kwargs_interp=None*)

Parameters

- **lens_model_list** – list of strings with lens model names
- **numerical_alpha_class** – an instance of a custom class for use in NumericalAlpha() lens model deflection angles as a lens model. See the documentation in Profiles.numerical_deflections
- **kwargs_interp** – interpolation keyword arguments specifying the numerics. See description in the Interpolate() class. Only applicable for 'INTERPOL' and 'INTERPOL_SCALED' models.

set_dynamic()
frees cache set by static model (if exists) and re-computes all lensing quantities each time a definition is called assuming different parameters are executed. This is the default mode if not specified as set_static()

Returns None

set_static(*kwargs_list*)

Parameters **kwargs_list** – list of keyword arguments for each profile

Returns kwargs_list

lenstronomy.LensModel.single_plane module

class SinglePlane(*lens_model_list*, *numerical_alpha_class=None*, *lens_redshift_list=None*, *z_source_convention=None*, *kwargs_interp=None*)
Bases: `lenstronomy.LensModel.profile_list_base.ProfileListBase`

class to handle an arbitrary list of lens models in a single lensing plane

alpha (*x*, *y*, *kwargs*, *k=None*)
deflection angles :param x: x-position (preferentially arcsec) :type x: numpy array :param y: y-position (preferentially arcsec) :type y: numpy array :param kwargs: list of keyword arguments of lens model parameters matching the lens model classes :param k: only evaluate the k-th lens model :return: deflection angles in units of arcsec

density (*r*, *kwargs*, *bool_list=None*)
3d mass density at radius r The integral in the LOS projection of this quantity results in the convergence quantity.

Parameters

- **r** – radius (in angular units)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **bool_list** – list of bools that are part of the output

Returns mass density at radius r (in angular units, modulo epsilon_crit)

fermat_potential (*x_image*, *y_image*, *kwargs_lens*, *x_source=None*, *y_source=None*, *k=None*)
fermat potential (negative sign means earlier arrival time)

Parameters

- **x_image** – image position
- **y_image** – image position
- **x_source** – source position
- **y_source** – source position
- **kwargs_lens** – list of keyword arguments of lens model parameters matching the lens model classes
- **k** –

Returns fermat potential in arcsec**2 without geometry term (second part of Eqn 1 in Suyu et al. 2013) as a list

hessian (*x*, *y*, *kwargs*, *k=None*)
hessian matrix :param x: x-position (preferentially arcsec) :type x: numpy array :param y: y-position (preferentially arcsec) :type y: numpy array :param kwargs: list of keyword arguments of lens model parameters matching the lens model classes :param k: only evaluate the k-th lens model :return: f_xx, f_xy, f_yx, f_yy components

mass_2d (*r*, *kwargs*, *bool_list=None*)
computes the mass enclosed a projected (2d) radius r

The mass definition is such that:

$$\alpha = mass_2d/r/\pi$$

with alpha is the deflection angle

Parameters

- **r** – radius (in angular units)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **bool_list** – list of bools that are part of the output

Returns projected mass (in angular units, modulo epsilon_crit)

mass_3d(*r*, *kwargs*, *bool_list=None*)
computes the mass within a 3d sphere of radius r

if you want to have physical units of kg, you need to multiply by this factor: const.arcsec ** 2 * self._cosmo.dd * self._cosmo.ds / self._cosmo.dds * const.Mpc * const.c ** 2 / (4 * np.pi * const.G) grav_pot = -const.G * mass_dim / (r * const.arcsec * self._cosmo.dd * const.Mpc)

Parameters

- **r** – radius (in angular units)
- **kwargs** – list of keyword arguments of lens model parameters matching the lens model classes
- **bool_list** – list of bools that are part of the output

Returns mass (in angular units, modulo epsilon_crit)

potential(*x*, *y*, *kwargs*, *k=None*)
lensing potential :param x: x-position (preferentially arcsec) :type x: numpy array :param y: y-position (preferentially arcsec) :type y: numpy array :param kwargs: list of keyword arguments of lens model parameters matching the lens model classes :param k: only evaluate the k-th lens model :return: lensing potential in units of arcsec^2

ray_shooting(*x*, *y*, *kwargs*, *k=None*)
maps image to source position (inverse deflection) :param x: x-position (preferentially arcsec) :type x: numpy array :param y: y-position (preferentially arcsec) :type y: numpy array :param kwargs: list of keyword arguments of lens model parameters matching the lens model classes :param k: only evaluate the k-th lens model :return: source plane positions corresponding to (x, y) in the image plane

Module contents

lenstronomy.LightModel package

Subpackages

lenstronomy.LightModel.Profiles package

Submodules

lenstronomy.LightModel.Profiles.chameleon module

class Chameleon
Bases: `object`

class of the Chameleon model (See Dutton+ 2011, Suyu+2014) an elliptical truncated double isothermal profile

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

function(*x*, *y*, *amp*, *w_c*, *w_t*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – ra-coordinate

- **y** – dec-coordinate
- **w_c** –
- **w_t** –
- **amp** – amplitude of first power-law flux
- **e1** – eccentricity parameter
- **e2** – eccentricity parameter
- **center_x** – center
- **center_y** – center

Returns flux of chameleon profile

light_3d(*r*, *amp*, *w_c*, *w_t*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **r** – 3d radius
- **w_c** –
- **w_t** –
- **amp** – amplitude of first power-law flux
- **e1** – eccentricity parameter
- **e2** – eccentricity parameter
- **center_x** – center
- **center_y** – center

Returns 3d flux of chameleon profile at radius r

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'w_c', 'w_t', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

class DoubleChameleon

Bases: `object`

class of the double Chameleon model. See Dutton+2011, Suyu+2014 for the single Chameleon model.

__init__()

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

function(*x*, *y*, *amp*, *ratio*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **ratio** –
- **w_c1** –
- **w_t1** –
- **e11** –

- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns

light_3d (*r*, *amp*, *ratio*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *center_x=0*, *center_y=0*)

Parameters

- **r** – 3d radius
- **amp** –
- **ratio** – ratio of first to second amplitude of Chameleon surface brightness
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **center_x** –
- **center_y** –

Returns 3d light density at radius r

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e11': -0.8,

param_names = ['amp', 'ratio', 'w_c1', 'w_t1', 'e11', 'e21', 'w_c2', 'w_t2', 'e12', 'e

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e11': 0.8, '

class TripleChameleon

Bases: `object`

class of the Chameleon model (See Suyu+2014) an elliptical truncated double isothermal profile

__init__()

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

function (*x*, *y*, *amp*, *ratio12*, *ratio13*, *w_c1*, *w_t1*, *e11*, *e21*, *w_c2*, *w_t2*, *e12*, *e22*, *w_c3*, *w_t3*, *e13*, *e23*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –

- **amp** –
- **ratio12** – ratio of first to second amplitude
- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **w_c3** –
- **w_t3** –
- **e13** –
- **e23** –
- **center_x** –
- **center_y** –

Returns

light_3d(*r, amp, ratio12, ratio13, w_c1, w_t1, e11, e21, w_c2, w_t2, e12, e22, w_c3, w_t3, e13, e23, center_x=0, center_y=0*)

Parameters

- **r** – 3d light radius
- **amp** –
- **ratio12** – ratio of first to second amplitude
- **ratio13** – ratio of first to third amplitude
- **w_c1** –
- **w_t1** –
- **e11** –
- **e21** –
- **w_c2** –
- **w_t2** –
- **e12** –
- **e22** –
- **w_c3** –
- **w_t3** –
- **e13** –
- **e23** –

- **center_x** –
- **center_y** –

Returns

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e11': -0.8, '

param_names = ['amp', 'ratio12', 'ratio13', 'w_c1', 'w_t1', 'e11', 'e21', 'w_c2', 'w_t

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e11': 0.8, '

lenstronomy.LightModel.Profiles.ellipsoid module

class Ellipsoid

Bases: `object`

class for an universal surface brightness within an ellipsoid

__init__()

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

function(*x*, *y*, *amp*, *radius*, *e1*, *e2*, *center_x*, *center_y*)

Parameters

- **x** –
- **y** –
- **amp** – surface brightness within the ellipsoid
- **radius** – radius (product average of semi-major and semi-minor axis) of the ellipsoid
- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – center
- **center_y** – center

Returns surface brightness

lenstronomy.LightModel.Profiles.gaussian module

class Gaussian

Bases: `object`

class for Gaussian light profile The two-dimensional Gaussian profile amplitude is defined such that the 2D integral leads to the 'amp' value.

profile name in LightModel module: 'GAUSSIAN'

__init__()

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

function(*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)

surface brightness per angular unit

Parameters

- **x** – coordinate on the sky

- **y** – coordinate on the sky
- **amp** – amplitude, such that 2D integral leads to this value
- **sigma** – sigma of Gaussian in each direction
- **center_x** – center of profile
- **center_y** – center of profile

Returns surface brightness at (x, y)

light_3d(*r*, *amp*, *sigma*)
3D brightness per angular volume element

Parameters

- **r** – 3d distance from center of profile
- **amp** – amplitude, such that 2D integral leads to this value
- **sigma** – sigma of Gaussian in each direction

Returns 3D brightness per angular volume element

total_flux(*amp*, *sigma*, *center_x=0*, *center_y=0*)
integrated flux of the profile

Parameters

- **amp** – amplitude, such that 2D integral leads to this value
- **sigma** – sigma of Gaussian in each direction
- **center_x** – center of profile
- **center_y** – center of profile

Returns total flux

class GaussianEllipse
Bases: `object`

class for Gaussian light profile with ellipticity

profile name in LightModel module: 'GAUSSIAN_ELLIPSE'

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

function(*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – coordinate on the sky
- **y** – coordinate on the sky
- **amp** – amplitude, such that 2D integral leads to this value
- **sigma** – sigma of Gaussian in each direction
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus
- **center_x** – center of profile
- **center_y** – center of profile

Returns surface brightness at (x, y)

light_3d(*r*, *amp*, *sigma*, *e1=0*, *e2=0*)
3D brightness per angular volume element

Parameters

- **r** – 3d distance from center of profile
- **amp** – amplitude, such that 2D integral leads to this value
- **sigma** – sigma of Gaussian in each direction
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus

Returns 3D brightness per angular volume element

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'sigma', 'e1', 'e2', 'center_x', 'center_y']

total_flux(*amp*, *sigma=None*, *e1=None*, *e2=None*, *center_x=None*, *center_y=None*)
total integrated flux of profile

Parameters

- **amp** – amplitude, such that 2D integral leads to this value
- **sigma** – sigma of Gaussian in each direction
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus
- **center_x** – center of profile
- **center_y** – center of profile

Returns total flux

upper_limit_default = {'amp': 1000, 'center_x': 100, 'center_y': 100, 'e1': -0.5,

class MultiGaussian
Bases: object

class for elliptical pseudo Jaffe lens light (2d projected light/mass distribution

profile name in LightModel module: 'MULTI_GAUSSIAN'

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

function(*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)
surface brightness per angular unit

Parameters

- **x** – coordinate on the sky
- **y** – coordinate on the sky
- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles
- **center_x** – center of profile
- **center_y** – center of profile

Returns surface brightness at (x, y)

function_split (*x*, *y*, *amp*, *sigma*, *center_x=0*, *center_y=0*)
split surface brightness in individual components

Parameters

- **x** – coordinate on the sky
- **y** – coordinate on the sky
- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles
- **center_x** – center of profile
- **center_y** – center of profile

Returns list of arrays of surface brightness

light_3d (*r*, *amp*, *sigma*)
3D brightness per angular volume element

Parameters

- **r** – 3d distance from center of profile
- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles

Returns 3D brightness per angular volume element

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'sigma', 'center_x', 'center_y']

total_flux (*amp*, *sigma*, *center_x=0*, *center_y=0*)
total integrated flux of profile

Parameters

- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles
- **center_x** – center of profile
- **center_y** – center of profile

Returns total flux

upper_limit_default = {'amp': 1000, 'center_x': 100, 'center_y': 100, 'e1': -0.5,

class MultiGaussianEllipse
Bases: `object`

class for elliptical multi Gaussian profile

profile name in LightModel module: 'MULTI_GAUSSIAN_ELLIPSE'

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

function (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
surface brightness per angular unit

Parameters

- **x** – coordinate on the sky
- **y** – coordinate on the sky
- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus
- **center_x** – center of profile
- **center_y** – center of profile

Returns surface brightness at (x, y)

function_split (*x*, *y*, *amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
split surface brightness in individual components

Parameters

- **x** – coordinate on the sky
- **y** – coordinate on the sky
- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus
- **center_x** – center of profile
- **center_y** – center of profile

Returns list of arrays of surface brightness

light_3d (*r*, *amp*, *sigma*, *e1=0*, *e2=0*)
3D brightness per angular volume element

Parameters

- **r** – 3d distance from center of profile
- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles
- **e1** – eccentricity modulus
- **e2** – eccentricity modulus

Returns 3D brightness per angular volume element

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'sigma', 'e1', 'e2', 'center_x', 'center_y']

total_flux (*amp*, *sigma*, *e1*, *e2*, *center_x=0*, *center_y=0*)
total integrated flux of profile

Parameters

- **amp** – list of amplitudes of individual Gaussian profiles
- **sigma** – list of widths of individual Gaussian profiles

- **e1** – eccentricity modulus
- **e2** – eccentricity modulus
- **center_x** – center of profile
- **center_y** – center of profile

Returns total flux

`upper_limit_default = {'amp': 1000, 'center_x': 100, 'center_y': 100, 'e1': -0.5,`

lenstronomy.LightModel.Profiles.hernquist module

class Hernquist
Bases: `object`

class for pseudo Jaffe lens light (2d projected light/mass distribution

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

function(*x*, *y*, *amp*, *Rs*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **Rs** – scale radius: half-light radius = Rs / 0.551
- **center_x** –
- **center_y** –

Returns

light_3d(*r*, *amp*, *Rs*)

Parameters

- **r** –
- **amp** –
- **Rs** –

Returns

class HernquistEllipse
Bases: `object`

class for elliptical pseudo Jaffe lens light (2d projected light/mass distribution

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

function(*x*, *y*, *amp*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –

- **amp** –
- **Rs** –
- **e1** –
- **e2** –
- **center_x** –
- **center_y** –

Returns

light_3d(*r, amp, Rs, e1=0, e2=0*)

Parameters

- **r** –
- **amp** –
- **Rs** –
- **e1** –
- **e2** –

Returns

lower_limit_default = {'Rs': 0, 'amp': 0, 'center_x': -100, 'center_y': -100, 'e1'

param_names = ['amp', 'Rs', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Rs': 100, 'amp': 100, 'center_x': 100, 'center_y': 100, 'e

lenstronomy.LightModel.Profiles.interpolation module

class Interpol

Bases: `object`

class which uses an interpolation of an image to compute the surface brightness

parameters are 'image': 2d numpy array of surface brightness per square arc second (not integrated flux per pixel!) 'center_x': coordinate of center of image in angular units (i.e. arc seconds) 'center_y': coordinate of center of image in angular units (i.e. arc seconds) 'phi_G': rotation of image relative to the rectangular ra-to-dec orientation 'scale': arcseconds per pixel of the image to be interpolated

__init__()

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

static coord2image_pixel(*ra, dec, center_x, center_y, phi_G, scale*)

Parameters

- **ra** – angular coordinate
- **dec** – angular coordinate
- **center_x** – center of image in angular coordinates
- **center_y** – center of image in angular coordinates
- **phi_G** – rotation angle
- **scale** – pixel scale of image

Returns pixel coordinates

delete_cache()
delete the cached interpolated image

function(*x, y, image=None, amp=1, center_x=0, center_y=0, phi_G=0, scale=1*)

Parameters

- **x** – x-coordinate to evaluate surface brightness
- **y** – y-coordinate to evaluate surface brightness
- **image** (*2d numpy array*) – pixelized surface brightness (an image) to be used to interpolate in units of surface brightness (flux per square arc seconds, not flux per pixel!)
- **amp** – amplitude of surface brightness scaling in respect of original input image
- **center_x** – center of interpolated image
- **center_y** – center of interpolated image
- **phi_G** – rotation angle of simulated image in respect to input gird
- **scale** – pixel scale (in angular units) of the simulated image

Returns surface brightness from the model at coordinates (x, y)

image_interp(*x, y, image*)

lower_limit_default = {'amp': 0, 'center_x': -1000, 'center_y': -1000, 'phi_G': -3.

param_names = ['image', 'amp', 'center_x', 'center_y', 'phi_G', 'scale']

static total_flux(*image, scale, amp=1, center_x=0, center_y=0, phi_G=0*)

sums up all the image surface brightness (image pixels defined in surface brightness at the coordinate of the pixel) times pixel area

Parameters

- **image** – pixelized surface brightness used to interpolate in units of surface brightness (flux per square arc seconds, not flux per pixel!)
- **scale** – scale of the pixel in units of angle
- **amp** – linear scaling parameter of the surface brightness multiplicative with the initial image
- **center_x** – center of image in angular coordinates
- **center_y** – center of image in angular coordinates
- **phi_G** – rotation angle

Returns total flux of the image

upper_limit_default = {'amp': 1000000, 'center_x': 1000, 'center_y': 1000, 'phi_G':

lenstronomy.LightModel.Profiles.moffat module

class Moffat
Bases: `object`

this class contains functions to evaluate a Moffat surface brightness profile

$$I(r) = I_0 * (1 + (r/\alpha)^2)^{-\beta}$$

with $I_0 = amp$.

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

function(*x*, *y*, *amp*, *alpha*, *beta*, *center_x=0*, *center_y=0*)
2D Moffat profile

Parameters

- **x** – x-position (angle)
- **y** – y-position (angle)
- **amp** – normalization
- **alpha** – scale
- **beta** – exponent
- **center_x** – x-center
- **center_y** – y-center

Returns surface brightness

lenstronomy.LightModel.Profiles.nie module

class NIE
Bases: *lenstronomy.LightModel.Profiles.profile_base.LightProfileBase*

non-divergent isothermal ellipse (projected) This is effectively the convergence profile of the NIE lens model with an amplitude 'amp' rather than an Einstein radius 'theta_E'

function(*x*, *y*, *amp*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **amp** – surface brightness normalization
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale (square averaged of minor and major axis)
- **center_x** – center of profile
- **center_y** – center of profile

Returns surface brightness of NIE profile

light_3d(*r*, *amp*, *e1*, *e2*, *s_scale*, *center_x=0*, *center_y=0*)
3d light distribution (in spherical regime)

Parameters

- **r** – 3d radius

- **amp** – surface brightness normalization
- **e1** – eccentricity component
- **e2** – eccentricity component
- **s_scale** – smoothing scale (square averaged of minor and major axis)
- **center_x** – center of profile
- **center_y** – center of profile

Returns light density at 3d radius

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, 'e

param_names = ['amp', 'e1', 'e2', 's_scale', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

lenstronomy.LightModel.Profiles.p_jaffe module

class PJaffe

Bases: `object`

class for pseudo Jaffe lens light (2d projected light/mass distribution)

__init__()

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

function(*x, y, amp, Ra, Rs, center_x=0, center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **Ra** –
- **Rs** –
- **center_x** –
- **center_y** –

Returns

light_3d(*r, amp, Ra, Rs*)

Parameters

- **r** –
- **amp** –
- **Rs** –
- **Ra** –

Returns

lower_limit_default = {'Ra': 0, 'Rs': 0, 'amp': 0, 'center_x': -100, 'center_y':

param_names = ['amp', 'Ra', 'Rs', 'center_x', 'center_y']

upper_limit_default = {'Ra': 100, 'Rs': 100, 'amp': 100, 'center_x': 100, 'center_

class PJaffeEllipse
Bases: `object`

calss for elliptical pseudo Jaffe lens light

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

function(*x*, *y*, *amp*, *Ra*, *Rs*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **Ra** –
- **Rs** –
- **center_x** –
- **center_y** –

Returns

light_3d(*r*, *amp*, *Ra*, *Rs*, *e1=0*, *e2=0*)

Parameters

- **r** –
- **amp** –
- **Ra** –
- **Rs** –

Returns

lower_limit_default = {'Ra': 0, 'Rs': 0, 'amp': 0, 'center_x': -100, 'center_y':

param_names = ['amp', 'Ra', 'Rs', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'Ra': 100, 'Rs': 100, 'amp': 100, 'center_x': 100, 'center_

lenstronomy.LightModel.Profiles.power_law module

class PowerLaw
Bases: `object`

class for power-law elliptical light distribution

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

function(*x*, *y*, *amp*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*)

Parameters

- **x** – ra-coordinate
- **y** – dec-coordinate

- **amp** – amplitude of flux
- **gamma** – projected power-law slope
- **e1** – ellipticity
- **e2** – ellipticity
- **center_x** – center
- **center_y** – center

Returns projected flux

light_3d(*r, amp, gamma, e1=0, e2=0*)

Parameters

- **r** –
- **amp** –
- **gamma** –
- **e1** –
- **e2** –

Returns

lower_limit_default = {'amp': 0, 'center_x': -100, 'center_y': -100, 'e1': -0.5, '

param_names = ['amp', 'gamma', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'amp': 100, 'center_x': 100, 'center_y': 100, 'e1': 0.5, 'e

lenstronomy.LightModel.Profiles.profile_base module

class LightProfileBase

Bases: `object`

base class of all light profiles

__init__()

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

function(**args, **kwargs*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **kwargs** – keyword arguments of profile

Returns surface brightness, raise as definition is not defined

light_3d(**args, **kwargs*)

Parameters

- **r** – 3d radius
- **kwargs** – keyword arguments of profile

Returns 3d light profile, raise as definition is not defined

lenstronomy.LightModel.Profiles.sersic module

class Sersic(*smoothing=1e-05*, *sersic_major_axis=False*)

Bases: `lenstronomy.LensModel.Profiles.sersic_utils.SersicUtil`

this class contains functions to evaluate a spherical Sersic function

$$I(R) = I_0 \exp \left[-b_n (R/R_{\rm Sersic})^{\frac{1}{n}} \right]$$

with $I_0 = amp$ and with $b_n \approx 1.999n - 0.327$

function(*x*, *y*, *amp*, *R_sersic*, *n_sersic*, *center_x=0*, *center_y=0*, *max_R_frac=1000.0*)

Parameters

- **x** –
- **y** –
- **amp** – surface brightness/amplitude value at the half light radius
- **R_sersic** – semi-major axis half light radius
- **n_sersic** – Sersic index
- **center_x** – center in x-coordinate
- **center_y** – center in y-coordinate
- **max_R_frac** – maximum window outside which the mass is zeroed, in units of R_sersic (float)

Returns Sersic profile value at (x, y)

lower_limit_default = {'R_sersic': 0, 'amp': 0, 'center_x': -100, 'center_y': -100

param_names = ['amp', 'R_sersic', 'n_sersic', 'center_x', 'center_y']

upper_limit_default = {'R_sersic': 100, 'amp': 100, 'center_x': 100, 'center_y': 1

class SersicElliptic(*smoothing=1e-05*, *sersic_major_axis=False*)

Bases: `lenstronomy.LensModel.Profiles.sersic_utils.SersicUtil`

this class contains functions to evaluate an elliptical Sersic function

$$I(R) = I_0 \exp \left[-b_n (R/R_{\rm Sersic})^{\frac{1}{n}} \right]$$

with $I_0 = amp$, $R = \sqrt{q\theta^2_x + \theta^2_y/q}$ and with $b_n \approx 1.999n - 0.327$

function(*x*, *y*, *amp*, *R_sersic*, *n_sersic*, *e1*, *e2*, *center_x=0*, *center_y=0*, *max_R_frac=1000.0*)

Parameters

- **x** –
- **y** –
- **amp** – surface brightness/amplitude value at the half light radius
- **R_sersic** – half light radius (either semi-major axis or product average of semi-major and semi-minor axis)
- **n_sersic** – Sersic index
- **e1** – eccentricity parameter e1

- **e2** – eccentricity parameter e2
- **center_x** – center in x-coordinate
- **center_y** – center in y-coordinate
- **max_R_frac** – maximum window outside which the mass is zeroed, in units of R_sersic (float)

Returns Sersic profile value at (x, y)

lower_limit_default = {'R_sersic': 0, 'amp': 0, 'center_x': -100, 'center_y': -100

param_names = ['amp', 'R_sersic', 'n_sersic', 'e1', 'e2', 'center_x', 'center_y']

upper_limit_default = {'R_sersic': 100, 'amp': 100, 'center_x': 100, 'center_y': 1

class CoreSersic(*smoothing=1e-05*, *sersic_major_axis=False*)

Bases: *lenstronomy.LensModel.Profiles.sersic_utils.SersicUtil*

this class contains the Core-Sersic function introduced by e.g. Trujillo et al. 2004

$$I(R) = I' \left[1 + (R_b/R)^{\alpha}\right]^{\gamma/\alpha} \exp -b_n \left[(R^{\alpha} + R_b^{\alpha})/R_e^{\alpha}\right]^{1/(n\alpha)}$$

with

$$I' = I_b 2^{-\gamma/\alpha} \exp\left[b_n 2^{1/(n\alpha)} (R_b/R_e)^{1/n}\right]$$

where I_b is the intensity at the break radius and $R = \sqrt{q\theta_x^2 + \theta_y^2/q}$.

function(*x*, *y*, *amp*, *R_sersic*, *Rb*, *n_sersic*, *gamma*, *e1*, *e2*, *center_x=0*, *center_y=0*, *alpha=3.0*, *max_R_frac=1000.0*)

Parameters

- **x** –
- **y** –
- **amp** – surface brightness/amplitude value at the half light radius
- **R_sersic** – half light radius (either semi-major axis or product average of semi-major and semi-minor axis)
- **Rb** – "break" core radius
- **n_sersic** – Sersic index
- **gamma** – inner power-law exponent
- **e1** – eccentricity parameter e1
- **e2** – eccentricity parameter e2
- **center_x** – center in x-coordinate
- **center_y** – center in y-coordinate
- **alpha** – sharpness of the transition between the cusp and the outer Sersic profile (float)
- **max_R_frac** – maximum window outside which the mass is zeroed, in units of R_sersic (float)

Returns Cored Sersic profile value at (x, y)

lower_limit_default = {'Rb': 0, 'amp': 0, 'center_x': -100, 'center_y': -100, 'e1'

param_names = ['amp', 'R_sersic', 'Rb', 'n_sersic', 'gamma', 'e1', 'e2', 'center_x', '

upper_limit_default = {'Rb': 100, 'amp': 100, 'center_x': 100, 'center_y': 100, 'e

lenstronomy.LightModel.Profiles.shapelets module

class Shapelets(*interpolation=False*, *precalc=False*, *stable_cut=True*, *cut_scale=5*)

Bases: `object`

class for 2d cartesian Shapelets.

Sources: Refregier 2003: Shapelets: I. A Method for Image Analysis https://arxiv.org/abs/astro-ph/0105178 Refregier 2003: Shapelets: II. A Method for Weak Lensing Measurements https://arxiv.org/abs/astro-ph/0105179

For one dimension, the shapelets are defined as

$$\phi_n(x) \equiv \left[2^n \pi^{1/2} n!\right]]^{-1/2} H_n(x) e^{-\frac{x^2}{2}}$$

This basis is orthonormal. The dimensional basis function is

$$B_n(x;\beta) \equiv \beta^{-1/2} \phi_n(\beta^{-1} x)$$

which are orthonormal as well.

The two-dimensional basis function is

$$\phi_{\mathbf{n}}(fx) \equiv \phi_{n1}(x1)\phi_{n2}(x2)$$

where $\mathbf{n} \equiv (n1, n2)$ and $\mathbf{x} \equiv (x1, x2)$.

The dimensional two-dimentional basis function is

$$B_{\mathbf{n}}(\mathbf{x};\beta) \equiv \beta^{-1/2} \phi_{\mathbf{n}}(\beta^{-1}\mathbf{x}).$$

H_n(*n*, *x*)

constructs the Hermite polynomial of order n at position x (dimensionless)

Parameters

- **n** – The n'the basis function.
- **x** (`float or numpy array.`) – 1-dim position (dimensionless)

Returns array– H_n(x).

__init__(*interpolation=False*, *precalc=False*, *stable_cut=True*, *cut_scale=5*)

load interpolation of the Hermite polynomials in a range [-30,30] in order n<= 150

Parameters

- **interpolation** – boolean; if True, uses interpolated pre-calculated shapelets in the evaluation
- **precalc** – boolean; if True interprets as input (x, y) as pre-calculated normalized shapelets
- **stable_cut** – boolean; if True, sets the values outside of $\sqrt{(n_{\max} + 1)\,\beta s_{\rm cutscale} = 0}$.
- **cut_scale** – float, scaling parameter where to cut the shapelets. This is for numerical reasons such that the polynomials in the Hermite function do not get unstable.

function (*x*, *y*, *amp*, *beta*, *n1*, *n2*, *center_x*, *center_y*)
2d cartesian shapelet

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **amp** – amplitude of shapelet
- **beta** – scale factor of shapelet
- **n1** – x-order of Hermite polynomial
- **n2** – y-order of Hermite polynomial
- **center_x** – center in x
- **center_y** – center in y

Returns flux surface brighness at (x, y)

hermval (*x*, *n_array*, *tensor=True*)
computes the Hermit polynomial as numpy.polynomial.hermite.hermval difference: for values more than sqrt(n_max + 1) * cut_scale, the value is set to zero this should be faster and numerically stable

Parameters

- **x** – array of values
- **n_array** – list of coeffs in H_n
- **tensor** – see numpy.polynomial.hermite.hermval

Returns see numpy.polynomial.hermite.hermval

lower_limit_default = {'amp': 0, 'beta': 0.01, 'center_x': -100, 'center_y': -100,

param_names = ['amp', 'beta', 'n1', 'n2', 'center_x', 'center_y']

phi_n (*n*, *x*)
constructs the 1-dim basis function (formula (1) in Refregier et al. 2001)

Parameters

- **n** (*int.*) – The n'the basis function.
- **x** (*float or numpy array.*) – 1-dim position (dimensionless)

Returns array– phi_n(x).

pre_calc (*x*, *y*, *beta*, *n_order*, *center_x*, *center_y*)
calculates the H_n(x) and H_n(y) for a given x-array and y-array for the full order in the polynomials

Parameters

- **x** – x-coordinates (numpy array)
- **y** – 7-coordinates (numpy array)
- **beta** – shapelet scale
- **n_order** – order of shapelets
- **center_x** – shapelet center
- **center_y** – shapelet center

Returns list of H_n(x) and H_n(y)

upper_limit_default = {'amp': 100, 'beta': 100, 'center_x': 100, 'center_y': 100,

class ShapeletSet

Bases: `object`

class to operate on entire shapelet set limited by a maximal polynomial order n_max, such that n1 + n2 <= n_max

__init__()

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

decomposition(*image*, *x*, *y*, *n_max*, *beta*, *deltaPix*, *center_x=0*, *center_y=0*)

decomposes an image into the shapelet coefficients in same order as for the function call

Parameters

- **image** –
- **x** –
- **y** –
- **n_max** –
- **beta** –
- **center_x** –
- **center_y** –

Returns

function(*x*, *y*, *amp*, *n_max*, *beta*, *center_x=0*, *center_y=0*)

Parameters

- **x** – x-coordinates
- **y** – y-coordinates
- **amp** – array of amplitudes in pre-defined order of shapelet basis functions
- **beta** – shapelet scale
- **n_max** – maximum polynomial order in Hermite polynomial
- **center_x** – shapelet center
- **center_y** – shapelet center

Returns surface brightness of combined shapelet set

function_split(*x*, *y*, *amp*, *n_max*, *beta*, *center_x=0*, *center_y=0*)

splits shapelet set in list of individual shapelet basis function responses

Parameters

- **x** – x-coordinates
- **y** – y-coordinates
- **amp** – array of amplitudes in pre-defined order of shapelet basis functions
- **beta** – shapelet scale
- **n_max** – maximum polynomial order in Hermite polynomial
- **center_x** – shapelet center
- **center_y** – shapelet center

Returns list of individual shapelet basis function responses

lower_limit_default = {'beta': 0.01, 'center_x': -100, 'center_y': -100}

param_names = ['amp', 'n_max', 'beta', 'center_x', 'center_y']

shapelet_basis_2d(*num_order*, *beta*, *numPix*, *deltaPix=1*, *center_x=0*, *center_y=0*)

Parameters

- **num_order** – max shapelet order
- **beta** – shapelet scale
- **numPix** – number of pixel of the grid

Returns list of shapelets drawn on pixel grid, centered.

upper_limit_default = {'beta': 100, 'center_x': 100, 'center_y': 100}

lenstronomy.LightModel.Profiles.shapelets_polar module

class ShapeletsPolar

Bases: `object`

2D polar Shapelets, see Massey & Refregier 2005

__init__()

load interpolation of the Hermite polynomials in a range [-30,30] in order n<= 150 :return:

function(*x*, *y*, *amp*, *beta*, *n*, *m*, *complex_bool*, *center_x*, *center_y*)

Parameters

- **x** – x-coordinate, numpy array
- **y** – y-ccordinate, numpy array
- **amp** – amplitude normalization
- **beta** – shaplet scale
- **n** – order of polynomial
- **m** – rotational invariance
- **complex_bool** – boolean; if True uses complex value of function _chi_n_m()
- **center_x** – center of shapelet
- **center_y** – center of shapelet

Returns amplitude of shapelet at possition (x, y)

index2poly(*index*)

manages the convention from an iterative index to the specific polynomial n, m, (real/imaginary part)

Parameters **index** – int, index of list

Returns n, m bool

lower_limit_default = {'amp': 0, 'beta': 0, 'center_x': -100, 'center_y': -100, 'm

static num_param(*n_max*)

Parameters **n_max** – maximal polynomial order

Returns number of basis components

param_names = ['amp', 'beta', 'n', 'm', 'center_x', 'center_y']

param_names_latex = {'I_0', '$\\beta$', 'm', 'n', 'x_0', 'y_0'}

static poly2index (*n, m, complex_bool*)

Parameters

- **n** – non-negative integer
- **m** – integer, running from -n to n in steps of two
- **complex_bool** – bool, if True, assigns complex part

Returns

upper_limit_default = {'amp': 100, 'beta': 100, 'center_x': 100, 'center_y': 100,

class ShapeletsPolarExp

Bases: object

2D exponential shapelets, Berge et al. 2019

__init__ ()

load interpolation of the Hermite polynomials in a range [-30,30] in order n<= 150 :return:

function (*x, y, amp, beta, n, m, complex_bool, center_x, center_y*)

Parameters

- **x** – x-coordinate, numpy array
- **y** – y-ccordinate, numpy array
- **amp** – amplitude normalization
- **beta** – shaplet scale
- **n** – order of polynomial
- **m** – rotational invariance
- **complex_bool** – boolean; if True uses complex value of function _chi_n_m()
- **center_x** – center of shapelet
- **center_y** – center of shapelet

Returns amplitude of shapelet at possition (x, y)

index2poly (*index*)

Parameters **index** –

Returns

lower_limit_default = {'amp': 0, 'beta': 0, 'center_x': -100, 'center_y': -100, 'm

static num_param (*n_max*)

Parameters **n_max** – maximal polynomial order

Returns number of basis components

param_names = ['amp', 'beta', 'n', 'm', 'center_x', 'center_y']

static poly2index (*n, m, complex_bool*)

Parameters

- **n** –

- **m** –
- **complex_bool** –

Returns index convention, integer

upper_limit_default = {'amp': 100, 'beta': 100, 'center_x': 100, 'center_y': 100,

class ShapeletSetPolar(*exponential=False*)
Bases: object

class to operate on entire shapelet set

__init__(*exponential=False*)
Initialize self. See help(type(self)) for accurate signature.

decomposition(*image, x, y, n_max, beta, deltaPix, center_x=0, center_y=0*)
decomposes an image into the shapelet coefficients in same order as for the function call :param image: :param x: :param y: :param n_max: :param beta: :param center_x: :param center_y: :return:

function(*x, y, amp, n_max, beta, center_x=0, center_y=0*)

Parameters

- **x** –
- **y** –
- **amp** –
- **n_max** –
- **beta** –
- **center_x** –
- **center_y** –

Returns

function_split(*x, y, amp, n_max, beta, center_x=0, center_y=0*)

index2poly(*index*)

Parameters index – index of coefficient in the convention here

Returns n, m, complex_bool

lower_limit_default = {'beta': 0, 'center_x': -100, 'center_y': -100}

param_names = ['amp', 'n_max', 'beta', 'center_x', 'center_y']

upper_limit_default = {'beta': 100, 'center_x': 100, 'center_y': 100}

lenstronomy.LightModel.Profiles.starlets module

class SLIT_Starlets(*thread_count=1, fast_inverse=True, second_gen=False, show_pysap_plots=False, force_no_pysap=False*)
Bases: object

Decomposition of an image using the Isotropic Undecimated Walevet Transform, also known as “starlet” or “B-spline”, using the ‘a trous’ algorithm.

Astronomical data (galaxies, stars, . . .) are often very sparsely represented in the starlet basis.

Based on Starck et al. : https://ui.adsabs.harvard.edu/abs/2007ITIP. . . 16..297S/abstract

__init__(*thread_count=1*, *fast_inverse=True*, *second_gen=False*, *show_pysap_plots=False*, *force_no_pysap=False*)

Load pySAP package if found, and initialize the Starlet transform.

Parameters

- **thread_count** – number of threads used for pySAP computations
- **fast_inverse** – if True, reconstruction is simply the sum of each scale (only for 1st generation starlet transform)
- **second_gen** – if True, uses the second generation of starlet transform
- **show_pysap_plots** – if True, displays pySAP plots when calling the decomposition method
- **force_no_pysap** – if True, does not load pySAP and computes starlet transforms in python.

decomposition(*image*, *n_scales*)

1D starlet transform from starlet coefficients stored in coeffs

Parameters

- **image** – 2D image to be decomposed, ndarray with shape (sqrt(n_pixels), sqrt(n_pixels))
- **n_scales** – number of decomposition scales

Returns reconstructed signal as 1D array of shape (n_scales*n_pixels,)

decomposition_2d(*image*, *n_scales*)

2D starlet transform from starlet coefficients stored in coeffs

Parameters

- **image** – 2D image to be decomposed, ndarray with shape (sqrt(n_pixels), sqrt(n_pixels))
- **n_scales** – number of decomposition scales

Returns reconstructed signal as 2D array of shape (n_scales, sqrt(n_pixels), sqrt(n_pixels))

delete_cache()

delete the cached interpolated image

function(*x*, *y*, *amp=None*, *n_scales=None*, *n_pixels=None*, *scale=1*, *center_x=0*, *center_y=0*)

1D inverse starlet transform from starlet coefficients stored in coeffs Follows lenstronomy conventions for light profiles.

Parameters

- **amp** – decomposition coefficients ('amp' to follow conventions in other light profile) This is an ndarray with shape (n_scales, sqrt(n_pixels), sqrt(n_pixels)) or (n_scales*n_pixels,)
- **n_scales** – number of decomposition scales
- **n_pixels** – number of pixels in a single scale

Returns reconstructed signal as 1D array of shape (n_pixels,)

function_2d(*coeffs*, *n_scales*, *n_pixels*)

2D inverse starlet transform from starlet coefficients stored in coeffs

Parameters

- **coeffs** – decomposition coefficients, ndarray with shape (n_scales, sqrt(n_pixels), sqrt(n_pixels))
- **n_scales** – number of decomposition scales

Returns reconstructed signal as 2D array of shape (sqrt(n_pixels), sqrt(n_pixels))

lower_limit_default = {'amp': [0], 'center_x': -1000, 'center_y': -1000, 'n_pixels'

param_names = ['amp', 'n_scales', 'n_pixels', 'scale', 'center_x', 'center_y']

param_names_latex = {'I_0', '$n_{\\rm pix}$', '$n_{\\rm scales}$', 'x_0', 'y_0',

upper_limit_default = {'amp': [100000000.0], 'center_x': 1000, 'center_y': 1000, 'n

lenstronomy.LightModel.Profiles.starlets_util module

transform(*img*, *n_scales*, *second_gen=False*)

Performs starlet decomposition of an 2D array.

Parameters

- **img** – input image
- **n_scales** – number of decomposition scales
- **second_gen** – if True, 'second generation' starlets are used

inverse_transform(*wave*, *fast=True*, *second_gen=False*)

Reconstructs an image fron its starlet decomposition coefficients

Parameters

- **wave** – input coefficients, with shape (n_scales, np.sqrt(n_pixel), np.sqrt(n_pixel))
- **fast** – if True, and only with second_gen is False, simply sums up all scales to reconstruct the image
- **second_gen** – if True, 'second generation' starlets are used

lenstronomy.LightModel.Profiles.uniform module

class Uniform

Bases: `object`

uniform light profile. This profile can also compensate for an inaccurate background subtraction. name for profile: 'UNIFORM'

__init__()

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

function(*x*, *y*, *amp*)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **amp** – surface brightness

Returns constant flux

lower_limit_default = {'amp': -100}

param_names = ['amp']

param_names_latex = {'I_0'}

upper_limit_default = {'amp': 100}

Module contents

Submodules

lenstronomy.LightModel.light_model module

class LightModel(*light_model_list*, *deflection_scaling_list=None*, *source_redshift_list=None*, *smoothing=0.001*, *sersic_major_axis=None*)

Bases: `lenstronomy.LightModel.linear_basis.LinearBasis`

class to handle extended surface brightness profiles (for e.g. source and lens light)

all profiles come with a surface_brightness parameterization (in units per square angle and independent of the pixel scale). The parameter 'amp' is the linear scaling parameter of surface brightness. Some functional forms come with a total_flux() definition that provide the integral of the surface brightness for a given set of parameters.

The SimulationAPI module allows to use astronomical magnitudes to be used and translated into the surface brightness conventions of this module given a magnitude zero point.

__init__(*light_model_list*, *deflection_scaling_list=None*, *source_redshift_list=None*, *smoothing=0.001*, *sersic_major_axis=None*)

Parameters

- **light_model_list** – list of light models
- **deflection_scaling_list** – list of floats indicating a relative scaling of the deflection angle from the reduced angles in the lens model definition (optional, only possible in single lens plane with multiple source planes)
- **source_redshift_list** – list of redshifts for the different light models (optional and only used in multi-plane lensing in conjunction with a cosmology model)
- **smoothing** – smoothing factor for certain models (deprecated)
- **sersic_major_axis** – boolean or None, if True, uses the semi-major axis as the definition of the Sersic half-light radius, if False, uses the product average of semi-major and semi-minor axis. If None, uses the convention in the lenstronomy yaml setting (which by default is =False)

lenstronomy.LightModel.light_param module

class LightParam(*light_model_list*, *kwargs_fixed*, *kwargs_lower=None*, *kwargs_upper=None*, *param_type='light'*, *linear_solver=True*)

Bases: `object`

class manages the parameters corresponding to the LightModel() module. Also manages linear parameter handling.

__init__(*light_model_list*, *kwargs_fixed*, *kwargs_lower=None*, *kwargs_upper=None*, *param_type='light'*, *linear_solver=True*)

Parameters

- **light_model_list** – list of light models
- **kwargs_fixed** – list of keyword arguments corresponding to parameters held fixed during sampling

- **kwargs_lower** – list of keyword arguments indicating hard lower limit of the parameter space
- **kwargs_upper** – list of keyword arguments indicating hard upper limit of the parameter space
- **param_type** – string (optional), adding specifications in the output strings (such as lens light or source light)
- **linear_solver** – bool, if True fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

get_params (*args*, *i*)

Parameters

- **args** – list of floats corresponding ot the arguments being sampled
- **i** – int, index of the first argument that is managed/read-out by this class

Returns keyword argument list of the light profile, index after reading out the arguments corresponding to this class

num_param (*latex_style=False*)

Parameters **latex_style** – boolena; if True, returns latex strings for plotting

Returns int, list of strings with param names

num_param_linear ()

Returns number of linear basis set coefficients

param_name_list

set_params (*kwargs_list*)

Parameters **kwargs_list** – list of keyword arguments of the light profile (free parameter as well as optionally the fixed ones)

Returns list of floats corresponding to the free parameters

Module contents

lenstronomy.Plots package

Submodules

lenstronomy.Plots.chain_plot module

plot_chain_list (*chain_list*, *index=0*, *num_average=100*)

plots the output of a chain of samples (MCMC or PSO) with the some diagnostics of convergence. This routine is an example and more tests might be appropriate to analyse a specific chain.

Parameters

- **chain_list** – list of chains with arguments [type string, samples etc. . .]
- **index** – index of chain to be plotted
- **num_average** – in chains, number of steps to average over in plotting diagnostics

Returns plotting instance figure, axes (potentially multiple)

plot_chain (*chain*, *param_list*)

plot_mcmc_behaviour (*ax*, *samples_mcmc*, *param_mcmc*, *dist_mcmc=None*, *num_average=100*)
plots the MCMC behaviour and looks for convergence of the chain

Parameters

- **ax** – matplotlib.axis instance
- **samples_mcmc** – parameters sampled 2d numpy array
- **param_mcmc** – list of parameters
- **dist_mcmc** – log likelihood of the chain
- **num_average** – number of samples to average (should coincide with the number of samples in the emcee process)

Returns

psf_iteration_compare (*kwargs_psf*, ***kwargs*)

Parameters

- **kwargs_psf** – keyword arguments that initiate a PSF() class
- **kwargs** – kwargs to send to matplotlib.pyplot.matshow()

Returns

lenstronomy.Plots.lens_plot module

lens_model_plot (*ax*, *lensModel*, *kwargs_lens*, *numPix=500*, *deltaPix=0.01*, *sourcePos_x=0*, *sourcePos_y=0*, *point_source=False*, *with_caustics=False*, *with_convergence=True*, *coord_center_ra=0*, *coord_center_dec=0*, *coord_inverse=False*, *fast_caustic=True*, ***kwargs*)
plots a lens model (convergence) and the critical curves and caustics

Parameters

- **ax** – matplotlib axis instance
- **lensModel** – LensModel() class instance
- **kwargs_lens** – lens model keyword argument list
- **numPix** – total number of pixels (for convergence map)
- **deltaPix** – width of pixel (total frame size is deltaPix x numPix)
- **sourcePos_x** – float, x-position of point source (image positions computed by the lens equation)
- **sourcePos_y** – float, y-position of point source (image positions computed by the lens equation)
- **point_source** – bool, if True, illustrates and computes the image positions of the point source
- **with_caustics** – bool, if True, illustrates the critical curve and caustics of the system
- **with_convergence** – bool, if True, illustrates the convergence map
- **coord_center_ra** – float, x-coordinate of the center of the frame
- **coord_center_dec** – float, y-coordinate of the center of the frame

- **coord_inverse** – bool, if True, inverts the x-coordinates to go from right-to-left (effectively the RA definition)
- **fast_caustic** – boolean, if True, uses faster but less precise caustic calculation (might have troubles for the outer caustic (inner critical curve)
- **with_convergence** – boolean, if True, plots the convergence of the deflector

Returns

arrival_time_surface(*ax*, *lensModel*, *kwargs_lens*, *numPix=500*, *deltaPix=0.01*, *sourcePos_x=0*, *sourcePos_y=0*, *with_caustics=False*, *point_source=False*, *n_levels=10*, *kwargs_contours=None*, *image_color_list=None*, *letter_font_size=20*, *name_list=None*)

Parameters

- **ax** – matplotlib axis instance
- **lensModel** – LensModel() class instance
- **kwargs_lens** – lens model keyword argument list
- **numPix** –
- **deltaPix** –
- **sourcePos_x** –
- **sourcePos_y** –
- **with_caustics** –
- **point_source** –
- **name_list** (*list of strings, longer or equal the number of point sources*) – list of names of images

Returns

curved_arc_illustration(*ax*, *lensModel*, *kwargs_lens*, *with_centroid=True*, *stretch_scale=0.1*, *color='k'*)

Parameters

- **ax** – matplotlib axis instance
- **lensModel** – LensModel() instance
- **kwargs_lens** – list of lens model keyword arguments (only those of CURVED_ARC considered
- **with_centroid** – plots the center of the curvature radius
- **stretch_scale** – float, relative scale of banana to the tangential and radial stretches (effectively intrinsic source size)
- **color** – string, matplotlib color for plot

Returns matplotlib axis instance

plot_arc(*ax*, *tangential_stretch*, *radial_stretch*, *curvature*, *direction*, *center_x*, *center_y*, *stretch_scale=0.1*, *with_centroid=True*, *linewidth=1*, *color='k'*, *dtan_dtan=0*)

Parameters

- **ax** – matplotlib.axes instance
- **tangential_stretch** – float, stretch of intrinsic source in tangential direction

- **radial_stretch** – float, stretch of intrinsic source in radial direction
- **curvature** – 1/curvature radius
- **direction** – float, angle in radian
- **center_x** – center of source in image plane
- **center_y** – center of source in image plane
- **with_centroid** – plots the center of the curvature radius
- **stretch_scale** – float, relative scale of banana to the tangential and radial stretches (effectively intrinsic source size)
- **linewidth** – linewidth
- **color** (*string in matplotlib color convention*) – color
- **dtan_dtan** – tangential eigenvector differential in tangential direction (not implemented yet as illustration)

Returns

distortions (*lensModel*, *kwargs_lens*, *num_pix=100*, *delta_pix=0.05*, *center_ra=0*, *center_dec=0*, *differential_scale=0.0001*, *smoothing_scale=None*, ***kwargs*)

Parameters

- **lensModel** – LensModel instance
- **kwargs_lens** – lens model keyword argument list
- **num_pix** – number of pixels per axis
- **delta_pix** – pixel scale per axis
- **center_ra** – center of the grid
- **center_dec** – center of the grid
- **differential_scale** – scale of the finite derivative length in units of angles
- **smoothing_scale** – float or None, Gaussian FWHM of a smoothing kernel applied before plotting

Returns matplotlib instance with different panels

lenstronomy.Plots.model_band_plot module

class ModelBandPlot (*multi_band_list*, *kwargs_model*, *model*, *error_map*, *cov_param*, *param*, *kwargs_params*, *likelihood_mask_list=None*, *band_index=0*, *arrow_size=0.02*, *cmap_string='gist_heat'*, *fast_caustic=True*)

Bases: `lenstronomy.Analysis.image_reconstruction.ModelBand`

class to plot a single band given the modeling results

__init__ (*multi_band_list*, *kwargs_model*, *model*, *error_map*, *cov_param*, *param*, *kwargs_params*, *likelihood_mask_list=None*, *band_index=0*, *arrow_size=0.02*, *cmap_string='gist_heat'*, *fast_caustic=True*)

Parameters

- **multi_band_list** – list of imaging data configuration [[kwargs_data, kwargs_psf, kwargs_numerics], [. . .]]

- **kwargs_model** – model keyword argument list for the full multi-band modeling
- **model** – 2d numpy array of modeled image for the specified band
- **error_map** – 2d numpy array of size of the image, additional error in the pixels coming from PSF uncertainties
- **cov_param** – covariance matrix of the linear inversion
- **param** – 1d numpy array of the linear coefficients of this imaging band
- **kwargs_params** – keyword argument of keyword argument lists of the different model components selected for the imaging band, NOT including linear amplitudes (not required as being overwritten by the param list)
- **likelihood_mask_list** – list of 2d numpy arrays of likelihood masks (for all bands)
- **band_index** – integer of the band to be considered in this class
- **arrow_size** – size of the scale and orientation arrow
- **cmap_string** – string of color map (or cmap matplotlib object)
- **fast_caustic** – boolean; if True, uses fast (but less accurate) caustic calculation method

absolute_residual_plot(*ax*, *v_min=-1*, *v_max=1*, *font_size=15*, *text='Residuals'*, *colorbar_label='(f$_{model}$-f$_{data}$)'*)

Parameters **ax** –

Returns

convergence_plot(*ax*, *text='Convergence'*, *v_min=None*, *v_max=None*, *font_size=15*, *colorbar_label='$\\log_{10}\\ \\kappa$'*, ***kwargs*)

Parameters **ax** – matplotib axis instance

Returns convergence plot in ax instance

data_plot(*ax*, *v_min=None*, *v_max=None*, *text='Observed'*, *font_size=15*, *colorbar_label='log$_{10}$ flux'*, ***kwargs*)

Parameters **ax** –

Returns

decomposition_plot(*ax*, *text='Reconstructed'*, *v_min=None*, *v_max=None*, *unconvolved=False*, *point_source_add=False*, *font_size=15*, *source_add=False*, *lens_light_add=False*, ***kwargs*)

Parameters

- **ax** –
- **text** –
- **v_min** –
- **v_max** –
- **unconvolved** –
- **point_source_add** –
- **source_add** –
- **lens_light_add** –
- **kwargs** – kwargs to send matplotlib.pyplot.matshow()

Returns

deflection_plot(*ax*, *v_min=None*, *v_max=None*, *axis=0*, *with_caustics=False*, *image_name_list=None*, *text='Deflection model'*, *font_size=15*, *colorbar_label='arcsec'*)

Returns

error_map_source_plot(*ax*, *numPix*, *deltaPix_source*, *v_min=None*, *v_max=None*, *with_caustics=False*, *font_size=15*, *point_source_position=True*)

plots the uncertainty in the surface brightness in the source from the linear inversion by taking the diagonal elements of the covariance matrix of the inversion of the basis set to be propagated to the source plane. #TODO illustration of the uncertainties in real space with the full covariance matrix is subtle. The best way is probably to draw realizations from the covariance matrix.

Parameters

- **ax** – matplotlib axis instance
- **numPix** – number of pixels in plot per axis
- **deltaPix_source** – pixel spacing in the source resolution illustrated in plot
- **v_min** – minimum plotting scale of the map
- **v_max** – maximum plotting scale of the map
- **with_caustics** – plot the caustics on top of the source reconstruction (may take some time)
- **font_size** – font size of labels
- **point_source_position** – boolean, if True, plots a point at the position of the point source

Returns plot of source surface brightness errors in the reconstruction on the axis instance

magnification_plot(*ax*, *v_min=-10*, *v_max=10*, *image_name_list=None*, *font_size=15*, *no_arrow=False*, *text='Magnification model'*, *colorbar_label='$\\det\\(\\mathsf{A}^{-1})$'*, ***kwargs*)

Parameters

- **ax** – matplotib axis instance
- **v_min** – minimum range of plotting
- **v_max** – maximum range of plotting
- **kwargs** – kwargs to send to matplotlib.pyplot.matshow()

Returns

model_plot(*ax*, *v_min=None*, *v_max=None*, *image_names=False*, *colorbar_label='log$_{10}$ flux'*, *font_size=15*, *text='Reconstructed'*, ***kwargs*)

Parameters

- **ax** – matplotib axis instance
- **v_min** –
- **v_max** –

Returns

normalized_residual_plot(*ax*, *v_min=-6*, *v_max=6*, *font_size=15*, *text='Normalized Residuals'*, *colorbar_label='(f${}_{\\rm model}$ - f${}_{\\rm data}$)/$\\sigma$'*, *no_arrow=False*, *color_bar=True*, ***kwargs*)

Parameters

- **ax** –
- **v_min** –
- **v_max** –
- **kwargs** – kwargs to send to matplotlib.pyplot.matshow()
- **color_bar** – Option to display the color bar

Returns

plot_extinction_map(*ax*, *v_min=None*, *v_max=None*, ***kwargs*)

Parameters

- **ax** –
- **v_min** –
- **v_max** –

Returns

plot_main(*with_caustics=False*)

print the main plots together in a joint frame

Returns

plot_separate()

plot the different model components separately

Returns

plot_subtract_from_data_all()

subtract model components from data

Returns

source(*numPix*, *deltaPix*, *center=None*, *image_orientation=True*)

Parameters

- **numPix** – number of pixels per axes
- **deltaPix** – pixel size
- **image_orientation** – bool, if True, uses frame in orientation of the image, otherwise in RA-DEC coordinates

Returns 2d surface brightness grid of the reconstructed source and Coordinates() instance of source grid

source_plot(*ax*, *numPix*, *deltaPix_source*, *center=None*, *v_min=None*, *v_max=None*, *with_caustics=False*, *caustic_color='yellow'*, *font_size=15*, *plot_scale='log'*, *scale_size=0.1*, *text='Reconstructed source'*, *colorbar_label='log$_{10}$ flux'*, *point_source_position=True*, ***kwargs*)

Parameters

- **ax** –
- **numPix** –

- **deltaPix_source** –
- **center** – [center_x, center_y], if specified, uses this as the center
- **v_min** –
- **v_max** –
- **caustic_color** –
- **font_size** –
- **plot_scale** – string, log or linear, scale of surface brightness plot
- **kwargs** –

Returns

subtract_from_data_plot (*ax, text='Subtracted', v_min=None, v_max=None, point_source_add=False, source_add=False, lens_light_add=False, font_size=15*)

lenstronomy.Plots.model_plot module

class ModelPlot (*multi_band_list, kwargs_model, kwargs_params, image_likelihood_mask_list=None, bands_compute=None, multi_band_type='multi-linear', source_marg=False, linear_prior=None, arrow_size=0.02, cmap_string='gist_heat', fast_caustic=True, linear_solver=True*)

Bases: `object`

class that manages the summary plots of a lens model The class uses the same conventions as being used in the FittingSequence and interfaces with the ImSim module. The linear inversion is re-done given the likelihood settings in the init of this class (make sure this is the same as you perform the FittingSequence) to make sure the linear amplitude parameters are computed as they are not part of the output of the FittingSequence results.

__init__ (*multi_band_list, kwargs_model, kwargs_params, image_likelihood_mask_list=None, bands_compute=None, multi_band_type='multi-linear', source_marg=False, linear_prior=None, arrow_size=0.02, cmap_string='gist_heat', fast_caustic=True, linear_solver=True*)

Parameters

- **multi_band_list** – list of [[kwargs_data, kwargs_psf, kwargs_numerics], [], ..]
- **multi_band_type** – string, option when having multiple imaging data sets modelled simultaneously. Options are: - 'multi-linear': linear amplitudes are inferred on single data set - 'linear-joint': linear amplitudes ae jointly inferred - 'single-band': single band
- **kwargs_model** – model keyword arguments
- **bands_compute** – (optional), bool list to indicate which band to be included in the modeling
- **image_likelihood_mask_list** – list of image likelihood mask (same size as image_data with 1 indicating being evaluated and 0 being left out)
- **kwargs_params** – keyword arguments of 'kwargs_lens', 'kwargs_source' etc as coming as kwargs_result from FittingSequence class
- **source_marg** –
- **linear_prior** –
- **arrow_size** –

- **cmap_string** –
- **fast_caustic** – boolean; if True, uses fast (but less accurate) caustic calculation method
- **linear_solver** – bool, if True (default) fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

absolute_residual_plot(*band_index=0*, ***kwargs*)
illustrates absolute residuals between data and model fit

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

convergence_plot(*band_index=0*, ***kwargs*)
illustrates lensing convergence in data frame

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

data_plot(*band_index=0*, ***kwargs*)
illustrates data

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

decomposition_plot(*band_index=0*, ***kwargs*)
illustrates decomposition of model components

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

deflection_plot(*band_index=0*, ***kwargs*)
illustrates lensing deflections on the field of view of the data frame

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

error_map_source_plot(*band_index=0*, ***kwargs*)
illustrates surface brightness variance in the reconstruction in the source plane

Parameters

- **band_index** – index of band

- **kwargs** – arguments of plotting

Returns plot instance

magnification_plot (*band_index=0*, ***kwargs*)
illustrates lensing magnification in the field of view of the data frame

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

model_plot (*band_index=0*, ***kwargs*)
illustrates model

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

normalized_residual_plot (*band_index=0*, ***kwargs*)
illustrates normalized residuals between data and model fit

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

plot_extinction_map (*band_index=0*, ***kwargs*)

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance of differential extinction map

plot_main (*band_index=0*, ***kwargs*)
plot a set of 'main' modelling diagnostics

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

plot_separate (*band_index=0*)
plot a set of 'main' modelling diagnostics

Parameters **band_index** – index of band

Returns plot instance

plot_subtract_from_data_all (*band_index=0*)
plot a set of 'main' modelling diagnostics

Parameters **band_index** – index of band

Returns plot instance

reconstruction_all_bands(***kwargs*)

Parameters **kwargs** – arguments of plotting

Returns 3 x n_data plot with data, model, reduced residual plots of all the images/bands that are being modeled

source(*band_index=0*, ***kwargs*)

Parameters

- **band_index** – index of band
- **kwargs** – keyword arguments accessible in model_band_plot.source()

Returns 2d array of source surface brightness

source_plot(*band_index=0*, ***kwargs*)
illustrates reconstructed source (de-lensed de-convolved)

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

subtract_from_data_plot(*band_index=0*, ***kwargs*)
subtracts individual model components from the data

Parameters

- **band_index** – index of band
- **kwargs** – arguments of plotting

Returns plot instance

lenstronomy.Plots.plot_util module

sqrt(*inputArray*, *scale_min=None*, *scale_max=None*)
Performs sqrt scaling of the input numpy array.

Parameters

- **inputArray** (`numpy array`) – image data array
- **scale_min** (`float`) – minimum data value
- **scale_max** (`float`) – maximum data value

Return type numpy array

Returns image data array

text_description(*ax*, *d*, *text*, *color='w'*, *backgroundcolor='k'*, *flipped=False*, *font_size=15*)

scale_bar(*ax*, *d*, *dist=1.0*, *text='1"'*, *color='w'*, *font_size=15*, *flipped=False*)

Parameters

- **ax** – matplotlib.axes instance
- **d** – diameter of frame

- **dist** – distance scale printed
- **text** – string printed on scale bar
- **color** – color of scale bar
- **font_size** – font size
- **flipped** – boolean

Returns None, updated ax instance

coordinate_arrows (*ax*, *d*, *coords*, *color='w'*, *font_size=15*, *arrow_size=0.05*)

Parameters

- **ax** – matplotlib axes instance
- **d** – diameter of frame in ax
- **coords** – lenstronomy.Data.coord_transforms Coordinates() instance
- **color** – color string
- **font_size** – font size of length scale
- **arrow_size** – size of arrow

Returns updated ax instance

plot_line_set (*ax*, *coords*, *line_set_list_x*, *line_set_list_y*, *origin=None*, *flipped_x=False*, *points_only=False*, *pixel_offset=True*, **args*, ***kwargs*)

plotting a line set on a matplotlib instance where the coordinates are defined in pixel units with the lower left corner (defined as origin) is by default (0, 0). The coordinates are moved by 0.5 pixels to be placed in the center of the pixel in accordance with the matplotlib.matshow() routine.

Parameters

- **ax** – matplotlib.axis instance
- **coords** – Coordinates() class instance
- **origin** – [x0, y0], lower left pixel coordinate in the frame of the pixels
- **line_set_list_x** – numpy arrays corresponding of different disconnected regions of the line (e.g. caustic or critical curve)
- **line_set_list_y** – numpy arrays corresponding of different disconnected regions of the line (e.g. caustic or critical curve)
- **color** – string with matplotlib color
- **flipped_x** – bool, if True, flips x-axis
- **points_only** – bool, if True, sets plotting keywords to plot single points without connecting lines
- **pixel_offset** – boolean; if True (default plotting), the coordinates are shifted a half a pixel to match with the matshow() command to center the coordinates in the pixel center

Returns plot with line sets on matplotlib axis in pixel coordinates

image_position_plot (*ax*, *coords*, *ra_image*, *dec_image*, *color='w'*, *image_name_list=None*, *origin=None*, *flipped_x=False*, *pixel_offset=True*)

Parameters

- **ax** – matplotlib axis instance

- **coords** – Coordinates() class instance or inherited class (such as PixelGrid(), or Data())
- **ra_image** – Ra/x-coordinates of image positions (list of arrays in angular units)
- **dec_image** – Dec/y-coordinates of image positions (list of arrays in angular units)
- **color** – color of ticks and text
- **image_name_list** – list of strings for names of the images in the same order as the positions
- **origin** – [x0, y0], lower left pixel coordinate in the frame of the pixels
- **flipped_x** – bool, if True, flips x-axis
- **pixel_offset** – boolean; if True (default plotting), the coordinates are shifted a half a pixel to match with the matshow() command to center the coordinates in the pixel center

Returns matplotlib axis instance with images plotted on

source_position_plot (*ax*, *coords*, *ra_source*, *dec_source*, *marker='*'*, *markersize=10*, ***kwargs*)

Parameters

- **ax** – matplotlib axis instance
- **coords** – Coordinates() class instance or inherited class (such as PixelGrid(), or Data())
- **ra_source** – list of source position in angular units
- **dec_source** – list of source position in angular units
- **marker** – marker style for matplotlib
- **markersize** – marker size for matplotlib

Returns matplotlib axis instance with images plotted on

result_string (*x*, *weights=None*, *title_fmt='.2f'*, *label=None*)

Parameters

- **x** – marginalized 1-d posterior
- **weights** – weights of posteriors (optional)
- **title_fmt** – format to what digit the results are presented
- **label** – string of parameter label (optional)

Returns string with mean $\pm$ quartile

cmap_conf (*cmap_string*)

configures matplotlib color map

Parameters **cmap_string** – string of cmap name, or cmap instance

Returns cmap instance with setting for bad pixels and values below the threshold

Module contents

lenstronomy.PointSource package

Subpackages

lenstronomy.PointSource.Types package

Submodules

lenstronomy.PointSource.Types.base_ps module

class PSBase(*lens_model=None*, *fixed_magnification=False*, *additional_image=False*)
Bases: `object`

base point source type class

__init__(*lens_model=None*, *fixed_magnification=False*, *additional_image=False*)

Parameters

- **lens_model** – instance of the LensModel() class
- **fixed_magnification** – bool. If True, magnification ratio of point sources is fixed to the one given by the lens model
- **additional_image** – bool. If True, search for additional images of the same source is conducted.

image_amplitude(*kwargs_ps*, **args*, ***kwargs*)
amplitudes as observed on the sky

Parameters

- **kwargs_ps** – keyword argument of point source model
- **kwargs** – keyword arguments of function call

Returns numpy array of amplitudes

image_position(*kwargs_ps*, ***kwargs*)
on-sky position

Parameters **kwargs_ps** – keyword argument of point source model

Returns numpy array of x, y image positions

source_amplitude(*kwargs_ps*, ***kwargs*)
intrinsic source amplitudes (without lensing magnification, but still apparent)

Parameters

- **kwargs_ps** – keyword argument of point source model
- **kwargs** – keyword arguments of function call (which are not used for this object

Returns numpy array of amplitudes

source_position(*kwargs_ps*, ***kwargs*)
original unlensed position

Parameters **kwargs_ps** – keyword argument of point source model

Returns numpy array of x, y source positions

update_lens_model(*lens_model_class*)
update LensModel() and LensEquationSolver() instance

Parameters **lens_model_class** – LensModel() class instance

Returns internal lensModel class updated

lenstronomy.PointSource.Types.lensed_position module

class LensedPositions(*lens_model=None*, *fixed_magnification=False*, *additional_image=False*)

Bases: `lenstronomy.PointSource.Types.base_ps.PSBase`

class of a a lensed point source parameterized as the (multiple) observed image positions Name within the PointSource module: 'LENSED_POSITION' parameters: ra_image, dec_image, point_amp If fixed_magnification=True, than 'source_amp' is a parameter instead of 'point_amp'

image_amplitude(*kwargs_ps*, *kwargs_lens=None*, *x_pos=None*, *y_pos=None*, *magnification_limit=None*, *kwargs_lens_eqn_solver=None*)

image brightness amplitudes

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only used when requiring the lens equation solver
- **x_pos** – pre-computed image position (no lens equation solver applied)
- **y_pos** – pre-computed image position (no lens equation solver applied)
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details

Returns array of image amplitudes

image_position(*kwargs_ps*, *kwargs_lens=None*, *magnification_limit=None*, *kwargs_lens_eqn_solver=None*)

on-sky image positions

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only used when requiring the lens equation solver
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details

Returns image positions in x, y as arrays

source_amplitude(*kwargs_ps*, *kwargs_lens=None*)

intrinsic brightness amplitude of point source When brightnesses are defined in magnified on-sky positions, the intrinsic brightness is computed as the mean in the magnification corrected image position brightnesses.

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), used when brightness are defined in magnified on-sky positions

Returns brightness amplitude (as numpy array)

source_position (*kwargs_ps*, *kwargs_lens=None*)
original source position (prior to lensing)

Parameters

- **kwargs_ps** – point source keyword arguments
- **kwargs_lens** – lens model keyword argument list (required to ray-trace back in the source plane)

Returns x, y position (as numpy arrays)

lenstronomy.PointSource.Types.source_position module

class SourcePositions (*lens_model=None*, *fixed_magnification=False*, *additional_image=False*)
Bases: *lenstronomy.PointSource.Types.base_ps.PSBase*

class of a single point source defined in the original source coordinate position that is lensed. The lens equation is solved to compute the image positions for the specified source position.

Name within the PointSource module: 'SOURCE_POSITION' parameters: ra_source, dec_source, source_amp, mag_pert (optional) If fixed_magnification=True, than 'source_amp' is a parameter instead of 'point_amp' mag_pert is a list of fractional magnification pertubations applied to point source images

image_amplitude (*kwargs_ps*, *kwargs_lens=None*, *x_pos=None*, *y_pos=None*, *magnification_limit=None*, *kwargs_lens_eqn_solver=None*)
image brightness amplitudes

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only ignored when providing image positions directly
- **x_pos** – pre-computed image position (no lens equation solver applied)
- **y_pos** – pre-computed image position (no lens equation solver applied)
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details

Returns array of image amplitudes

image_position (*kwargs_ps*, *kwargs_lens=None*, *magnification_limit=None*, *kwargs_lens_eqn_solver=None*)
on-sky image positions

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only used when requiring the lens equation solver
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit

- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details

Returns image positions in x, y as arrays

source_amplitude (*kwargs_ps*, *kwargs_lens=None*)
intrinsic brightness amplitude of point source When brightnesses are defined in magnified on-sky positions, the intrinsic brightness is computed as the mean in the magnification corrected image position brightnesses.

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), used when brightness are defined in magnified on-sky positions

Returns brightness amplitude (as numpy array)

source_position (*kwargs_ps*, ***kwargs*)
original source position (prior to lensing)

Parameters **kwargs_ps** – point source keyword arguments

Returns x, y position (as numpy arrays)

lenstronomy.PointSource.Types.unlensed module

class Unlensed (*lens_model=None*, *fixed_magnification=False*, *additional_image=False*)
Bases: *lenstronomy.PointSource.Types.base_ps.PSBase*

class of a single point source in the image plane, aka star Name within the PointSource module: 'UNLENSED' This model can deal with arrays of point sources. parameters: ra_image, dec_image, point_amp

image_amplitude (*kwargs_ps*, ***kwargs*)
amplitudes as observed on the sky

Parameters

- **kwargs_ps** – keyword argument of point source model
- **kwargs** – keyword arguments of function call (which are not used for this object

Returns numpy array of amplitudes

image_position (*kwargs_ps*, ***kwargs*)
on-sky position

Parameters **kwargs_ps** – keyword argument of point source model

Returns numpy array of x, y image positions

source_amplitude (*kwargs_ps*, ***kwargs*)
intrinsic source amplitudes

Parameters

- **kwargs_ps** – keyword argument of point source model
- **kwargs** – keyword arguments of function call (which are not used for this object

Returns numpy array of amplitudes

source_position (*kwargs_ps*, ***kwargs*)
original physical position (identical for this object)

Parameters **kwargs_ps** – keyword argument of point source model

Returns numpy array of x, y source positions

Module contents

Submodules

lenstronomy.PointSource.point_source module

class PointSource(*point_source_type_list, lensModel=None, fixed_magnification_list=None, additional_images_list=None, flux_from_point_source_list=None, magnification_limit=None, save_cache=False, kwargs_lens_eqn_solver=None*)

Bases: `object`

__init__(*point_source_type_list, lensModel=None, fixed_magnification_list=None, additional_images_list=None, flux_from_point_source_list=None, magnification_limit=None, save_cache=False, kwargs_lens_eqn_solver=None*)

Parameters

- **point_source_type_list** – list of point source types
- **lensModel** – instance of the LensModel() class
- **fixed_magnification_list** – list of booleans (same length as point_source_type_list). If True, magnification ratio of point sources is fixed to the one given by the lens model. This option then requires to provide a 'source_amp' amplitude of the source brightness instead of 'point_amp' the list of image brightnesses.
- **additional_images_list** – list of booleans (same length as point_source_type_list). If True, search for additional images of the same source is conducted.
- **flux_from_point_source_list** – list of booleans (optional), if set, will only return image positions (for imaging modeling) for the subset of the point source lists that =True. This option enables to model imaging data with transient point sources, when the point source positions are measured and present at a different time than the imaging data
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit
- **save_cache** – bool, saves image positions and only if delete_cache is executed, a new solution of the lens equation is conducted with the lens model parameters provided. This can increase the speed as multiple times the image positions are requested for the same lens model. Attention in usage!
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details ,such as: min_distance=0.01, search_window=5, precision_limit=10**(-10), num_iter_max=100

check_image_positions(*kwargs_ps, kwargs_lens, tolerance=0.001*)

checks whether the point sources in kwargs_ps satisfy the lens equation with a tolerance (computed by ray-tracing in the source plane)

Parameters

- **kwargs_ps** – point source keyword argument list

- **kwargs_lens** – lens model keyword argument list
- **tolerance** – Eucledian distance between the source positions ray-traced backwards to be tolerated

Returns bool: True, if requirement on tolerance is fulfilled, False if not.

classmethod check_positive_flux(*kwargs_ps*)
check whether inferred linear parameters are positive

Parameters **kwargs_ps** – point source keyword argument list

Returns bool, True, if all 'point_amp' parameters are positive semi-definite

delete_lens_model_cache()
deletes the variables saved for a specific lens model

Returns None

image_amplitude(*kwargs_ps*, *kwargs_lens*, *k=None*)
returns the image amplitudes

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list
- **k** – None, int or list of int's to select a subset of the point source models in the return

Returns list of image amplitudes per model component

image_position(*kwargs_ps*, *kwargs_lens*, *k=None*, *original_position=False*)
image positions as observed on the sky of the point sources

Parameters

- **kwargs_ps** – point source parameter keyword argument list
- **kwargs_lens** – lens model keyword argument list
- **k** – None, int or boolean list; only returns a subset of the model predictions
- **original_position** – boolean (only applies to 'LENSED_POSITION' models), returns the image positions in the model parameters and does not re-compute images (which might be differently ordered) in case of the lens equation solver

Returns list of: list of image positions per point source model component

linear_param_from_kwargs(*kwargs_list*)
inverse function of update_linear() returning the linear amplitude list for the keyword argument list

Parameters **kwargs_list** (`list of keyword arguments`) – model parameters including the linear amplitude parameters

Returns list of linear amplitude parameters

Return type list

linear_response_set(*kwargs_ps*, *kwargs_lens=None*, *with_amp=False*)

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list

- **with_amp** – bool, if True returns the image amplitude derived from kwargs_ps, otherwise the magnification of the lens model

Returns ra_pos, dec_pos, amp, n

num_basis (*kwargs_ps*, *kwargs_lens*)
number of basis functions for linear inversion

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list

Returns int

point_source_list (*kwargs_ps*, *kwargs_lens*, *k=None*, *with_amp=True*)
returns the coordinates and amplitudes of all point sources in a single array

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list
- **k** – None, int or list of int's to select a subset of the point source models in the return
- **with_amp** – bool, if False, ignores the amplitude parameters in the return and instead provides ones for each point source image

Returns ra_array, dec_array, amp_array

set_amplitudes (*amp_list*, *kwargs_ps*)
translates the amplitude parameters into the convention of the keyword argument list currently only used in SimAPI to transform magnitudes to amplitudes in the lenstronomy conventions

Parameters

- **amp_list** – list of model amplitudes for each point source model
- **kwargs_ps** – list of point source keywords

Returns overwrites kwargs_ps with new amplitudes

set_save_cache (*save_cache*)
set the save cache boolean to new value

Parameters **save_cache** – bool, if True, saves (or uses a previously saved) values

Returns updated class and sub-class instances to either save or not save the point source information in cache

source_amplitude (*kwargs_ps*, *kwargs_lens*)
intrinsic (unlensed) point source amplitudes

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list

Returns list of intrinsic (unlensed) point source amplitudes

source_position (*kwargs_ps*, *kwargs_lens*)
intrinsic source positions of the point sources

Parameters

- **kwargs_ps** – keyword argument list of point source models
- **kwargs_lens** – keyword argument list of lens models

Returns list of source positions for each point source model

update_lens_model(*lens_model_class*)

Parameters **lens_model_class** – instance of LensModel class

Returns update instance of lens model class

update_linear(*param*, *i*, *kwargs_ps*, *kwargs_lens*)

Parameters

- **param** – list of floats corresponding ot the parameters being sampled
- **i** – index of the first parameter relevant for this class
- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list

Returns kwargs_ps with updated linear parameters, index of the next parameter relevant for another class

update_search_window(*search_window*, *x_center*, *y_center*, *min_distance=None*, *only_from_unspecified=False*)

update the search area for the lens equation solver

Parameters

- **search_window** – search_window: window size of the image position search with the lens equation solver.
- **x_center** – center of search window
- **y_center** – center of search window
- **min_distance** – minimum search distance
- **only_from_unspecified** – bool, if True, only sets keywords that previously have not been set

Returns updated self instances

lenstronomy.PointSource.point_source_cached module

class PointSourceCached(*point_source_model*, *save_cache=False*)

Bases: `object`

This class is the same as PointSource() except that it saves image and source positions in cache. This speeds-up repeated calls for the same source and lens model and avoids duplicating the lens equation solving. Attention: cache needs to be deleted before calling functions with different lens and point source parameters.

__init__(*point_source_model*, *save_cache=False*)

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

delete_lens_model_cache()

image_amplitude(*kwargs_ps*, *kwargs_lens=None*, *magnification_limit=None*, *kwargs_lens_eqn_solver=None*)

image brightness amplitudes

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only used when requiring the lens equation solver
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details

Returns array of image amplitudes

image_position(*kwargs_ps*, *kwargs_lens=None*, *magnification_limit=None*, *kwargs_lens_eqn_solver=None*)
on-sky image positions

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only used when requiring the lens equation solver
- **magnification_limit** – float >0 or None, if float is set and additional images are computed, only those images will be computed that exceed the lensing magnification (absolute value) limit
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details

Returns image positions in x, y as arrays

set_save_cache(*save_bool*)

source_amplitude(*kwargs_ps*, *kwargs_lens=None*)
intrinsic brightness amplitude of point source

Parameters

- **kwargs_ps** – keyword arguments of the point source model
- **kwargs_lens** – keyword argument list of the lens model(s), only used when positions are defined in image plane and have to be ray-traced back

Returns brightness amplitude (as numpy array)

source_position(*kwargs_ps*, *kwargs_lens=None*)
original source position (prior to lensing)

Parameters

- **kwargs_ps** – point source keyword arguments
- **kwargs_lens** – lens model keyword argument list (only used when required)

Returns x, y position

update_lens_model(*lens_model_class*)

lenstronomy.PointSource.point_source_param module

class PointSourceParam(*model_list*, *kwargs_fixed*, *num_point_source_list=None*, *linear_solver=True*, *fixed_magnification_list=None*, *kwargs_lower=None*, *kwargs_upper=None*)

Bases: `object`

Point source parameters

__init__(*model_list*, *kwargs_fixed*, *num_point_source_list=None*, *linear_solver=True*, *fixed_magnification_list=None*, *kwargs_lower=None*, *kwargs_upper=None*)

Parameters

- **model_list** – list of point source model names
- **kwargs_fixed** – list of keyword arguments with parameters to be held fixed
- **num_point_source_list** – list of number of point sources per point source model class
- **linear_solver** – bool, if True, does not return linear parameters for the sampler (will be solved linearly instead)
- **fixed_magnification_list** – list of booleans, if entry is True, keeps one overall scaling among the point sources in this class

add_fix_linear(*kwargs_fixed*)

updates fixed keyword argument list with linear parameters

Parameters **kwargs_fixed** – list of keyword arguments held fixed during sampling

Returns updated keyword argument list

get_params(*args*, *i*)

Parameters

- **args** – sorted list of floats corresponding to the parameters being sampled
- **i** – int, index of first entry relevant for being managed by this class

Returns keyword argument list of point sources, index relevant for the next class

num_param()

number of parameters and their names

Returns int, list of parameter names

num_param_linear()

Returns number of linear parameters

set_params(*kwargs_list*)

Parameters **kwargs_list** – keyword argument list

Returns sorted list of parameters being sampled extracted from kwargs_list

Module contents

lenstronomy.Sampling package

Subpackages

lenstronomy.Sampling.Likelihoods package

Submodules

lenstronomy.Sampling.Likelihoods.image_likelihood module

class ImageLikelihood(*multi_band_list*, *multi_band_type*, *kwargs_model*, *bands_compute=None*, *image_likelihood_mask_list=None*, *source_marg=False*, *linear_prior=None*, *check_positive_flux=False*, *kwargs_pixelbased=None*, *linear_solver=True*)

Bases: `object`

manages imaging data likelihoods

__init__(*multi_band_list*, *multi_band_type*, *kwargs_model*, *bands_compute=None*, *image_likelihood_mask_list=None*, *source_marg=False*, *linear_prior=None*, *check_positive_flux=False*, *kwargs_pixelbased=None*, *linear_solver=True*)

Parameters

- **bands_compute** – list of bools with same length as data objects, indicates which "band" to include in the fitting
- **image_likelihood_mask_list** – list of boolean 2d arrays of size of images marking the pixels to be evaluated in the likelihood
- **source_marg** – marginalization addition on the imaging likelihood based on the covariance of the inferred linear coefficients
- **linear_prior** – float or list of floats (when multi-linear setting is chosen) indicating the range of linear amplitude priors when computing the marginalization term.
- **check_positive_flux** – bool, option to punish models that do not have all positive linear amplitude parameters
- **kwargs_pixelbased** – keyword arguments with various settings related to the pixel-based solver (see SLITronomy documentation)
- **linear_solver** – bool, if True (default) fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

logL(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_special=None*, *kwargs_extinction=None*)

Parameters

- **kwargs_lens** – lens model keyword argument list according to LensModel module
- **kwargs_source** – source light keyword argument list according to LightModel module
- **kwargs_lens_light** – deflector light (not lensed) keyword argument list according to LightModel module
- **kwargs_ps** – point source keyword argument list according to PointSource module
- **kwargs_special** – special keyword argument list as part of the Param module
- **kwargs_extinction** – extinction parameter keyword argument list according to LightModel module

Returns log likelihood of the data given the model

num_data

Returns number of image data points

num_param_linear (*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_special=None*, *kwargs_extinction=None*)

Returns number of linear parameters solved for during the image reconstruction process

reset_point_source_cache (*cache=True*)

Parameters **cache** – boolean

Returns None

lenstronomy.Sampling.Likelihoods.position_likelihood module

class PositionLikelihood (*point_source_class*, *image_position_uncertainty=0.005*, *astrometric_likelihood=False*, *image_position_likelihood=False*, *ra_image_list=None*, *dec_image_list=None*, *source_position_likelihood=False*, *check_matched_source_position=False*, *source_position_tolerance=0.001*, *source_position_sigma=0.001*, *force_no_add_image=False*, *restrict_image_number=False*, *max_num_images=None*)

Bases: `object`

likelihood of positions of multiply imaged point sources

__init__ (*point_source_class*, *image_position_uncertainty=0.005*, *astrometric_likelihood=False*, *image_position_likelihood=False*, *ra_image_list=None*, *dec_image_list=None*, *source_position_likelihood=False*, *check_matched_source_position=False*, *source_position_tolerance=0.001*, *source_position_sigma=0.001*, *force_no_add_image=False*, *restrict_image_number=False*, *max_num_images=None*)

Parameters

- **point_source_class** – Instance of PointSource() class
- **image_position_uncertainty** – uncertainty in image position uncertainty (1-sigma Gaussian radially), this is applicable for astrometric uncertainties as well as if image positions are provided as data
- **astrometric_likelihood** – bool, if True, evaluates the astrometric uncertainty of the predicted and modeled image positions with an offset 'delta_x_image' and 'delta_y_image'
- **image_position_likelihood** – bool, if True, evaluates the likelihood of the model predicted image position given the data/measured image positions
- **ra_image_list** – list or RA image positions per model component
- **dec_image_list** – list or DEC image positions per model component
- **source_position_likelihood** – bool, if True, ray-traces image positions back to source plane and evaluates relative errors in respect ot the position_uncertainties in the image plane (image_position_uncertainty)
- **check_matched_source_position** – bool, if True, checks whether multiple images are a solution of the same source
- **source_position_tolerance** – tolerance level (in arc seconds in the source plane) of the different images

- **source_position_sigma** – r.m.s. value corresponding to a 1-sigma Gaussian likelihood accepted by the model precision in matching the source position
- **force_no_add_image** – bool, if True, will punish additional images appearing in the frame of the modelled image(first calculate them)
- **restrict_image_number** – bool, if True, searches for all appearing images in the frame of the data and compares with max_num_images
- **max_num_images** – integer, maximum number of appearing images. Default is the number of images given in the Param() class

static astrometric_likelihood(*kwargs_ps*, *kwargs_special*, *sigma*)

evaluates the astrometric uncertainty of the model plotted point sources (only available for 'LENSED_POSITION' point source model) and predicted image position by the lens model including an astrometric correction term.

Parameters

- **kwargs_ps** – point source model kwargs list
- **kwargs_special** – kwargs list, should include the astrometric corrections 'delta_x', 'delta_y'
- **sigma** – 1-sigma Gaussian uncertainty in the astrometry

Returns log likelihood of the astrometirc correction between predicted image positions and model placement of the point sources

check_additional_images(*kwargs_ps*, *kwargs_lens*)

checks whether additional images have been found and placed in kwargs_ps of the first point source model #TODO check for all point source models :param kwargs_ps: point source kwargs :return: bool, True if more image positions are found than originally been assigned

image_position_likelihood(*kwargs_ps*, *kwargs_lens*, *sigma*)

computes the likelihood of the model predicted image position relative to measured image positions with an astrometric error. This routine requires the 'ra_image_list' and 'dec_image_list' being declared in the initiation of the class

Parameters

- **kwargs_ps** – point source keyword argument list
- **kwargs_lens** – lens model keyword argument list
- **sigma** – 1-sigma uncertainty in the measured position of the images

Returns log likelihood of the model predicted image positions given the data/measured image positions.

logL(*kwargs_lens*, *kwargs_ps*, *kwargs_special*, *verbose=False*)

Parameters

- **kwargs_lens** – lens model parameter keyword argument list
- **kwargs_ps** – point source model parameter keyword argument list
- **kwargs_special** – special keyword arguments
- **verbose** – bool

Returns log likelihood of the optional likelihoods being computed

num_data

Returns integer, number of data points associated with the class instance

source_position_likelihood(*kwargs_lens*, *kwargs_ps*, *sigma*, *hard_bound_rms=None*, *verbose=False*)

computes a likelihood/punishing factor of how well the source positions of multiple images match given the image position and a lens model. The likelihood level is computed in respect of a displacement in the image plane and transposed through the Hessian into the source plane.

Parameters

- **kwargs_lens** – lens model keyword argument list
- **kwargs_ps** – point source keyword argument list
- **sigma** – 1-sigma Gaussian uncertainty in the image plane
- **hard_bound_rms** – hard bound deviation between the mapping of the images back to the source plane (in source frame)
- **verbose** – bool, if True provides print statements with useful information.

Returns log likelihood of the model reproducing the correct image positions given an image position uncertainty

lenstronomy.Sampling.Likelihoods.prior_likelihood module

class PriorLikelihood(*prior_lens=None*, *prior_source=None*, *prior_lens_light=None*, *prior_ps=None*, *prior_special=None*, *prior_extinction=None*, *prior_lens_kde=None*, *prior_source_kde=None*, *prior_lens_light_kde=None*, *prior_ps_kde=None*, *prior_special_kde=None*, *prior_extinction_kde=None*, *prior_lens_lognormal=None*, *prior_source_lognormal=None*, *prior_lens_light_lognormal=None*, *prior_ps_lognormal=None*, *prior_special_lognormal=None*, *prior_extinction_lognormal=None*)

Bases: `object`

class containing additional Gaussian priors to be folded into the likelihood

__init__(*prior_lens=None*, *prior_source=None*, *prior_lens_light=None*, *prior_ps=None*, *prior_special=None*, *prior_extinction=None*, *prior_lens_kde=None*, *prior_source_kde=None*, *prior_lens_light_kde=None*, *prior_ps_kde=None*, *prior_special_kde=None*, *prior_extinction_kde=None*, *prior_lens_lognormal=None*, *prior_source_lognormal=None*, *prior_lens_light_lognormal=None*, *prior_ps_lognormal=None*, *prior_special_lognormal=None*, *prior_extinction_lognormal=None*)

Parameters

- **prior_lens** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_source** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_lens_light** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_ps** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_special** – list of [param_name, mean, 1-sigma priors]
- **prior_extinction** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_lens_kde** – list of [index_model, param_name, samples]
- **prior_source_kde** – list of [index_model, param_name, samples]
- **prior_lens_light_kde** – list of [index_model, param_name, samples]

- **prior_ps_kde** – list of [index_model, param_name, samples]
- **prior_special_kde** – list of [param_name, samples]
- **prior_extinction_kde** – list of [index_model, param_name, samples]
- **prior_lens_lognormal** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_source_lognormal** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_lens_light_lognormal** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_ps_lognormal** – list of [index_model, param_name, mean, 1-sigma priors]
- **prior_special_lognormal** – list of [param_name, mean, 1-sigma priors]
- **prior_extinction_lognormal** – list of [index_model, param_name, mean, 1-sigma priors]

logL(*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_special=None*, *kwargs_extinction=None*)

Parameters **kwargs_lens** – lens model parameter list

Returns log likelihood of lens center

lenstronomy.Sampling.Likelihoods.time_delay_likelihood module

class TimeDelayLikelihood(*time_delays_measured*, *time_delays_uncertainties*, *lens_model_class*, *point_source_class*)

Bases: `object`

class to compute the likelihood of a model given a measurement of time delays

__init__(*time_delays_measured*, *time_delays_uncertainties*, *lens_model_class*, *point_source_class*)

Parameters

- **time_delays_measured** – relative time delays (in days) in respect to the first image of the point source
- **time_delays_uncertainties** – time-delay uncertainties in same order as time_delay_measured. Alternatively a full covariance matrix that describes the likelihood.
- **lens_model_class** – instance of the LensModel() class
- **point_source_class** – instance of the PointSource() class, note: the first point source type is the one the time delays are imposed on

logL(*kwargs_lens*, *kwargs_ps*, *kwargs_cosmo*)

routine to compute the log likelihood of the time delay distance :param kwargs_lens: lens model kwargs list :param kwargs_ps: point source kwargs list :param kwargs_cosmo: cosmology and other kwargs :return: log likelihood of the model given the time delay data

num_data

Returns number of time delay measurements

Module contents

lenstronomy.Sampling.Pool package

Submodules

lenstronomy.Sampling.Pool.multiprocessing module

this file is taken from schwimmbad (https://github.com/adrn/schwimmbad) and an explicit fork by Aymeric Galan to replace the multiprocessing with the multiprocess dependence as for multi-threading, multiprocessing is not supporting dill (only pickle) which is required.

The class also extends with a `is_master()` definition

class MultiPool(*processes=None*, *initializer=None*, *initargs=()*, ***kwargs*)

Bases: `multiprocess.pool.Pool`

A modified version of `multiprocessing.pool.Pool` that has better behavior with regard to `KeyboardInterrupts` in the *map()* method. (Original author: Peter K. G. Williams)

__init__(*processes=None*, *initializer=None*, *initargs=()*, ***kwargs*)

Parameters

- **processes** (*int, optional*) – The number of worker processes to use; defaults to the number of CPUs.
- **initializer** (*callable, optional*) – If specified, a callable that will be invoked by each worker process when it starts.
- **initargs** (*iterable, optional*) – Arguments for `initializer`; it will be called as `initializer(*initargs)`.
- **kwargs** – Extra arguments passed to the `multiprocessing.pool.Pool` superclass.

static enabled()

is_master()

is_worker()

map(*func*, *iterable*, *chunksize=None*, *callback=None*)

Equivalent to the built-in `map()` function and `multiprocessing.pool.Pool.map()`, without catching `KeyboardInterrupt`.

Parameters

- **func** (*callable*) – A function or callable object that is executed on each element of the specified `tasks` iterable. This object must be picklable (i.e. it can't be a function scoped within a function or a `lambda` function). This should accept a single positional argument and return a single object.
- **iterable** (*iterable*) – A list or iterable of tasks. Each task can be itself an iterable (e.g., tuple) of values or data to pass in to the worker function.
- **callback** (*callable, optional*) – An optional callback function (or callable) that is called with the result from each worker run and is executed on the master process. This is useful for, e.g., saving results to a file, since the callback is only called on the master thread.

Returns A list of results from the output of each `worker()` call.

wait_timeout = 3600

lenstronomy.Sampling.Pool.pool module

this file is taken from schwimmbad (https://github.com/adrn/schwimmbad) and an explicit fork by Aymeric Galan to replace the multiprocessing with the multiprocess dependence as for multi-threading, multiprocessing is not supporting dill (only pickle) which is required.

Tests show that the MPI mode works with Python 3.7.2 but not with Python 3.7.0 on a specific system due to mpi4py dependencies and configurations.

Contributions by: - Peter K. G. Williams - Júlio Hoffimann Mendes - Dan Foreman-Mackey - Aymeric Galan - Simon Birrer

Implementations of four different types of processing pools:

- MPIPool: An MPI pool.
- MultiPool: A multiprocessing for local parallelization.
- SerialPool: A serial pool, which uses the built-in *map* function

choose_pool(*mpi=False*, *processes=1*, ***kwargs*)

Extends the capabilities of the schwimmbad.choose_pool method.

It handles the *use_dill* parameters in kwargs, that would otherwise raise an error when processes > 1. Any thread in the returned multiprocessing pool (e.g. processes > 1) also default

The requirement of schwimmbad relies on the master branch (as specified in requirements.txt). The 'use_dill' functionality can raise if not following the requirement specified.

Choose between the different pools given options from, e.g., argparse.

Parameters

- **mpi** (`bool, optional`) – Use the MPI processing pool, `MPIPool`. By default, `False`, will use the `SerialPool`.
- **processes** (`int, optional`) – Use the multiprocessing pool, `MultiPool`, with this number of processes. By default, `processes=1`, will use them:class:~*schwimmbad.serial.SerialPool*.
- **kwargs** (`keyword arguments`) – Any additional kwargs are passed in to the pool class initializer selected by the arguments.

Module contents

lenstronomy.Sampling.Samplers package

Submodules

lenstronomy.Sampling.Samplers.base_nested_sampler module

class NestedSampler(*likelihood_module*, *prior_type*, *prior_means*, *prior_sigmas*, *width_scale*, *sigma_scale*)

Bases: `object`

Base class for nested samplers

__init__(*likelihood_module*, *prior_type*, *prior_means*, *prior_sigmas*, *width_scale*, *sigma_scale*)

Parameters

- **likelihood_module** – likelihood_module like in likelihood.py (should be callable)
- **prior_type** – 'uniform' of 'gaussian', for converting the unit hypercube to param cube
- **prior_means** – if prior_type is 'gaussian', mean for each param
- **prior_sigmas** – if prior_type is 'gaussian', std dev for each param
- **width_scale** – scale the widths of the parameters space by this factor
- **sigma_scale** – if prior_type is 'gaussian', scale the gaussian sigma by this factor

log_likelihood(**args*, ***kwargs*)
compute the log-likelihood given list of parameters

Returns log-likelihood (from the likelihood module)

prior(**args*, ***kwargs*)
compute the mapping between the unit cube and parameter cube

Returns hypercube in parameter space

run(*kwargs_run*)
run the nested sampling algorithm

lenstronomy.Sampling.Samplers.dynesty_sampler module

class DynestySampler(*likelihood_module*, *prior_type='uniform'*, *prior_means=None*, *prior_sigmas=None*, *width_scale=1*, *sigma_scale=1*, *bound='multi'*, *sample='auto'*, *use_mpi=False*, *use_pool=None*)

Bases: `lenstronomy.Sampling.Samplers.base_nested_sampler.NestedSampler`

Wrapper for dynamical nested sampling algorithm Dynesty by J. Speagle

paper : https://arxiv.org/abs/1904.02180 doc : https://dynesty.readthedocs.io/

__init__(*likelihood_module*, *prior_type='uniform'*, *prior_means=None*, *prior_sigmas=None*, *width_scale=1*, *sigma_scale=1*, *bound='multi'*, *sample='auto'*, *use_mpi=False*, *use_pool=None*)

Parameters

- **likelihood_module** – likelihood_module like in likelihood.py (should be callable)
- **prior_type** – 'uniform' of 'gaussian', for converting the unit hypercube to param cube
- **prior_means** – if prior_type is 'gaussian', mean for each param
- **prior_sigmas** – if prior_type is 'gaussian', std dev for each param
- **width_scale** – scale the widths of the parameters space by this factor
- **sigma_scale** – if prior_type is 'gaussian', scale the gaussian sigma by this factor
- **bound** – specific to Dynesty, see https://dynesty.readthedocs.io
- **sample** – specific to Dynesty, see https://dynesty.readthedocs.io
- **use_mpi** – Use MPI computing if *True*
- **use_pool** – specific to Dynesty, see https://dynesty.readthedocs.io

log_likelihood(*x*)
compute the log-likelihood given list of parameters

Parameters x – parameter values

Returns log-likelihood (from the likelihood module)

prior(*u*)
compute the mapping between the unit cube and parameter cube

Parameters u – unit hypercube, sampled by the algorithm

Returns hypercube in parameter space

run(*kwargs_run*)
run the Dynesty nested sampler

see https://dynesty.readthedocs.io for content of kwargs_run

Parameters kwargs_run – kwargs directly passed to DynamicNestedSampler.run_nested

Returns samples, means, logZ, logZ_err, logL, results

lenstronomy.Sampling.Samplers.multinest_sampler module

class MultiNestSampler(*likelihood_module, prior_type='uniform', prior_means=None, prior_sigmas=None, width_scale=1, sigma_scale=1, output_dir=None, output_basename='-', remove_output_dir=False, use_mpi=False*)
Bases: *lenstronomy.Sampling.Samplers.base_nested_sampler.NestedSampler*

Wrapper for nested sampling algorithm MultInest by F. Feroz & M. Hobson papers : arXiv:0704.3704, arXiv:0809.3437, arXiv:1306.2144 pymultinest doc : https://johannesbuchner.github.io/PyMultiNest/pymultinest.html

__init__(*likelihood_module, prior_type='uniform', prior_means=None, prior_sigmas=None, width_scale=1, sigma_scale=1, output_dir=None, output_basename='-', remove_output_dir=False, use_mpi=False*)

Parameters

- **likelihood_module** – likelihood_module like in likelihood.py (should be callable)
- **prior_type** – 'uniform' of 'gaussian', for converting the unit hypercube to param cube
- **prior_means** – if prior_type is 'gaussian', mean for each param
- **prior_sigmas** – if prior_type is 'gaussian', std dev for each param
- **width_scale** – scale the widths of the parameters space by this factor
- **sigma_scale** – if prior_type is 'gaussian', scale the gaussian sigma by this factor
- **output_dir** – name of the folder that will contain output files
- **output_basename** – prefix for output files
- **remove_output_dir** – remove the output_dir folder after completion
- **use_mpi** – flag directly passed to MultInest sampler (NOT TESTED)

log_likelihood(*args, ndim, nparams*)
compute the log-likelihood given list of parameters

Parameters

- **args** – parameter values

- **ndim** – number of sampled parameters
- **nparams** – total number of parameters

Returns log-likelihood (from the likelihood module)

prior (*cube*, *ndim*, *nparams*)
compute the mapping between the unit cube and parameter cube (in-place)

Parameters

- **cube** – unit hypercube, sampled by the algorithm
- **ndim** – number of sampled parameters
- **nparams** – total number of parameters

run (*kwargs_run*)
run the MultiNest nested sampler

see https://johannesbuchner.github.io/PyMultiNest/pymultinest.html for content of kwargs_run

Parameters **kwargs_run** – kwargs directly passed to pymultinest.run

Returns samples, means, logZ, logZ_err, logL, stats

lenstronomy.Sampling.Samplers.polychord_sampler module

class DyPolyChordSampler (*likelihood_module*, *prior_type='uniform'*, *prior_means=None*, *prior_sigmas=None*, *width_scale=1*, *sigma_scale=1*, *output_dir=None*, *output_basename='-'*, *resume_dyn_run=False*, *polychord_settings=None*, *remove_output_dir=False*, *use_mpi=False*)
Bases: *lenstronomy.Sampling.Samplers.base_nested_sampler.NestedSampler*

Wrapper for dynamical nested sampling algorithm DyPolyChord by E. Higson, M. Hobson, W. Handley, A. Lasenby

papers : arXiv:1704.03459, arXiv:1804.06406 doc : https://dypolychord.readthedocs.io

__init__ (*likelihood_module*, *prior_type='uniform'*, *prior_means=None*, *prior_sigmas=None*, *width_scale=1*, *sigma_scale=1*, *output_dir=None*, *output_basename='-'*, *resume_dyn_run=False*, *polychord_settings=None*, *remove_output_dir=False*, *use_mpi=False*)

Parameters

- **likelihood_module** – likelihood_module like in likelihood.py (should be callable)
- **prior_type** – 'uniform' of 'gaussian', for converting the unit hypercube to param cube
- **prior_means** – if prior_type is 'gaussian', mean for each param
- **prior_sigmas** – if prior_type is 'gaussian', std dev for each param
- **width_scale** – scale the widths of the parameters space by this factor
- **sigma_scale** – if prior_type is 'gaussian', scale the gaussian sigma by this factor
- **output_dir** – name of the folder that will contain output files
- **output_basename** – prefix for output files
- **resume_dyn_run** – if True, previous resume files will not be deleted so that previous run can be resumed

- **polychord_settings** – settings dictionary to send to pypolychord. Check dypolychord documentation for details.
- **remove_output_dir** – remove the output_dir folder after completion
- **use_mpi** – Use MPI computing if *True*

log_likelihood(*args*)
compute the log-likelihood given list of parameters

Parameters **args** – parameter values

Returns log-likelihood (from the likelihood module)

prior(*cube*)
compute the mapping between the unit cube and parameter cube

'copy=True' below because cube can not be modified in-place (read-only)

Parameters **cube** – unit hypercube, sampled by the algorithm

Returns hypercube in parameter space

run(*dynamic_goal*, *kwargs_run*)
run the DyPolyChord dynamical nested sampler

see https://dypolychord.readthedocs.io for content of kwargs_run

Parameters

- **dynamic_goal** – 0 for evidence computation, 1 for posterior computation
- **kwargs_run** – kwargs directly passed to dyPolyChord.run_dypolychord

Returns samples, means, logZ, logZ_err, logL, ns_run

Module contents

Submodules

lenstronomy.Sampling.likelihood module

class LikelihoodModule(*kwargs_data_joint, kwargs_model, param_class, image_likelihood=True, check_bounds=True, check_matched_source_position=False, astrometric_likelihood=False, image_position_likelihood=False, source_position_likelihood=False, image_position_uncertainty=0.004, check_positive_flux=False, source_position_tolerance=0.001, source_position_sigma=0.001, force_no_add_image=False, source_marg=False, linear_prior=None, restrict_image_number=False, max_num_images=None, bands_compute=None, time_delay_likelihood=False, image_likelihood_mask_list=None, flux_ratio_likelihood=False, kwargs_flux_compute=None, prior_lens=None, prior_source=None, prior_extinction=None, prior_lens_light=None, prior_ps=None, prior_special=None, prior_lens_kde=None, prior_source_kde=None, prior_lens_light_kde=None, prior_ps_kde=None, prior_special_kde=None, prior_extinction_kde=None, prior_lens_lognormal=None, prior_source_lognormal=None, prior_extinction_lognormal=None, prior_lens_light_lognormal=None, prior_ps_lognormal=None, prior_special_lognormal=None, custom_logL_addition=None, kwargs_pixelbased=None*)

Bases: `object`

this class contains the routines to run a MCMC process the key components are: - imSim_class: an instance of a class that simulates one (or more) images and returns the likelihood, such as ImageModel(), Multiband(), MultiExposure() - param_class: instance of a Param() class that can cast the sorted list of parameters that are sampled into the conventions of the imSim_class

Additional arguments are supported for adding a time-delay likelihood etc (see __init__ definition)

__init__(*kwargs_data_joint, kwargs_model, param_class, image_likelihood=True, check_bounds=True, check_matched_source_position=False, astrometric_likelihood=False, image_position_likelihood=False, source_position_likelihood=False, image_position_uncertainty=0.004, check_positive_flux=False, source_position_tolerance=0.001, source_position_sigma=0.001, force_no_add_image=False, source_marg=False, linear_prior=None, restrict_image_number=False, max_num_images=None, bands_compute=None, time_delay_likelihood=False, image_likelihood_mask_list=None, flux_ratio_likelihood=False, kwargs_flux_compute=None, prior_lens=None, prior_source=None, prior_extinction=None, prior_lens_light=None, prior_ps=None, prior_special=None, prior_lens_kde=None, prior_source_kde=None, prior_lens_light_kde=None, prior_ps_kde=None, prior_special_kde=None, prior_extinction_kde=None, prior_lens_lognormal=None, prior_source_lognormal=None, prior_extinction_lognormal=None, prior_lens_light_lognormal=None, prior_ps_lognormal=None, prior_special_lognormal=None, custom_logL_addition=None, kwargs_pixelbased=None*)

initializing class

Parameters

- **param_class** – instance of a Param() class that can cast the sorted list of parameters that are sampled into the conventions of the imSim_class
- **image_likelihood** – bool, option to compute the imaging likelihood
- **source_position_likelihood** – bool, if True, ray-traces image positions back to source plane and evaluates relative errors in respect ot the position_uncertainties in the

image plane

- **check_bounds** – bool, option to punish the hard bounds in parameter space
- **check_matched_source_position** – bool, option to check whether point source position of solver finds a solution to match all the image positions in the same source plane coordinate
- **astrometric_likelihood** – bool, additional likelihood term of the predicted vs modelled point source position
- **image_position_uncertainty** – float, 1-sigma Gaussian uncertainty on the point source position (only used if point_source_likelihood=True)
- **check_positive_flux** – bool, option to punish models that do not have all positive linear amplitude parameters
- **source_position_tolerance** – float, punishment of check_solver occurs when image positions are predicted further away than this number
- **image_likelihood_mask_list** – list of boolean 2d arrays of size of images marking the pixels to be evaluated in the likelihood
- **force_no_add_image** – bool, if True: computes ALL image positions of the point source. If there are more images predicted than modelled, a punishment occurs
- **source_marg** – marginalization addition on the imaging likelihood based on the covariance of the inferred linear coefficients
- **linear_prior** – float or list of floats (when multi-linear setting is chosen) indicating the range of linear amplitude priors when computing the marginalization term.
- **restrict_image_number** – bool, if True: computes ALL image positions of the point source. If there are more images predicted than indicated in max_num_images, a punishment occurs
- **max_num_images** – int, see restrict_image_number
- **bands_compute** – list of bools with same length as data objects, indicates which "band" to include in the fitting
- **time_delay_likelihood** – bool, if True computes the time-delay likelihood of the FIRST point source
- **kwargs_flux_compute** – keyword arguments of how to compute the image position fluxes (see FluxRatioLikeliood)
- **custom_logL_addition** – a definition taking as arguments (kwargs_lens, kwargs_source, kwargs_lens_light, kwargs_ps, kwargs_special, kwargs_extinction) and returns a logL (punishing) value.
- **kwargs_pixelbased** – keyword arguments with various settings related to the pixel-based solver (see SLITronomy documentation)

static check_bounds(*args*, *lowerLimit*, *upperLimit*, *verbose=False*)
checks whether the parameter vector has left its bound, if so, adds a big number

effective_num_data_points(***kwargs*)
returns the effective number of data points considered in the X2 estimation to compute the reduced X2 value

likelihood(*a*)

logL(*args*, *verbose=False*)
routine to compute X2 given variable parameters for a MCMC/PSO chain

Parameters

- **args** (*tuple or list of floats*) – ordered parameter values that are being sampled
- **verbose** (*boolean*) – if True, makes print statements about individual likelihood components

Returns log likelihood of the data given the model (natural logarithm)

log_likelihood(*kwargs_return*, *verbose=False*)

Parameters

- **kwargs_return** (*keyword arguments*) – need to contain 'kwargs_lens', 'kwargs_source', 'kwargs_lens_light', 'kwargs_ps', 'kwargs_special'. These entries themselves are lists of keyword argument of the parameters entering the model to be evaluated
- **verbose** (*boolean*) – if True, makes print statements about individual likelihood components

Returns

- logL (float) log likelihood of the data given the model (natural logarithm)

negativelogL(*a*)
for minimizer function, the negative value of the logl value is requested

Parameters **a** – array of parameters

Returns -logL

num_data

Returns number of independent data points in the combined fitting

param_limits

lenstronomy.Sampling.parameters module

class Param(*kwargs_model*, *kwargs_fixed_lens=None*, *kwargs_fixed_source=None*, *kwargs_fixed_lens_light=None*, *kwargs_fixed_ps=None*, *kwargs_fixed_special=None*, *kwargs_fixed_extinction=None*, *kwargs_lower_lens=None*, *kwargs_lower_source=None*, *kwargs_lower_lens_light=None*, *kwargs_lower_ps=None*, *kwargs_lower_special=None*, *kwargs_lower_extinction=None*, *kwargs_upper_lens=None*, *kwargs_upper_source=None*, *kwargs_upper_lens_light=None*, *kwargs_upper_ps=None*, *kwargs_upper_special=None*, *kwargs_upper_extinction=None*, *kwargs_lens_init=None*, *linear_solver=True*, *joint_lens_with_lens=[]*, *joint_lens_light_with_lens_light=[]*, *joint_source_with_source=[]*, *joint_lens_with_light=[]*, *joint_source_with_point_source=[]*, *joint_lens_light_with_point_source=[]*, *joint_extinction_with_lens_light=[]*, *joint_lens_with_source_light=[]*, *mass_scaling_list=None*, *point_source_offset=False*, *general_scaling=None*, *num_point_source_list=None*, *image_plane_source_list=None*, *solver_type='NONE'*, *Ddt_sampling=None*, *source_size=False*, *num_tau0=0*, *lens_redshift_sampling_indexes=None*, *source_redshift_sampling_indexes=None*, *source_grid_offset=False*, *num_shapelet_lens=0*, *log_sampling_lens=[]*)
Bases: `object`

class that handles the parameter constraints. In particular when different model profiles share joint constraints.

Options between same model classes:

'joint_lens_with_lens':list [[i_lens, k_lens, ['param_name1', 'param_name2', ...]], [...], ...], joint parameter between two lens models

'joint_lens_light_with_lens_light':list [[i_lens_light, k_lens_light, ['param_name1', 'param_name2', ...]], [...], ...], joint parameter between two lens light models, the second adopts the value of the first

'joint_source_with_source':list [[i_source, k_source, ['param_name1', 'param_name2', ...]], [...], ...], joint parameter between two source surface brightness models, the second adopts the value of the first

Options between different model classes:

'joint_lens_with_light': list [[i_light, k_lens, ['param_name1', 'param_name2', ...]], [...], ...], joint parameter between lens model and lens light model

'joint_source_with_point_source': list [[i_point_source, k_source], [...], ...], joint position parameter between source light model and point source

'joint_lens_light_with_point_source': list [[i_point_source, k_lens_light], [...], ...], joint position parameter between lens model and lens light model

'joint_extinction_with_lens_light': list [[i_lens_light, k_extinction, ['param_name1', 'param_name2', ...]], [...], ...], joint parameters between the lens surface brightness and the optical depth models

'joint_lens_with_source_light': [[i_source, k_lens, ['param_name1', 'param_name2', ...]], [...], ...], joint parameter between lens model and source light model. Samples light model parameter only.

'mass_scaling_list': e.g. [False, 1, 1, False, 2, False, 1, ...] Links lens models to have their masses scaled together. In this example, masses with False are not scaled, masses labeled 1 are scaled together, and those labeled 2 are scaled together independent of 1, etc.

'general_scaling': { 'param1': [False, 1, 1, False, 1, ...], 'param2': [1, 1, 1, False, 2, 2, ...] } Generalized parameter scaling. Input should be a dictionary mapping parameter names to the masks defining which lens models are scaled together, in the same format as for 'mass_scaling_list'. For each scaled parameter, two special params will be added called '${param}_scale_factor' and '${param}_scale_pow', defining the scaling and power-law of each.

Each scale will be modified as *param = param_scale_factor * param**param_scale_pow*.

For example, say we want to jointly constrain the *sigma0* and *Rs* parameters of some lens models indexed by *i*, like so:

$$\sigma_{0,i} = \sigma_0^{ref} L_i^{\alpha}$$
$$r_{cut,i} = r_{cut}^{ref} L_i^{\beta}$$

To do this we can add the following. The lens models corresponding to entries of *1* will be scaled together, and those corresponding to *False* will not be. As in *mass_scaling_list*, subsets of models can be scaled independently by marking them *2*, *3*, etc.

```
>>> 'general_scaling': {
>>>     'sigma0': [False, 1, 1, False, 1, ...],
>>>     'Rs': [False, 1, 1, False, 1, ...],
>>> }
```

Then we can choose to fix the power-law and vary the scale factor like so:

```
>>> fixed_special = {'sigma0_scale_pow': [alpha*2], 'Rs_scale_pow': [beta]}
>>> kwargs_special_init = {'sigma0_scale_factor': [17.0], 'Rs_scale_factor': [8]}
>>> kwargs_special_sigma = {'sigma0_scale_factor': [10.0], 'Rs_scale_factor': [3]}
>>> kwargs_lower_special = {'sigma0_scale_factor': [0.5], 'Rs_scale_factor': [1]}
>>> kwargs_upper_special = {'sigma0_scale_factor': [40], 'Rs_scale_factor': [20]}
```

```
>>> special_params = [kwargs_special_init, kwargs_special_sigma, fixed_special,␣
↪kwargs_lower_special, kwargs_upper_special]
```

hierarchy is as follows: 1. Point source parameters are inferred 2. Lens light joint parameters are set 3. Lens model joint constraints are set 4. Lens model solver is applied 5. Joint source and point source is applied

Alternatively to the format of the linking of parameters with IDENTICAL names as listed above as: [[i_1, k_2, ['param_name1', 'param_name2', ...]], [...], ...] the following format of the arguments are supported to join parameters with DIFFERENT names: [[i_1, k_2, {'param_old1': 'param_new1', 'ra_0': 'center_x'}], [...], ...] Log10 sampling of the lens parameters : 'log_sampling_lens': [[i_lens, ['param_name1', 'param_name2', ...]], [...], ...], Sample the log10 of the lens model parameters.

__init__(*kwargs_model, kwargs_fixed_lens=None, kwargs_fixed_source=None, kwargs_fixed_lens_light=None, kwargs_fixed_ps=None, kwargs_fixed_special=None, kwargs_fixed_extinction=None, kwargs_lower_lens=None, kwargs_lower_source=None, kwargs_lower_lens_light=None, kwargs_lower_ps=None, kwargs_lower_special=None, kwargs_lower_extinction=None, kwargs_upper_lens=None, kwargs_upper_source=None, kwargs_upper_lens_light=None, kwargs_upper_ps=None, kwargs_upper_special=None, kwargs_upper_extinction=None, kwargs_lens_init=None, linear_solver=True, joint_lens_with_lens=[], joint_lens_light_with_lens_light=[], joint_source_with_source=[], joint_lens_with_light=[], joint_source_with_point_source=[], joint_lens_light_with_point_source=[], joint_extinction_with_lens_light=[], joint_lens_with_source_light=[], mass_scaling_list=None, point_source_offset=False, general_scaling=None, num_point_source_list=None, image_plane_source_list=None, solver_type='NONE', Ddt_sampling=None, source_size=False, num_tau0=0, lens_redshift_sampling_indexes=None, source_redshift_sampling_indexes=None, source_grid_offset=False, num_shapelet_lens=0, log_sampling_lens=[]*)

Parameters

- **kwargs_model** – keyword arguments to describe all model components used in class_creator.create_class_instances()
- **kwargs_fixed_lens** – fixed parameters for lens model (keyword argument list)
- **kwargs_fixed_source** – fixed parameters for source model (keyword argument list)
- **kwargs_fixed_lens_light** – fixed parameters for lens light model (keyword argument list)
- **kwargs_fixed_ps** – fixed parameters for point source model (keyword argument list)
- **kwargs_fixed_special** – fixed parameters for special model parameters (keyword arguments)
- **kwargs_fixed_extinction** – fixed parameters for extinction model parameters (keyword argument list)
- **kwargs_lower_lens** – lower limits for parameters of lens model (keyword argument list)
- **kwargs_lower_source** – lower limits for parameters of source model (keyword argument list)
- **kwargs_lower_lens_light** – lower limits for parameters of lens light model (keyword argument list)
- **kwargs_lower_ps** – lower limits for parameters of point source model (keyword argument list)

- **kwargs_lower_special** – lower limits for parameters of special model parameters (keyword arguments)
- **kwargs_lower_extinction** – lower limits for parameters of extinction model (keyword argument list)
- **kwargs_upper_lens** – upper limits for parameters of lens model (keyword argument list)
- **kwargs_upper_source** – upper limits for parameters of source model (keyword argument list)
- **kwargs_upper_lens_light** – upper limits for parameters of lens light model (keyword argument list)
- **kwargs_upper_ps** – upper limits for parameters of point source model (keyword argument list)
- **kwargs_upper_special** – upper limits for parameters of special model parameters (keyword arguments)
- **kwargs_upper_extinction** – upper limits for parameters of extinction model (keyword argument list)
- **kwargs_lens_init** – initial guess of lens model keyword arguments (only relevant as the starting point of the non-linear solver)
- **linear_solver** – bool, if True; avoids sampling the linear amplitude parameters 'amp' such that they get overwritten by the linear solver solution. Fixed 'amp' parameters will be overwritten if linear_solver = True.
- **joint_lens_with_lens** – list [[i_lens, k_lens, ['param_name1', 'param_name2', . . .]], [. . .], . . .], joint parameter between two lens models
- **joint_lens_light_with_lens_light** – list [[i_lens_light, k_lens_light, ['param_name1', 'param_name2', . . .]], [. . .], . . .], joint parameter between two lens light models, the second adopts the value of the first
- **joint_source_with_source** – [[i_source, k_source, ['param_name1', 'param_name2', . . .]], [. . .], . . .], joint parameter between two source surface brightness models, the second adopts the value of the first
- **joint_lens_with_light** – list [[i_light, k_lens, ['param_name1', 'param_name2', . . .]], [. . .], . . .], joint parameter between lens model and lens light model
- **joint_source_with_point_source** – list [[i_point_source, k_source], [. . .], . . .], joint position parameter between lens model and source light model
- **joint_lens_light_with_point_source** – list [[i_point_source, k_lens_light], [. . .], . . .], joint position parameter between lens model and lens light model
- **joint_extinction_with_lens_light** – list [[i_lens_light, k_extinction, ['param_name1', 'param_name2', . . .]], [. . .], . . .], joint parameters between the lens surface brightness and the optical depth models
- **joint_lens_with_source_light** – [[i_source, k_lens, ['param_name1', 'param_name2', . . .]], [. . .], . . .], joint parameter between lens model and source light model. Samples light model parameter only.
- **mass_scaling_list** – boolean list of length of lens model list (optional) models with identical integers will be scaled with the same additional scaling factor. First integer starts with 1 (not 0)

- **general_scaling** – { 'param_1': [list of booleans/integers defining which model to fit], 'param_2': [..], ..}
- **point_source_offset** – bool, if True, adds relative offsets ot the modeled image positions relative to the time-delay and lens equation solver
- **num_point_source_list** – list of number of point sources per point source model class
- **image_plane_source_list** – optional, list of booleans for the source_light components. If a component is set =True it will parameterized the positions in the image plane and ray-trace the parameters back to the source position on the fly during the fitting.
- **solver_type** – string, option for specific solver type see detailed instruction of the Solver4Point and Solver2Point classes
- **Ddt_sampling** – bool, if True, samples the time-delay distance D_dt (in units of Mpc)
- **source_size** – bool, if True, samples a source size parameters to be evaluated in the flux ratio likelihood
- **num_tau0** – integer, number of different optical depth re-normalization factors
- **lens_redshift_sampling_indexes** – list of integers corresponding to the lens model components whose redshifts are a free parameter (only has an effect in multi-plane lensing) with same indexes indicating joint redshift, in ascending numbering e.g. [-1, 0, 0, 1, 0, 2], -1 indicating not sampled fixed indexes
- **source_redshift_sampling_indexes** – list of integers corresponding to the source model components whose redshifts are a free parameter (only has an effect in multi-plane lensing) with same indexes indicating joint redshift, in ascending numbering e.g. [-1, 0, 0, 1, 0, 2], -1 indicating not sampled fixed indexes. These indexes are the sample as for the lens
- **source_grid_offset** – optional, if True when using a pixel-based modelling (e.g. with STARLETS-like profiles), adds two additional sampled parameters describing RA/Dec offsets between data coordinate grid and pixelated source plane coordinate grid.
- **num_shapelet_lens** – number of shapelet coefficients in the 'SHAPELETS_CART' or 'SHAPELETS_POLAR' mass profile.
- **log_sampling_lens** – Sample the log10 of the lens model parameters. Format : [[i_lens, ['param_name1', 'param_name2', …]], […], …],

args2kwargs (*args*, *bijective=False*)

Parameters

- **args** – tuple of parameter values (float, strings, …)
- **bijective** – boolean, if True (default) returns the parameters in the form as they are sampled (e.g. if image_plane_source_list is set =True it returns the position in the image plane coordinates), if False, returns the parameters in the form to render a model (e.g. image_plane_source_list positions are ray-traced back to the source plane).

Returns keyword arguments sorted in lenstronomy conventions

check_solver (*kwargs_lens*, *kwargs_ps*)

test whether the image positions map back to the same source position :param kwargs_lens: lens model keyword argument list :param kwargs_ps: point source model keyword argument list :return: Euclidean distance between the ray-shooting of the image positions

image2source_plane (*kwargs_source*, *kwargs_lens*, *image_plane=False*)
maps the image plane position definition of the source plane

Parameters

- **kwargs_source** – source light model keyword argument list
- **kwargs_lens** – lens model keyword argument list
- **image_plane** – boolean, if True, does not up map image plane parameters to source plane

Returns source light model keyword arguments with mapped position arguments from image to source plane

kwargs2args (*kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *kwargs_special=None*, *kwargs_extinction=None*)
inverse of getParam function :param kwargs_lens: keyword arguments depending on model options :param kwargs_source: keyword arguments depending on model options :param kwargs_lens_light: lens light model keyword argument list :param kwargs_ps: point source model keyword argument list :param kwargs_special: special keyword arguments :param kwargs_extinction: extinction model keyword argument list :return: numpy array of parameters

linear_solver
boolean to state whether linear solver is activated or not

Returns boolean

num_param ()

Returns number of parameters involved (int), list of parameter names

num_param_linear ()

Returns number of linear basis set coefficients that are solved for

num_point_source_images

Returns total number of point source images

param_limits ()

Returns lower and upper limits of the arguments being sampled

print_setting ()
prints the setting of the parameter class

Returns

update_kwargs_model (*kwargs_special*)
updates model keyword arguments with redshifts being sampled

Parameters **kwargs_special** – keyword arguments from SpecialParam() class return of sampling arguments

Returns kwargs_model, bool (True if kwargs_model has changed, else False)

update_lens_scaling (*kwargs_special*, *kwargs_lens*, *inverse=False*)
multiplies the scaling parameters of the profiles

Parameters

- **kwargs_special** – keyword arguments of the 'special' arguments
- **kwargs_lens** – lens model keyword argument list
- **inverse** – bool, if True, performs the inverse lens scaling for bijective transforms

Returns updated lens model keyword argument list

lenstronomy.Sampling.sampler module

class Sampler(*likelihoodModule*)

Bases: `object`

class which executes the different sampling methods Available are: affine-invariant ensemble sampling with emcee, ensemble slice sampling with zeus and a Particle Swarm Optimizer. These are examples and depending on your problem, you might find other/better solutions. Feel free to sample with your convenient sampler!

__init__(*likelihoodModule*)

Parameters likelihoodModule – instance of LikelihoodModule class

mcmc_emcee(*n_walkers*, *n_run*, *n_burn*, *mean_start*, *sigma_start*, *mpi=False*, *progress=False*, *threadCount=1*, *initpos=None*, *backend_filename=None*, *start_from_backend=False*)

Run MCMC with emcee. For details, please have a look at the documentation of the emcee packager.

Parameters

- **n_walkers** (*integer*) – number of walkers in the emcee process
- **n_run** (*integer*) – number of sampling (after burn-in) of the emcee
- **n_burn** (*integer*) – number of burn-in iterations (those will not be saved in the output sample)
- **mean_start** (*numpy array of length the number of parameters*) – mean of the parameter position of the initialising sample
- **sigma_start** (*numpy array of length the number of parameters*) – spread of the parameter values (uncorrelated in each dimension) of the initialising sample
- **mpi** (*bool*) – if True, initializes an MPIPool to allow for MPI execution of the sampler
- **progress** (*bool*) – if True, prints the progress bar
- **threadCount** (*integer*) – number of threats in multi-processing (not applicable for MPI)
- **initpos** (*numpy array of size num param x num walkser*) – initial walker position to start sampling (optional)
- **backend_filename** (*string*) – name of the HDF5 file where sampling state is saved (through emcee backend engine)
- **start_from_backend** (*bool*) – if True, start from the state saved in *backup_filename*. Otherwise, create a new backup file with name *backup_filename* (any already existing file is overwritten!).

Returns samples, ln likelihood value of samples

Return type numpy 2d array, numpy 1d array

mcmc_zeus(*n_walkers*, *n_run*, *n_burn*, *mean_start*, *sigma_start*, *mpi=False*, *threadCount=1*, *progress=False*, *initpos=None*, *backend_filename=None*, ***kwargs_zeus*)

Lightning fast MCMC with zeus: https://github.com/minaskar/zeus

For the full list of arguments for the EnsembleSampler and callbacks, see see the zeus docs.

If you use the zeus sampler, you should cite the following papers: 2105.03468, 2002.06212.

Parameters

- **n_walkers** (*integer*) – number of walkers per parameter
- **n_run** (*integer*) – number of sampling steps
- **n_burn** (*integer*) – number of burn-in steps
- **mean_start** (*numpy array of length the number of parameters*) – mean of the parameter position of the initialising sample
- **sigma_start** (*numpy array of length the number of parameters*) – spread of the parameter values (uncorrelated in each dimension) of the initialising sample
- **mpi** (*bool*) – if True, initializes an MPIPool to allow for MPI execution of the sampler
- **progress** (*bool*) –
- **initpos** (*numpy array of size num param x num walkser*) – initial walker position to start sampling (optional)
- **backend_filename** (*string*) – name of the HDF5 file where sampling state is saved (through zeus callback function)

Returns samples, ln likelihood value of samples

Return type numpy 2d array, numpy 1d array

pso (*n_particles*, *n_iterations*, *lower_start=None*, *upper_start=None*, *threadCount=1*, *init_pos=None*, *mpi=False*, *print_key='PSO'*)

Return the best fit for the lens model on catalogue basis with particle swarm optimizer.

Parameters

- **n_particles** – number of particles in the sampling process
- **n_iterations** – number of iterations of the swarm
- **lower_start** – numpy array, lower end parameter of the values of the starting particles
- **upper_start** – numpy array, upper end parameter of the values of the starting particles
- **threadCount** – number of threads in the computation (only applied if mpi=False)
- **init_pos** – numpy array, position of the initial best guess model
- **mpi** – bool, if True, makes instance of MPIPool to allow for MPI execution
- **print_key** – string, prints the process name in the progress bar (optional)

Returns kwargs_result (of best fit), [lnlikelihood of samples, positions of samples, velocity of samples])

simplex (*init_pos*, *n_iterations*, *method*, *print_key='SIMPLEX'*)

Parameters

- **init_pos** – starting point for the optimization
- **n_iterations** – maximum number of iterations
- **method** – the optimization method, default is 'Nelder-Mead'

Returns the best fit for the lens model using the optimization routine specified by method

lenstronomy.Sampling.special_param module

class SpecialParam(*Ddt_sampling=False*, *mass_scaling=False*, *num_scale_factor=1*, *general_scaling_params=None*, *kwargs_fixed=None*, *kwargs_lower=None*, *kwargs_upper=None*, *point_source_offset=False*, *source_size=False*, *num_images=0*, *num_tau0=0*, *num_z_sampling=0*, *source_grid_offset=False*)

Bases: `object`

class that handles special parameters that are not directly part of a specific model component. These includes cosmology relevant parameters, astrometric errors and overall scaling parameters.

__init__(*Ddt_sampling=False*, *mass_scaling=False*, *num_scale_factor=1*, *general_scaling_params=None*, *kwargs_fixed=None*, *kwargs_lower=None*, *kwargs_upper=None*, *point_source_offset=False*, *source_size=False*, *num_images=0*, *num_tau0=0*, *num_z_sampling=0*, *source_grid_offset=False*)

Parameters

- **Ddt_sampling** – bool, if True, samples the time-delay distance D_dt (in units of Mpc)
- **mass_scaling** – bool, if True, samples a mass scaling factor between different profiles
- **num_scale_factor** – int, number of independent mass scaling factors being sampled
- **kwargs_fixed** – keyword arguments, fixed parameters during sampling
- **kwargs_lower** – keyword arguments, lower bound of parameters being sampled
- **kwargs_upper** – keyword arguments, upper bound of parameters being sampled
- **point_source_offset** – bool, if True, adds relative offsets ot the modeled image positions relative to the time-delay and lens equation solver
- **num_images** – number of point source images such that the point source offset parameters match their numbers
- **source_size** – bool, if True, samples a source size parameters to be evaluated in the flux ratio likelihood
- **num_tau0** – integer, number of different optical depth re-normalization factors
- **num_z_sampling** – integer, number of different lens redshifts to be sampled
- **source_grid_offset** – bool, if True, samples two parameters (x, y) for the offset of the pixelated source plane grid coordinates. Warning: this is only defined for pixel-based source modelling (e.g. 'SLIT_STARLETS' light profile)

get_params(*args*, *i*)

Parameters

- **args** – argument list
- **i** – integer, list index to start the read out for this class

Returns keyword arguments related to args, index after reading out arguments of this class

num_param()

Returns integer, number of free parameters sampled (and managed) by this class, parameter names (list of strings)

set_params(*kwargs_special*)

Parameters **kwargs_special** – keyword arguments with parameter settings

Returns argument list of the sampled parameters extracted from kwargs_special

lenstronomy.Sampling.param_group module

This module provides helper classes for managing sample parameters. This is for internal use, if you are not modifying lenstronomy sampling to include new parameters you can safely ignore this.

class ModelParamGroup

Bases: `object`

This abstract class represents any lenstronomy fitting parameters used in the Param class.

Subclasses should implement num_params(), set_params(), and get_params() to convert parameters from lenstronomy's semantic dictionary format to a flattened array format and back.

This class also contains three static methods to easily aggregate groups of parameter classes, called *compose_num_params()*, *compose_set_params()*, and *compose_get_params()*.

static compose_get_params(*each_group*, *flat_args*, *i*, **args*, ***kwargs*)

Converts a flattened array of parameters to lenstronomy semantic parameters in dictionary format. Combines the results for a set of arbitrarily many parameter groups.

Parameters

- **each_group** (*list*) – collection of parameter groups. Should each be subclasses of ModelParamGroup.
- **flat_args** (*list*) – the input array of parameters
- **i** (*int*) – the index in *flat_args* to start at
- **args** – Extra arguments to be passed to each call of *set_params()*
- **kwargs** – Extra keyword arguments to be passed to each call of *set_params()*

Returns As in each individual *get_params()*, a 2-tuple of (dictionary of params, new index)

static compose_num_params(*each_group*, **args*, ***kwargs*)

Aggregates the number of parameters for a group of parameter groups, calling each instance's *num_params()* method and combining the results

Parameters

- **each_group** (*list*) – collection of parameter groups. Should each be subclasses of ModelParamGroup.
- **args** – Extra arguments to be passed to each call of *num_params()*
- **kwargs** – Extra keyword arguments to be passed to each call of *num_params()*

Returns As in each individual *num_params()*, a 2-tuple of (num params, list of param names)

static compose_set_params(*each_group*, *param_kwargs*, **args*, ***kwargs*)

Converts lenstronomy semantic arguments in dictionary format to a flattened list of floats for use in optimization/fitting algorithms. Combines the results for a set of arbitrarily many parameter groups.

Parameters

- **each_group** (*list*) – collection of parameter groups. Should each be subclasses of ModelParamGroup.
- **param_kwargs** (*dict*) – the kwargs to process
- **args** – Extra arguments to be passed to each call of *set_params()*

- **kwargs** – Extra keyword arguments to be passed to each call of *set_params()*

Returns As in each individual *set_params()*, a list of floats

get_params (*args, i*)

Converts a flattened array of parameters back into a lenstronomy dictionary, starting at index i.

Parameters

- **args** (*list*) – flattened arguments to convert to lenstronomy format
- **i** (*int*) – index to begin at in args

Returns dictionary of parameters

num_params ()

Tells the number of parameters that this group samples and their names.

Returns 2-tuple of (num param, list of names)

set_params (*kwargs*)

Converts lenstronomy semantic parameters in dictionary format into a flattened array of parameters.

The flattened array is for use in optimization algorithms, e.g. MCMC, Particle swarm, etc.

Returns flattened array of parameters as floats

class SingleParam (*on*)

Bases: *lenstronomy.Sampling.param_group.ModelParamGroup*

Helper for handling parameters which are a single float.

Subclasses should define:

Parameters

- **on** (*bool*) – Whether this parameter is sampled
- **param_names** – List of strings, the name of each parameter
- **_kwargs_lower** – Dictionary. Lower bounds of each parameter
- **_kwargs_upper** – Dictionary. Upper bounds of each parameter

__init__ (*on*)

Parameters on (*bool*) – Whether this paramter should be sampled

get_params (*args, i, kwargs_fixed*)

Converts a flattened array of parameters back into a lenstronomy dictionary, starting at index i.

Parameters

- **args** (*list*) – flattened arguments to convert to lenstronomy format
- **i** (*int*) – index to begin at in args
- **kwargs_fixed** (*dict*) – Dictionary of fixed arguments

Returns dictionary of parameters

kwargs_lower

kwargs_upper

num_params (*kwargs_fixed*)

Tells the number of parameters that this group samples and their names.

Parameters kwargs_fixed (*dict*) – Dictionary of fixed arguments

Returns 2-tuple of (num param, list of names)

on

set_params (*kwargs*, *kwargs_fixed*)
Converts lenstronomy semantic parameters in dictionary format into a flattened array of parameters.

The flattened array is for use in optimization algorithms, e.g. MCMC, Particle swarm, etc.

Parameters

- **kwargs** (*dict*) – lenstronomy parameters to flatten
- **kwargs_fixed** (*dict*) – Dictionary of fixed arguments

Returns flattened array of parameters as floats

class ArrayParam (*on*)
Bases: *lenstronomy.Sampling.param_group.ModelParamGroup*

Helper for handling parameters which are an array of values. Examples include mass_scaling, which is an array of scaling parameters, and wavelet or gaussian decompositions which have different coefficients for each mode.

Subclasses should define:

Parameters

- **on** (*bool*) – Whether this parameter is sampled
- **param_names** – Dictionary mapping the name of each parameter to the number of values needed.
- **_kwargs_lower** – Dictionary. Lower bounds of each parameter
- **_kwargs_upper** – Dictionary. Upper bounds of each parameter

__init__ (*on*)

Parameters on (*bool*) – Whether this paramter should be sampled

get_params (*args*, *i*, *kwargs_fixed*)
Converts a flattened array of parameters back into a lenstronomy dictionary, starting at index i.

Parameters

- **args** (*list*) – flattened arguments to convert to lenstronomy format
- **i** (*int*) – index to begin at in args
- **kwargs_fixed** (*dict*) – Dictionary of fixed arguments

Returns dictionary of parameters

kwargs_lower

kwargs_upper

num_params (*kwargs_fixed*)
Tells the number of parameters that this group samples and their names.

Parameters kwargs_fixed (*dict*) – Dictionary of fixed arguments

Returns 2-tuple of (num param, list of names)

on

set_params (*kwargs*, *kwargs_fixed*)

Converts lenstronomy semantic parameters in dictionary format into a flattened array of parameters.

The flattened array is for use in optimization algorithms, e.g. MCMC, Particle swarm, etc.

Parameters

- **kwargs** (`dict`) – lenstronomy parameters to flatten
- **kwargs_fixed** (`dict`) – Dictionary of fixed arguments

Returns flattened array of parameters as floats

Module contents

lenstronomy.SimulationAPI package

Subpackages

lenstronomy.SimulationAPI.ObservationConfig package

Submodules

lenstronomy.SimulationAPI.ObservationConfig.DES module

Provisional DES instrument and observational settings. See Optics and Observation Conditions spreadsheet at https://docs.google.com/spreadsheets/d/1pMUB_OOZWwXON2dd5oP8PekhCT5MBBZJO1HV7IMZg4Y/edit?usp=sharing for list of sources.

class DES (*band='g'*, *psf_type='GAUSSIAN'*, *coadd_years=3*)

Bases: `object`

class contains DES instrument and observation configurations

__init__ (*band='g'*, *psf_type='GAUSSIAN'*, *coadd_years=3*)

Parameters

- **band** – string, 'g', 'r', 'i', 'z', or 'Y' supported. Determines obs dictionary.
- **psf_type** – string, type of PSF ('GAUSSIAN' supported).
- **coadd_years** – int, number of years corresponding to num_exposures in obs dict. Currently supported: 1-6.

camera = None

Parameters

- **read_noise** – std of noise generated by read-out (in units of electrons)
- **pixel_scale** – scale (in arcseconds) of pixels
- **ccd_gain** – electrons/ADU (analog-to-digital unit).

kwargs_single_band ()

Returns merged kwargs from camera and obs dicts

lenstronomy.SimulationAPI.ObservationConfig.Euclid module

Provisional Euclid instrument and observational settings. See Optics and Observation Conditions spreadsheet at https://docs.google.com/spreadsheets/d/1pMUB_OOZWwXON2dd5oP8PekhCT5MBBZJO1HV7IMZg4Y/edit?usp=sharing for list of sources.

class Euclid(*band='VIS'*, *psf_type='GAUSSIAN'*, *coadd_years=6*)

Bases: `object`

class contains Euclid instrument and observation configurations

__init__(*band='VIS'*, *psf_type='GAUSSIAN'*, *coadd_years=6*)

Parameters

- **band** – string, only 'VIS' supported. Determines obs dictionary.
- **psf_type** – string, type of PSF ('GAUSSIAN' supported).
- **coadd_years** – int, number of years corresponding to num_exposures in obs dict. Currently supported: 2-6.

camera = None

Parameters

- **read_noise** – std of noise generated by read-out (in units of electrons)
- **pixel_scale** – scale (in arcseconds) of pixels
- **ccd_gain** – electrons/ADU (analog-to-digital unit).

kwargs_single_band()

Returns merged kwargs from camera and obs dicts

lenstronomy.SimulationAPI.ObservationConfig.HST module

Provisional HST instrument and observational settings. See Optics and Observation Conditions spreadsheet at https://docs.google.com/spreadsheets/d/1pMUB_OOZWwXON2dd5oP8PekhCT5MBBZJO1HV7IMZg4Y/edit?usp=sharing for list of sources.

class HST(*band='TDLMC_F160W'*, *psf_type='PIXEL'*, *coadd_years=None*)

Bases: `object`

class contains HST instrument and observation configurations

__init__(*band='TDLMC_F160W'*, *psf_type='PIXEL'*, *coadd_years=None*)

Parameters

- **band** – string, 'WFC3_F160W' or 'TDLMC_F160W' supported. Determines obs dictionary.
- **psf_type** – string, type of PSF ('GAUSSIAN', 'PIXEL' supported).
- **coadd_years** – int, number of years corresponding to num_exposures in obs dict. Currently supported: None.

camera = None

Parameters

- **read_noise** – std of noise generated by read-out (in units of electrons)

- **pixel_scale** – scale (in arcseconds) of pixels
- **ccd_gain** – electrons/ADU (analog-to-digital unit).

kwargs_single_band()

Returns merged kwargs from camera and obs dicts

lenstronomy.SimulationAPI.ObservationConfig.LSST module

Provisional LSST instrument and observational settings. See Optics and Observation Conditions spreadsheet at https://docs.google.com/spreadsheets/d/1pMUB_OOZWwXON2dd5oP8PekhCT5MBBZJO1HV7IMZg4Y/edit?usp=sharing for list of sources.

class LSST(*band='g'*, *psf_type='GAUSSIAN'*, *coadd_years=10*)
Bases: `object`

class contains LSST instrument and observation configurations

__init__(*band='g'*, *psf_type='GAUSSIAN'*, *coadd_years=10*)

Parameters

- **band** – string, 'u', 'g', 'r', 'i', 'z' or 'y' supported. Determines obs dictionary.
- **psf_type** – string, type of PSF ('GAUSSIAN' supported).
- **coadd_years** – int, number of years corresponding to num_exposures in obs dict. Currently supported: 1-10.

camera = None

Parameters

- **read_noise** – std of noise generated by read-out (in units of electrons)
- **pixel_scale** – scale (in arcseconds) of pixels
- **ccd_gain** – electrons/ADU (analog-to-digital unit).

kwargs_single_band()

Returns merged kwargs from camera and obs dicts

Module contents

Submodules

lenstronomy.SimulationAPI.data_api module

class DataAPI(*numpix*, *kwargs_pixel_grid=None*, ***kwargs_single_band*)
Bases: *lenstronomy.SimulationAPI.observation_api.SingleBand*

This class is a wrapper of the general description of data in SingleBand() to translate those quantities into configurations in the core lenstronomy Data modules to simulate images according to those quantities. This class is meant to be an example of a wrapper. More possibilities in terms of PSF and data type options are available. Have a look in the specific modules if you are interested in.

__init__(*numpix*, *kwargs_pixel_grid=None*, ***kwargs_single_band*)

Parameters

- **numpix** – number of pixels per axis in the simulation to be modelled
- **kwargs_pixel_grid** – if None, uses default pixel grid option if defined, must contain keyword arguments PixelGrid() class
- **kwargs_single_band** – keyword arguments used to create instance of SingleBand class

data_class
creates a Data() instance of lenstronomy based on knowledge of the observation

Returns instance of Data() class

kwargs_data

Returns keyword arguments for ImageData class instance

lenstronomy.SimulationAPI.model_api module

class ModelAPI(*lens_model_list=None, z_lens=None, z_source=None, lens_redshift_list=None, source_light_model_list=None, lens_light_model_list=None, point_source_model_list=None, source_redshift_list=None, cosmo=None, z_source_convention=None*)
Bases: `object`

This class manages the model choices. The role is to return instances of the lenstronomy LightModel, LensModel, PointSource modules according to the options chosen by the user. Currently, all other model choices are equivalent to the ones provided by LightModel, LensModel, PointSource. The current options of the class instance only describe a subset of possibilities.

__init__(*lens_model_list=None, z_lens=None, z_source=None, lens_redshift_list=None, source_light_model_list=None, lens_light_model_list=None, point_source_model_list=None, source_redshift_list=None, cosmo=None, z_source_convention=None*)
TODO: make inputs follow the kwargs_model of the class_creator instances of 'kwargs_model', # i.e. multi-plane options, perhaps others

Parameters

- **lens_model_list** – list of strings with lens model names
- **z_lens** – redshift of the deflector (only considered when operating in single plane mode). Is only needed for specific functions that require a cosmology.
- **z_source** – redshift of the source: Needed in multi_plane option only, not required for the core functionalities in the single plane mode. This will be the redshift of the source plane (if not further specified the 'source_redshift_list') and the point source redshift (regardless of 'source_redshift_list')
- **lens_redshift_list** – list of deflector redshift (corresponding to the lens model list), only applicable in multi_plane mode.
- **source_light_model_list** – list of strings with source light model names (lensed light profiles)
- **lens_light_model_list** – list of strings with lens light model names (not lensed light profiles)
- **point_source_model_list** – list of strings with point source model names
- **source_redshift_list** – list of redshifts of the source profiles (optional)

- **cosmo** – instance of the astropy cosmology class. If not specified, uses the default cosmology.
- **z_source_convention** – float, redshift of a source to define the reduced deflection angles of the lens models. If None, 'z_source' is used.

lens_light_model_class

Returns instance of lenstronomy LightModel class describing the non-lensed light profiles

lens_model_class

Returns instance of lenstronomy LensModel class

physical2lensing_conversion(*kwargs_mass*)

Parameters kwargs_mass – list of keyword arguments of all the lens models. Einstein radius 'theta_E' are replaced by 'sigma_v', velocity dispersion in km/s, 'alpha_Rs' and 'Rs' of NFW profiles are replaced by 'M200' and 'concentration'

Returns kwargs_lens in reduced deflection angles compatible with the lensModel instance of this module

point_source_model_class

Returns instance of lenstronomy PointSource class describing the point sources (lensed and unlensed)

source_model_class

Returns instance of lenstronomy LightModel class describing the source light profiles

lenstronomy.SimulationAPI.observation_api module

class Instrument(*pixel_scale*, *read_noise=None*, *ccd_gain=None*)

Bases: `object`

basic access points to instrument properties

__init__(*pixel_scale*, *read_noise=None*, *ccd_gain=None*)

Parameters

- **read_noise** – std of noise generated by read-out (in units of electrons)
- **pixel_scale** – scale (in arcseconds) of pixels
- **ccd_gain** – electrons/ADU (analog-to-digital unit). A gain of 8 means that the camera digitizes the CCD signal so that each ADU corresponds to 8 photoelectrons.

class Observation(*exposure_time*, *sky_brightness=None*, *seeing=None*, *num_exposures=1*, *psf_type='GAUSSIAN'*, *kernel_point_source=None*, *truncation=5*, *point_source_supersampling_factor=1*)

Bases: `object`

basic access point to observation properties

__init__(*exposure_time*, *sky_brightness=None*, *seeing=None*, *num_exposures=1*, *psf_type='GAUSSIAN'*, *kernel_point_source=None*, *truncation=5*, *point_source_supersampling_factor=1*)

Parameters

- **exposure_time** – exposure time per image (in seconds)

- **sky_brightness** – sky brightness (in magnitude per square arcseconds)
- **seeing** – full width at half maximum of the PSF (if not specific psf_model is specified)
- **num_exposures** – number of exposures that are combined
- **psf_type** – string, type of PSF ('GAUSSIAN' and 'PIXEL' supported)
- **kernel_point_source** – 2d numpy array, model of PSF centered with odd number of pixels per axis (optional when psf_type='PIXEL' is chosen)
- **point_source_supersampling_factor** – int, supersampling factor of kernel_point_source (optional when psf_type='PIXEL' is chosen)

exposure_time
total exposure time

Returns summed exposure time

kwargs_psf
keyword arguments to initiate a PSF() class

Returns kwargs_psf

psf_class
creates instance of PSF() class based on knowledge of the observations For the full possibility of how to create such an instance, see the PSF() class documentation

Returns instance of PSF() class

update_observation(*exposure_time=None, sky_brightness=None, seeing=None, num_exposures=None, psf_type=None, kernel_point_source=None*)
updates class instance with new properties if specific argument is not None

Parameters

- **exposure_time** – exposure time per image (in seconds)
- **sky_brightness** – sky brightness (in magnitude per square arcseconds)
- **seeing** – full width at half maximum of the PSF (if not specific psf_model is specified)
- **num_exposures** – number of exposures that are combined
- **psf_type** – string, type of PSF ('GAUSSIAN' and 'PIXEL' supported)
- **kernel_point_source** – 2d numpy array, model of PSF centered with odd number of pixels per axis (optional when psf_type='PIXEL' is chosen)

Returns None, updated class instance

class SingleBand(*pixel_scale, exposure_time, magnitude_zero_point, read_noise=None, ccd_gain=None, sky_brightness=None, seeing=None, num_exposures=1, psf_type='GAUSSIAN', kernel_point_source=None, truncation=5, point_source_supersampling_factor=1, data_count_unit='e-', background_noise=None*)
Bases: `lenstronomy.SimulationAPI.observation_api.Instrument`, `lenstronomy.SimulationAPI.observation_api.Observation`

class that combines Instrument and Observation

__init__(*pixel_scale, exposure_time, magnitude_zero_point, read_noise=None, ccd_gain=None, sky_brightness=None, seeing=None, num_exposures=1, psf_type='GAUSSIAN', kernel_point_source=None, truncation=5, point_source_supersampling_factor=1, data_count_unit='e-', background_noise=None*)

Parameters

- **read_noise** – std of noise generated by read-out (in units of electrons)
- **pixel_scale** – scale (in arcseconds) of pixels
- **ccd_gain** – electrons/ADU (analog-to-digital unit). A gain of 8 means that the camera digitizes the CCD signal so that each ADU corresponds to 8 photoelectrons.
- **exposure_time** – exposure time per image (in seconds)
- **sky_brightness** – sky brightness (in magnitude per square arcseconds in units of electrons)
- **seeing** – Full-Width-at-Half-Maximum (FWHM) of PSF
- **magnitude_zero_point** – magnitude in which 1 count (e-) per second per arcsecond square is registered
- **num_exposures** – number of exposures that are combined
- **point_source_supersampling_factor** – int, supersampling factor of kernel_point_source (optional when psf_type='PIXEL' is chosen)
- **data_count_unit** – string, unit of the data (not noise properties - see other definitions), 'e-': (electrons assumed to be IID), 'ADU': (analog-to-digital unit)
- **background_noise** – sqrt(variance of background) as a total contribution from read-noise, sky brightness etc in units of the data_count_units (e- or ADU) If you set this parameter, it will use this value regardless of the values of read_noise, sky_brightness

background_noise

Gaussian sigma of noise level per pixel in counts (e- or ADU) per second

Returns sqrt(variance) of background noise level in data units

estimate_noise (*image*)

Parameters **image** – noisy data, background subtracted

Returns estimated noise map sqrt(variance) for each pixel as estimated from the instrument and observation

flux_iid (*flux_per_second*)

IID counts. This can be used by lenstronomy to estimate the Poisson errors keeping the assumption that the counts are IIDs (even if they are not).

Parameters **flux_per_second** – flux count per second in the units set in this class (ADU or e-)

Returns IID count number

flux_noise (*flux*)

Parameters **flux** – float or array, units of count_unit/seconds, needs to be positive semi-definite in the flux value

Returns Gaussian approximation of Poisson statistics in IIDs sqrt(variance)

magnitude2cps (*magnitude*)

converts an apparent magnitude to counts per second (in units of the data)

The zero point of an instrument, by definition, is the magnitude of an object that produces one count (or data number, DN) per second. The magnitude of an arbitrary object producing DN counts in an observation of length EXPTIME is therefore: m = -2.5 x log10(DN / EXPTIME) + ZEROPOINT

Parameters **magnitude** – magnitude of object

Returns counts per second of object

noise_for_model (*model*, *background_noise=True*, *poisson_noise=True*, *seed=None*)

Parameters

- **model** – 2d numpy array of modelled image (with pixels in units of data specified in class)
- **background_noise** – bool, if True, adds background noise
- **poisson_noise** – bool, if True, adds Poisson noise of modelled flux
- **seed** – int, seed number to be used to render the noise properties. If None, then uses the current numpy.random seed to render the noise properties.

Returns noise realization corresponding to the model

sky_brightness

Returns sky brightness (counts per square arcseconds in unit of data (e- or ADU's) per unit time)

lenstronomy.SimulationAPI.observation_constructor module

observation_constructor (*instrument_name*, *observation_name*)

Parameters

- **instrument_name** – string, name of instrument referenced in this file
- **observation_name** – string, name of observation referenced in this file

Returns instance of the SimulationAPI.data_type instance

lenstronomy.SimulationAPI.point_source_variability module

class PointSourceVariability (*source_x*, *source_y*, *variability_func*, *numpix*, *kwargs_single_band*, *kwargs_model*, *kwargs_numerics*, *kwargs_lens*, *kwargs_source_mag=None*, *kwargs_lens_light_mag=None*, *kwargs_ps_mag=None*)

Bases: `object`

This class enables to plug in a variable point source in the source plane to be added on top of a fixed lens and extended surface brightness model. The class inherits SimAPI and additionally requires the lens and light model parameters as well as a position in the source plane.

The intrinsic source variability can be defined by the user and additional uncorrelated variability in the image plane can be plugged in as well (e.g. due to micro-lensing)

__init__ (*source_x*, *source_y*, *variability_func*, *numpix*, *kwargs_single_band*, *kwargs_model*, *kwargs_numerics*, *kwargs_lens*, *kwargs_source_mag=None*, *kwargs_lens_light_mag=None*, *kwargs_ps_mag=None*)

Parameters

- **source_x** – RA of source position
- **source_y** – DEC of source position

- **variability_func** – function that returns a brightness (in magnitude) as a function of time t
- **numpix** – number of pixels per axis
- **kwargs_single_band** –
- **kwargs_model** –
- **kwargs_numerics** –
- **kwargs_lens** –
- **kwargs_source_mag** –
- **kwargs_lens_light_mag** –
- **kwargs_ps_mag** –

delays

Returns time delays

image_bkg

Returns 2d numpy array, image of the extended light components without the variable source

image_time(*time=0*)

Parameters **time** – time relative to the definition of t=0 for the first appearing image

Returns image with time variable source at given time

point_source_time(*t*)

Parameters **t** – time (in units of days)

Returns image plane parameters of the point source observed at t

lenstronomy.SimulationAPI.sim_api module

class SimAPI(*numpix*, *kwargs_single_band*, *kwargs_model*)

Bases: *lenstronomy.SimulationAPI.data_api.DataAPI*, *lenstronomy.SimulationAPI.model_api.ModelAPI*

This class manages the model parameters in regard of the data specified in SingleBand. In particular, this API translates models specified in units of astronomical magnitudes into the amplitude parameters used in the LightModel module of lenstronomy. Optionally, this class can also handle inputs with cosmology dependent lensing quantities and translates them to the optical quantities being used in the lenstronomy LensModel module. All other model choices are equivalent to the ones provided by LightModel, LensModel, PointSource modules

__init__(*numpix*, *kwargs_single_band*, *kwargs_model*)

Parameters

- **numpix** – number of pixels per axis
- **kwargs_single_band** – keyword arguments specifying the class instance of DataAPI
- **kwargs_model** – keyword arguments specifying the class instance of ModelAPI

image_model_class(*kwargs_numerics=None*)

Parameters **kwargs_numerics** – keyword arguments list of Numerics module

Returns instance of the ImageModel class with all the specified configurations

magnitude2amplitude(*kwargs_lens_light_mag=None*, *kwargs_source_mag=None*, *kwargs_ps_mag=None*)
'magnitude' definition are in APPARENT magnitudes as observed on the sky, not intrinsic!

Parameters

- **kwargs_lens_light_mag** – keyword argument list as for LightModel module except that 'amp' parameters are 'magnitude' parameters.
- **kwargs_source_mag** – keyword argument list as for LightModel module except that 'amp' parameters are 'magnitude' parameters.
- **kwargs_ps_mag** – keyword argument list as for PointSource module except that 'amp' parameters are 'magnitude' parameters.

Returns value of the lenstronomy 'amp' parameter such that the total flux of the profile type results in this magnitude for all the light models. These keyword arguments conform with the lenstronomy LightModel syntax.

Module contents

lenstronomy.Util package

Submodules

lenstronomy.Util.analysis_util module

half_light_radius(*lens_light*, *x_grid*, *y_grid*, *center_x=0*, *center_y=0*)

Parameters

- **lens_light** – array of surface brightness
- **x_grid** – x-axis coordinates
- **y_grid** – y-axis coordinates
- **center_x** – center of light
- **center_y** – center of light

Returns

radial_profile(*light_grid*, *x_grid*, *y_grid*, *center_x=0*, *center_y=0*, *n=None*)
computes radial profile

Parameters

- **light_grid** – array of surface brightness
- **x_grid** – x-axis coordinates
- **y_grid** – y-axis coordinates
- **center_x** – center of light
- **center_y** – center of light
- **n** – number of discrete steps

Returns I(r), r with r in units of the coordinate grid

azimuthalAverage (*image*, *center=None*)
Calculate the azimuthally averaged radial profile.

image - The 2D image center - The [x,y] pixel coordinates used as the center. The default is None, which then uses the center of the image (including fractional pixels). :return: I(r) (averaged), r of bin edges in units of pixels of the 2D image

moments (*I_xy_input*, *x*, *y*)
compute quadrupole moments from a light distribution

Parameters

- **I_xy_input** – light distribution
- **x** – x-coordinates of I_xy
- **y** – y-coordinates of I_xy

Returns Q_xx, Q_xy, Q_yy

ellipticities (*I_xy*, *x*, *y*)
compute ellipticities of a light distribution

Parameters

- **I_xy** – surface brightness I(x, y) as array
- **x** – x-coordinates in same shape as I_xy
- **y** – y-coordinates in same shape as I_xy

Returns reduced shear moments g1, g2

bic_model (*logL*, *num_data*, *num_param*)
Bayesian information criteria

Parameters

- **logL** – log likelihood value
- **num_data** – numbers of data
- **num_param** – numbers of model parameters

Returns BIC value

profile_center (*kwargs_list*, *center_x=None*, *center_y=None*)
utility routine that results in the centroid estimate for the profile estimates

Parameters

- **kwargs_list** – light parameter keyword argument list (can be light or mass)
- **center_x** – None or center
- **center_y** – None or center

Returns center_x, center_y

lenstronomy.Util.class_creator module

create_class_instances(*lens_model_list=None, z_lens=None, z_source=None, z_source_convention=None, lens_redshift_list=None, kwargs_interp=None, multi_plane=False, observed_convention_index=None, source_light_model_list=None, lens_light_model_list=None, point_source_model_list=None, fixed_magnification_list=None, flux_from_point_source_list=None, additional_images_list=None, kwargs_lens_eqn_solver=None, source_deflection_scaling_list=None, source_redshift_list=None, cosmo=None, index_lens_model_list=None, index_source_light_model_list=None, index_lens_light_model_list=None, index_point_source_model_list=None, optical_depth_model_list=None, index_optical_depth_model_list=None, band_index=0, tau0_index_list=None, all_models=False, point_source_magnification_limit=None, surface_brightness_smoothing=0.001, sersic_major_axis=None*)

Parameters

- **lens_model_list** – list of strings indicating the type of lens models
- **z_lens** – redshift of the deflector (for single lens plane mode, but only relevant when computing physical quantities)
- **z_source** – redshift of source (for single source plane mode, or for multiple source planes the redshift of the point source). In regard to this redshift the reduced deflection angles are defined in the lens model.
- **z_source_convention** – float, redshift of a source to define the reduced deflection angles of the lens models. If None, 'z_source' is used.
- **lens_redshift_list** –
- **multi_plane** –
- **kwargs_interp** – interpolation keyword arguments specifying the numerics. See description in the Interpolate() class. Only applicable for 'INTERPOL' and 'INTERPOL_SCALED' models.
- **observed_convention_index** –
- **source_light_model_list** –
- **lens_light_model_list** –
- **point_source_model_list** –
- **fixed_magnification_list** –
- **flux_from_point_source_list** – list of bools (optional), if set, will only return image positions (for imaging modeling) for the subset of the point source lists that =True. This option enables to model
- **additional_images_list** –
- **kwargs_lens_eqn_solver** – keyword arguments specifying the numerical settings for the lens equation solver see LensEquationSolver() class for details
- **source_deflection_scaling_list** – List of floats for each source ligth model (optional, and only applicable for single-plane lensing. The factors re-scale the reduced deflection angles described from the lens model. =1 means identical source position as

without this option. This option enables multiple source planes. The geometric difference between the different source planes needs to be pre-computed and is cosmology dependent.

- **source_redshift_list** –
- **cosmo** – astropy.cosmology instance
- **index_lens_model_list** –
- **index_source_light_model_list** –
- **index_lens_light_model_list** –
- **index_point_source_model_list** –
- **optical_depth_model_list** – list of strings indicating the optical depth model to compute (differential) extinctions from the source
- **index_optical_depth_model_list** –
- **band_index** – int, index of band to consider. Has an effect if only partial models are considered for a specific band
- **tau0_index_list** – list of integers of the specific extinction scaling parameter tau0 for each band
- **all_models** – bool, if True, will make class instances of all models ignoring potential keywords that are excluding specific models as indicated.
- **point_source_magnification_limit** – float >0 or None, if set and additional images are computed, then it will cut the point sources computed to the limiting (absolute) magnification
- **surface_brightness_smoothing** – float, smoothing scale of light profile (minimal distance to the center of a profile) this can help to avoid inaccuracies in the very center of a cuspy light profile
- **sersic_major_axis** – boolean or None, if True, uses the semi-major axis as the definition of the Sersic half-light radius, if False, uses the product average of semi-major and semi-minor axis. If None, uses the convention in the lenstronomy yaml setting (which by default is =False)

Returns

create_image_model(*kwargs_data*, *kwargs_psf*, *kwargs_numerics*, *kwargs_model*, *image_likelihood_mask=None*)

Parameters

- **kwargs_data** – ImageData keyword arguments
- **kwargs_psf** – PSF keyword arguments
- **kwargs_numerics** – numerics keyword arguments for Numerics() class
- **kwargs_model** – model keyword arguments
- **image_likelihood_mask** – image likelihood mask (same size as image_data with 1 indicating being evaluated and 0 being left out)

Returns ImageLinearFit() instance

create_im_sim(*multi_band_list*, *multi_band_type*, *kwargs_model*, *bands_compute=None*, *image_likelihood_mask_list=None*, *band_index=0*, *kwargs_pixelbased=None*, *linear_solver=True*)

Parameters

- **multi_band_list** – list of [[kwargs_data, kwargs_psf, kwargs_numerics], [], ..]
- **multi_band_type** – string, option when having multiple imaging data sets modelled simultaneously. Options are: - 'multi-linear': linear amplitudes are inferred on single data set - 'linear-joint': linear amplitudes ae jointly inferred - 'single-band': single band
- **kwargs_model** – model keyword arguments
- **bands_compute** – (optional), bool list to indicate which band to be included in the modeling
- **image_likelihood_mask_list** – list of image likelihood mask (same size as image_data with 1 indicating being evaluated and 0 being left out)
- **band_index** – integer, index of the imaging band to model (only applied when using 'single-band' as option)
- **kwargs_pixelbased** – keyword arguments with various settings related to the pixel-based solver (see SLITronomy documentation)
- **linear_solver** – bool, if True (default) fixes the linear amplitude parameters 'amp' (avoid sampling) such that they get overwritten by the linear solver solution.

Returns MultiBand class instance

lenstronomy.Util.constants module

delay_arcsec2days(*delay_arcsec*, *ddt*)

given a delay in arcsec^2 and a Delay distance, the delay is computed in days

Parameters

- **delay_arcsec** – gravitational delay in units of arcsec^2 (e.g. Fermat potential)
- **ddt** – Time delay distance (in units of Mpc)

Returns time-delay in units of days

lenstronomy.Util.correlation module

correlation_2D(*image*)

#TODO document normalization output in units

Parameters **image** – 2d image

Returns 2d fourier transform

power_spectrum_2d(*image*)

Parameters **image** – 2d numpy array

Returns 2d power spectrum in frequency units of the pixels

power_spectrum_1d(*image*)

Parameters **image** – 2d numpy array

Returns 1d radially averaged power spectrum of image in frequency units of pixels, radius in units of pixels

lenstronomy.Util.data_util module

bkg_noise (*readout_noise*, *exposure_time*, *sky_brightness*, *pixel_scale*, *num_exposures=1*)
computes the expected Gaussian background noise of a pixel in units of counts/second

Parameters

- **readout_noise** – noise added per readout
- **exposure_time** – exposure time per exposure (in seconds)
- **sky_brightness** – counts per second per unit arcseconds square
- **pixel_scale** – size of pixel in units arcseonds
- **num_exposures** – number of exposures (with same exposure time) to be co-added

Returns estimated Gaussian noise sqrt(variance)

flux_noise (*cps_pixel*, *exposure_time*)
computes the variance of the shot noise Gaussian approximation of Poisson noise term

Parameters

- **cps_pixel** – counts per second of the intensity per pixel unit
- **exposure_time** – total exposure time (in units seconds or equivalent unit as cps_pixel)

Returns sqrt(variance) of pixel value

magnitude2cps (*magnitude*, *magnitude_zero_point*)
converts an apparent magnitude to counts per second

The zero point of an instrument, by definition, is the magnitude of an object that produces one count (or data number, DN) per second. The magnitude of an arbitrary object producing DN counts in an observation of length EXPTIME is therefore: m = -2.5 x log10(DN / EXPTIME) + ZEROPOINT

Parameters

- **magnitude** – astronomical magnitude
- **magnitude_zero_point** – magnitude zero point (astronomical magnitude with 1 count per second)

Returns counts per second of astronomical object

cps2magnitude (*cps*, *magnitude_zero_point*)

Parameters

- **cps** – float, count-per-second
- **magnitude_zero_point** – magnitude zero point

Returns magnitude for given counts

absolute2apparent_magnitude (*absolute_magnitude*, *d_parsec*)
converts absolute to apparent magnitudes

Parameters

- **absolute_magnitude** – absolute magnitude of object
- **d_parsec** – distance to object in units parsec

Returns apparent magnitude

adu2electrons (*adu*, *ccd_gain*)
converts analog-to-digital units into electron counts

Parameters

- **adu** – counts in analog-to-digital unit
- **ccd_gain** – CCD gain, meaning how many electrons are counted per unit ADU

Returns counts in electrons

electrons2adu (*electrons*, *ccd_gain*)
converts electron counts into analog-to-digital unit

Parameters

- **electrons** – number of electrons received on detector
- **ccd_gain** – CCD gain, meaning how many electrons are counted per unit ADU

Returns adu value in Analog-to-digital units corresponding to electron count

lenstronomy.Util.derivative_util module

routines to compute derivatives of spherical functions

d_r_dx (*x*, *y*)
derivative of r with respect to x :param x: :param y: :return:

d_r_dy (*x*, *y*)
differential dr/dy

Parameters

- **x** –
- **y** –

Returns

d_r_dxx (*x*, *y*)
second derivative dr/dxdx :param x: :param y: :return:

d_r_dyy (*x*, *y*)
second derivative dr/dxdx :param x: :param y: :return:

d_r_dxy (*x*, *y*)
second derivative dr/dxdx :param x: :param y: :return:

d_phi_dx (*x*, *y*)
angular derivative in respect to x when phi = arctan2(y, x)

Parameters

- **x** –
- **y** –

Returns

d_phi_dy (*x*, *y*)
angular derivative in respect to y when phi = arctan2(y, x)

Parameters

- **x** –

- **y** –

Returns

d_phi_dxx (*x*, *y*)
second derivative of the orientation angle

Parameters

- **x** –
- **y** –

Returns

d_phi_dyy (*x*, *y*)
second derivative of the orientation angle in dydy

Parameters

- **x** –
- **y** –

Returns

d_phi_dxy (*x*, *y*)
second derivative of the orientation angle in dxdy

Parameters

- **x** –
- **y** –

Returns

d_x_diffr_dx (*x*, *y*)
derivative of d(x/r)/dx equivalent to second order derivatives dr_dxx

Parameters

- **x** –
- **y** –

Returns

d_y_diffr_dy (*x*, *y*)
derivative of d(y/r)/dy equivalent to second order derivatives dr_dyy

Parameters

- **x** –
- **y** –

Returns

d_y_diffr_dx (*x*, *y*)
derivative of d(y/r)/dx equivalent to second order derivatives dr_dxy

Parameters

- **x** –
- **y** –

Returns

d_x_diffr_dy (*x*, *y*)
derivative of d(x/r)/dy equivalent to second order derivatives dr_dyx

Parameters

- **x** –
- **y** –

Returns

lenstronomy.Util.image_util module

add_layer2image (*grid2d*, *x_pos*, *y_pos*, *kernel*, *order=1*)
adds a kernel on the grid2d image at position x_pos, y_pos with an interpolated subgrid pixel shift of order=order

Parameters

- **grid2d** – 2d pixel grid (i.e. image)
- **x_pos** – x-position center (pixel coordinate) of the layer to be added
- **y_pos** – y-position center (pixel coordinate) of the layer to be added
- **kernel** – the layer to be added to the image
- **order** – interpolation order for sub-pixel shift of the kernel to be added

Returns image with added layer, cut to original size

add_layer2image_int (*grid2d*, *x_pos*, *y_pos*, *kernel*)
adds a kernel on the grid2d image at position x_pos, y_pos at integer positions of pixel

Parameters

- **grid2d** – 2d pixel grid (i.e. image)
- **x_pos** – x-position center (pixel coordinate) of the layer to be added
- **y_pos** – y-position center (pixel coordinate) of the layer to be added
- **kernel** – the layer to be added to the image

Returns image with added layer

add_background (*image*, *sigma_bkd*)
Generates background noise to image. To generate a noisy image with background noise, generate image_noisy = image + add_background(image, sigma_bkd)

Parameters

- **image** – pixel values of image
- **sigma_bkd** – background noise (sigma)

Returns a realisation of Gaussian noise of the same size as image

add_poisson (*image*, *exp_time*)
Generates a poison (or Gaussian) distributed noise with mean given by surface brightness. To generate a noisy image with Poisson noise, perform image_noisy = image + add_poisson(image, exp_time)

Parameters

- **image** – pixel values (photon counts per unit exposure time)
- **exp_time** – exposure time

Returns Poisson noise realization of input image

rotateImage (*img*, *angle*)
querries scipy.ndimage.rotate routine :param img: image to be rotated :param angle: angle to be rotated (radian) :return: rotated image

re_size_array (*x_in*, *y_in*, *input_values*, *x_out*, *y_out*)
resizes 2d array (i.e. image) to new coordinates. So far only works with square output aligned with coordinate axis.

Parameters

- **x_in** –
- **y_in** –
- **input_values** –
- **x_out** –
- **y_out** –

Returns

symmetry_average (*image*, *symmetry*)
symmetry averaged image

Parameters

- **image** –
- **symmetry** –

Returns

findOverlap (*x_mins*, *y_mins*, *min_distance*)
finds overlapping solutions, deletes multiples and deletes non-solutions and if it is not a solution, deleted as well

coordInImage (*x_coord*, *y_coord*, *num_pix*, *deltapix*)
checks whether image positions are within the pixel image in units of arcsec if not: remove it

Returns image positions within the pixel image

re_size (*image*, *factor=1*)
re-sizes image with nx x ny to nx/factor x ny/factor

Parameters

- **image** – 2d image with shape (nx,ny)
- **factor** – integer >=1

Returns

rebin_image (*bin_size*, *image*, *wht_map*, *sigma_bkg*, *ra_coords*, *dec_coords*, *idex_mask*)
re-bins pixels, updates cutout image, wht_map, sigma_bkg, coordinates, PSF

Parameters **bin_size** – number of pixels (per axis) to merge

Returns

rebin_coord_transform (*factor*, *x_at_radec_0*, *y_at_radec_0*, *Mpix2coord*, *Mcoord2pix*)
adopt coordinate system and transformation between angular and pixel coordinates of a re-binned image

stack_images (*image_list*, *wht_list*, *sigma_list*)
stacks images and saves new image as a fits file

Returns

cut_edges (*image*, *num_pix*)
cuts out the edges of a 2d image and returns re-sized image to numPix center is well defined for odd pixel sizes.

Parameters

- **image** – 2d numpy array
- **num_pix** – square size of cut out image

Returns cutout image with size numPix

radial_profile (*data*, *center*)
computes radial profile

Parameters

- **data** – 2d numpy array
- **center** – center [x, y] from which pixel to compute the radial profile

Returns radial profile (in units pixel)

gradient_map (*image*)
computes gradients of images with the sobel transform

Parameters **image** – 2d numpy array

Returns array of same size as input, with gradients between neighboring pixels

lenstronomy.Util.kernel_util module

routines that manipulate convolution kernels

de_shift_kernel (*kernel*, *shift_x*, *shift_y*, *iterations=20*, *fractional_step_size=1*)
de-shifts a shifted kernel to the center of a pixel. This is performed iteratively.

The input kernel is the solution of a linear interpolated shift of a sharper kernel centered in the middle of the pixel. To find the de-shifted kernel, we perform an iterative correction of proposed de-shifted kernels and compare its shifted version with the input kernel.

Parameters

- **kernel** – (shifted) kernel, e.g. a star in an image that is not centered in the pixel grid
- **shift_x** – x-offset relative to the center of the pixel (sub-pixel shift)
- **shift_y** – y-offset relative to the center of the pixel (sub-pixel shift)
- **iterations** – number of repeated iterations of shifting a new de-shifted kernel and apply corrections
- **fractional_step_size** – float (0, 1] correction factor relative to previous proposal (can be used for stability

Returns de-shifted kernel such that the interpolated shift boy (shift_x, shift_y) results in the input kernel

center_kernel (*kernel*, *iterations=20*)
given a kernel that might not be perfectly centered, this routine computes its light weighted center and then moves the center in an iterative process such that it is centered

Parameters

- **kernel** – 2d array (odd numbers)

- **iterations** – int, number of iterations

Returns centered kernel

kernel_norm(*kernel*)

Parameters **kernel** –

Returns normalisation of the psf kernel

subgrid_kernel(*kernel*, *subgrid_res*, *odd=False*, *num_iter=100*)
creates a higher resolution kernel with subgrid resolution as an interpolation of the original kernel in an iterative approach

Parameters

- **kernel** (*2d numpy array with square odd size*) – initial kernel
- **subgrid_res** (*integer*) – subgrid resolution required
- **odd** (*boolean*) – forces odd axis size return (-1 in size if even)
- **num_iter** (*integer*) – number of iterations in the de-shifting and enhancement

Returns kernel with higher resolution (larger)

Return type 2d numpy array with n x subgrid size (-1 if result is even and odd=True)

kernel_pixelsize_change(*kernel*, *deltaPix_in*, *deltaPix_out*)
change the pixel size of a given kernel

Parameters

- **kernel** –
- **deltaPix_in** –
- **deltaPix_out** –

Returns

cut_psf(*psf_data*, *psf_size*, *normalisation=True*)
cut the psf properly

Parameters

- **psf_data** – image of PSF
- **psf_size** – size of psf

Returns re-sized and re-normalized PSF

pixel_kernel(*point_source_kernel*, *subgrid_res=7*)
converts a pixelised kernel of a point source to a kernel representing a uniform extended pixel

Parameters

- **point_source_kernel** –
- **subgrid_res** –

Returns convolution kernel for an extended pixel

kernel_average_pixel(*kernel_super*, *supersampling_factor*)
computes the effective convolution kernel assuming a uniform surface brightness on the scale of a pixel

Parameters

- **kernel_super** – supersampled PSF of a point source (odd number per axis

- **supersampling_factor** – supersampling factor (int)

Returns

kernel_gaussian (*kernel_numPix*, *deltaPix*, *fwhm*)

split_kernel (*kernel_super*, *supersampling_kernel_size*, *supersampling_factor*, *normalized=True*)
pixel kernel and subsampling kernel such that the convolution of both applied on an image can be performed, i.e. smaller subsampling PSF and hole in larger PSF

Parameters

- **kernel_super** – super-sampled kernel
- **supersampling_kernel_size** – size of super-sampled PSF in units of degraded pixels
- **normalized** – boolean, if True returns a split kernel that is area normalized=1 representing a convolution kernel

Returns degraded kernel with hole and super-sampled kernel

degrade_kernel (*kernel_super*, *degrading_factor*)

Parameters

- **kernel_super** – higher resolution kernel (odd number per axis)
- **degrading_factor** – degrading factor (effectively the super-sampling resolution of the kernel given

Returns degraded kernel with odd axis number with the sum of the flux/values in the kernel being preserved

averaging_even_kernel (*kernel_high_res*, *subgrid_res*)
makes a lower resolution kernel based on the kernel_high_res (odd numbers) and the subgrid_res (even number), both meant to be centered.

Parameters

- **kernel_high_res** – high resolution kernel with even subsampling resolution, centered
- **subgrid_res** – subsampling resolution (even number)

Returns averaged undersampling kernel

cutout_source (*x_pos*, *y_pos*, *image*, *kernelsize*, *shift=True*)
cuts out point source (e.g. PSF estimate) out of image and shift it to the center of a pixel

Parameters

- **x_pos** –
- **y_pos** –
- **image** –
- **kernelsize** –

Returns

fwhm_kernel (*kernel*)

Parameters **kernel** –

Returns

estimate_amp (*data, x_pos, y_pos, psf_kernel*)
estimates the amplitude of a point source located at x_pos, y_pos

Parameters

- **data** –
- **x_pos** –
- **y_pos** –
- **psf_kernel** –

Returns

mge_kernel (*kernel, order=5*)
azimutal Multi-Gaussian expansion of a pixelized kernel

Parameters **kernel** – 2d numpy array

Returns

match_kernel_size (*image, size*)
matching kernel/image to a dedicated size by either expanding the image with zeros at the edges or chopping of the edges.

Parameters

- **image** – 2d array (square with odd number of pixels)
- **size** – integer (odd number)

Returns image with matched size, either by cutting or by adding zeros in the outskirts

lenstronomy.Util.mask_util module

mask_center_2d (*center_x, center_y, r, x_grid, y_grid*)

Parameters

- **center_x** – x-coordinate of center position of circular mask
- **center_y** – y-coordinate of center position of circular mask
- **r** – radius of mask in pixel values
- **x_grid** – x-coordinate grid
- **y_grid** – y-coordinate grid

Returns mask array of shape x_grid with =0 inside the radius and =1 outside

Return type array of size of input grid with integers 0 or 1

mask_azimuthal (*x, y, center_x, center_y, r*)

Parameters

- **x** – x-coordinates (1d or 2d array numpy array)
- **y** – y-coordinates (1d or 2d array numpy array)
- **center_x** – center of azimuthal mask in x
- **center_y** – center of azimuthal mask in y
- **r** – radius of azimuthal mask

Returns array with zeros outside r and ones inside azimuthal radius r

Return type array of size of input grid with integers 0 or 1

mask_ellipse (*x, y, center_x, center_y, a, b, angle*)

Parameters

- **x** – x-coordinates of pixels
- **y** – y-coordinates of pixels
- **center_x** – center of mask
- **center_y** – center of mask
- **a** – major axis
- **b** – minor axis
- **angle** – angle of major axis

Returns mask (list of zeros and ones)

Return type array of size of input grid with integers 0 or 1

mask_half_moon (*x, y, center_x, center_y, r_in, r_out, phi0=0, delta_phi=6.283185307179586*)

Parameters

- **x** –
- **y** –
- **center_x** –
- **center_y** –
- **r_in** –
- **r_out** –
- **phi0** –
- **delta_phi** –

Returns

Return type array of size of input grid with integers 0 or 1

lenstronomy.Util.multi_gauss_expansion module

gaussian (*R, sigma, amp*)

Parameters

- **R** – radius
- **sigma** – gaussian sigma
- **amp** – normalization

Returns Gaussian function

mge_1d (*r_array, flux_r, N=20, linspace=False*)

Parameters

- **r_array** – list or radii (numpy array)

- **flux_r** – list of flux values (numpy array)
- **N** – number of Gaussians

Returns amplitudes and Gaussian sigmas for the best 1d flux profile

de_projection_3d(*amplitudes*, *sigmas*)
de-projects a gaussian (or list of multiple Gaussians from a 2d projected to a 3d profile) :param amplitudes: :param sigmas: :return:

lenstronomy.Util.numba_util module

jit(*nopython=True*, *cache=True*, *parallel=False*, *fastmath=False*, *error_model='numpy'*, *inline='never'*)

generated_jit(*nopython=True*, *cache=True*, *parallel=False*, *fastmath=False*, *error_model='numpy'*)

Wrapper around numba.generated_jit. Allows you to redirect a function to another based on its type

- see the Numba docs for more info

nan_to_num
Implements a Numba equivalent to np.nan_to_num (with copy=False!) array or scalar in Numba. Behaviour is the same as np.nan_to_num with copy=False, although it only supports 1-dimensional arrays and scalar inputs.

nan_to_num_arr
Part of the Numba implementation of np.nan_to_num - see nan_to_num

nan_to_num_single
Part of the Numba implementation of np.nan_to_num - see nan_to_num

lenstronomy.Util.param_util module

cart2polar(*x*, *y*, *center_x=0*, *center_y=0*)
transforms cartesian coords [x,y] into polar coords [r,phi] in the frame of the lens center

Parameters

- **x** (*array of size (n)*) – set of x-coordinates
- **y** (*array of size (n)*) – set of x-coordinates
- **center_x** (*float*) – rotation point
- **center_y** (*float*) – rotation point

Returns array of same size with coords [r,phi]

polar2cart(*r*, *phi*, *center*)
transforms polar coords [r,phi] into cartesian coords [x,y] in the frame of the lense center

Parameters

- **r** (*array of size n or float*) – radial coordinate (distance) to the center
- **phi** (*array of size n or float*) – angular coordinate
- **center** (*array of size (2)*) – rotation point

Returns array of same size with coords [x,y]

Raises AttributeError, KeyError

shear_polar2cartesian(*phi*, *gamma*)

Parameters

- **phi** – shear angle (radian)
- **gamma** – shear strength

Returns shear components gamma1, gamma2

shear_cartesian2polar (*gamma1*, *gamma2*)

Parameters

- **gamma1** – cartesian shear component
- **gamma2** – cartesian shear component

Returns shear angle, shear strength

phi_q2_ellipticity

transforms orientation angle and axis ratio into complex ellipticity moduli e1, e2

Parameters

- **phi** – angle of orientation (in radian)
- **q** – axis ratio minor axis / major axis

Returns eccentricities e1 and e2 in complex ellipticity moduli

ellipticity2phi_q

transforms complex ellipticity moduli in orientation angle and axis ratio

Parameters

- **e1** – eccentricity in x-direction
- **e2** – eccentricity in xy-direction

Returns angle in radian, axis ratio (minor/major)

transform_e1e2_product_average (*x*, *y*, *e1*, *e2*, *center_x*, *center_y*)

maps the coordinates x, y with eccentricities e1 e2 into a new elliptical coordinate system such that R = sqrt(R_major * R_minor)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate
- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – center of distortion
- **center_y** – center of distortion

Returns distorted coordinates x', y'

transform_e1e2_square_average (*x*, *y*, *e1*, *e2*, *center_x*, *center_y*)

maps the coordinates x, y with eccentricities e1 e2 into a new elliptical coordinate system such that R = sqrt(R_major**2 + R_minor**2)

Parameters

- **x** – x-coordinate
- **y** – y-coordinate

- **e1** – eccentricity
- **e2** – eccentricity
- **center_x** – center of distortion
- **center_y** – center of distortion

Returns distorted coordinates x', y'

lenstronomy.Util.prob_density module

class SkewGaussian

Bases: object

class for the Skew Gaussian distribution

map_mu_sigma_skw (*mu*, *sigma*, *skw*)

map to parameters e, w, a

Parameters

- **mu** – mean
- **sigma** – standard deviation
- **skw** – skewness

Returns e, w, a

pdf (*x*, *e=0.0*, *w=1.0*, *a=0.0*)

probability density function see: https://en.wikipedia.org/wiki/Skew_normal_distribution

Parameters

- **x** – input value
- **e** –
- **w** –
- **a** –

Returns

pdf_skew (*x*, *mu*, *sigma*, *skw*)

function with different parameterisation

Parameters

- **x** –
- **mu** – mean
- **sigma** – sigma
- **skw** – skewness

Returns

class KDE1D (*values*)

Bases: object

class that allows to compute likelihoods based on a 1-d posterior sample

__init__ (*values*)

Parameters **values** – 1d numpy array of points representing a PDF

likelihood(*x*)

Parameters **x** – position where to evaluate the density

Returns likelihood given the sample distribution

compute_lower_upper_errors(*sample*, *num_sigma=1*)
computes the upper and lower sigma from the median value. This functions gives good error estimates for skewed pdf's

Parameters

- **sample** – 1-D sample
- **num_sigma** – integer, number of sigmas to be returned

Returns median, lower_sigma, upper_sigma

lenstronomy.Util.sampling_util module

unit2uniform(*x*, *vmin*, *vmax*)
mapping from uniform distribution on parameter space to uniform distribution on unit hypercube

uniform2unit(*theta*, *vmin*, *vmax*)
mapping from uniform distribution on unit hypercube to uniform distribution on parameter space

cube2args_uniform(*cube*, *lowers*, *uppers*, *num_dims*, *copy=False*)
mapping from uniform distribution on unit hypercube 'cube' to uniform distribution on parameter space

Parameters

- **cube** – list or 1D-array of parameter values on unit hypercube
- **lowers** – lower bounds for each parameter
- **uppers** – upper bounds for each parameter
- **num_dims** – parameter space dimension (= number of parameters)
- **copy** – If False, this function modifies 'cube' in-place. Default to False.

Returns hypercube mapped to parameters space

cube2args_gaussian(*cube*, *lowers*, *uppers*, *means*, *sigmas*, *num_dims*, *copy=False*)
mapping from uniform distribution on unit hypercube 'cube' to truncated gaussian distribution on parameter space, with mean 'mu' and std dev 'sigma'

Parameters

- **cube** – list or 1D-array of parameter values on unit hypercube
- **lowers** – lower bounds for each parameter
- **uppers** – upper bounds for each parameter
- **means** – gaussian mean for each parameter
- **sigmas** – gaussian std deviation for each parameter
- **num_dims** – parameter space dimension (= number of parameters)
- **copy** – If False, this function modifies 'cube' in-place. Default to False.

Returns hypercube mapped to parameters space

scale_limits (*lowers*, *uppers*, *scale*)

sample_ball (*p0*, *std*, *size=1*, *dist='uniform'*)
Produce a ball of walkers around an initial parameter value. this routine is from the emcee package as it became deprecated there

Parameters

- **p0** – The initial parameter values (array).
- **std** – The axis-aligned standard deviation (array).
- **size** – The number of samples to produce.
- **dist** – string, specifies the distribution being sampled, supports 'uniform' and 'normal'

sample_ball_truncated (*mean*, *sigma*, *lower_limit*, *upper_limit*, *size*)
samples gaussian ball with truncation at lower and upper limit of the distribution

Parameters

- **mean** – numpy array, mean of the distribution to be sampled
- **sigma** – numpy array, sigma of the distribution to be sampled
- **lower_limit** – numpy array, lower bound of to be sampled distribution
- **upper_limit** – numpy array, upper bound of to be sampled distribution
- **size** – number of tuples to be sampled

Returns realization of truncated normal distribution with shape (size, dim(parameters))

lenstronomy.Util.simulation_util module

data_configure_simple (*numPix*, *deltaPix*, *exposure_time=None*, *background_rms=None*, *center_ra=0*, *center_dec=0*, *inverse=False*)
configures the data keyword arguments with a coordinate grid centered at zero.

Parameters

- **numPix** – number of pixel (numPix x numPix)
- **deltaPix** – pixel size (in angular units)
- **exposure_time** – exposure time
- **background_rms** – background noise (Gaussian sigma)
- **center_ra** – RA at the center of the image
- **center_dec** – DEC at the center of the image
- **inverse** – if True, coordinate system is ra to the left, if False, to the right

Returns keyword arguments that can be used to construct a Data() class instance of lenstronomy

simulate_simple (*image_model_class*, *kwargs_lens=None*, *kwargs_source=None*, *kwargs_lens_light=None*, *kwargs_ps=None*, *no_noise=False*, *source_add=True*, *lens_light_add=True*, *point_source_add=True*)

Parameters

- **image_model_class** –
- **kwargs_lens** –

- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –
- **no_noise** –
- **source_add** –
- **lens_light_add** –
- **point_source_add** –

Returns

lenstronomy.Util.util module

merge_dicts (**dict_args*)
Given any number of dicts, shallow copy and merge into a new dict, precedence goes to key value pairs in latter dicts.

approx_theta_E (*ximg, yimg*)

sort_image_index (*ximg, yimg, xref, yref*)

Parameters

- **ximg** – x coordinates to sort
- **yimg** – y coordinates to sort
- **xref** – reference x coordinate
- **yref** – reference y coordinate

Returns indexes such that ximg[indexes],yimg[indexes] matches xref,yref

rotate

Parameters

- **xcoords** – x points
- **ycoords** – y points
- **angle** – angle in radians

Returns x points and y points rotated ccw by angle theta

map_coord2pix (*ra, dec, x_0, y_0, M*)
this routines performs a linear transformation between two coordinate systems. Mainly used to transform angular into pixel coordinates in an image

Parameters

- **ra** – ra coordinates
- **dec** – dec coordinates
- **x_0** – pixel value in x-axis of ra,dec = 0,0
- **y_0** – pixel value in y-axis of ra,dec = 0,0
- **M** – 2x2 matrix to transform angular to pixel coordinates

Returns transformed coordinate systems of input ra and dec

array2image(*array*, *nx=0*, *ny=0*)

returns the information contained in a 1d array into an n*n 2d array (only works when length of array is n**2, or nx and ny are provided)

Parameters **array** (*array of size n**2*) – image values

Returns 2d array

Raises AttributeError, KeyError

image2array(*image*)

returns the information contained in a 2d array into an n*n 1d array

Parameters **image** (*array of size (n,n)*) – image values

Returns 1d array

Raises AttributeError, KeyError

array2cube(*array*, *n_1*, *n_23*)

returns the information contained in a 1d array of shape (n_1*n_23*n_23) into 3d array with shape (n_1, sqrt(n_23), sqrt(n_23))

Parameters

- **array** (*1d array*) – image values
- **n_1** (*int*) – first dimension of returned array
- **n_23** (*int*) – square of second and third dimensions of returned array

Returns 3d array

Raises **ValueError** – when n_23 is not a perfect square

cube2array(*cube*)

returns the information contained in a 3d array of shape (n_1, n_2, n_3) into 1d array with shape (n_1*n_2*n_3)

Parameters **cube** (*3d array*) – image values

Returns 1d array

make_grid(*numPix*, *deltapix*, *subgrid_res=1*, *left_lower=False*)

creates pixel grid (in 1d arrays of x- and y- positions) default coordinate frame is such that (0,0) is in the center of the coordinate grid

Parameters

- **numPix** – number of pixels per axis Give an integers for a square grid, or a 2-length sequence (first, second axis length) for a non-square grid.
- **deltapix** – pixel size
- **subgrid_res** – sub-pixel resolution (default=1)

Returns x, y position information in two 1d arrays

make_grid_transformed(*numPix*, *Mpix2Angle*)

returns grid with linear transformation (deltaPix and rotation)

Parameters

- **numPix** – number of Pixels
- **Mpix2Angle** – 2-by-2 matrix to mat a pixel to a coordinate

Returns coordinate grid

make_grid_with_coordtransform(*numPix*, *deltapix*, *subgrid_res=1*, *center_ra=0*, *center_dec=0*, *left_lower=False*, *inverse=True*)
same as make_grid routine, but returns the transformation matrix and shift between coordinates and pixel

Parameters

- **numPix** – number of pixels per axis
- **deltapix** – pixel scale per axis
- **subgrid_res** – super-sampling resolution relative to the stated pixel size
- **center_ra** – center of the grid
- **center_dec** – center of the grid
- **left_lower** – sets the zero point at the lower left corner of the pixels
- **inverse** – bool, if true sets East as left, otherwise East is righrt

Returns ra_grid, dec_grid, ra_at_xy_0, dec_at_xy_0, x_at_radec_0, y_at_radec_0, Mpix2coord, Mcoord2pix

grid_from_coordinate_transform(*nx*, *ny*, *Mpix2coord*, *ra_at_xy_0*, *dec_at_xy_0*)
return a grid in x and y coordinates that satisfy the coordinate system

Parameters

- **nx** – number of pixels in x-axis
- **ny** – number of pixels in y-axis
- **Mpix2coord** – transformation matrix (2x2) of pixels into coordinate displacements
- **ra_at_xy_0** – RA coordinate at (x,y) = (0,0)
- **dec_at_xy_0** – DEC coordinate at (x,y) = (0,0)

Returns RA coordinate grid, DEC coordinate grid

get_axes(*x*, *y*)
computes the axis x and y of a given 2d grid

Parameters

- **x** –
- **y** –

Returns

averaging(*grid*, *numGrid*, *numPix*)
resize 2d pixel grid with numGrid to numPix and averages over the pixels

Parameters

- **grid** – higher resolution pixel grid
- **numGrid** – number of pixels per axis in the high resolution input image
- **numPix** – lower number of pixels per axis in the output image (numGrid/numPix is integer number)

Returns averaged pixel grid

displaceAbs(*x*, *y*, *sourcePos_x*, *sourcePos_y*)
calculates a grid of distances to the observer in angel

Parameters

- **x** (*numpy array*) – cartesian coordinates
- **y** (*numpy array*) – cartesian coordinates
- **sourcePos_x** (*float*) – source position
- **sourcePos_y** (*float*) – source position

Returns array of displacement

Raises AttributeError, KeyError

get_distance (*x_mins*, *y_mins*, *x_true*, *y_true*)

Parameters

- **x_mins** –
- **y_mins** –
- **x_true** –
- **y_true** –

Returns

compare_distance (*x_mapped*, *y_mapped*)

Parameters

- **x_mapped** – array of x-positions of remapped catalogue image
- **y_mapped** – array of y-positions of remapped catalogue image

Returns sum of distance square of positions

min_square_dist (*x_1*, *y_1*, *x_2*, *y_2*)

return minimum of quadratic distance of pairs (x1, y1) to pairs (x2, y2)

Parameters

- **x_1** –
- **y_1** –
- **x_2** –
- **y_2** –

Returns

selectBest (*array*, *criteria*, *numSelect*, *highest=True*)

Parameters

- **array** – numpy array to be selected from
- **criteria** – criteria of selection
- **highest** – bool, if false the lowest will be selected
- **numSelect** – number of elements to be selected

Returns

select_best (*array*, *criteria*, *num_select*, *highest=True*)

Parameters

- **array** – numpy array to be selected from

- **criteria** – criteria of selection
- **highest** – bool, if false the lowest will be selected
- **num_select** – number of elements to be selected

Returns

points_on_circle(*radius*, *num_points*, *connect_ends=True*)
returns a set of uniform points around a circle

Parameters

- **radius** – radius of the circle
- **num_points** – number of points on the circle
- **connect_ends** – boolean, if True, start and end point are the same

Returns x-coords, y-coords of points on the circle

local_minima_2d
finds (local) minima in a 2d grid applies less rigid criteria for maximum without second-order tangential minima criteria

Parameters

- **a** (*numpy array with length numPix**2 in float*) – 1d array of displacements from the source positions
- **x** (*numpy array with length numPix**2 in float*) – 1d coordinate grid in x-direction
- **y** (*numpy array with length numPix**2 in float*) – 1d coordinate grid in x-direction

Returns array of indices of local minima, values of those minima

Raises AttributeError, KeyError

neighborSelect
#TODO replace by from scipy.signal import argrelextrema for speed up >>> from scipy.signal import argrelextrema >>> x = np.array([2, 1, 2, 3, 2, 0, 1, 0]) >>> argrelextrema(x, np.greater) (array([3, 6]),) >>> y = np.array([[1, 2, 1, 2], … [2, 2, 0, 0], … [5, 3, 4, 4]]) … >>> argrelextrema(y, np.less, axis=1) (array([0, 2]), array([2, 1]))

finds (local) minima in a 2d grid

Parameters **a** (*numpy array with length numPix**2 in float*) – 1d array of displacements from the source positions

Returns array of indices of local minima, values of those minima

Raises AttributeError, KeyError

fwhm2sigma(*fwhm*)

Parameters **fwhm** – full-width-half-max value

Returns gaussian sigma (sqrt(var))

sigma2fwhm(*sigma*)

Parameters **sigma** –

Returns

hyper2F2_array(*a*, *b*, *c*, *d*, *x*)

Parameters

- **a** –
- **b** –
- **c** –
- **d** –
- **x** –

Returns

make_subgrid(*ra_coord*, *dec_coord*, *subgrid_res=2*)
return a grid with subgrid resolution

Parameters

- **ra_coord** –
- **dec_coord** –
- **subgrid_res** –

Returns

convert_bool_list(*n*, *k=None*)
returns a bool list of the length of the lens models

if k = None: returns bool list with True's if k is int, returns bool list with False's but k'th is True if k is a list of int, e.g. [0, 3, 5], returns a bool list with True's in the integers listed and False elsewhere if k is a boolean list, checks for size to match the numbers of models and returns it

Parameters

- **n** – integer, total lenght of output boolean list
- **k** – None, int, or list of ints

Returns bool list

Module contents

lenstronomy.Workflow package

Submodules

lenstronomy.Workflow.alignment_matching module

class AlignmentFitting(*multi_band_list*, *kwargs_model*, *kwargs_params*, *band_index=0*, *likelihood_mask_list=None*)
Bases: `object`

class which executes the different sampling methods

__init__(*multi_band_list*, *kwargs_model*, *kwargs_params*, *band_index=0*, *likelihood_mask_list=None*)
initialise the classes of the chain and for parameter options

pso(*n_particles=10*, *n_iterations=10*, *lowerLimit=-0.2*, *upperLimit=0.2*, *threadCount=1*, *mpi=False*, *print_key='default'*)
returns the best fit for the lense model on catalogue basis with particle swarm optimizer

class AlignmentLikelihood(*multi_band_list*, *kwargs_model*, *kwargs_params*, *band_index=0*, *likelihood_mask_list=None*)
Bases: `object`

__init__(*multi_band_list*, *kwargs_model*, *kwargs_params*, *band_index=0*, *likelihood_mask_list=None*)
initializes all the classes needed for the chain

computeLikelihood(*ctx*)

static get_args(*kwargs_data*)

Parameters **kwargs_data** –

Returns

likelihood(*a*)

num_param

setup()

update_data(*args*)

Parameters **args** –

Returns

update_multi_band(*args*)

Parameters **args** – list of parameters

Returns updated multi_band_list

lenstronomy.Workflow.fitting_sequence module

class FittingSequence(*kwargs_data_joint*, *kwargs_model*, *kwargs_constraints*, *kwargs_likelihood*, *kwargs_params*, *mpi=False*, *verbose=True*)
Bases: `object`

class to define a sequence of fitting applied, inherit the Fitting class this is a Workflow manager that allows to update model configurations before executing another step in the modelling The user can take this module as an example of how to create their own workflows or build their own around the FittingSequence

__init__(*kwargs_data_joint*, *kwargs_model*, *kwargs_constraints*, *kwargs_likelihood*, *kwargs_params*, *mpi=False*, *verbose=True*)

Parameters

- **kwargs_data_joint** – keyword argument specifying the data according to LikelihoodModule
- **kwargs_model** – keyword arguments to describe all model components used in class_creator.create_class_instances()
- **kwargs_constraints** – keyword arguments of the Param() class to handle parameter constraints during the sampling (except upper and lower limits and sampling input mean and width)
- **kwargs_likelihood** – keyword arguments of the Likelihood() class to handle parameters and settings of the likelihood
- **kwargs_params** – setting of the sampling bounds and initial guess mean and spread. The argument is organized as: 'lens_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'source_model': [kwargs_init, kwargs_sigma,

kwargs_fixed, kwargs_lower, kwargs_upper] 'lens_light_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'point_source_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'extinction_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'special': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper]

- **mpi** – MPI option (bool), if True, will launch an MPI Pool job for the steps in the fitting sequence where possible
- **verbose** – bool, if True prints temporary results and indicators of the fitting process

align_images (*n_particles=10*, *n_iterations=10*, *lowerLimit=-0.2*, *upperLimit=0.2*, *threadCount=1*, *compute_bands=None*)

aligns the coordinate systems of different exposures within a fixed model parameterisation by executing a PSO with relative coordinate shifts as free parameters

Parameters

- **n_particles** – number of particles in the Particle Swarm Optimization
- **n_iterations** – number of iterations in the optimization process
- **lowerLimit** – lower limit of relative shift
- **upperLimit** – upper limit of relative shift
- **compute_bands** – bool list, if multiple bands, this process can be limited to a subset of bands for which the coordinate system is being fit for best alignment to the model parameters

Returns 0, updated coordinate system for the band(s)

best_fit (*bijective=False*)

Parameters **bijective** – bool, if True, the mapping of image2source_plane and the mass_scaling parameterisation are inverted. If you do not use those options, there is no effect.

Returns best fit model of the current state of the FittingSequence class

best_fit_from_samples (*samples*, *logl*)

return best fit (max likelihood) value of samples in lenstronomy conventions

Parameters

- **samples** – samples of multi-dimensional parameter space
- **logl** – likelihood values for each sample

Returns kwargs_result in lenstronomy convention

best_fit_likelihood

returns the log likelihood of the best fit model of the current state of this class

Returns log likelihood, float

bic

Bayesian information criterion (BIC) of the model.

Returns bic value, float

fit_sequence (*fitting_list*)

Parameters **fitting_list** – list of [['string', {kwargs}], ..] with 'string being the specific fitting option and kwargs being the arguments passed to this option

Returns fitting results

fix_not_computed(*free_bands*)
fixes lens model parameters of imaging bands/frames that are not computed and frees the parameters of the other lens models to the initial kwargs_fixed options

Parameters **free_bands** – bool list of length of imaging bands in order of imaging bands, if False: set fixed lens model

Returns None

kwargs_fixed()
returns the updated kwargs_fixed from the update Manager

Returns list of fixed kwargs, see UpdateManager()

likelihoodModule

Returns Likelihood() class instance reflecting the current state of FittingSequence

mcmc(*n_burn*, *n_run*, *walkerRatio=None*, *n_walkers=None*, *sigma_scale=1*, *threadCount=1*, *init_samples=None*, *re_use_samples=True*, *sampler_type='EMCEE'*, *progress=True*, *backend_filename=None*, *start_from_backend=False*, ***kwargs_zeus*)
MCMC routine

Parameters

- **n_burn** – number of burn in iterations (will not be saved)
- **n_run** – number of MCMC iterations that are saved
- **walkerRatio** – ratio of walkers/number of free parameters
- **n_walkers** – integer, number of walkers of emcee (optional, if set, overwrites the walkerRatio input
- **sigma_scale** – scaling of the initial parameter spread relative to the width in the initial settings
- **threadCount** – number of CPU threads. If MPI option is set, threadCount=1
- **init_samples** – initial sample from where to start the MCMC process
- **re_use_samples** – bool, if True, re-uses the samples described in init_samples.nOtherwise starts from scratch.
- **sampler_type** – string, which MCMC sampler to be used. Options are: 'EMCEE', 'ZEUS'
- **progress** – boolean, if True shows progress bar in EMCEE
- **backend_filename** (`string`) – name of the HDF5 file where sampling state is saved (through emcee backend engine)
- **start_from_backend** (`bool`) – if True, start from the state saved in *backup_filename*. O therwise, create a new backup file with name *backup_filename* (any already existing file is overwritten!).
- **kwargs_zeus** – zeus-specific kwargs

Returns list of output arguments, e.g. MCMC samples, parameter names, logL distances of all samples specified by the specific sampler used

nested_sampling(*sampler_type='MULTINEST'*, *kwargs_run={}*, *prior_type='uniform'*, *width_scale=1*, *sigma_scale=1*, *output_basename='chain'*, *remove_output_dir=True*, *dypolychord_dynamic_goal=0.8*, *polychord_settings={}*, *dypolychord_seed_increment=200*, *output_dir='nested_sampling_chains'*, *dynesty_bound='multi'*, *dynesty_sample='auto'*)

Run (Dynamic) Nested Sampling algorithms, depending on the type of algorithm.

Parameters

- **sampler_type** – 'MULTINEST', 'DYPOLYCHORD', 'DYNESTY'
- **kwargs_run** – keywords passed to the core sampling method
- **prior_type** – 'uniform' of 'gaussian', for converting the unit hypercube to param cube
- **width_scale** – scale the width (lower/upper limits) of the parameters space by this factor
- **sigma_scale** – if prior_type is 'gaussian', scale the gaussian sigma by this factor
- **output_basename** – name of the folder in which the core MultiNest/PolyChord code will save output files
- **remove_output_dir** – if True, the above folder is removed after completion
- **dypolychord_dynamic_goal** – dynamic goal for DyPolyChord (trade-off between evidence (0) and posterior (1) computation)
- **polychord_settings** – settings dictionary to send to pypolychord. Check dypolychord documentation for details.
- **dypolychord_seed_increment** – seed increment for dypolychord with MPI. Check dypolychord documentation for details.
- **dynesty_bound** – see https://dynesty.readthedocs.io for details
- **dynesty_sample** – see https://dynesty.readthedocs.io for details

Returns list of output arguments : samples, mean inferred values, log-likelihood, log-evidence, error on log-evidence for each sample

param_class

Returns Param() class instance reflecting the current state of FittingSequence

psf_iteration(*compute_bands=None*, ***kwargs_psf_iter*)

iterative PSF reconstruction

Parameters

- **compute_bands** – bool list, if multiple bands, this process can be limited to a subset of bands
- **kwargs_psf_iter** – keyword arguments as used or available in PSFIteration.update_iterative() definition

Returns 0, updated PSF is stored in self.multi_band_list

pso(*n_particles*, *n_iterations*, *sigma_scale=1*, *print_key='PSO'*, *threadCount=1*)

Particle Swarm Optimization

Parameters

- **n_particles** – number of particles in the Particle Swarm Optimization
- **n_iterations** – number of iterations in the optimization process

- **sigma_scale** – scaling of the initial parameter spread relative to the width in the initial settings
- **print_key** – string, printed text when executing this routine
- **threadCount** – number of CPU threads. If MPI option is set, threadCount=1

Returns result of the best fit, the PSO chain of the best fit parameter after each iteration [lnlikelihood, parameters, velocities], list of parameters in same order as in chain

set_param_value(***kwargs*)
Set a parameter to a specific value. *kwargs* are below.

Parameters

- **lens** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **source** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **lens_light** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **ps** – [[i_model, ['param1', 'param2',. . .], [. . .]]

Returns 0, the value of the param is overwritten

Return type

simplex(*n_iterations*, *method='Nelder-Mead'*)
Downhill simplex optimization using the Nelder-Mead algorithm.

Parameters

- **n_iterations** – maximum number of iterations to perform
- **method** – the optimization method used, see documentation in scipy.optimize.minimize

Returns result of the best fit

update_settings(*kwargs_model=None*, *kwargs_constraints=None*, *kwargs_likelihood=None*, *lens_add_fixed=None*, *source_add_fixed=None*, *lens_light_add_fixed=None*, *ps_add_fixed=None*, *cosmo_add_fixed=None*, *lens_remove_fixed=None*, *source_remove_fixed=None*, *lens_light_remove_fixed=None*, *ps_remove_fixed=None*, *cosmo_remove_fixed=None*, *change_source_lower_limit=None*, *change_source_upper_limit=None*, *change_lens_lower_limit=None*, *change_lens_upper_limit=None*, *change_sigma_lens=None*, *change_sigma_source=None*, *change_sigma_lens_light=None*)
updates lenstronomy settings "on the fly"

Parameters

- **kwargs_model** – kwargs, specified keyword arguments overwrite the existing ones
- **kwargs_constraints** – kwargs, specified keyword arguments overwrite the existing ones
- **kwargs_likelihood** – kwargs, specified keyword arguments overwrite the existing ones
- **lens_add_fixed** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **source_add_fixed** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **lens_light_add_fixed** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **ps_add_fixed** – [[i_model, ['param1', 'param2',. . .], [. . .]]

- **cosmo_add_fixed** – [‘param1’, ‘param2’,. . .]
- **lens_remove_fixed** – [[i_model, [‘param1’, ‘param2’,. . .], [. . .]]
- **source_remove_fixed** – [[i_model, [‘param1’, ‘param2’,. . .], [. . .]]
- **lens_light_remove_fixed** – [[i_model, [‘param1’, ‘param2’,. . .], [. . .]]
- **ps_remove_fixed** – [[i_model, [‘param1’, ‘param2’,. . .], [. . .]]
- **cosmo_remove_fixed** – [‘param1’, ‘param2’,. . .]
- **change_lens_lower_limit** – [[i_model, [‘param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]
- **change_lens_upper_limit** – [[i_model, [‘param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]
- **change_source_lower_limit** – [[i_model, [‘param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]
- **change_source_upper_limit** – [[i_model, [‘’param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]
- **change_sigma_lens** – [[i_model, [‘param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]
- **change_sigma_source** – [[i_model, [‘param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]
- **change_sigma_lens_light** – [[i_model, [‘param_name1’, ‘param_name2’, . . .], [value1, value2, . . .]]]

Returns 0, the settings are overwritten for the next fitting step to come

update_state(*kwargs_update*)
updates current best fit state to the input model keywords specified.

Parameters **kwargs_update** – format of kwargs_result

Returns None

lenstronomy.Workflow.psf_fitting module

class PsfFitting(*image_model_class*)
Bases: `object`

class to find subsequently a better psf The method make use of a model and subtracts all the non-point source components of the model from the data. If the model is sufficient, then the data will be a (better) representation of the actual PSF. The method cuts out those point sources and combines them to update the estimate of the PSF. This is done in an iterative procedure as the model components of the extended features is PSF-dependent (hopefully not too much).

Various options can be chosen. There is no guarantee that the method works for specific data and models.

‘stacking_method’: ‘median’, ‘mean’; the different estimates of the PSF are stacked and combined together. The choices are: - ‘mean’: mean of pixel values as the estimator (not robust to outliers) - ‘median’: median of pixel values as the estimator (outlier rejection robust but needs >2 point sources in the data

‘block_center_neighbour’: angle, radius of neighbouring point sources around their centers the estimates is ignored.
Default is zero, meaning a not optimal subtraction of the neighbouring point sources might contaminate the estimate.

'keep_error_map': bool, if True, does not replace the error term associated with the PSF estimate. If false, re-estimates the variance between the PSF estimates.

'psf_symmetry': number of rotational invariant symmetries in the estimated PSF. =1 mean no additional symmetries. =4 means 90 deg symmetry. This is enforced by a rotatioanl stack according to the symmetry specified. These additional imposed symmetries can help stabelize the PSF estimate when there are limited constraints/number of point sources in the image.

The procedure only requires and changes the 'point_source_kernel' in the PSF() class and the 'psf_error_map'. Any previously set subgrid kernels or pixel_kernels are removed and constructed from the 'point_source_kernel'.

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

`static combine_psf`(*kernel_list_new*, *kernel_old*, *factor=1.0*, *stacking_option='median'*, *symmetry=1*)
updates psf estimate based on old kernel and several new estimates

Parameters

- **`kernel_list_new`** – list of new PSF kernels estimated from the point sources in the image (un-normalized)
- **`kernel_old`** – old PSF kernel
- **`factor`** – weight of updated estimate based on new and old estimate, factor=1 means new estimate, factor=0 means old estimate
- **`stacking_option`** – option of stacking, mean or median
- **`symmetry`** – imposed symmetry of PSF estimate

Returns updated PSF estimate and error_map associated with it

`cutout_psf`(*ra_image*, *dec_image*, *x*, *y*, *image_list*, *kernel_size*, *kernel_init*, *block_center_neighbour=0*)

Parameters

- **`ra_image`** – coordinate array of images in angles
- **`dec_image`** – coordinate array of images in angles
- **`x`** – image position array in x-pixel
- **`y`** – image position array in y-pixel
- **`image_list`** – list of images (i.e. data - all models subtracted, except a single point source)
- **`kernel_size`** – width in pixel of the kernel
- **`kernel_init`** – initial guess of kernel (pixels that are masked are replaced by those values)
- **`block_center_neighbour`** – angle, radius of neighbouring point sources around their centers the estimates is ignored. Default is zero, meaning a not optimal subtraction of the neighbouring point sources might contaminate the estimate.

Returns list of de-shifted kernel estimates

`static cutout_psf_single`(*x*, *y*, *image*, *mask*, *kernel_size*, *kernel_init*)

Parameters

- **`x`** – x-coordinate of point source

- **y** – y-coordinate of point source
- **image** – image (i.e. data - all models subtracted, except a single point source)
- **mask** – mask of pixels in the image not to be considered in the PSF estimate (being replaced by kernel_init)
- **kernel_size** – width in pixel of the kernel
- **kernel_init** – initial guess of kernel (pixels that are masked are replaced by those values)

Returns estimate of the PSF based on the image and position of the point source

error_map_estimate(*kernel*, *star_cutout_list*, *amp*, *x_pos*, *y_pos*, *error_map_radius=None*, *block_center_neighbour=0*)

provides a psf_error_map based on the goodness of fit of the given PSF kernel on the point source cutouts, their estimated amplitudes and positions

Parameters

- **kernel** – PSF kernel
- **star_cutout_list** – list of 2d arrays of cutouts of the point sources with all other model components subtracted
- **amp** – list of amplitudes of the estimated PSF kernel
- **x_pos** – pixel position (in original data unit, not in cutout) of the point sources (same order as amp and star cutouts)
- **y_pos** – pixel position (in original data unit, not in cutout) of the point sources (same order as amp and star cutouts)
- **error_map_radius** – float, radius (in arc seconds) of the outermost error in the PSF estimate (e.g. to avoid double counting of overlapping PSF erros)
- **block_center_neighbour** – angle, radius of neighbouring point sources around their centers the estimates is ignored. Default is zero, meaning a not optimal subtraction of the neighbouring point sources might contaminate the estimate.

Returns relative uncertainty in the psf model (in quadrature) per pixel based on residuals achieved in the image

error_map_estimate_new(*psf_kernel*, *psf_kernel_list*, *ra_image*, *dec_image*, *point_amp*, *supersampling_factor*, *error_map_radius=None*)

relative uncertainty in the psf model (in quadrature) per pixel based on residuals achieved in the image

Parameters

- **psf_kernel** – PSF kernel (super-sampled)
- **psf_kernel_list** – list of individual best PSF kernel estimates
- **ra_image** – image positions in angles
- **dec_image** – image positions in angles
- **point_amp** – image amplitude
- **supersampling_factor** – super-sampling factor
- **error_map_radius** – radius (in angle) to cut the error map

Returns psf error map such that square of the uncertainty gets boosted by error_map * (psf * amp)**2

image_single_point_source (*image_model_class*, *kwargs_params*)
return model without including the point source contributions as a list (for each point source individually)

Parameters

- **image_model_class** – ImageModel class instance
- **kwargs_params** – keyword arguments of model component keyword argument lists

Returns list of images with point source isolated

static mask_point_source (*x_pos*, *y_pos*, *x_grid*, *y_grid*, *radius*, *i=0*)

Parameters

- **x_pos** – x-position of list of point sources
- **y_pos** – y-position of list of point sources
- **x_grid** – x-coordinates of grid
- **y_grid** – y-coordinates of grid
- **i** – index of point source not to mask out
- **radius** – radius to mask out other point sources

Returns a mask of the size of the image with cutouts around the position

static point_like_source_cutouts (*x_pos*, *y_pos*, *image_list*, *cutout_size*)
cutouts of point-like objects

Parameters

- **x_pos** – list of image positions in pixel units
- **y_pos** – list of image position in pixel units
- **image_list** – list of 2d numpy arrays with cleaned images, with all contaminating sources removed except the point-like object to be cut out.
- **cutout_size** – odd integer, size of cutout.

Returns list of cutouts

psf_estimate_individual (*ra_image*, *dec_image*, *point_amp*, *residuals*, *cutout_size*, *kernel_guess*, *supersampling_factor*, *block_center_neighbour*)

Parameters

- **ra_image** – list; position in angular units of the image
- **dec_image** – list; position in angular units of the image
- **point_amp** – list of model amplitudes of point sources
- **residuals** – data - model
- **cutout_size** – pixel size of cutout around single star/quasar to be considered for the psf reconstruction
- **kernel_guess** – initial guess of super-sampled PSF
- **supersampling_factor** – int, super-sampling factor
- **block_center_neighbour** –

Returns list of best-guess PSF’s for each star based on the residual patterns

update_iterative(*kwargs_psf*, *kwargs_params*, *num_iter=10*, *keep_psf_error_map=True*, *no_break=True*, *verbose=True*, ***kwargs_psf_update*)

Parameters

- **kwargs_psf** – keyword arguments to construct the PSF() class
- **kwargs_params** – keyword arguments of the parameters of the model components (e.g. 'kwargs_lens' etc)
- **num_iter** – number of iterations in the PSF fitting and image fitting process
- **keep_psf_error_map** – boolean, if True keeps previous psf_error_map
- **no_break** – boolean, if True, runs until the end regardless of the next step getting worse, and then reads out the overall best fit
- **verbose** – print statements informing about progress of iterative procedure
- **kwargs_psf_update** – keyword arguments providing the settings for a single iteration of the PSF, as being passed to update_psf() method

Returns keyword argument of PSF constructor for PSF() class with updated PSF

update_psf(*kwargs_psf*, *kwargs_params*, *stacking_method='median'*, *psf_symmetry=1*, *psf_iter_factor=0.2*, *block_center_neighbour=0*, *error_map_radius=None*, *block_center_neighbour_error_map=None*, *new_procedure=True*)

Parameters

- **kwargs_psf** – keyword arguments to construct the PSF() class
- **kwargs_params** – keyword arguments of the parameters of the model components (e.g. 'kwargs_lens' etc)
- **stacking_method** – 'median', 'mean'; the different estimates of the PSF are stacked and combined together. The choices are: 'mean': mean of pixel values as the estimator (not robust to outliers) 'median': median of pixel values as the estimator (outlier rejection robust but needs >2 point sources in the data
- **psf_symmetry** – number of rotational invariant symmetries in the estimated PSF. =1 mean no additional symmetries. =4 means 90 deg symmetry. This is enforced by a rotatioanl stack according to the symmetry specified. These additional imposed symmetries can help stabelize the PSF estimate when there are limited constraints/number of point sources in the image.
- **psf_iter_factor** – factor in (0, 1] of ratio of old vs new PSF in the update in the iteration.
- **block_center_neighbour** – angle, radius of neighbouring point sources around their centers the estimates is ignored. Default is zero, meaning a not optimal subtraction of the neighbouring point sources might contaminate the estimate.
- **block_center_neighbour_error_map** – angle, radius of neighbouring point sources around their centers the estimates of the ERROR MAP is ignored. If None, then the value of block_center_neighbour is used (recommended)
- **error_map_radius** – float, radius (in arc seconds) of the outermost error in the PSF estimate (e.g. to avoid double counting of overlapping PSF errors), if None, all of the pixels are considered (unless blocked through other means)
- **new_procedure** – boolean, uses post lenstronomy 1.9.2 procedure which is more optimal for super-sampled PSF's

Returns kwargs_psf_new, logL_after, error_map

lenstronomy.Workflow.update_manager module

class UpdateManager(*kwargs_model, kwargs_constraints, kwargs_likelihood, kwargs_params*)
Bases: `object`

this class manages the parameter constraints as they may evolve through the steps of the modeling. This includes: keeping certain parameters fixed during one modelling step

__init__(*kwargs_model, kwargs_constraints, kwargs_likelihood, kwargs_params*)

Parameters

- **kwargs_model** – keyword arguments to describe all model components used in class_creator.create_class_instances()
- **kwargs_constraints** – keyword arguments of the Param() class to handle parameter constraints during the sampling (except upper and lower limits and sampling input mean and width)
- **kwargs_likelihood** – keyword arguments of the Likelihood() class to handle parameters and settings of the likelihood
- **kwargs_params** – setting of the sampling bounds and initial guess mean and spread. The argument is organized as: 'lens_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'source_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'lens_light_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'point_source_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'extinction_model': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper] 'special': [kwargs_init, kwargs_sigma, kwargs_fixed, kwargs_lower, kwargs_upper]

best_fit(*bijective=False*)
best fit (max likelihood) position for all the model parameters

Parameters bijective – boolean, if True, returns the parameters in the argument of the sampling that might deviate from the convention of the ImSim module. For example, if parameterized in the image position, the parameters remain in the image plane rather than being mapped to the source plane.

Returns kwargs_result with all the keyword arguments of the best fit for the model components

fix_image_parameters(*image_index=0*)
fixes all parameters that are only assigned to a specific image. This allows to sample only parameters that constraint by the fitting of a sub-set of the images.

Parameters image_index – index

Returns None

fixed_kwargs

init_kwargs

Returns keyword arguments for all model components of the initial mean model proposition in the sampling

param_class
creating instance of lenstronomy Param() class. It uses the keyword arguments in self.kwargs_constraints as __init__() arguments, as well as self.kwargs_model, and the set of kwargs_fixed___, kwargs_lower___, kwargs_upper___ arguments for lens, lens_light, source, point source, extinction and special parameters.

Returns instance of the Param class with the recent options and bounds

parameter_state

Returns parameter state saved in this class

set_init_state()

set the current state of the parameters to the initial one.

Returns

sigma_kwargs

Returns keyword arguments for all model components of the initial 1-sigma width proposition in the sampling

update_fixed(*lens_add_fixed=None*, *source_add_fixed=None*, *lens_light_add_fixed=None*, *ps_add_fixed=None*, *special_add_fixed=None*, *lens_remove_fixed=None*, *source_remove_fixed=None*, *lens_light_remove_fixed=None*, *ps_remove_fixed=None*, *special_remove_fixed=None*)

adds or removes the values of the keyword arguments that are stated in the _add_fixed to the existing fixed arguments. convention for input arguments are: [[i_model, ['param_name1', 'param_name2', …], [value1, value2, … (optional)], [], …]

Parameters

- **lens_add_fixed** – added fixed parameter in lens model
- **source_add_fixed** – added fixed parameter in source model
- **lens_light_add_fixed** – added fixed parameter in lens light model
- **ps_add_fixed** – added fixed parameter in point source model
- **special_add_fixed** – added fixed parameter in special model
- **lens_remove_fixed** – remove fixed parameter in lens model
- **source_remove_fixed** – remove fixed parameter in source model
- **lens_light_remove_fixed** – remove fixed parameter in lens light model
- **ps_remove_fixed** – remove fixed parameter in point source model
- **special_remove_fixed** – remove fixed parameter in special model

Returns updated kwargs fixed

update_limits(*change_source_lower_limit=None*, *change_source_upper_limit=None*, *change_lens_lower_limit=None*, *change_lens_upper_limit=None*)

updates the limits (lower and upper) of the update manager instance

Parameters

- **change_source_lower_limit** – [[i_model, ['param_name1', 'param_name2', …], [value1, value2, …]]]
- **change_lens_lower_limit** – [[i_model, ['param_name1', 'param_name2', …], [value1, value2, …]]]
- **change_source_upper_limit** – [[i_model, ['param_name1', 'param_name2', …], [value1, value2, …]]]
- **change_lens_upper_limit** – [[i_model, ['param_name1', 'param_name2', …], [value1, value2, …]]]

Returns updates internal state of lower and upper limits accessible from outside

update_options (*kwargs_model=None, kwargs_constraints=None, kwargs_likelihood=None*)
updates the options by overwriting the kwargs with the new ones being added/changed WARNING: some updates may not be valid depending on the model options. Use carefully!

Parameters

- **kwargs_model** – keyword arguments to describe all model components used in class_creator.create_class_instances() that are updated from previous arguments
- **kwargs_constraints** –
- **kwargs_likelihood** –

Returns kwargs_model, kwargs_constraints, kwargs_likelihood

update_param_state (*kwargs_lens=None, kwargs_source=None, kwargs_lens_light=None, kwargs_ps=None, kwargs_special=None, kwargs_extinction=None*)
updates the temporary state of the parameters being saved. ATTENTION: Any previous knowledge gets lost if you call this function

Parameters

- **kwargs_lens** –
- **kwargs_source** –
- **kwargs_lens_light** –
- **kwargs_ps** –
- **kwargs_special** –
- **kwargs_extinction** –

Returns

update_param_value (*lens=None, source=None, lens_light=None, ps=None*)
Set a model parameter to a specific value.

Parameters

- **lens** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **source** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **lens_light** – [[i_model, ['param1', 'param2',. . .], [. . .]]
- **ps** – [[i_model, ['param1', 'param2',. . .], [. . .]]

Returns 0, the value of the param is overwritten

update_sigmas (*change_sigma_lens=None, change_sigma_source=None, change_sigma_lens_light=None*)
updates individual estimated uncertainty levels for the initialization of search and sampling algorithms

Parameters

- **change_sigma_lens** – [[i_model, ['param_name1', 'param_name2', . . .], [value1, value2, . . .]]]
- **change_sigma_source** – [[i_model, ['param_name1', 'param_name2', . . .], [value1, value2, . . .]]]
- **change_sigma_lens_light** – [[i_model, ['param_name1', 'param_name2', . . .], [value1, value2, . . .]]]

Returns updated internal state of the spread to initialize samplers

Module contents

Module contents

6.1.10 History

0.0.1 (2018-01-09)

- First release on PyPI.

0.0.2 (2018-01-16)

- Improved testing and stability

0.0.6 (2018-01-29)

- Added feature to align coordinate system of different images

0.1.0 (2018-02-25)

- Major design update

0.1.1 (2018-03-05)

- minor update to facilitate options without lensing

0.2.0 (2018-03-10)

- ellipticity parameter handling changed
- time-delay distance sampling included
- parameter handling for sampling more flexible
- removed redundancies in the light and mass profiles

0.2.1 (2018-03-19)

- updated documentation
- improved sub-sampling of the PSF

0.2.2 (2018-03-25)

- improved parameter handling
- minor bugs with parameter handling fixed

0.2.8 (2018-05-31)

- improved GalKin module
- minor improvements in PSF reconstruction
- mass-to-light ratio parameterization

0.3.1 (2018-07-21)

- subgrid psf sampling for inner parts of psf exclusively
- minor stability improvements
- cleaner likelihood definition
- additional Chameleon lens and light profiles

0.3.3 (2018-08-21)

- minor updates, better documentation and handling of parameters

0.4.1-3 (2018-11-27)

- various multi-band modelling frameworks added
- lens models added
- Improved fitting sequence, solver and psf iteration

0.5.0 (2019-1-30)

- Workflow module redesign
- improved parameter handling
- improved PSF subsampling module
- relative astrometric precision of point sources implemented

0.6.0 (2019-2-26)

- Simulation API module for mock generations
- Multi-source plane modelling

0.7.0 (2019-4-13)

- New design of Likelihood module
- Updated design of FittingSequence
- Exponential Shapelets implemented

0.8.0 (2019-5-23)

- New design of Numerics module
- New design of PSF and Data module
- New design of multi-band fitting module

0.8.1 (2019-5-23)

- PSF numerics improved and redundancies removed.

0.8.2 (2019-5-27)

- psf_construction simplified
- parameter handling for catalogue modelling improved

0.9.0 (2019-7-06)

- faster fft convolutions
- re-design of multi-plane lensing module
- re-design of plotting module
- nested samplers implemented
- Workflow module with added features

0.9.1 (2019-7-21)

- non-linear solver for 4 point sources updated
- new lens models added
- updated Workflow module
- implemented differential extinction

0.9.2 (2019-8-29)

- non-linear solver for 4 point sources updated
- Moffat PSF for GalKin in place
- Likelihood module for point sources and catalogue data improved
- Design improvements in the LensModel module
- minor stability updates

0.9.3 (2019-9-25)

- improvements in SimulationAPI design
- improvements in astrometric uncertainty handling of parameters
- local arc lens model description and differentials

1.0.0 (2019-9-25)

- marking version as 5 - Stable/production mode

1.0.1 (2019-10-01)

- compatible with emcee 3.0.0
- removed CosmoHammer MCMC sampling

1.1.0 (2019-11-5)

- plotting routines split in different files
- curved arc parameterization and eigenvector differentials
- numerical differentials as part of the LensModel core class

1.2.0 (2019-11-17)

- Analysis module re-designed
- GalKin module partially re-designed
- Added cosmography module
- parameterization of cartesian shear coefficients changed

1.2.4 (2020-01-02)

- First implementation of a LightCone module for numerical ray-tracing
- Improved cosmology sampling from time-delay cosmography measurements
- TNFW profile lensing potential implemented

1.3.0 (2020-01-10)

- image position likelihood description improved

1.4.0 (2020-03-26)

- Major re-design of GalKin module, added new anisotropy modeling and IFU aperture type
- Updated design of the Analysis.kinematicsAPI sub-module
- Convention and redundancy in the Cosmo module changed
- NIE, SIE and SPEMD model consistent with their ellipticity and Einstein radius definition
- added cored-Sersic profile
- dependency for PSO to CosmoHammer removed
- MPI and multi-threading for PSO and MCMC improved and compatible with python3

1.5.0 (2020-04-05)

- Re-naming SPEMD to PEMD, SPEMD_SMOOTH to SPEMD
- adaptive numerics improvement
- multi-processing improvements

1.5.1 (2020-06-20)

- bug fix in Hession of POINT_SOURCE model
- EPL model from Tessore et al. 2015 implemented
- multi-observation mode for kinematics calculation

1.6.0 (2020-09-07)

- SLITronomy integration
- observation configuration templates and examples
- lens equation solver arguments in single sub-kwargs
- adapted imports to latest scipy release
- iterative PSF reconstruction improved
- multipole lens model

1.7.0 (2020-12-16)

- cosmo.NFWParam mass definition changed
- QuadOptimizer re-factored
- interpol light model support for non-square grid
- add functionality to psf error map
- fix in multiband reconstruction
- observational config for ZTF
- short-hand class imports

1.8.0 (2021-03-21)

- EPL numba version
- numba configuration variables can be set globally with configuration file
- Series of curved arc models available
- single plane hessian return all for differentials
- elliptical density slice lens model
- vectorized lens and light interpolation models
- updated installation description
- fast caustic calculation replacing matplotlib with skitlearn
- multi-patch illustration class and plotting routines
- updated PSF iteration procedure with more settings

1.8.1 (2021-04-19)

- illustration plots for curved arcs updated
- documentation of elliptical lens models updated

1.8.2 (2021-06-08)

- JOSS paper added
- improved testing documentation and tox compatibility
- TNFW_ELLIPSE lens model implemented
- ULDM lens model implemented

1.9.0 (2021-07-15)

- re-defined half light radius in Sersic profile
- re-named parameter in 'CONVERGENCE' profile
- improved numerics in Galkin
- configuration import design changed

1.9.1 (2021-08-27)

- re-defined amplitude normalization in NIE and CHAMELEON light profiles
- bug fix in sky brightness errors (SimulationAPI)

1.9.2 (2021-12-12)

- support for astropy v5
- new PSF iteration procedure implemented
- updated caustic plotting feature
- magnification perturbations in point source amplitudes
- analytic point source solver for SIE+shear

1.9.3 (2021-12-22)

- changed syntax to be compatible with python3 version <3.9

1.10.0 (2022-03-23)

- schwimmbad dependency to pip version
- ellipticity definition in lensing potential changed
- Implemented Cored steep ellipsoid approximation of NFW and Hernquist profile

1.10.1 (2022-03-26)

- install requirements changed

1.10.2 (2022-03-27)

- requirement from sklearn changed to scikit-learn

1.10.3 (2022-04-18)

- class_creator update
- conda-forge linked and installation updated

1.10.4 (2022-07-25)

- Zeus sampler implemented
- Nautilus sampler implemented
- Roman telescope configuration added
- double power-law mass profile
- generalized NFW profile
- enabled to turn off linear solver in fitting

1.11.0 (2022-09-26)

- transitioning to project repository
- logo update
- line of sight lensing module
- documentation improvements
- lens equation solver numerics improved

1.11.1 (2023-03-07)

- psf_error_map definition changed
- added JWST configurations
- minor change in Sersic light profile
- simplified LensCosmo class
- NFW c-rho0 inversion extended in range
- added stretch_plot and shear_plot to lens_plot
- minor bug fix for critical_curve_caustic
- enable the change of kwargs_sigma initial guess parameters in FittingSequence
- improve zeus and nautilus sampler implementations
- added EPL_boxydisky lens profile
- added primary beam to image simulation (for interferrometic data)

Python Module Index

l

lenstronomy, 395
lenstronomy.Analysis, 41
lenstronomy.Analysis.image_reconstruction, 27
lenstronomy.Analysis.kinematics_api, 29
lenstronomy.Analysis.lens_profile, 33
lenstronomy.Analysis.light2mass, 35
lenstronomy.Analysis.light_profile, 36
lenstronomy.Analysis.multi_patch_reconstruction, 38
lenstronomy.Analysis.td_cosmography, 39
lenstronomy.Conf, 42
lenstronomy.Conf.config_loader, 42
lenstronomy.Cosmo, 50
lenstronomy.Cosmo.background, 42
lenstronomy.Cosmo.cosmo_solver, 43
lenstronomy.Cosmo.kde_likelihood, 44
lenstronomy.Cosmo.lcdm, 44
lenstronomy.Cosmo.lens_cosmo, 45
lenstronomy.Cosmo.nfw_param, 48
lenstronomy.Data, 56
lenstronomy.Data.coord_transforms, 50
lenstronomy.Data.image_noise, 52
lenstronomy.Data.imaging_data, 53
lenstronomy.Data.pixel_grid, 54
lenstronomy.Data.psf, 55
lenstronomy.GalKin, 76
lenstronomy.GalKin.analytic_kinematics, 56
lenstronomy.GalKin.anisotropy, 58
lenstronomy.GalKin.aperture, 62
lenstronomy.GalKin.aperture_types, 63
lenstronomy.GalKin.cosmo, 66
lenstronomy.GalKin.galkin, 66
lenstronomy.GalKin.galkin_model, 68
lenstronomy.GalKin.light_profile, 69
lenstronomy.GalKin.numeric_kinematics, 71
lenstronomy.GalKin.observation, 73
lenstronomy.GalKin.psf, 73
lenstronomy.GalKin.velocity_util, 74
lenstronomy.ImSim, 98
lenstronomy.ImSim.de_lens, 90
lenstronomy.ImSim.image2source_mapping, 90
lenstronomy.ImSim.image_linear_solve, 92
lenstronomy.ImSim.image_model, 96
lenstronomy.ImSim.MultiBand, 82
lenstronomy.ImSim.MultiBand.joint_linear, 76
lenstronomy.ImSim.MultiBand.multi_data_base, 77
lenstronomy.ImSim.MultiBand.multi_linear, 78
lenstronomy.ImSim.MultiBand.single_band_multi_model, 80
lenstronomy.ImSim.Numerics, 90
lenstronomy.ImSim.Numerics.adaptive_numerics, 82
lenstronomy.ImSim.Numerics.convolution, 83
lenstronomy.ImSim.Numerics.grid, 85
lenstronomy.ImSim.Numerics.numba_convolution, 86
lenstronomy.ImSim.Numerics.numerics, 87
lenstronomy.ImSim.Numerics.partial_image, 88
lenstronomy.ImSim.Numerics.point_source_rendering, 89
lenstronomy.LensModel, 272
lenstronomy.LensModel.convergence_integrals, 258
lenstronomy.LensModel.lens_model, 259
lenstronomy.LensModel.lens_model_extensions, 264
lenstronomy.LensModel.lens_param, 268
lenstronomy.LensModel.LightConeSim, 99

lenstronomy.LensModel.LightConeSim.light_cone, 98
lenstronomy.LensModel.LineOfSight, 104
lenstronomy.LensModel.LineOfSight.LOSModels, 102
lenstronomy.LensModel.LineOfSight.LOSModels.los, 99
lenstronomy.LensModel.LineOfSight.LOSModels.los_minimal, 101
lenstronomy.LensModel.LineOfSight.single_plane_los, 102
lenstronomy.LensModel.MultiPlane, 109
lenstronomy.LensModel.MultiPlane.multi_plane, 104
lenstronomy.LensModel.MultiPlane.multi_plane_base, 108
lenstronomy.LensModel.profile_integrals, 269
lenstronomy.LensModel.profile_list_base, 270
lenstronomy.LensModel.Profiles, 247
lenstronomy.LensModel.Profiles.arc_perturbations, 109
lenstronomy.LensModel.Profiles.base_profile, 110
lenstronomy.LensModel.Profiles.chameleon, 112
lenstronomy.LensModel.Profiles.cnfw, 121
lenstronomy.LensModel.Profiles.cnfw_ellipse, 123
lenstronomy.LensModel.Profiles.const_mag, 123
lenstronomy.LensModel.Profiles.constant_shift, 125
lenstronomy.LensModel.Profiles.convergence, 125
lenstronomy.LensModel.Profiles.coreBurkert, 126
lenstronomy.LensModel.Profiles.cored_density, 129
lenstronomy.LensModel.Profiles.cored_density_2, 132
lenstronomy.LensModel.Profiles.cored_density_exp, 135
lenstronomy.LensModel.Profiles.cored_density_mst, 137
lenstronomy.LensModel.Profiles.cored_steep_ellipsoid, 138
lenstronomy.LensModel.Profiles.curved_arc_const, 143
lenstronomy.LensModel.Profiles.curved_arc_sis_mst, 145
lenstronomy.LensModel.Profiles.curved_arc_spp, 147
lenstronomy.LensModel.Profiles.curved_arc_spt, 150
lenstronomy.LensModel.Profiles.curved_arc_tan_diff, 151
lenstronomy.LensModel.Profiles.dipole, 153
lenstronomy.LensModel.Profiles.elliptical_density_s 154
lenstronomy.LensModel.Profiles.epl, 155
lenstronomy.LensModel.Profiles.epl_numba, 159
lenstronomy.LensModel.Profiles.flexion, 161
lenstronomy.LensModel.Profiles.flexionfg, 161
lenstronomy.LensModel.Profiles.gauss_decomposition, 162
lenstronomy.LensModel.Profiles.gaussian_ellipse_kap 170
lenstronomy.LensModel.Profiles.gaussian_ellipse_pot 172
lenstronomy.LensModel.Profiles.gaussian_kappa, 174
lenstronomy.LensModel.Profiles.gaussian_potential, 176
lenstronomy.LensModel.Profiles.general_nfw, 176
lenstronomy.LensModel.Profiles.hernquist, 180
lenstronomy.LensModel.Profiles.hernquist_ellipse, 182
lenstronomy.LensModel.Profiles.hernquist_ellipse_cs 184
lenstronomy.LensModel.Profiles.hessian, 185
lenstronomy.LensModel.Profiles.interpol, 186
lenstronomy.LensModel.Profiles.multi_gaussian_kappa 190
lenstronomy.LensModel.Profiles.multipole, 193
lenstronomy.LensModel.Profiles.nfw, 194
lenstronomy.LensModel.Profiles.nfw_ellipse, 198
lenstronomy.LensModel.Profiles.nfw_ellipse_cse, 200
lenstronomy.LensModel.Profiles.nfw_mass_concentrati 201
lenstronomy.LensModel.Profiles.nfw_vir_trunc, 203
lenstronomy.LensModel.Profiles.nie, 203
lenstronomy.LensModel.Profiles.nie_potential, 206
lenstronomy.LensModel.Profiles.numerical_deflectio

208
lenstronomy.LensModel.Profiles.p_jaffe, 209
lenstronomy.LensModel.Profiles.p_jaffe_ellipse, 212
lenstronomy.LensModel.Profiles.pemd, 213
lenstronomy.LensModel.Profiles.point_mass, 215
lenstronomy.LensModel.Profiles.sersic, 216
lenstronomy.LensModel.Profiles.sersic_ellipse_kappa, 217
lenstronomy.LensModel.Profiles.sersic_ellipse_potential, 218
lenstronomy.LensModel.Profiles.sersic_utils, 218
lenstronomy.LensModel.Profiles.shapelet_pot_cartesian, 220
lenstronomy.LensModel.Profiles.shapelet_pot_polar, 221
lenstronomy.LensModel.Profiles.shear, 221
lenstronomy.LensModel.Profiles.sie, 224
lenstronomy.LensModel.Profiles.sis, 227
lenstronomy.LensModel.Profiles.sis_truncate, 229
lenstronomy.LensModel.Profiles.spemd, 229
lenstronomy.LensModel.Profiles.spep, 231
lenstronomy.LensModel.Profiles.splcore, 233
lenstronomy.LensModel.Profiles.spp, 236
lenstronomy.LensModel.Profiles.tnfw, 238
lenstronomy.LensModel.Profiles.tnfw_ellipse, 241
lenstronomy.LensModel.Profiles.uldm, 243
lenstronomy.LensModel.QuadOptimizer, 251
lenstronomy.LensModel.QuadOptimizer.multi_plane_fast, 247
lenstronomy.LensModel.QuadOptimizer.optimizer, 248
lenstronomy.LensModel.QuadOptimizer.param_manager, 249
lenstronomy.LensModel.single_plane, 270
lenstronomy.LensModel.Solver, 257
lenstronomy.LensModel.Solver.lens_equation_solver, 251
lenstronomy.LensModel.Solver.solver, 255
lenstronomy.LensModel.Solver.solver2point, 256
lenstronomy.LensModel.Solver.solver4point, 256
lenstronomy.LensModel.Util, 258
lenstronomy.LensModel.Util.epl_util, 257
lenstronomy.LightModel, 299
lenstronomy.LightModel.light_model, 298
lenstronomy.LightModel.light_param, 298
lenstronomy.LightModel.Profiles, 298
lenstronomy.LightModel.Profiles.chameleon, 272
lenstronomy.LightModel.Profiles.ellipsoid, 276
lenstronomy.LightModel.Profiles.gaussian, 276
lenstronomy.LightModel.Profiles.hernquist, 281
lenstronomy.LightModel.Profiles.interpolation, 282
lenstronomy.LightModel.Profiles.moffat, 283
lenstronomy.LightModel.Profiles.nie, 284
lenstronomy.LightModel.Profiles.p_jaffe, 285
lenstronomy.LightModel.Profiles.power_law, 286
lenstronomy.LightModel.Profiles.profile_base, 287
lenstronomy.LightModel.Profiles.sersic, 288
lenstronomy.LightModel.Profiles.shapelets, 290
lenstronomy.LightModel.Profiles.shapelets_polar, 293
lenstronomy.LightModel.Profiles.starlets, 295
lenstronomy.LightModel.Profiles.starlets_util, 297
lenstronomy.LightModel.Profiles.uniform, 297
lenstronomy.Plots, 311
lenstronomy.Plots.chain_plot, 299
lenstronomy.Plots.lens_plot, 300
lenstronomy.Plots.model_band_plot, 302
lenstronomy.Plots.model_plot, 306
lenstronomy.Plots.plot_util, 309
lenstronomy.PointSource, 321
lenstronomy.PointSource.point_source, 316
lenstronomy.PointSource.point_source_cached, 319
lenstronomy.PointSource.point_source_param, 321
lenstronomy.PointSource.Types, 316
lenstronomy.PointSource.Types.base_ps, 312
lenstronomy.PointSource.Types.lensed_position, 313
lenstronomy.PointSource.Types.source_position,

314
lenstronomy.PointSource.Types.unlensed, 315
lenstronomy.Sampling, 347
lenstronomy.Sampling.likelihood, 333
lenstronomy.Sampling.Likelihoods, 327
lenstronomy.Sampling.Likelihoods.image_likelihood, 322
lenstronomy.Sampling.Likelihoods.position_likelihood, 323
lenstronomy.Sampling.Likelihoods.prior_likelihood, 325
lenstronomy.Sampling.Likelihoods.time_delay_likelihood, 326
lenstronomy.Sampling.param_group, 344
lenstronomy.Sampling.parameters, 335
lenstronomy.Sampling.Pool, 328
lenstronomy.Sampling.Pool.multiprocessing, 327
lenstronomy.Sampling.Pool.pool, 328
lenstronomy.Sampling.sampler, 341
lenstronomy.Sampling.Samplers, 332
lenstronomy.Sampling.Samplers.base_nested_sampler, 328
lenstronomy.Sampling.Samplers.dynesty_sampler, 329
lenstronomy.Sampling.Samplers.multinest_sampler, 330
lenstronomy.Sampling.Samplers.polychord_sampler, 331
lenstronomy.Sampling.special_param, 343
lenstronomy.SimulationAPI, 356
lenstronomy.SimulationAPI.data_api, 349
lenstronomy.SimulationAPI.model_api, 350
lenstronomy.SimulationAPI.observation_api, 351
lenstronomy.SimulationAPI.observation_constructor, 354
lenstronomy.SimulationAPI.ObservationConfig, 349
lenstronomy.SimulationAPI.ObservationConfig.DES, 347
lenstronomy.SimulationAPI.ObservationConfig.Euclid, 348
lenstronomy.SimulationAPI.ObservationConfig.HST, 348
lenstronomy.SimulationAPI.ObservationConfig.LSST, 349
lenstronomy.SimulationAPI.point_source_variability, 354
lenstronomy.SimulationAPI.sim_api, 355
lenstronomy.Util, 381
lenstronomy.Util.analysis_util, 356
lenstronomy.Util.class_creator, 358
lenstronomy.Util.constants, 360
lenstronomy.Util.correlation, 360
lenstronomy.Util.data_util, 361
lenstronomy.Util.derivative_util, 362
lenstronomy.Util.image_util, 364
lenstronomy.Util.kernel_util, 366
lenstronomy.Util.mask_util, 369
lenstronomy.Util.multi_gauss_expansion, 370
lenstronomy.Util.numba_util, 371
lenstronomy.Util.param_util, 371
lenstronomy.Util.prob_density, 373
lenstronomy.Util.sampling_util, 374
lenstronomy.Util.simulation_util, 375
lenstronomy.Util.util, 376
lenstronomy.Workflow, 395
lenstronomy.Workflow.alignment_matching, 381
lenstronomy.Workflow.fitting_sequence, 382
lenstronomy.Workflow.psf_fitting, 387
lenstronomy.Workflow.update_manager, 392

Index

Symbols

__init__() (*AdaptiveConvolution method*), 82
__init__() (*AdaptiveGrid method*), 85
__init__() (*AlignmentFitting method*), 381
__init__() (*AlignmentLikelihood method*), 382
__init__() (*AnalyticKinematics method*), 57
__init__() (*Anisotropy method*), 58
__init__() (*Aperture method*), 62
__init__() (*ArcPerturbations method*), 109
__init__() (*ArrayParam method*), 346
__init__() (*Background method*), 42
__init__() (*CNFW method*), 121
__init__() (*CNFW_ELLIPSE method*), 123
__init__() (*CSE method*), 138
__init__() (*CSEMajorAxisSet method*), 140
__init__() (*CSEProductAvg method*), 141
__init__() (*CSEProductAvgSet method*), 142
__init__() (*CTNFWGaussDec method*), 168
__init__() (*Chameleon method*), 112, 272
__init__() (*Colin method*), 62
__init__() (*Const method*), 59
__init__() (*Coordinates method*), 50
__init__() (*CoredDensityMST method*), 137
__init__() (*Cosmo method*), 66
__init__() (*CurvedArcConstMST method*), 143
__init__() (*CurvedArcSISMST method*), 146
__init__() (*CurvedArcSPP method*), 148
__init__() (*CurvedArcSPT method*), 150
__init__() (*CurvedArcTanDiff method*), 151
__init__() (*DES method*), 347
__init__() (*DataAPI method*), 349
__init__() (*DoubleChameleon method*), 114, 273
__init__() (*DoubleChameleonPointMass method*), 120
__init__() (*DyPolyChordSampler method*), 331
__init__() (*DynestySampler method*), 329
__init__() (*EPL method*), 156
__init__() (*EPLMajorAxis method*), 158
__init__() (*EPL_numba method*), 159
__init__() (*Ellipsoid method*), 276
__init__() (*Euclid method*), 348
__init__() (*FWHMGaussianConvolution method*), 84
__init__() (*FittingSequence method*), 382
__init__() (*Flexionfg method*), 161
__init__() (*Frame method*), 64
__init__() (*Galkin method*), 67
__init__() (*GalkinModel method*), 68
__init__() (*GalkinObservation method*), 73
__init__() (*GaussDecompositionAbstract method*), 164
__init__() (*Gaussian method*), 276
__init__() (*GaussianEllipse method*), 277
__init__() (*GaussianEllipseKappa method*), 170
__init__() (*GaussianEllipseKappaSet method*), 162
__init__() (*GaussianEllipsePotential method*), 172
__init__() (*GaussianKappa method*), 174
__init__() (*GeneralizedOM method*), 61
__init__() (*HST method*), 348
__init__() (*Hernquist method*), 281
__init__() (*HernquistEllipse method*), 281
__init__() (*HernquistEllipseCSE method*), 184
__init__() (*Hernquist_Ellipse method*), 182
__init__() (*IFUShells method*), 65
__init__() (*Image2SourceMapping method*), 91
__init__() (*ImageData method*), 53
__init__() (*ImageLikelihood method*), 322
__init__() (*ImageLinearFit method*), 92
__init__() (*ImageModel method*), 96
__init__() (*ImageNoise method*), 52
__init__() (*Instrument method*), 351
__init__() (*Interpol method*), 186, 282
__init__() (*InterpolScaled method*), 188
__init__() (*InvertCosmo method*), 44
__init__() (*Isotropic method*), 60
__init__() (*JointLinear method*), 76
__init__() (*KDE1D method*), 373
__init__() (*KDELikelihood method*), 44
__init__() (*KinematicsAPI method*), 29

__init__() (*LCDM method*), 45
__init__() (*LOS method*), 100
__init__() (*LSST method*), 349
__init__() (*LensCosmo method*), 45
__init__() (*LensEquationSolver method*), 251
__init__() (*LensModel method*), 260
__init__() (*LensModelExtensions method*), 264
__init__() (*LensParam method*), 268
__init__() (*LensProfileAnalysis method*), 33
__init__() (*LensProfileBase method*), 110
__init__() (*LensedLocation method*), 107
__init__() (*LightCone method*), 98
__init__() (*LightModel method*), 298
__init__() (*LightParam method*), 298
__init__() (*LightProfile method*), 69
__init__() (*LightProfileAnalysis method*), 36
__init__() (*LightProfileBase method*), 287
__init__() (*LikelihoodModule method*), 333
__init__() (*MGEConvolution method*), 85
__init__() (*MassSlice method*), 99
__init__() (*ModelAPI method*), 350
__init__() (*ModelBand method*), 28
__init__() (*ModelBandPlot method*), 302
__init__() (*ModelPlot method*), 306
__init__() (*Moffat method*), 284
__init__() (*MultiBandImageReconstruction method*), 27
__init__() (*MultiDataBase method*), 78
__init__() (*MultiGaussian method*), 278
__init__() (*MultiGaussianConvolution method*), 84
__init__() (*MultiGaussianEllipse method*), 279
__init__() (*MultiGaussianKappa method*), 190
__init__() (*MultiGaussianKappaEllipse method*), 191
__init__() (*MultiLinear method*), 78
__init__() (*MultiNestSampler method*), 330
__init__() (*MultiPatchReconstruction method*), 38
__init__() (*MultiPlane method*), 104
__init__() (*MultiPlaneBase method*), 108
__init__() (*MultiPool method*), 327
__init__() (*MultiplaneFast method*), 247
__init__() (*NFW method*), 194
__init__() (*NFWEllipseGaussDec method*), 167
__init__() (*NFWMC method*), 202
__init__() (*NFWParam method*), 49
__init__() (*NFWVirTrunc method*), 203
__init__() (*NFW_ELLIPSE method*), 198
__init__() (*NFW_ELLIPSE_CSE method*), 200
__init__() (*NIE method*), 203
__init__() (*NIEMajorAxis method*), 205
__init__() (*NIEPotentialMajorAxis method*), 207
__init__() (*NIE_POTENTIAL method*), 206
__init__() (*NestedSampler method*), 328
__init__() (*NumbaConvolution method*), 87
__init__() (*NumericKinematics method*), 71
__init__() (*NumericalAlpha method*), 208
__init__() (*Numerics method*), 87
__init__() (*Observation method*), 351
__init__() (*Optimizer method*), 248
__init__() (*OsipkovMerritt method*), 61
__init__() (*PEMD method*), 214
__init__() (*PJaffe method*), 209, 285
__init__() (*PJaffeEllipse method*), 286
__init__() (*PJaffe_Ellipse method*), 213
__init__() (*PSBase method*), 312
__init__() (*PSF method*), 55, 73
__init__() (*PSFGaussian method*), 74
__init__() (*PSFMoffat method*), 74
__init__() (*Param method*), 337
__init__() (*PartialImage method*), 88
__init__() (*PixelGrid method*), 54
__init__() (*PixelKernelConvolution method*), 83
__init__() (*PointMass method*), 215
__init__() (*PointSource method*), 316
__init__() (*PointSourceCached method*), 319
__init__() (*PointSourceParam method*), 321
__init__() (*PointSourceRendering method*), 89
__init__() (*PointSourceVariability method*), 354
__init__() (*PolarShapelets method*), 221
__init__() (*PositionLikelihood method*), 323
__init__() (*PowerLaw method*), 286
__init__() (*PowerLawFixedShear method*), 249
__init__() (*PowerLawParamManager method*), 250
__init__() (*PriorLikelihood method*), 325
__init__() (*ProfileIntegrals method*), 269
__init__() (*ProfileListBase method*), 270
__init__() (*PsfFitting method*), 388
__init__() (*Radial method*), 60
__init__() (*RegularGrid method*), 86
__init__() (*SIE method*), 224
__init__() (*SLIT_Starlets method*), 295
__init__() (*SPEMD method*), 230
__init__() (*SPEP method*), 231
__init__() (*Sampler method*), 341
__init__() (*SersicEllipse method*), 218
__init__() (*SersicEllipseKappa method*), 217
__init__() (*SersicUtil method*), 218
__init__() (*ShapeletSet method*), 292
__init__() (*ShapeletSetPolar method*), 295
__init__() (*Shapelets method*), 290
__init__() (*ShapeletsPolar method*), 293
__init__() (*ShapeletsPolarExp method*), 294
__init__() (*ShearGammaPsi method*), 222
__init__() (*ShearReduced method*), 223
__init__() (*Shell method*), 64
__init__() (*SimAPI method*), 355
__init__() (*SingleBand method*), 352
__init__() (*SingleBandMultiModel method*), 80

__init__() (*SingleParam method*), 345
__init__() (*SinglePlaneLOS method*), 102
__init__() (*Slit method*), 63
__init__() (*Solver method*), 255
__init__() (*Solver2Point method*), 256
__init__() (*Solver4Point method*), 256
__init__() (*SolverFlatLCDM method*), 43
__init__() (*SpecialParam method*), 343
__init__() (*SubgridKernelConvolution method*), 83
__init__() (*TDCosmography method*), 39
__init__() (*TNFW method*), 238
__init__() (*TNFW_ELLIPSE method*), 241
__init__() (*TimeDelayLikelihood method*), 326
__init__() (*TripleChameleon method*), 117, 274
__init__() (*Uniform method*), 297
__init__() (*UpdateManager method*), 392

A

a_z() (*Background static method*), 42
absolute2apparent_magnitude() (*in module lenstronomy.Util.data_util*), 361
absolute_residual_plot() (*ModelBandPlot method*), 303
absolute_residual_plot() (*ModelPlot method*), 307
AdaptiveConvolution (*class in lenstronomy.ImSim.Numerics.adaptive_numerics*), 82
AdaptiveGrid (*class in lenstronomy.ImSim.Numerics.grid*), 85
add_background() (*in module lenstronomy.Util.image_util*), 364
add_fix_linear() (*PointSourceParam method*), 321
add_fixed_lens() (*Solver method*), 255
add_fixed_lens() (*Solver2Point method*), 256
add_fixed_lens() (*Solver4Point method*), 257
add_layer2image() (*in module lenstronomy.Util.image_util*), 364
add_layer2image_int() (*in module lenstronomy.Util.image_util*), 364
add_poisson() (*in module lenstronomy.Util.image_util*), 364
adu2electrons() (*in module lenstronomy.Util.data_util*), 361
align_images() (*FittingSequence method*), 383
AlignmentFitting (*class in lenstronomy.Workflow.alignment_matching*), 381
AlignmentLikelihood (*class in lenstronomy.Workflow.alignment_matching*), 381
alpha() (*LensModel method*), 260
alpha() (*MultiPlane method*), 105
alpha() (*SinglePlane method*), 270
alpha() (*SinglePlaneLOS method*), 102
alpha() (*SPLCORE method*), 233
alpha2rho0() (*GNFW method*), 176
alpha2rho0() (*NFW static method*), 195
alpha2rho0() (*TNFW static method*), 238
alpha_abs() (*GaussianKappa method*), 174
alpha_abs() (*SersicUtil method*), 218
alpha_ext() (*ElliSLICE method*), 154
alpha_in() (*ElliSLICE method*), 154
alpha_r() (*CNFW method*), 121
alpha_r() (*CoredDensity static method*), 129
alpha_r() (*CoredDensity2 static method*), 132
alpha_radial() (*CoredDensityExp static method*), 135
alpha_radial() (*Uldm static method*), 244
AnalyticKinematics (*class in lenstronomy.GalKin.analytic_kinematics*), 56
angle() (*DipoleUtil static method*), 153
Anisotropy (*class in lenstronomy.GalKin.anisotropy*), 58
anisotropy_solution() (*Anisotropy method*), 58
anisotropy_solution() (*Const method*), 59
anisotropy_solution() (*GeneralizedOM static method*), 61
anisotropy_solution() (*Isotropic static method*), 60
anisotropy_solution() (*OsipkovMerritt static method*), 61
anisotropy_solution() (*Radial static method*), 60
Aperture (*class in lenstronomy.GalKin.aperture*), 62
aperture_select() (*Aperture method*), 62
aperture_select() (*Frame method*), 64
aperture_select() (*IFUShells method*), 65
aperture_select() (*Shell method*), 64
aperture_select() (*Slit method*), 63
approx_theta_E() (*in module lenstronomy.Util.util*), 376
ArcPerturbations (*class in lenstronomy.LensModel.Profiles.arc_perturbations*), 109
arcsec2phys_lens() (*Cosmo method*), 66
arcsec2phys_lens() (*LensCosmo method*), 45
arcsec2phys_source() (*LensCosmo method*), 46
args2kwargs() (*Param method*), 339
args_to_kwargs() (*PowerLawFixedShear method*), 249
args_to_kwargs() (*PowerLawFixedShearMultipole method*), 249
args_to_kwargs() (*PowerLawFreeShear method*), 250
args_to_kwargs() (*PowerLawFreeShearMultipole method*), 250
array2cube() (*in module lenstronomy.Util.util*), 377
array2image() (*in module lenstronomy.Util.util*), 376

array_from_partial() (PartialImage method), 89
array_masked2image() (ImageLinearFit method), 92
ArrayParam (class in lenstronomy.Sampling.param_group), 346
arrival_time() (LensModel method), 261
arrival_time() (MultiPlane method), 105
arrival_time_surface() (in module lenstronomy.Plots.lens_plot), 301
astrometric_likelihood() (PositionLikelihood static method), 324
averaging() (in module lenstronomy.Util.util), 378
averaging_even_kernel() (in module lenstronomy.Util.kernel_util), 368
azimuthalAverage() (in module lenstronomy.Util.analysis_util), 356

B

b_n() (SersicUtil static method), 219
Background (class in lenstronomy.Cosmo.background), 42
background_noise (SingleBand attribute), 353
background_rms (ImageNoise attribute), 52
band_setup() (MultiBandImageReconstruction method), 28
best_fit() (FittingSequence method), 383
best_fit() (UpdateManager method), 392
best_fit_from_samples() (FittingSequence method), 383
best_fit_likelihood (FittingSequence attribute), 383
beta_r() (Anisotropy method), 58
beta_r() (Colin static method), 62
beta_r() (Const static method), 59
beta_r() (GeneralizedOM static method), 61
beta_r() (Isotropic static method), 60
beta_r() (OsipkovMerritt static method), 61
beta_r() (Radial static method), 60
bic (FittingSequence attribute), 383
bic_model() (in module lenstronomy.Util.analysis_util), 357
bkg_noise() (in module lenstronomy.Util.data_util), 361
bounds() (PowerLawParamManager method), 250
brentq_inline (in module lenstronomy.LensModel.Util.epl_util), 257
brentq_nojit() (in module lenstronomy.LensModel.Util.epl_util), 257

C

C_D (ImageNoise attribute), 52
C_D_model() (ImageNoise method), 52
c_M_z() (NFWParam static method), 49
c_rho0() (NFWParam method), 49
camera (DES attribute), 347
camera (Euclid attribute), 348
camera (HST attribute), 348
camera (LSST attribute), 349
candidate_solutions() (LensEquationSolver method), 251
cart2polar() (in module lenstronomy.Util.param_util), 371
cart_to_pol (in module lenstronomy.LensModel.Util.epl_util), 257
CartShapelets (class in lenstronomy.LensModel.Profiles.shapelet_pot_cartesian), 220
caustic_area() (LensModelExtensions method), 264
cBurkGamma() (CoreBurkert method), 126
cBurkPot() (CoreBurkert method), 127
cdot (in module lenstronomy.LensModel.Util.epl_util), 257
center (PixelGrid attribute), 55
center_deflector() (in module lenstronomy.LensModel.Profiles.curved_arc_spp), 149
center_kernel() (in module lenstronomy.Util.kernel_util), 366
Chameleon (class in lenstronomy.LensModel.Profiles.chameleon), 112
Chameleon (class in lenstronomy.LightModel.Profiles.chameleon), 272
check_additional_images() (PositionLikelihood method), 324
check_bounds() (LikelihoodModule static method), 334
check_df() (GalkinModel method), 68
check_image_positions() (PointSource method), 316
check_positive_flux() (ImageLinearFit method), 92
check_positive_flux() (lenstronomy.PointSource.point_source.PointSource class method), 317
check_solver() (Param method), 339
check_solver() (Solver method), 255
check_solver_error() (in module lenstronomy.Analysis.image_reconstruction), 29
chi_square() (MultiplaneFast method), 247
choose_pool() (in module lenstronomy.Sampling.Pool.pool), 328
cmap_conf() (in module lenstronomy.Plots.plot_util), 311
CNFW (class in lenstronomy.LensModel.Profiles.cnfw), 121
CNFW_ELLIPSE (class in lenstronomy.LensModel.Profiles.cnfw_ellipse), 123

cnfwGamma() (*CNFW method*), 121
co_moving2angle_source() (*MultiPlane method*), 105
Colin (*class in lenstronomy.GalKin.anisotropy*), 62
com() (*DipoleUtil static method*), 153
combine_psf() (*PsfFitting static method*), 388
compare_distance() (*in module lenstronomy.Util.util*), 379
compose_get_params() (*ModelParamGroup static method*), 344
compose_num_params() (*ModelParamGroup static method*), 344
compose_set_params() (*ModelParamGroup static method*), 344
compute_lower_upper_errors() (*in module lenstronomy.Util.prob_density*), 374
computeLikelihood() (*AlignmentLikelihood method*), 382
cone_instance() (*LightCone method*), 99
Const (*class in lenstronomy.GalKin.anisotropy*), 59
ConstMag (*class in lenstronomy.LensModel.Profiles.const_mag*), 123
constraint_lensmodel() (*Solver method*), 255
constraint_lensmodel() (*Solver2Point method*), 256
constraint_lensmodel() (*Solver4Point method*), 257
conventions_conf() (*in module lenstronomy.Conf.config_loader*), 42
Convergence (*class in lenstronomy.LensModel.Profiles.convergence*), 125
convergence_peak() (*LensProfileAnalysis method*), 33
convergence_plot() (*ModelBandPlot method*), 303
convergence_plot() (*ModelPlot method*), 307
convert_bool_list() (*in module lenstronomy.Util.util*), 381
convert_params() (*SPEMD static method*), 230
convolution2d() (*FWHMGaussianConvolution method*), 84
convolution2d() (*MGEConvolution method*), 85
convolution2d() (*MultiGaussianConvolution method*), 84
convolution2d() (*PixelKernelConvolution method*), 83
convolution2d() (*SubgridKernelConvolution method*), 83
convolution_class (*Numerics attribute*), 88
convolve2d() (*AdaptiveConvolution method*), 82
convolve2d() (*NumbaConvolution method*), 87
coord2image_pixel() (*Interpol static method*), 282
coordinate_arrows() (*in module lenstronomy.Plots.plot_util*), 310
coordinate_grid() (*Coordinates method*), 50
coordinate_grid() (*Coordinates1D method*), 51
Coordinates (*class in lenstronomy.Data.coord_transforms*), 50
Coordinates1D (*class in lenstronomy.Data.coord_transforms*), 51
coordinates_evaluate (*AdaptiveGrid attribute*), 85
coordinates_evaluate (*Numerics attribute*), 88
coordinates_evaluate (*RegularGrid attribute*), 86
coordInImage() (*in module lenstronomy.Util.image_util*), 365
copy_transpose() (*PixelKernelConvolution method*), 83
coreBurkAlpha() (*CoreBurkert method*), 127
CoreBurkert (*class in lenstronomy.LensModel.Profiles.coreBurkert*), 126
CoredDensity (*class in lenstronomy.LensModel.Profiles.cored_density*), 129
CoredDensity2 (*class in lenstronomy.LensModel.Profiles.cored_density_2*), 132
CoredDensityExp (*class in lenstronomy.LensModel.Profiles.cored_density_exp*), 135
CoredDensityMST (*class in lenstronomy.LensModel.Profiles.cored_density_mst*), 137
CoreSersic (*class in lenstronomy.LightModel.Profiles.sersic*), 289
correlation_2D() (*in module lenstronomy.Util.correlation*), 360
Cosmo (*class in lenstronomy.GalKin.cosmo*), 66
cosmo2angular_diameter_distances() (*in module lenstronomy.Cosmo.cosmo_solver*), 43
covariance_matrix() (*in module lenstronomy.Data.image_noise*), 52
cps2magnitude() (*in module lenstronomy.Util.data_util*), 361
create_class_instances() (*in module lenstronomy.Util.class_creator*), 358
create_im_sim() (*in module lenstronomy.Util.class_creator*), 359
create_image_model() (*in module lenstronomy.Util.class_creator*), 359
critical_curve_caustics() (*LensModelExtensions method*), 264
critical_curve_tiling() (*LensModelExtensions method*), 265
CSE (*class in lenstronomy.LensModel.Profiles.cored_steep_ellipsoid*), 138

CSEMajorAxis (*class in lenstronomy.LensModel.Profiles.cored_steep_ellipsoid*), 139
CSEMajorAxisSet (*class in lenstronomy.LensModel.Profiles.cored_steep_ellipsoid*), 140
CSEProductAvg (*class in lenstronomy.LensModel.Profiles.cored_steep_ellipsoid*), 141
CSEProductAvgSet (*class in lenstronomy.LensModel.Profiles.cored_steep_ellipsoid*), 142
CTNFWGaussDec (*class in lenstronomy.LensModel.Profiles.gauss_decomposition*), 168
cube2args_gaussian() (*in module lenstronomy.Util.sampling_util*), 374
cube2args_uniform() (*in module lenstronomy.Util.sampling_util*), 374
cube2array() (*in module lenstronomy.Util.util*), 377
curl() (*LensModel method*), 261
curved_arc_estimate() (*LensModelExtensions method*), 265
curved_arc_finite_area() (*LensModelExtensions method*), 265
curved_arc_illustration() (*in module lenstronomy.Plots.lens_plot*), 301
CurvedArcConst (*class in lenstronomy.LensModel.Profiles.curved_arc_const*), 144
CurvedArcConstMST (*class in lenstronomy.LensModel.Profiles.curved_arc_const*), 143
CurvedArcSISMST (*class in lenstronomy.LensModel.Profiles.curved_arc_sis_mst*), 145
CurvedArcSPP (*class in lenstronomy.LensModel.Profiles.curved_arc_spp*), 147
CurvedArcSPT (*class in lenstronomy.LensModel.Profiles.curved_arc_spt*), 150
CurvedArcTanDiff (*class in lenstronomy.LensModel.Profiles.curved_arc_tan_diff*), 151
cut_edges() (*in module lenstronomy.Util.image_util*), 365
cut_psf() (*in module lenstronomy.Util.kernel_util*), 367
cutout_psf() (*PsfFitting method*), 388
cutout_psf_single() (*PsfFitting static method*), 388
cutout_source() (*in module lenstronomy.Util.kernel_util*), 368

D

d_alpha_dr() (*CoredDensity static method*), 129
d_alpha_dr() (*CoredDensity2 static method*), 132
d_alpha_dr() (*GaussianKappa method*), 174
d_alpha_dr() (*SersicUtil method*), 219
D_d() (*LCDM method*), 44
D_ds() (*LCDM method*), 45
D_dt() (*LCDM method*), 45
d_phi_dx() (*in module lenstronomy.Util.derivative_util*), 362
d_phi_dxx() (*in module lenstronomy.Util.derivative_util*), 363
d_phi_dxy() (*in module lenstronomy.Util.derivative_util*), 363
d_phi_dy() (*in module lenstronomy.Util.derivative_util*), 362
d_phi_dyy() (*in module lenstronomy.Util.derivative_util*), 363
d_r_dx() (*in module lenstronomy.Util.derivative_util*), 362
d_r_dxx() (*in module lenstronomy.Util.derivative_util*), 362
d_r_dxy() (*in module lenstronomy.Util.derivative_util*), 362
d_r_dy() (*in module lenstronomy.Util.derivative_util*), 362
d_r_dyy() (*in module lenstronomy.Util.derivative_util*), 362
D_s() (*LCDM method*), 45
d_x_diffr_dx() (*in module lenstronomy.Util.derivative_util*), 363
d_x_diffr_dy() (*in module lenstronomy.Util.derivative_util*), 363
d_xy() (*Background method*), 42
d_y_diffr_dx() (*in module lenstronomy.Util.derivative_util*), 363
d_y_diffr_dy() (*in module lenstronomy.Util.derivative_util*), 363
data (*ImageData attribute*), 54
data_class (*DataAPI attribute*), 350
data_configure_simple() (*in module lenstronomy.Util.simulation_util*), 375
data_plot() (*ModelBandPlot method*), 303
data_plot() (*ModelPlot method*), 307
data_response (*ImageLinearFit attribute*), 92
data_response (*JointLinear attribute*), 76
DataAPI (*class in lenstronomy.SimulationAPI.data_api*), 349
dd (*LensCosmo attribute*), 46
dds (*LensCosmo attribute*), 46
ddt (*LensCosmo attribute*), 46
ddt() (*Background method*), 43
ddt2h0() (*in module lenstronomy.Cosmo.cosmo_solver*), 43

ddt_dd_from_time_delay_and_kinematics() (*TDCosmography method*), 39
ddt_from_time_delay() (*TDCosmography static method*), 40
de_projection_3d() (*in module lenstronomy.Util.multi_gauss_expansion*), 371
de_shift_kernel() (*in module lenstronomy.Util.kernel_util*), 366
decomposition() (*ShapeletSet method*), 292
decomposition() (*ShapeletSetPolar method*), 295
decomposition() (*SLIT_Starlets method*), 296
decomposition_2d() (*SLIT_Starlets method*), 296
decomposition_plot() (*ModelBandPlot method*), 303
decomposition_plot() (*ModelPlot method*), 307
deflection_from_kappa_grid() (*in module lenstronomy.LensModel.convergence_integrals*), 259
deflection_from_kappa_grid_adaptive() (*in module lenstronomy.LensModel.convergence_integrals*), 259
deflection_kernel() (*in module lenstronomy.LensModel.convergence_integrals*), 259
deflection_plot() (*ModelBandPlot method*), 304
deflection_plot() (*ModelPlot method*), 307
degrade_kernel() (*in module lenstronomy.Util.kernel_util*), 368
delay_arcsec2days() (*in module lenstronomy.Util.constants*), 360
delays (*PointSourceVariability attribute*), 355
delete_anisotropy_cache() (*Anisotropy method*), 59
delete_cache() (*AnalyticKinematics method*), 57
delete_cache() (*GeneralizedOM method*), 62
delete_cache() (*Interpol method*), 283
delete_cache() (*LightProfile method*), 69
delete_cache() (*NumericKinematics method*), 71
delete_cache() (*SLIT_Starlets method*), 296
delete_lens_model_cache() (*PointSource method*), 317
delete_lens_model_cache() (*PointSourceCached method*), 319
density() (*CNFW method*), 121
density() (*CoreBurkert method*), 127
density() (*CoredDensity static method*), 129
density() (*CoredDensity2 static method*), 132
density() (*CoredDensityExp method*), 135
density() (*GaussianEllipsePotential method*), 172
density() (*GaussianKappa method*), 174
density() (*GNFW static method*), 176
density() (*Hernquist method*), 180
density() (*Hernquist_Ellipse method*), 182
density() (*MultiGaussianKappa method*), 190
density() (*MultiGaussianKappaEllipse method*), 191
density() (*NFW method*), 195
density() (*PJaffe method*), 209
density() (*SersicUtil method*), 219
density() (*SIE static method*), 224
density() (*SinglePlane method*), 271
density() (*SinglePlaneLOS method*), 102
density() (*SIS static method*), 227
density() (*SPLCORE static method*), 233
density() (*SPP static method*), 236
density() (*TNFW static method*), 238
density() (*Uldm method*), 244
density_2d() (*CNFW method*), 122
density_2d() (*CoreBurkert method*), 127
density_2d() (*CoredDensity method*), 129
density_2d() (*CoredDensity2 method*), 132
density_2d() (*CoredDensityExp method*), 135
density_2d() (*GaussDecompositionAbstract method*), 164
density_2d() (*GaussianEllipseKappa method*), 170
density_2d() (*GaussianEllipseKappaSet method*), 162
density_2d() (*GaussianEllipsePotential method*), 172
density_2d() (*GaussianKappa method*), 174
density_2d() (*GNFW method*), 177
density_2d() (*Hernquist method*), 180
density_2d() (*Hernquist_Ellipse method*), 183
density_2d() (*MultiGaussianKappa method*), 190
density_2d() (*MultiGaussianKappaEllipse method*), 191
density_2d() (*NFW method*), 195
density_2d() (*PJaffe method*), 209
density_2d() (*ProfileIntegrals method*), 269
density_2d() (*SIE static method*), 224
density_2d() (*SIS static method*), 227
density_2d() (*SPLCORE method*), 233
density_2d() (*SPP static method*), 236
density_2d() (*TNFW method*), 239
density_2d() (*Uldm method*), 244
density_lens() (*Chameleon method*), 112
density_lens() (*CNFW method*), 122
density_lens() (*CNFW_ELLIPSE method*), 123
density_lens() (*CoredDensity method*), 130
density_lens() (*CoredDensity2 method*), 133
density_lens() (*CoredDensityExp method*), 135
density_lens() (*DoubleChameleon method*), 114
density_lens() (*EPL method*), 156
density_lens() (*GNFW method*), 177
density_lens() (*Hernquist method*), 180
density_lens() (*Hernquist_Ellipse method*), 183
density_lens() (*LensProfileBase method*), 110
density_lens() (*NFW method*), 195

density_lens() (*NFW_ELLIPSE method*), 198
density_lens() (*NIE method*), 203
density_lens() (*PEMD method*), 214
density_lens() (*SIE method*), 225
density_lens() (*SIS method*), 227
density_lens() (*SPEP method*), 232
density_lens() (*SPLCORE method*), 234
density_lens() (*SPP method*), 236
density_lens() (*TNFW_ELLIPSE method*), 241
density_lens() (*TripleChameleon method*), 117
density_lens() (*Uldm method*), 244
derivatives (*EPL_numba attribute*), 159
derivatives() (*ArcPerturbations method*), 109
derivatives() (*CartShapelets method*), 220
derivatives() (*Chameleon method*), 112
derivatives() (*CNFW method*), 122
derivatives() (*CNFW_ELLIPSE method*), 123
derivatives() (*ConstMag method*), 124
derivatives() (*Convergence method*), 125
derivatives() (*CoreBurkert method*), 127
derivatives() (*CoredDensity method*), 130
derivatives() (*CoredDensity2 method*), 133
derivatives() (*CoredDensityExp method*), 135
derivatives() (*CoredDensityMST method*), 137
derivatives() (*CSE method*), 138
derivatives() (*CSEMajorAxis method*), 140
derivatives() (*CSEMajorAxisSet method*), 140
derivatives() (*CSEProductAvg method*), 141
derivatives() (*CSEProductAvgSet method*), 142
derivatives() (*CurvedArcConst method*), 144
derivatives() (*CurvedArcConstMST method*), 143
derivatives() (*CurvedArcSISMST method*), 146
derivatives() (*CurvedArcSPP method*), 148
derivatives() (*CurvedArcSPT method*), 150
derivatives() (*CurvedArcTanDiff method*), 151
derivatives() (*Dipole method*), 153
derivatives() (*DoubleChameleon method*), 115
derivatives() (*DoubleChameleonPointMass method*), 120
derivatives() (*ElliSLICE method*), 154
derivatives() (*EPL method*), 156
derivatives() (*EPLMajorAxis method*), 158
derivatives() (*Flexion method*), 161
derivatives() (*Flexionfg method*), 161
derivatives() (*GaussDecompositionAbstract method*), 165
derivatives() (*Gaussian method*), 176
derivatives() (*GaussianEllipseKappa method*), 170
derivatives() (*GaussianEllipseKappaSet method*), 163
derivatives() (*GaussianEllipsePotential method*), 173
derivatives() (*GaussianKappa method*), 175
derivatives() (*GNFW method*), 177
derivatives() (*Hernquist method*), 180
derivatives() (*Hernquist_Ellipse method*), 183
derivatives() (*HernquistEllipseCSE method*), 184
derivatives() (*Hessian method*), 185
derivatives() (*Interpol method*), 186
derivatives() (*InterpolScaled method*), 188
derivatives() (*LensProfileBase method*), 111
derivatives() (*MultiGaussianKappa method*), 190
derivatives() (*MultiGaussianKappaEllipse method*), 192
derivatives() (*Multipole method*), 193
derivatives() (*NFW method*), 195
derivatives() (*NFW_ELLIPSE method*), 198
derivatives() (*NFW_ELLIPSE_CSE method*), 200
derivatives() (*NFWMC method*), 202
derivatives() (*NIE method*), 204
derivatives() (*NIE_POTENTIAL method*), 206
derivatives() (*NIEMajorAxis method*), 205
derivatives() (*NIEPotentialMajorAxis method*), 207
derivatives() (*NumericalAlpha method*), 208
derivatives() (*PEMD method*), 214
derivatives() (*PJaffe method*), 210
derivatives() (*PJaffe_Ellipse method*), 213
derivatives() (*PointMass method*), 215
derivatives() (*PolarShapelets method*), 221
derivatives() (*Sersic method*), 217
derivatives() (*SersicEllipse method*), 218
derivatives() (*SersicEllipseKappa method*), 217
derivatives() (*Shear method*), 221
derivatives() (*ShearGammaPsi method*), 222
derivatives() (*ShearReduced method*), 223
derivatives() (*Shift method*), 125
derivatives() (*SIE method*), 225
derivatives() (*SIS method*), 228
derivatives() (*SIS_truncate method*), 229
derivatives() (*SPEMD method*), 230
derivatives() (*SPEP method*), 232
derivatives() (*SPLCORE method*), 234
derivatives() (*SPP method*), 236
derivatives() (*TNFW method*), 239
derivatives() (*TNFW_ELLIPSE method*), 242
derivatives() (*TripleChameleon method*), 117
derivatives() (*Uldm method*), 244
DES (*class in lenstronomy.SimulationAPI.ObservationConfig.DES*), 347
Dipole (*class in lenstronomy.LensModel.Profiles.dipole*), 153
DipoleUtil (*class in lenstronomy.LensModel.Profiles.dipole*), 153
dispersion() (*Galkin method*), 67
dispersion_map() (*Galkin method*), 67

displace_psf() (*PSF method*), 74
displace_psf() (*PSFGaussian method*), 74
displace_psf() (*PSFMoffat method*), 74
displace_PSF_gaussian() (*in module lenstronomy.GalKin.velocity_util*), 74
displace_PSF_moffat() (*in module lenstronomy.GalKin.velocity_util*), 75
displaceAbs() (*in module lenstronomy.Util.util*), 378
distort_vector() (*LOS static method*), 100
distortions() (*in module lenstronomy.Plots.lens_plot*), 302
do_interp() (*Interpol method*), 187
DoubleChameleon (*class in lenstronomy.LensModel.Profiles.chameleon*), 114
DoubleChameleon (*class in lenstronomy.LightModel.Profiles.chameleon*), 273
DoubleChameleonPointMass (*class in lenstronomy.LensModel.Profiles.chameleon*), 119
draw_cdf_Y() (*in module lenstronomy.GalKin.velocity_util*), 75
draw_hernquist() (*in module lenstronomy.GalKin.velocity_util*), 75
draw_light() (*AnalyticKinematics static method*), 57
draw_light() (*NumericKinematics method*), 71
draw_light_2d() (*LightProfile method*), 69
draw_light_2d_linear() (*LightProfile method*), 70
draw_light_3d() (*LightProfile method*), 70
draw_moffat_r() (*in module lenstronomy.GalKin.velocity_util*), 75
draw_xy() (*in module lenstronomy.GalKin.velocity_util*), 75
ds (*LensCosmo attribute*), 46
ds_dds_from_kinematics() (*TDCosmography static method*), 40
DynestySampler (*class in lenstronomy.Sampling.Samplers.dynesty_sampler*), 329
DyPolyChordSampler (*class in lenstronomy.Sampling.Samplers.polychord_sampler*), 331

E

effective_einstein_radius() (*LensProfileAnalysis method*), 33
effective_num_data_points() (*LikelihoodModule method*), 334
electrons2adu() (*in module lenstronomy.Util.data_util*), 362
ell_to_pol (*in module lenstronomy.LensModel.Util.epl_util*), 257
Ellipsoid (*class in lenstronomy.LightModel.Profiles.ellipsoid*), 276
ellipticities() (*in module lenstronomy.Util.analysis_util*), 357
ellipticity() (*LightProfileAnalysis method*), 36
ellipticity2phi_q (*in module lenstronomy.Util.param_util*), 372
ElliSLICE (*class in lenstronomy.LensModel.Profiles.elliptical_density_slice*), 154
enabled() (*MultiPool static method*), 327
EPL (*class in lenstronomy.LensModel.Profiles.epl*), 155
EPL_numba (*class in lenstronomy.LensModel.Profiles.epl_numba*), 159
EPLMajorAxis (*class in lenstronomy.LensModel.Profiles.epl*), 158
epsilon_crit (*Cosmo attribute*), 66
error_map_estimate() (*PsfFitting method*), 389
error_map_estimate_new() (*PsfFitting method*), 389
error_map_source() (*ImageLinearFit method*), 93
error_map_source() (*SingleBandMultiModel method*), 80
error_map_source_plot() (*ModelBandPlot method*), 304
error_map_source_plot() (*ModelPlot method*), 307
error_response() (*ImageLinearFit method*), 93
error_response() (*JointLinear method*), 76
estimate_amp() (*in module lenstronomy.Util.kernel_util*), 368
estimate_noise() (*SingleBand method*), 353
Euclid (*class in lenstronomy.SimulationAPI.ObservationConfig.Euclid*), 348
exposure_map (*ImageNoise attribute*), 52
exposure_time (*Observation attribute*), 352
extinction_map() (*ImageModel method*), 96

F

F() (*SolverFlatLCDM method*), 43
F() (*TNFW method*), 238
F_() (*NFW method*), 194
f_interp() (*Interpol method*), 187
f_x_interp() (*Interpol method*), 187
f_xx_interp() (*Interpol method*), 187
f_xy_interp() (*Interpol method*), 187
f_y_interp() (*Interpol method*), 187
f_yy_interp() (*Interpol method*), 187
fermat_potential() (*LensModel method*), 261
fermat_potential() (*SinglePlane method*), 271
fermat_potential() (*SinglePlaneLOS method*), 102
fermat_potential() (*TDCosmography method*), 40

findBrightImage() (LensEquationSolver method), 252
findOverlap() (in module lenstronomy.Util.image_util), 365
fit_sequence() (FittingSequence method), 383
FittingSequence (class in lenstronomy.Workflow.fitting_sequence), 382
fix_image_parameters() (UpdateManager method), 392
fix_not_computed() (FittingSequence method), 384
fixed_kwargs (UpdateManager attribute), 392
Flexion (class in lenstronomy.LensModel.Profiles.flexion), 161
flexion() (LensModel method), 262
Flexionfg (class in lenstronomy.LensModel.Profiles.flexionfg), 161
flux_array2image_low_high() (AdaptiveGrid method), 85
flux_array2image_low_high() (RegularGrid method), 86
flux_components() (LightProfileAnalysis method), 36
flux_iid() (SingleBand method), 353
flux_noise() (in module lenstronomy.Util.data_util), 361
flux_noise() (SingleBand method), 353
Frame (class in lenstronomy.GalKin.aperture_types), 63
frame_select() (in module lenstronomy.GalKin.aperture_types), 64
function (EPL_numba attribute), 160
function() (ArcPerturbations method), 110
function() (CartShapelets method), 220
function() (Chameleon method), 112, 272
function() (CNFW method), 122
function() (CNFW_ELLIPSE method), 123
function() (ConstMag method), 124
function() (Convergence method), 126
function() (CoreBurkert method), 127
function() (CoredDensity method), 130
function() (CoredDensity2 method), 133
function() (CoredDensityExp method), 136
function() (CoredDensityMST method), 137
function() (CoreSersic method), 289
function() (CSE method), 139
function() (CSEMajorAxis method), 140
function() (CSEMajorAxisSet method), 141
function() (CSEProductAvg method), 142
function() (CSEProductAvgSet method), 143
function() (CurvedArcConst method), 145
function() (CurvedArcConstMST method), 144
function() (CurvedArcSISMST method), 146
function() (CurvedArcSPP method), 148
function() (CurvedArcSPT method), 150
function() (CurvedArcTanDiff method), 152
function() (Dipole method), 153
function() (DoubleChameleon method), 115, 273
function() (DoubleChameleonPointMass method), 120
function() (Ellipsoid method), 276
function() (ElliSLICE method), 154
function() (EPL method), 156
function() (EPLMajorAxis method), 158
function() (Flexion method), 161
function() (Flexionfg method), 161
function() (GaussDecompositionAbstract method), 165
function() (Gaussian method), 176, 276
function() (GaussianEllipse method), 277
function() (GaussianEllipseKappa method), 171
function() (GaussianEllipseKappaSet method), 163
function() (GaussianEllipsePotential method), 173
function() (GaussianKappa method), 175
function() (Hernquist method), 181, 281
function() (Hernquist_Ellipse method), 183
function() (HernquistEllipse method), 281
function() (HernquistEllipseCSE method), 184
function() (Hessian method), 185
function() (Interpol method), 187, 283
function() (InterpolScaled method), 189
function() (LensProfileBase method), 111
function() (LightProfileBase method), 287
function() (Moffat method), 284
function() (MultiGaussian method), 278
function() (MultiGaussianEllipse method), 279
function() (MultiGaussianKappa method), 190
function() (MultiGaussianKappaEllipse method), 192
function() (Multipole method), 193
function() (NFW method), 196
function() (NFW_ELLIPSE method), 199
function() (NFW_ELLIPSE_CSE method), 200
function() (NFWMC method), 202
function() (NIE method), 204, 284
function() (NIE_POTENTIAL method), 206
function() (NIEMajorAxis method), 205
function() (NIEPotentialMajorAxis method), 207
function() (NumericalAlpha method), 208
function() (PEMD method), 214
function() (PJaffe method), 210, 285
function() (PJaffe_Ellipse method), 213
function() (PJaffeEllipse method), 286
function() (PointMass method), 216
function() (PolarShapelets method), 221
function() (PowerLaw method), 286
function() (Sersic method), 217, 288
function() (SersicEllipse method), 218
function() (SersicEllipseKappa method), 217

function() (*SersicElliptic method*), 288
function() (*Shapelets method*), 290
function() (*ShapeletSet method*), 292
function() (*ShapeletSetPolar method*), 295
function() (*ShapeletsPolar method*), 293
function() (*ShapeletsPolarExp method*), 294
function() (*Shear method*), 221
function() (*ShearGammaPsi static method*), 222
function() (*ShearReduced method*), 223
function() (*Shift method*), 125
function() (*SIE method*), 225
function() (*SIS method*), 228
function() (*SIS_truncate method*), 229
function() (*SLIT_Starlets method*), 296
function() (*SPEMD method*), 230
function() (*SPEP method*), 232
function() (*SPLCORE method*), 234
function() (*SPP method*), 236
function() (*TNFW method*), 239
function() (*TNFW_ELLIPSE method*), 242
function() (*TripleChameleon method*), 118, 274
function() (*Uldm method*), 245
function() (*Uniform method*), 297
function_2d() (*SLIT_Starlets method*), 296
function_split() (*MultiGaussian method*), 279
function_split() (*MultiGaussianEllipse method*), 280
function_split() (*ShapeletSet method*), 292
function_split() (*ShapeletSetPolar method*), 295
fwhm (*PSF attribute*), 56
fwhm2sigma() (*in module lenstronomy.Util.util*), 380
fwhm_kernel() (*in module lenstronomy.Util.kernel_util*), 368
FWHMGaussianConvolution (*class in lenstronomy.ImSim.Numerics.convolution*), 84

G

g_() (*NFW method*), 196
Galkin (*class in lenstronomy.GalKin.galkin*), 66
galkin_settings() (*KinematicsAPI method*), 30
GalkinModel (*class in lenstronomy.GalKin.galkin_model*), 68
GalkinObservation (*class in lenstronomy.GalKin.observation*), 73
gamma() (*LensModel method*), 262
gauss_decompose() (*GaussDecompositionAbstract method*), 165
gauss_decompose() (*GaussDecompositionAbstract3D method*), 168
GaussDecompositionAbstract (*class in lenstronomy.LensModel.Profiles.gauss_decomposition*), 164
GaussDecompositionAbstract3D (*class in lenstronomy.LensModel.Profiles.gauss_decomposition*), 168
Gaussian (*class in lenstronomy.LensModel.Profiles.gaussian_potential*), 176
Gaussian (*class in lenstronomy.LightModel.Profiles.gaussian*), 276
gaussian() (*in module lenstronomy.Util.multi_gauss_expansion*), 370
GaussianEllipse (*class in lenstronomy.LightModel.Profiles.gaussian*), 277
GaussianEllipseKappa (*class in lenstronomy.LensModel.Profiles.gaussian_ellipse_kappa*), 170
GaussianEllipseKappaSet (*class in lenstronomy.LensModel.Profiles.gauss_decomposition*), 162
GaussianEllipsePotential (*class in lenstronomy.LensModel.Profiles.gaussian_ellipse_potential*), 172
GaussianKappa (*class in lenstronomy.LensModel.Profiles.gaussian_kappa*), 174
GeneralizedOM (*class in lenstronomy.GalKin.anisotropy*), 61
generated_jit() (*in module lenstronomy.Util.numba_util*), 371
geo_shapiro_delay() (*MultiPlane method*), 105
geo_shapiro_delay() (*MultiPlaneBase method*), 108
geomlinspace() (*in module lenstronomy.LensModel.Util.epl_util*), 257
get_args() (*AlignmentLikelihood static method*), 382
get_axes() (*in module lenstronomy.Util.util*), 378
get_cosmo() (*InvertCosmo method*), 44
get_distance() (*in module lenstronomy.Util.util*), 379
get_distance_from_center() (*SersicUtil method*), 219
get_kappa_1d() (*CTNFWGaussDec method*), 169
get_kappa_1d() (*GaussDecompositionAbstract method*), 166
get_kappa_1d() (*NFWEllipseGaussDec method*), 167
get_kappa_1d() (*SersicEllipseGaussDec method*), 166
get_param_WLS() (*in module lenstronomy.ImSim.de_lens*), 90
get_params() (*ArrayParam method*), 346
get_params() (*LensParam method*), 269
get_params() (*LightParam method*), 299
get_params() (*ModelParamGroup method*), 345
get_params() (*PointSourceParam method*), 321
get_params() (*SingleParam method*), 345

get_params() (*SpecialParam method*), 343
get_scale() (*CTNFWGaussDec method*), 169
get_scale() (*GaussDecompositionAbstract method*), 166
get_scale() (*NFWEllipseGaussDec method*), 168
get_scale() (*SersicEllipseGaussDec method*), 167
GNFW (*class in lenstronomy.LensModel.Profiles.general_nfw*), 176
gradient_map() (*in module lenstronomy.Util.image_util*), 366
grav_pot() (*Hernquist method*), 181
grav_pot() (*PJaffe method*), 210
grav_pot() (*SIE method*), 226
grav_pot() (*SIS method*), 228
grav_pot() (*SPP method*), 236
grav_potential() (*AnalyticKinematics method*), 57
grav_potential() (*NumericKinematics method*), 71
grid_class (*Numerics attribute*), 88
grid_from_coordinate_transform() (*in module lenstronomy.Util.util*), 378
grid_points_spacing (*RegularGrid attribute*), 86
grid_supersampling_factor (*Numerics attribute*), 88

H

h (*LensCosmo attribute*), 46
h_() (*NFW method*), 196
H_n() (*CartShapelets method*), 220
H_n() (*Shapelets method*), 290
half_light_radius() (*in module lenstronomy.Util.analysis_util*), 356
half_light_radius() (*LightProfileAnalysis method*), 36
hermval() (*Shapelets method*), 291
Hernquist (*class in lenstronomy.LensModel.Profiles.hernquist*), 180
Hernquist (*class in lenstronomy.LightModel.Profiles.hernquist*), 281
Hernquist_Ellipse (*class in lenstronomy.LensModel.Profiles.hernquist_ellipse*), 182
HernquistEllipse (*class in lenstronomy.LightModel.Profiles.hernquist*), 281
HernquistEllipseCSE (*class in lenstronomy.LensModel.Profiles.hernquist_ellipse_cse*), 184
Hessian (*class in lenstronomy.LensModel.Profiles.hessian*), 185
hessian (*EPL_numba attribute*), 160
hessian() (*ArcPerturbations method*), 110
hessian() (*CartShapelets method*), 220
hessian() (*Chameleon method*), 113
hessian() (*CNFW method*), 122
hessian() (*CNFW_ELLIPSE method*), 123
hessian() (*ConstMag method*), 124
hessian() (*Convergence method*), 126
hessian() (*CoreBurkert method*), 128
hessian() (*CoredDensity method*), 130
hessian() (*CoredDensity2 method*), 133
hessian() (*CoredDensityExp method*), 136
hessian() (*CoredDensityMST method*), 138
hessian() (*CSE method*), 139
hessian() (*CSEMajorAxis method*), 140
hessian() (*CSEMajorAxisSet method*), 141
hessian() (*CSEProductAvg method*), 142
hessian() (*CSEProductAvgSet method*), 143
hessian() (*CurvedArcConst method*), 145
hessian() (*CurvedArcConstMST method*), 144
hessian() (*CurvedArcSISMST method*), 146
hessian() (*CurvedArcSPP method*), 148
hessian() (*CurvedArcSPT method*), 150
hessian() (*CurvedArcTanDiff method*), 152
hessian() (*Dipole method*), 153
hessian() (*DoubleChameleon method*), 116
hessian() (*DoubleChameleonPointMass method*), 120
hessian() (*ElliSLICE method*), 155
hessian() (*EPL method*), 157
hessian() (*EPLMajorAxis method*), 158
hessian() (*Flexion method*), 161
hessian() (*Flexionfg method*), 162
hessian() (*GaussDecompositionAbstract method*), 166
hessian() (*Gaussian method*), 176
hessian() (*GaussianEllipseKappa method*), 171
hessian() (*GaussianEllipseKappaSet method*), 163
hessian() (*GaussianEllipsePotential method*), 173
hessian() (*GaussianKappa method*), 175
hessian() (*GNFW method*), 177
hessian() (*Hernquist method*), 181
hessian() (*Hernquist_Ellipse method*), 183
hessian() (*HernquistEllipseCSE method*), 184
hessian() (*Hessian method*), 185
hessian() (*Interpol method*), 187
hessian() (*InterpolScaled method*), 189
hessian() (*LensModel method*), 262
hessian() (*LensProfileBase method*), 111
hessian() (*MultiGaussianKappa method*), 191
hessian() (*MultiGaussianKappaEllipse method*), 192
hessian() (*MultiPlane method*), 105
hessian() (*Multipole method*), 193
hessian() (*NFW method*), 196
hessian() (*NFW_ELLIPSE method*), 199
hessian() (*NFW_ELLIPSE_CSE method*), 201
hessian() (*NFWMC method*), 202
hessian() (*NIE method*), 204
hessian() (*NIE_POTENTIAL method*), 207

hessian() (*NIEMajorAxis method*), 206
hessian() (*NIEPotentialMajorAxis method*), 208
hessian() (*NumericalAlpha method*), 208
hessian() (*PEMD method*), 215
hessian() (*PJaffe method*), 211
hessian() (*PJaffe_Ellipse method*), 213
hessian() (*PointMass method*), 216
hessian() (*PolarShapelets method*), 221
hessian() (*Sersic method*), 217
hessian() (*SersicEllipse method*), 218
hessian() (*SersicEllipseKappa method*), 218
hessian() (*Shear method*), 222
hessian() (*ShearGammaPsi method*), 222
hessian() (*ShearReduced method*), 223
hessian() (*Shift method*), 125
hessian() (*SIE method*), 226
hessian() (*SinglePlane method*), 271
hessian() (*SinglePlaneLOS method*), 103
hessian() (*SIS method*), 228
hessian() (*SIS_truncate method*), 229
hessian() (*SPEMD method*), 230
hessian() (*SPEP method*), 232
hessian() (*SPLCORE method*), 234
hessian() (*SPP method*), 237
hessian() (*TNFW method*), 239
hessian() (*TNFW_ELLIPSE method*), 242
hessian() (*TripleChameleon method*), 118
hessian() (*Uldm method*), 245
hessian_eigenvectors() (*LensModelExtensions method*), 265
HST (*class in lenstronomy.SimulationAPI.ObservationConfig.HST*), 348
hyp_2F1() (*in module lenstronomy.GalKin.velocity_util*), 74
hyper2F2_array() (*in module lenstronomy.Util.util*), 380

I

IFUShells (*class in lenstronomy.GalKin.aperture_types*), 65
image() (*ImageModel method*), 96
image2array() (*in module lenstronomy.Util.util*), 377
image2array_masked() (*ImageLinearFit method*), 93
image2source() (*Image2SourceMapping method*), 91
image2source_plane() (*Param method*), 339
Image2SourceMapping (*class in lenstronomy.ImSim.image2source_mapping*), 90
image_amplitude() (*LensedPositions method*), 313
image_amplitude() (*PointSource method*), 317
image_amplitude() (*PointSourceCached method*), 319
image_amplitude() (*PSBase method*), 312
image_amplitude() (*SourcePositions method*), 314
image_amplitude() (*Unlensed method*), 315
image_bkg (*PointSourceVariability attribute*), 355
image_flux_joint() (*Image2SourceMapping method*), 91
image_flux_split() (*Image2SourceMapping method*), 91
image_from_partial() (*PartialImage method*), 89
image_interp() (*Interpol method*), 283
image_joint() (*MultiPatchReconstruction method*), 38
image_linear_solve() (*ImageLinearFit method*), 93
image_linear_solve() (*JointLinear method*), 76
image_linear_solve() (*MultiLinear method*), 79
image_linear_solve() (*SingleBandMultiModel method*), 80
image_model_class (*ModelBand attribute*), 28
image_model_class() (*SimAPI method*), 355
image_pixelbased_solve() (*ImageLinearFit method*), 93
image_position() (*LensedPositions method*), 313
image_position() (*PointSource method*), 317
image_position() (*PointSourceCached method*), 320
image_position() (*PSBase method*), 312
image_position() (*SourcePositions method*), 314
image_position() (*Unlensed method*), 315
image_position_analytical() (*LensEquationSolver method*), 252
image_position_from_source() (*LensEquationSolver method*), 253
image_position_lenstronomy() (*LensEquationSolver method*), 253
image_position_likelihood() (*PositionLikelihood method*), 324
image_position_plot() (*in module lenstronomy.Plots.plot_util*), 310
image_position_stochastic() (*LensEquationSolver method*), 254
image_single_point_source() (*PsfFitting method*), 389
image_time() (*PointSourceVariability method*), 355
ImageData (*class in lenstronomy.Data.imaging_data*), 53
ImageLikelihood (*class in lenstronomy.Sampling.Likelihoods.image_likelihood*), 322
ImageLinearFit (*class in lenstronomy.ImSim.image_linear_solve*), 92
ImageModel (*class in lenstronomy.ImSim.image_model*), 96
ImageNoise (*class in lenstronomy.Data.image_noise*),

52
index2poly() (*ShapeletSetPolar method*), 295
index2poly() (*ShapeletsPolar method*), 293
index2poly() (*ShapeletsPolarExp method*), 294
index_array (*PartialImage attribute*), 89
init_kwargs (*UpdateManager attribute*), 392
Instrument (*class in lenstronomy.SimulationAPI.observation_api*), 351
Interpol (*class in lenstronomy.LensModel.Profiles.interpol*), 186
Interpol (*class in lenstronomy.LightModel.Profiles.interpolation*), 282
interpol_instance() (*MassSlice method*), 99
InterpolScaled (*class in lenstronomy.LensModel.Profiles.interpol*), 188
inverse_transform() (*in module lenstronomy.LightModel.Profiles.starlets_util*), 297
InvertCosmo (*class in lenstronomy.Cosmo.cosmo_solver*), 43
is_master() (*MultiPool method*), 327
is_not_empty() (*SPEMD static method*), 231
is_worker() (*MultiPool method*), 327
Isotropic (*class in lenstronomy.GalKin.anisotropy*), 59

J

jit() (*in module lenstronomy.Util.numba_util*), 371
JointLinear (*class in lenstronomy.ImSim.MultiBand.joint_linear*), 76

K

K() (*Anisotropy method*), 58
K() (*Colin static method*), 62
K() (*Const static method*), 59
K() (*GeneralizedOM method*), 61
K() (*Isotropic static method*), 59
K() (*OsipkovMerritt static method*), 60
K() (*Radial static method*), 60
k_bn() (*SersicUtil method*), 219
k_Re() (*SersicUtil method*), 219
kappa() (*LensModel method*), 262
kappa() (*NFWVirTrunc method*), 203
kappa() (*NIEMajorAxis static method*), 206
kappa2proj_mass() (*LensCosmo method*), 46
kappa_r() (*CoredDensity static method*), 131
kappa_r() (*CoredDensity2 static method*), 134
kappa_r() (*CoredDensityExp static method*), 136
kappa_r() (*Uldm static method*), 245
KDE1D (*class in lenstronomy.Util.prob_density*), 373
KDELikelihood (*class in lenstronomy.Cosmo.kde_likelihood*), 44
kernel_average_pixel() (*in module lenstronomy.Util.kernel_util*), 367
kernel_difference() (*MGEConvolution method*), 85
kernel_gaussian() (*in module lenstronomy.Util.kernel_util*), 368
kernel_norm() (*in module lenstronomy.Util.kernel_util*), 367
kernel_pixel (*PSF attribute*), 56
kernel_pixelsize_change() (*in module lenstronomy.Util.kernel_util*), 367
kernel_point_source (*PSF attribute*), 56
kernel_point_source_supersampled() (*PSF method*), 56
kinematic_lens_profiles() (*KinematicsAPI method*), 30
kinematic_light_profile() (*KinematicsAPI method*), 31
kinematics_modeling_settings() (*KinematicsAPI method*), 31
KinematicsAPI (*class in lenstronomy.Analysis.kinematics_api*), 29
kwargs2args() (*Param method*), 340
kwargs_data (*DataAPI attribute*), 350
kwargs_fixed() (*FittingSequence method*), 384
kwargs_lower (*ArrayParam attribute*), 346
kwargs_lower (*SingleParam attribute*), 345
kwargs_model (*ModelBand attribute*), 28
kwargs_psf (*Observation attribute*), 352
kwargs_single_band() (*DES method*), 347
kwargs_single_band() (*Euclid method*), 348
kwargs_single_band() (*HST method*), 349
kwargs_single_band() (*LSST method*), 349
kwargs_to_args() (*PowerLawParamManager static method*), 251
kwargs_upper (*ArrayParam attribute*), 346
kwargs_upper (*SingleParam attribute*), 345

L

LCDM (*class in lenstronomy.Cosmo.lcdm*), 44
left_multiply() (*LOS static method*), 100
lens_light_model_class (*ModelAPI attribute*), 351
lens_model_class (*ModelAPI attribute*), 351
lens_model_joint() (*MultiPatchReconstruction method*), 38
lens_model_plot() (*in module lenstronomy.Plots.lens_plot*), 300
lens_surface_brightness() (*ImageModel method*), 97
LensCosmo (*class in lenstronomy.Cosmo.lens_cosmo*), 45
LensedLocation (*class in lenstronomy.LensModel.MultiPlane.multi_plane*), 107

LensedPositions (*class in lenstronomy.PointSource.Types.lensed_position*), 313
LensEquationSolver (*class in lenstronomy.LensModel.Solver.lens_equation_solver*), 251
LensModel (*class in lenstronomy.LensModel.lens_model*), 259
LensModelExtensions (*class in lenstronomy.LensModel.lens_model_extensions*), 264
LensParam (*class in lenstronomy.LensModel.lens_param*), 268
LensProfileAnalysis (*class in lenstronomy.Analysis.lens_profile*), 33
LensProfileBase (*class in lenstronomy.LensModel.Profiles.base_profile*), 110
lenstronomy (*module*), 395
lenstronomy.Analysis (*module*), 41
lenstronomy.Analysis.image_reconstruction (*module*), 27
lenstronomy.Analysis.kinematics_api (*module*), 29
lenstronomy.Analysis.lens_profile (*module*), 33
lenstronomy.Analysis.light2mass (*module*), 35
lenstronomy.Analysis.light_profile (*module*), 36
lenstronomy.Analysis.multi_patch_reconstruction (*module*), 38
lenstronomy.Analysis.td_cosmography (*module*), 39
lenstronomy.Conf (*module*), 42
lenstronomy.Conf.config_loader (*module*), 42
lenstronomy.Cosmo (*module*), 50
lenstronomy.Cosmo.background (*module*), 42
lenstronomy.Cosmo.cosmo_solver (*module*), 43
lenstronomy.Cosmo.kde_likelihood (*module*), 44
lenstronomy.Cosmo.lcdm (*module*), 44
lenstronomy.Cosmo.lens_cosmo (*module*), 45
lenstronomy.Cosmo.nfw_param (*module*), 48
lenstronomy.Data (*module*), 56
lenstronomy.Data.coord_transforms (*module*), 50
lenstronomy.Data.image_noise (*module*), 52
lenstronomy.Data.imaging_data (*module*), 53
lenstronomy.Data.pixel_grid (*module*), 54
lenstronomy.Data.psf (*module*), 55
lenstronomy.GalKin (*module*), 76
lenstronomy.GalKin.analytic_kinematics (*module*), 56
lenstronomy.GalKin.anisotropy (*module*), 58
lenstronomy.GalKin.aperture (*module*), 62
lenstronomy.GalKin.aperture_types (*module*), 63
lenstronomy.GalKin.cosmo (*module*), 66
lenstronomy.GalKin.galkin (*module*), 66
lenstronomy.GalKin.galkin_model (*module*), 68
lenstronomy.GalKin.light_profile (*module*), 69
lenstronomy.GalKin.numeric_kinematics (*module*), 71
lenstronomy.GalKin.observation (*module*), 73
lenstronomy.GalKin.psf (*module*), 73
lenstronomy.GalKin.velocity_util (*module*), 74
lenstronomy.ImSim (*module*), 98
lenstronomy.ImSim.de_lens (*module*), 90
lenstronomy.ImSim.image2source_mapping (*module*), 90
lenstronomy.ImSim.image_linear_solve (*module*), 92
lenstronomy.ImSim.image_model (*module*), 96
lenstronomy.ImSim.MultiBand (*module*), 82
lenstronomy.ImSim.MultiBand.joint_linear (*module*), 76
lenstronomy.ImSim.MultiBand.multi_data_base (*module*), 77
lenstronomy.ImSim.MultiBand.multi_linear (*module*), 78
lenstronomy.ImSim.MultiBand.single_band_multi_model (*module*), 80
lenstronomy.ImSim.Numerics (*module*), 90
lenstronomy.ImSim.Numerics.adaptive_numerics (*module*), 82
lenstronomy.ImSim.Numerics.convolution (*module*), 83
lenstronomy.ImSim.Numerics.grid (*module*), 85
lenstronomy.ImSim.Numerics.numba_convolution (*module*), 86
lenstronomy.ImSim.Numerics.numerics (*module*), 87
lenstronomy.ImSim.Numerics.partial_image (*module*), 88
lenstronomy.ImSim.Numerics.point_source_rendering (*module*), 89
lenstronomy.LensModel (*module*), 272
lenstronomy.LensModel.convergence_integrals (*module*), 258
lenstronomy.LensModel.lens_model (*module*), 259

lenstronomy.LensModel.lens_model_extensions (*module*), 264
lenstronomy.LensModel.lens_param (*module*), 268
lenstronomy.LensModel.LightConeSim (*module*), 99
lenstronomy.LensModel.LightConeSim.light_cone (*module*), 98
lenstronomy.LensModel.LineOfSight (*module*), 104
lenstronomy.LensModel.LineOfSight.LOSModels (*module*), 102
lenstronomy.LensModel.LineOfSight.LOSModels.los (*module*), 99
lenstronomy.LensModel.LineOfSight.LOSModels.los_minimal (*module*), 101
lenstronomy.LensModel.LineOfSight.single_plane_los (*module*), 102
lenstronomy.LensModel.MultiPlane (*module*), 109
lenstronomy.LensModel.MultiPlane.multi_plane (*module*), 104
lenstronomy.LensModel.MultiPlane.multi_plane_base (*module*), 108
lenstronomy.LensModel.profile_integrals (*module*), 269
lenstronomy.LensModel.profile_list_base (*module*), 270
lenstronomy.LensModel.Profiles (*module*), 247
lenstronomy.LensModel.Profiles.arc_perturbations (*module*), 109
lenstronomy.LensModel.Profiles.base_profile (*module*), 110
lenstronomy.LensModel.Profiles.chameleon (*module*), 112
lenstronomy.LensModel.Profiles.cnfw (*module*), 121
lenstronomy.LensModel.Profiles.cnfw_ellipse (*module*), 123
lenstronomy.LensModel.Profiles.const_mag (*module*), 123
lenstronomy.LensModel.Profiles.constant_shift (*module*), 125
lenstronomy.LensModel.Profiles.convergence (*module*), 125
lenstronomy.LensModel.Profiles.coreBurkert (*module*), 126
lenstronomy.LensModel.Profiles.cored_density (*module*), 129
lenstronomy.LensModel.Profiles.cored_density_2 (*module*), 132
lenstronomy.LensModel.Profiles.cored_density_exp (*module*), 135
lenstronomy.LensModel.Profiles.cored_density_mst (*module*), 137
lenstronomy.LensModel.Profiles.cored_steep_ellipso (*module*), 138
lenstronomy.LensModel.Profiles.curved_arc_const (*module*), 143
lenstronomy.LensModel.Profiles.curved_arc_sis_mst (*module*), 145
lenstronomy.LensModel.Profiles.curved_arc_spp (*module*), 147
lenstronomy.LensModel.Profiles.curved_arc_spt (*module*), 150
lenstronomy.LensModel.Profiles.curved_arc_tan_diff (*module*), 151
lenstronomy.LensModel.Profiles.dipole (*module*), 153
lenstronomy.LensModel.Profiles.elliptical_density_s (*module*), 154
lenstronomy.LensModel.Profiles.epl (*module*), 155
lenstronomy.LensModel.Profiles.epl_numba (*module*), 159
lenstronomy.LensModel.Profiles.flexion (*module*), 161
lenstronomy.LensModel.Profiles.flexionfg (*module*), 161
lenstronomy.LensModel.Profiles.gauss_decomposition (*module*), 162
lenstronomy.LensModel.Profiles.gaussian_ellipse_kap (*module*), 170
lenstronomy.LensModel.Profiles.gaussian_ellipse_pot (*module*), 172
lenstronomy.LensModel.Profiles.gaussian_kappa (*module*), 174
lenstronomy.LensModel.Profiles.gaussian_potential (*module*), 176
lenstronomy.LensModel.Profiles.general_nfw (*module*), 176
lenstronomy.LensModel.Profiles.hernquist (*module*), 180
lenstronomy.LensModel.Profiles.hernquist_ellipse (*module*), 182
lenstronomy.LensModel.Profiles.hernquist_ellipse_cs (*module*), 184
lenstronomy.LensModel.Profiles.hessian (*module*), 185
lenstronomy.LensModel.Profiles.interpol (*module*), 186
lenstronomy.LensModel.Profiles.multi_gaussian_kappa (*module*), 190
lenstronomy.LensModel.Profiles.multipole (*module*), 193
lenstronomy.LensModel.Profiles.nfw (*module*), 194

lenstronomy.LensModel.Profiles.nfw_ellipse (*module*), 198
lenstronomy.LensModel.Profiles.nfw_ellipse_cse (*module*), 200
lenstronomy.LensModel.Profiles.nfw_mass_concentration (*module*), 201
lenstronomy.LensModel.Profiles.nfw_vir_trunc (*module*), 203
lenstronomy.LensModel.Profiles.nie (*module*), 203
lenstronomy.LensModel.Profiles.nie_potential (*module*), 206
lenstronomy.LensModel.Profiles.numerical_deflections (*module*), 208
lenstronomy.LensModel.Profiles.p_jaffe (*module*), 209
lenstronomy.LensModel.Profiles.p_jaffe_ellipse (*module*), 212
lenstronomy.LensModel.Profiles.pemd (*module*), 213
lenstronomy.LensModel.Profiles.point_mass (*module*), 215
lenstronomy.LensModel.Profiles.sersic (*module*), 216
lenstronomy.LensModel.Profiles.sersic_ellipse_kappa (*module*), 217
lenstronomy.LensModel.Profiles.sersic_ellipse_potential (*module*), 218
lenstronomy.LensModel.Profiles.sersic_utils (*module*), 218
lenstronomy.LensModel.Profiles.shapelet_pot_cartesian (*module*), 220
lenstronomy.LensModel.Profiles.shapelet_pot_polar (*module*), 221
lenstronomy.LensModel.Profiles.shear (*module*), 221
lenstronomy.LensModel.Profiles.sie (*module*), 224
lenstronomy.LensModel.Profiles.sis (*module*), 227
lenstronomy.LensModel.Profiles.sis_truncate (*module*), 229
lenstronomy.LensModel.Profiles.spemd (*module*), 229
lenstronomy.LensModel.Profiles.spep (*module*), 231
lenstronomy.LensModel.Profiles.splcore (*module*), 233
lenstronomy.LensModel.Profiles.spp (*module*), 236
lenstronomy.LensModel.Profiles.tnfw (*module*), 238
lenstronomy.LensModel.Profiles.tnfw_ellipse (*module*), 241
lenstronomy.LensModel.Profiles.uldm (*module*), 243
lenstronomy.LensModel.QuadOptimizer (*module*), 251
lenstronomy.LensModel.QuadOptimizer.multi_plane_fas (*module*), 247
lenstronomy.LensModel.QuadOptimizer.optimizer (*module*), 248
lenstronomy.LensModel.QuadOptimizer.param_manager (*module*), 249
lenstronomy.LensModel.single_plane (*module*), 270
lenstronomy.LensModel.Solver (*module*), 257
lenstronomy.LensModel.Solver.lens_equation_solver (*module*), 251
lenstronomy.LensModel.Solver.solver (*module*), 255
lenstronomy.LensModel.Solver.solver2point (*module*), 256
lenstronomy.LensModel.Solver.solver4point (*module*), 256
lenstronomy.LensModel.Util (*module*), 258
lenstronomy.LensModel.Util.epl_util (*module*), 257
lenstronomy.LightModel (*module*), 299
lenstronomy.LightModel.light_model (*module*), 298
lenstronomy.LightModel.light_param (*module*), 298
lenstronomy.LightModel.Profiles (*module*), 298
lenstronomy.LightModel.Profiles.chameleon (*module*), 272
lenstronomy.LightModel.Profiles.ellipsoid (*module*), 276
lenstronomy.LightModel.Profiles.gaussian (*module*), 276
lenstronomy.LightModel.Profiles.hernquist (*module*), 281
lenstronomy.LightModel.Profiles.interpolation (*module*), 282
lenstronomy.LightModel.Profiles.moffat (*module*), 283
lenstronomy.LightModel.Profiles.nie (*module*), 284
lenstronomy.LightModel.Profiles.p_jaffe (*module*), 285
lenstronomy.LightModel.Profiles.power_law (*module*), 286
lenstronomy.LightModel.Profiles.profile_base (*module*), 287
lenstronomy.LightModel.Profiles.sersic (*module*), 288
lenstronomy.LightModel.Profiles.shapelets

(module), 290
lenstronomy.LightModel.Profiles.shapelets_polar (module), 293
lenstronomy.LightModel.Profiles.starlets (module), 295
lenstronomy.LightModel.Profiles.starlets_util (module), 297
lenstronomy.LightModel.Profiles.uniform (module), 297
lenstronomy.Plots (module), 311
lenstronomy.Plots.chain_plot (module), 299
lenstronomy.Plots.lens_plot (module), 300
lenstronomy.Plots.model_band_plot (module), 302
lenstronomy.Plots.model_plot (module), 306
lenstronomy.Plots.plot_util (module), 309
lenstronomy.PointSource (module), 321
lenstronomy.PointSource.point_source (module), 316
lenstronomy.PointSource.point_source_cached (module), 319
lenstronomy.PointSource.point_source_param (module), 321
lenstronomy.PointSource.Types (module), 316
lenstronomy.PointSource.Types.base_ps (module), 312
lenstronomy.PointSource.Types.lensed_position (module), 313
lenstronomy.PointSource.Types.source_position (module), 314
lenstronomy.PointSource.Types.unlensed (module), 315
lenstronomy.Sampling (module), 347
lenstronomy.Sampling.likelihood (module), 333
lenstronomy.Sampling.Likelihoods (module), 327
lenstronomy.Sampling.Likelihoods.image_likelihood (module), 322
lenstronomy.Sampling.Likelihoods.position_likelihood (module), 323
lenstronomy.Sampling.Likelihoods.prior_likelihood (module), 325
lenstronomy.Sampling.Likelihoods.time_delay_likelihood (module), 326
lenstronomy.Sampling.param_group (module), 344
lenstronomy.Sampling.parameters (module), 335
lenstronomy.Sampling.Pool (module), 328
lenstronomy.Sampling.Pool.multiprocessing (module), 327
lenstronomy.Sampling.Pool.pool (module), 328
lenstronomy.Sampling.sampler (module), 341
lenstronomy.Sampling.Samplers (module), 332
lenstronomy.Sampling.Samplers.base_nested_sampler (module), 328
lenstronomy.Sampling.Samplers.dynesty_sampler (module), 329
lenstronomy.Sampling.Samplers.multinest_sampler (module), 330
lenstronomy.Sampling.Samplers.polychord_sampler (module), 331
lenstronomy.Sampling.special_param (module), 343
lenstronomy.SimulationAPI (module), 356
lenstronomy.SimulationAPI.data_api (module), 349
lenstronomy.SimulationAPI.model_api (module), 350
lenstronomy.SimulationAPI.observation_api (module), 351
lenstronomy.SimulationAPI.observation_constructor (module), 354
lenstronomy.SimulationAPI.ObservationConfig (module), 349
lenstronomy.SimulationAPI.ObservationConfig.DES (module), 347
lenstronomy.SimulationAPI.ObservationConfig.Euclid (module), 348
lenstronomy.SimulationAPI.ObservationConfig.HST (module), 348
lenstronomy.SimulationAPI.ObservationConfig.LSST (module), 349
lenstronomy.SimulationAPI.point_source_variability (module), 354
lenstronomy.SimulationAPI.sim_api (module), 355
lenstronomy.Util (module), 381
lenstronomy.Util.analysis_util (module), 356
lenstronomy.Util.class_creator (module), 358
lenstronomy.Util.constants (module), 360
lenstronomy.Util.correlation (module), 360
lenstronomy.Util.data_util (module), 361
lenstronomy.Util.derivative_util (module), 362
lenstronomy.Util.image_util (module), 364
lenstronomy.Util.kernel_util (module), 366
lenstronomy.Util.mask_util (module), 369
lenstronomy.Util.multi_gauss_expansion (module), 370
lenstronomy.Util.numba_util (module), 371
lenstronomy.Util.param_util (module), 371

lenstronomy.Util.prob_density (*module*), 373
lenstronomy.Util.sampling_util (*module*), 374
lenstronomy.Util.simulation_util (*module*), 375
lenstronomy.Util.util (*module*), 376
lenstronomy.Workflow (*module*), 395
lenstronomy.Workflow.alignment_matching (*module*), 381
lenstronomy.Workflow.fitting_sequence (*module*), 382
lenstronomy.Workflow.psf_fitting (*module*), 387
lenstronomy.Workflow.update_manager (*module*), 392
light2mass_interpol() (*in module lenstronomy.Analysis.light2mass*), 35
light_2d() (*LightProfile method*), 70
light_2d_finite() (*LightProfile method*), 70
light_3d() (*Chameleon method*), 273
light_3d() (*DoubleChameleon method*), 274
light_3d() (*Gaussian method*), 277
light_3d() (*GaussianEllipse method*), 278
light_3d() (*Hernquist method*), 281
light_3d() (*HernquistEllipse method*), 282
light_3d() (*LightProfile method*), 70
light_3d() (*LightProfileBase method*), 287
light_3d() (*MultiGaussian method*), 279
light_3d() (*MultiGaussianEllipse method*), 280
light_3d() (*NIE method*), 284
light_3d() (*PJaffe method*), 285
light_3d() (*PJaffeEllipse method*), 286
light_3d() (*PowerLaw method*), 287
light_3d() (*TripleChameleon method*), 275
light_3d_interp() (*LightProfile method*), 70
LightCone (*class in lenstronomy.LensModel.LightConeSim.light_cone*), 98
LightModel (*class in lenstronomy.LightModel.light_model*), 298
LightParam (*class in lenstronomy.LightModel.light_param*), 298
LightProfile (*class in lenstronomy.GalKin.light_profile*), 69
LightProfileAnalysis (*class in lenstronomy.Analysis.light_profile*), 36
LightProfileBase (*class in lenstronomy.LightModel.Profiles.profile_base*), 287
likelihood() (*AlignmentLikelihood method*), 382
likelihood() (*KDE1D method*), 374
likelihood() (*LikelihoodModule method*), 334
likelihood_data_given_model() (*ImageLinearFit method*), 94
likelihood_data_given_model() (*JointLinear method*), 77
likelihood_data_given_model() (*MultiLinear method*), 79
likelihood_data_given_model() (*SingleBandMultiModel method*), 81
LikelihoodModule (*class in lenstronomy.Sampling.likelihood*), 333
likelihoodModule (*FittingSequence attribute*), 384
linear_param_from_kwargs() (*ImageLinearFit method*), 94
linear_param_from_kwargs() (*PointSource method*), 317
linear_response_matrix() (*ImageLinearFit method*), 94
linear_response_matrix() (*JointLinear method*), 77
linear_response_matrix() (*SingleBandMultiModel method*), 81
linear_response_set() (*PointSource method*), 317
linear_solver (*Param attribute*), 340
local_lensing_effect() (*LensProfileAnalysis method*), 34
local_minima_2d (*in module lenstronomy.Util.util*), 380
log_likelihood() (*DynestySampler method*), 329
log_likelihood() (*DyPolyChordSampler method*), 332
log_likelihood() (*ImageData method*), 54
log_likelihood() (*LikelihoodModule method*), 335
log_likelihood() (*MultiNestSampler method*), 330
log_likelihood() (*NestedSampler method*), 329
logL() (*ImageLikelihood method*), 322
logL() (*LikelihoodModule method*), 334
logL() (*MultiplaneFast method*), 247
logL() (*PositionLikelihood method*), 324
logL() (*PriorLikelihood method*), 326
logL() (*TimeDelayLikelihood method*), 326
logLikelihood() (*KDELikelihood method*), 44
LOS (*class in lenstronomy.LensModel.LineOfSight.LOSModels.los*), 99
LOSMinimal (*class in lenstronomy.LensModel.LineOfSight.LOSModels.los_minimal*), 101
lower_limit_default (*CartShapelets attribute*), 220
lower_limit_default (*Chameleon attribute*), 113, 273
lower_limit_default (*CNFW attribute*), 122
lower_limit_default (*CNFW_ELLIPSE attribute*), 123
lower_limit_default (*ConstMag attribute*), 124

lower_limit_default (*Convergence attribute*), 126
lower_limit_default (*CoreBurkert attribute*), 128
lower_limit_default (*CoredDensity attribute*), 131
lower_limit_default (*CoredDensity2 attribute*), 134
lower_limit_default (*CoredDensityExp attribute*), 136
lower_limit_default (*CoredDensityMST attribute*), 138
lower_limit_default (*CoreSersic attribute*), 289
lower_limit_default (*CSE attribute*), 139
lower_limit_default (*CSEMajorAxis attribute*), 140
lower_limit_default (*CSEProductAvg attribute*), 142
lower_limit_default (*CTNFWGaussDec attribute*), 169
lower_limit_default (*CurvedArcConst attribute*), 145
lower_limit_default (*CurvedArcConstMST attribute*), 144
lower_limit_default (*CurvedArcSISMST attribute*), 147
lower_limit_default (*CurvedArcSPP attribute*), 149
lower_limit_default (*CurvedArcSPT attribute*), 151
lower_limit_default (*CurvedArcTanDiff attribute*), 152
lower_limit_default (*Dipole attribute*), 153
lower_limit_default (*DoubleChameleon attribute*), 116, 274
lower_limit_default (*DoubleChameleonPointMass attribute*), 121
lower_limit_default (*ElliSLICE attribute*), 155
lower_limit_default (*EPL attribute*), 157
lower_limit_default (*EPL_numba attribute*), 160
lower_limit_default (*Flexion attribute*), 161
lower_limit_default (*Flexionfg attribute*), 162
lower_limit_default (*Gaussian attribute*), 176
lower_limit_default (*GaussianEllipse attribute*), 278
lower_limit_default (*GaussianEllipseKappa attribute*), 171
lower_limit_default (*GaussianEllipseKappaSet attribute*), 164
lower_limit_default (*GaussianEllipsePotential attribute*), 173
lower_limit_default (*GaussianKappa attribute*), 175
lower_limit_default (*GNFW attribute*), 178
lower_limit_default (*Hernquist attribute*), 181
lower_limit_default (*Hernquist_Ellipse attribute*), 183
lower_limit_default (*HernquistEllipse attribute*), 282
lower_limit_default (*HernquistEllipseCSE attribute*), 184
lower_limit_default (*Hessian attribute*), 186
lower_limit_default (*Interpol attribute*), 188, 283
lower_limit_default (*InterpolScaled attribute*), 190
lower_limit_default (*LOS attribute*), 100
lower_limit_default (*LOSMinimal attribute*), 101
lower_limit_default (*MultiGaussian attribute*), 279
lower_limit_default (*MultiGaussianEllipse attribute*), 280
lower_limit_default (*MultiGaussianKappa attribute*), 191
lower_limit_default (*MultiGaussianKappaEllipse attribute*), 192
lower_limit_default (*Multipole attribute*), 194
lower_limit_default (*NFW attribute*), 196
lower_limit_default (*NFW_ELLIPSE attribute*), 199
lower_limit_default (*NFW_ELLIPSE_CSE attribute*), 201
lower_limit_default (*NFWEllipseGaussDec attribute*), 168
lower_limit_default (*NFWMC attribute*), 202
lower_limit_default (*NIE attribute*), 204, 285
lower_limit_default (*NIE_POTENTIAL attribute*), 207
lower_limit_default (*PEMD attribute*), 215
lower_limit_default (*PJaffe attribute*), 211, 285
lower_limit_default (*PJaffe_Ellipse attribute*), 213
lower_limit_default (*PJaffeEllipse attribute*), 286
lower_limit_default (*PointMass attribute*), 216
lower_limit_default (*PolarShapelets attribute*), 221
lower_limit_default (*PowerLaw attribute*), 287
lower_limit_default (*Sersic attribute*), 217, 288
lower_limit_default (*SersicEllipse attribute*), 218
lower_limit_default (*SersicEllipseGaussDec attribute*), 167
lower_limit_default (*SersicEllipseKappa attribute*), 218
lower_limit_default (*SersicElliptic attribute*), 289
lower_limit_default (*Shapelets attribute*), 291

lower_limit_default (*ShapeletSet attribute*), 293
lower_limit_default (*ShapeletSetPolar attribute*), 295
lower_limit_default (*ShapeletsPolar attribute*), 293
lower_limit_default (*ShapeletsPolarExp attribute*), 294
lower_limit_default (*Shear attribute*), 222
lower_limit_default (*ShearGammaPsi attribute*), 223
lower_limit_default (*ShearReduced attribute*), 224
lower_limit_default (*Shift attribute*), 125
lower_limit_default (*SIE attribute*), 226
lower_limit_default (*SIS attribute*), 228
lower_limit_default (*SIS_truncate attribute*), 229
lower_limit_default (*SLIT_Starlets attribute*), 297
lower_limit_default (*SPEMD attribute*), 231
lower_limit_default (*SPEP attribute*), 232
lower_limit_default (*SPLCORE attribute*), 234
lower_limit_default (*SPP attribute*), 237
lower_limit_default (*TNFW attribute*), 240
lower_limit_default (*TNFW_ELLIPSE attribute*), 243
lower_limit_default (*TripleChameleon attribute*), 119, 276
lower_limit_default (*Uldm attribute*), 246
lower_limit_default (*Uniform attribute*), 297
LSST (*class in lenstronomy.SimulationAPI.ObservationConfig.LSST*), 349

M

M200() (*NFWParam static method*), 48
M_r200() (*NFWParam method*), 49
magnification() (*LensModel method*), 263
magnification_finite() (*LensModelExtensions method*), 266
magnification_finite_adaptive() (*LensModelExtensions method*), 266
magnification_plot() (*ModelBandPlot method*), 304
magnification_plot() (*ModelPlot method*), 308
magnitude2amplitude() (*SimAPI method*), 355
magnitude2cps() (*in module lenstronomy.Util.data_util*), 361
magnitude2cps() (*SingleBand method*), 353
make_grid() (*in module lenstronomy.Util.util*), 377
make_grid_transformed() (*in module lenstronomy.Util.util*), 377
make_grid_with_coordtransform() (*in module lenstronomy.Util.util*), 377
make_subgrid() (*in module lenstronomy.Util.util*), 381
map() (*MultiPool method*), 327
map_coord2pix() (*Coordinates method*), 50
map_coord2pix() (*in module lenstronomy.Util.util*), 376
map_mu_sigma_skw() (*SkewGaussian method*), 373
map_pix2coord() (*Coordinates method*), 50
marginalisation_const() (*in module lenstronomy.ImSim.de_lens*), 90
marginalization_new() (*in module lenstronomy.ImSim.de_lens*), 90
mask_azimuthal() (*in module lenstronomy.Util.mask_util*), 369
mask_center_2d() (*in module lenstronomy.Util.mask_util*), 369
mask_ellipse() (*in module lenstronomy.Util.mask_util*), 370
mask_half_moon() (*in module lenstronomy.Util.mask_util*), 370
mask_point_source() (*PsfFitting static method*), 390
mass_2d() (*CNFW method*), 122
mass_2d() (*CoreBurkert method*), 128
mass_2d() (*CoredDensity method*), 131
mass_2d() (*CoredDensity2 static method*), 134
mass_2d() (*CoredDensityExp method*), 136
mass_2d() (*GaussianEllipsePotential method*), 173
mass_2d() (*GaussianKappa method*), 175
mass_2d() (*GNFW method*), 178
mass_2d() (*Hernquist method*), 181
mass_2d() (*Hernquist_Ellipse method*), 183
mass_2d() (*NFW method*), 196
mass_2d() (*PJaffe method*), 211
mass_2d() (*SIE method*), 226
mass_2d() (*SinglePlane method*), 271
mass_2d() (*SinglePlaneLOS method*), 103
mass_2d() (*SIS static method*), 228
mass_2d() (*SPLCORE method*), 234
mass_2d() (*SPP method*), 237
mass_2d() (*TNFW method*), 240
mass_2d() (*Uldm method*), 246
mass_2d_lens() (*GaussianEllipsePotential method*), 173
mass_2d_lens() (*GaussianKappa method*), 175
mass_2d_lens() (*Hernquist method*), 182
mass_2d_lens() (*Hernquist_Ellipse method*), 183
mass_2d_lens() (*LensProfileBase method*), 111
mass_2d_lens() (*NFW method*), 196
mass_2d_lens() (*SIE method*), 226
mass_2d_lens() (*SIS method*), 228
mass_2d_lens() (*SPLCORE method*), 235
mass_2d_lens() (*SPP method*), 237
mass_3d() (*CNFW method*), 122

mass_3d() (*CoreBurkert method*), 128
mass_3d() (*CoredDensity static method*), 131
mass_3d() (*CoredDensity2 static method*), 134
mass_3d() (*CoredDensityExp static method*), 137
mass_3d() (*GaussianEllipsePotential method*), 173
mass_3d() (*GaussianKappa method*), 175
mass_3d() (*GNFW static method*), 178
mass_3d() (*Hernquist method*), 182
mass_3d() (*Hernquist_Ellipse method*), 184
mass_3d() (*NFW method*), 197
mass_3d() (*NumericKinematics method*), 72
mass_3d() (*PJaffe method*), 211
mass_3d() (*SIE static method*), 227
mass_3d() (*SinglePlane method*), 272
mass_3d() (*SinglePlaneLOS method*), 103
mass_3d() (*SIS static method*), 228
mass_3d() (*SPLCORE method*), 235
mass_3d() (*SPP static method*), 237
mass_3d() (*TNFW method*), 240
mass_3d() (*Uldm method*), 246
mass_3d_lens() (*Chameleon method*), 113
mass_3d_lens() (*CNFW method*), 123
mass_3d_lens() (*CNFW_ELLIPSE method*), 123
mass_3d_lens() (*CoredDensity method*), 131
mass_3d_lens() (*CoredDensity2 method*), 134
mass_3d_lens() (*CoredDensityExp method*), 137
mass_3d_lens() (*DoubleChameleon method*), 116
mass_3d_lens() (*EPL method*), 157
mass_3d_lens() (*GaussianEllipsePotential method*), 173
mass_3d_lens() (*GaussianKappa method*), 175
mass_3d_lens() (*GNFW method*), 178
mass_3d_lens() (*Hernquist method*), 182
mass_3d_lens() (*Hernquist_Ellipse method*), 184
mass_3d_lens() (*LensProfileBase method*), 111
mass_3d_lens() (*MultiGaussianKappa method*), 191
mass_3d_lens() (*MultiGaussianKappaEllipse method*), 192
mass_3d_lens() (*NFW method*), 197
mass_3d_lens() (*NFW_ELLIPSE method*), 200
mass_3d_lens() (*NIE method*), 204
mass_3d_lens() (*PEMD method*), 215
mass_3d_lens() (*PJaffe method*), 211
mass_3d_lens() (*PJaffe_Ellipse method*), 213
mass_3d_lens() (*SIE method*), 227
mass_3d_lens() (*SIS method*), 228
mass_3d_lens() (*SPEP method*), 232
mass_3d_lens() (*SPLCORE method*), 235
mass_3d_lens() (*SPP method*), 237
mass_3d_lens() (*TNFW_ELLIPSE method*), 243
mass_3d_lens() (*TripleChameleon method*), 119
mass_3d_lens() (*Uldm method*), 246
mass_enclosed_2d() (*ProfileIntegrals method*), 269
mass_enclosed_3d() (*ProfileIntegrals method*), 270
mass_fraction_within_radius() (*LensProfileAnalysis method*), 34
mass_in_coin() (*LensCosmo method*), 46
mass_in_theta_E() (*LensCosmo method*), 46
mass_ratio() (*DipoleUtil static method*), 153
mass_tot() (*Hernquist method*), 182
mass_tot() (*PJaffe method*), 211
MassSlice (*class in lenstronomy.LensModel.LightConeSim.light_cone*), 99
match_kernel_size() (*in module lenstronomy.Util.kernel_util*), 369
mcmc() (*FittingSequence method*), 384
mcmc_emcee() (*Sampler method*), 341
mcmc_zeus() (*Sampler method*), 341
merge_dicts() (*in module lenstronomy.Util.util*), 376
mge_1d() (*in module lenstronomy.Util.multi_gauss_expansion*), 370
mge_kernel() (*in module lenstronomy.Util.kernel_util*), 369
MGEConvolution (*class in lenstronomy.ImSim.Numerics.convolution*), 85
min_approx (*in module lenstronomy.LensModel.Util.epl_util*), 258
min_square_dist() (*in module lenstronomy.Util.util*), 379
model (*ModelBand attribute*), 29
model_name (*CNFW attribute*), 123
model_name (*Convergence attribute*), 126
model_name (*CoredDensity2 attribute*), 134
model_plot() (*ModelBandPlot method*), 304
model_plot() (*ModelPlot method*), 308
ModelAPI (*class in lenstronomy.SimulationAPI.model_api*), 350
ModelBand (*class in lenstronomy.Analysis.image_reconstruction*), 28
ModelBandPlot (*class in lenstronomy.Plots.model_band_plot*), 302
ModelParamGroup (*class in lenstronomy.Sampling.param_group*), 344
ModelPlot (*class in lenstronomy.Plots.model_plot*), 306
Moffat (*class in lenstronomy.LightModel.Profiles.moffat*), 283
moffat_fwhm_alpha() (*in module lenstronomy.GalKin.velocity_util*), 75
moffat_r() (*in module lenstronomy.GalKin.velocity_util*), 75
moments() (*in module lenstronomy.Util.analysis_util*), 357
mst_invariant_differential() (*LensProfileAnalysis method*), 34

multi_gaussian_decomposition() (*LightProfileAnalysis method*), 37
multi_gaussian_decomposition_ellipse() (*LightProfileAnalysis method*), 37
multi_gaussian_lens() (*LensProfileAnalysis method*), 35
MultiBandImageReconstruction (*class in lenstronomy.Analysis.image_reconstruction*), 27
MultiDataBase (*class in lenstronomy.ImSim.MultiBand.multi_data_base*), 77
MultiGaussian (*class in lenstronomy.LightModel.Profiles.gaussian*), 278
MultiGaussianConvolution (*class in lenstronomy.ImSim.Numerics.convolution*), 84
MultiGaussianEllipse (*class in lenstronomy.LightModel.Profiles.gaussian*), 279
MultiGaussianKappa (*class in lenstronomy.LensModel.Profiles.multi_gaussian_kappa*), 190
MultiGaussianKappaEllipse (*class in lenstronomy.LensModel.Profiles.multi_gaussian_kappa*), 191
MultiLinear (*class in lenstronomy.ImSim.MultiBand.multi_linear*), 78
MultiNestSampler (*class in lenstronomy.Sampling.Samplers.multinest_sampler*), 330
MultiPatchReconstruction (*class in lenstronomy.Analysis.multi_patch_reconstruction*), 38
MultiPlane (*class in lenstronomy.LensModel.MultiPlane.multi_plane*), 104
MultiPlaneBase (*class in lenstronomy.LensModel.MultiPlane.multi_plane_base*), 108
MultiplaneFast (*class in lenstronomy.LensModel.QuadOptimizer.multi_plane_fast*), 247
Multipole (*class in lenstronomy.LensModel.Profiles.multipole*), 193
MultiPool (*class in lenstronomy.Sampling.Pool.multiprocessing*), 327

N

nan_to_num (*in module lenstronomy.Util.numba_util*), 371
nan_to_num_arr (*in module lenstronomy.Util.numba_util*), 371
nan_to_num_single (*in module lenstronomy.Util.numba_util*), 371
negativelogL() (*LikelihoodModule method*), 335
neighborSelect (*in module lenstronomy.Util.util*), 380
nested_sampling() (*FittingSequence method*), 384
NestedSampler (*class in lenstronomy.Sampling.Samplers.base_nested_sampler*), 328
NFW (*class in lenstronomy.LensModel.Profiles.nfw*), 194
nfw_angle2physical() (*LensCosmo method*), 46
NFW_ELLIPSE (*class in lenstronomy.LensModel.Profiles.nfw_ellipse*), 198
NFW_ELLIPSE_CSE (*class in lenstronomy.LensModel.Profiles.nfw_ellipse_cse*), 200
nfw_M_theta_r200() (*LensCosmo method*), 46
nfw_Mz() (*NFWParam method*), 49
nfw_physical2angle() (*LensCosmo method*), 47
nfwAlpha() (*GNFW method*), 179
nfwAlpha() (*NFW method*), 197
nfwAlpha() (*TNFW method*), 240
NFWEllipseGaussDec (*class in lenstronomy.LensModel.Profiles.gauss_decomposition*), 167
nfwGamma() (*GNFW method*), 179
nfwGamma() (*NFW method*), 197
nfwGamma() (*TNFW method*), 240
NFWMC (*class in lenstronomy.LensModel.Profiles.nfw_mass_concentration*), 201
NFWParam (*class in lenstronomy.Cosmo.nfw_param*), 48
nfwParam_physical() (*LensCosmo method*), 46
nfwPot() (*NFW method*), 197
nfwPot() (*TNFW method*), 241
NFWVirTrunc (*class in lenstronomy.LensModel.Profiles.nfw_vir_trunc*), 203
NIE (*class in lenstronomy.LensModel.Profiles.nie*), 203
NIE (*class in lenstronomy.LightModel.Profiles.nie*), 284
NIE_POTENTIAL (*class in lenstronomy.LensModel.Profiles.nie_potential*), 206
NIEMajorAxis (*class in lenstronomy.LensModel.Profiles.nie*), 205
NIEPotentialMajorAxis (*class in lenstronomy.LensModel.Profiles.nie_potential*), 207
noise_for_model() (*SingleBand method*), 354
norm_residuals (*ModelBand attribute*), 29
normalized_residual_plot() (*ModelBandPlot method*), 304
normalized_residual_plot() (*ModelPlot method*), 308
num_bands (*MultiDataBase attribute*), 78
num_basis() (*PointSource method*), 318
num_data (*ImageLikelihood attribute*), 322
num_data (*LikelihoodModule attribute*), 335
num_data (*PositionLikelihood attribute*), 324

num_data (*TimeDelayLikelihood attribute*), 326
num_data_evaluate (*ImageLinearFit attribute*), 95
num_data_evaluate (*MultiDataBase attribute*), 78
num_grid_points_axes (*RegularGrid attribute*), 86
num_param (*AlignmentLikelihood attribute*), 382
num_param() (*LensParam method*), 269
num_param() (*LightParam method*), 299
num_param() (*Param method*), 340
num_param() (*PointSourceParam method*), 321
num_param() (*ShapeletsPolar static method*), 293
num_param() (*ShapeletsPolarExp static method*), 294
num_param() (*SpecialParam method*), 343
num_param_linear() (*ImageLikelihood method*), 323
num_param_linear() (*ImageLinearFit method*), 95
num_param_linear() (*LightParam method*), 299
num_param_linear() (*MultiDataBase method*), 78
num_param_linear() (*Param method*), 340
num_param_linear() (*PointSourceParam method*), 321
num_param_linear() (*SingleBandMultiModel method*), 81
num_params() (*ArrayParam method*), 346
num_params() (*ModelParamGroup method*), 345
num_params() (*SingleParam method*), 345
num_partial (*PartialImage attribute*), 89
num_pixel (*PixelGrid attribute*), 55
num_pixel_axes (*PixelGrid attribute*), 55
num_point_source_images (*Param attribute*), 340
num_response_list (*MultiDataBase attribute*), 78
num_segments (*Aperture attribute*), 63
num_segments (*Frame attribute*), 64
num_segments (*IFUShells attribute*), 65
num_segments (*Shell attribute*), 65
num_segments (*Slit attribute*), 63
numba_conf() (*in module lenstronomy.Conf.config_loader*), 42
NumbaConvolution (*class in lenstronomy.ImSim.Numerics.numba_convolution*), 86
NumericalAlpha (*class in lenstronomy.LensModel.Profiles.numerical_deflections*), 208
NumericKinematics (*class in lenstronomy.GalKin.numeric_kinematics*), 71
Numerics (*class in lenstronomy.ImSim.Numerics.numerics*), 87

O

Observation (*class in lenstronomy.SimulationAPI.observation_api*), 351
observation_constructor() (*in module lenstronomy.SimulationAPI.observation_constructor*), 354
observed2flat_convention() (*MultiPlane method*), 106
on (*ArrayParam attribute*), 346
on (*SingleParam attribute*), 346
optimize() (*Optimizer method*), 249
Optimizer (*class in lenstronomy.LensModel.QuadOptimizer.optimizer*), 248
OsipkovMerritt (*class in lenstronomy.GalKin.anisotropy*), 60

P

Param (*class in lenstronomy.Sampling.parameters*), 335
param_chi_square_penalty() (*PowerLawParamManager method*), 251
param_class (*FittingSequence attribute*), 385
param_class (*UpdateManager attribute*), 392
param_conv() (*EPL method*), 157
param_conv() (*NIE method*), 205
param_conv() (*NIE_POTENTIAL method*), 207
param_convert() (*Chameleon method*), 113
param_limits (*LikelihoodModule attribute*), 335
param_limits() (*Param method*), 340
param_name_list (*LightParam attribute*), 299
param_names (*CartShapelets attribute*), 220
param_names (*Chameleon attribute*), 114, 273
param_names (*CNFW attribute*), 123
param_names (*CNFW_ELLIPSE attribute*), 123
param_names (*ConstMag attribute*), 124
param_names (*Convergence attribute*), 126
param_names (*CoreBurkert attribute*), 128
param_names (*CoredDensity attribute*), 131
param_names (*CoredDensity2 attribute*), 134
param_names (*CoredDensityExp attribute*), 137
param_names (*CoredDensityMST attribute*), 138
param_names (*CoreSersic attribute*), 289
param_names (*CSE attribute*), 139
param_names (*CSEMajorAxis attribute*), 140
param_names (*CSEProductAvg attribute*), 142
param_names (*CTNFWGaussDec attribute*), 169
param_names (*CurvedArcConst attribute*), 145
param_names (*CurvedArcConstMST attribute*), 144
param_names (*CurvedArcSISMST attribute*), 147
param_names (*CurvedArcSPP attribute*), 149
param_names (*CurvedArcSPT attribute*), 151
param_names (*CurvedArcTanDiff attribute*), 152
param_names (*Dipole attribute*), 153
param_names (*DoubleChameleon attribute*), 117, 274
param_names (*DoubleChameleonPointMass attribute*), 121
param_names (*ElliSLICE attribute*), 155
param_names (*EPL attribute*), 157

param_names (*EPL_numba attribute*), 160
param_names (*EPLMajorAxis attribute*), 159
param_names (*Flexion attribute*), 161
param_names (*Flexionfg attribute*), 162
param_names (*Gaussian attribute*), 176
param_names (*GaussianEllipse attribute*), 278
param_names (*GaussianEllipseKappa attribute*), 171
param_names (*GaussianEllipseKappaSet attribute*), 164
param_names (*GaussianEllipsePotential attribute*), 174
param_names (*GaussianKappa attribute*), 175
param_names (*GNFW attribute*), 179
param_names (*Hernquist attribute*), 182
param_names (*Hernquist_Ellipse attribute*), 184
param_names (*HernquistEllipse attribute*), 282
param_names (*HernquistEllipseCSE attribute*), 184
param_names (*Hessian attribute*), 186
param_names (*Interpol attribute*), 188, 283
param_names (*InterpolScaled attribute*), 190
param_names (*LOS attribute*), 101
param_names (*LOSMinimal attribute*), 101
param_names (*MultiGaussian attribute*), 279
param_names (*MultiGaussianEllipse attribute*), 280
param_names (*MultiGaussianKappa attribute*), 191
param_names (*MultiGaussianKappaEllipse attribute*), 193
param_names (*Multipole attribute*), 194
param_names (*NFW attribute*), 198
param_names (*NFW_ELLIPSE attribute*), 200
param_names (*NFW_ELLIPSE_CSE attribute*), 201
param_names (*NFWEllipseGaussDec attribute*), 168
param_names (*NFWMC attribute*), 202
param_names (*NIE attribute*), 205, 285
param_names (*NIE_POTENTIAL attribute*), 207
param_names (*NIEMajorAxis attribute*), 206
param_names (*NIEPotentialMajorAxis attribute*), 208
param_names (*PEMD attribute*), 215
param_names (*PJaffe attribute*), 212, 285
param_names (*PJaffe_Ellipse attribute*), 213
param_names (*PJaffeEllipse attribute*), 286
param_names (*PointMass attribute*), 216
param_names (*PolarShapelets attribute*), 221
param_names (*PowerLaw attribute*), 287
param_names (*Sersic attribute*), 217, 288
param_names (*SersicEllipse attribute*), 218
param_names (*SersicEllipseGaussDec attribute*), 167
param_names (*SersicEllipseKappa attribute*), 218
param_names (*SersicElliptic attribute*), 289
param_names (*Shapelets attribute*), 291
param_names (*ShapeletSet attribute*), 293
param_names (*ShapeletSetPolar attribute*), 295
param_names (*ShapeletsPolar attribute*), 293
param_names (*ShapeletsPolarExp attribute*), 294
param_names (*Shear attribute*), 222
param_names (*ShearGammaPsi attribute*), 223
param_names (*ShearReduced attribute*), 224
param_names (*Shift attribute*), 125
param_names (*SIE attribute*), 227
param_names (*SIS attribute*), 228
param_names (*SIS_truncate attribute*), 229
param_names (*SLIT_Starlets attribute*), 297
param_names (*SPEMD attribute*), 231
param_names (*SPEP attribute*), 233
param_names (*SPLCORE attribute*), 235
param_names (*SPP attribute*), 238
param_names (*TNFW attribute*), 241
param_names (*TNFW_ELLIPSE attribute*), 243
param_names (*TripleChameleon attribute*), 119, 276
param_names (*Uldm attribute*), 246
param_names (*Uniform attribute*), 297
param_names_latex (*ShapeletsPolar attribute*), 294
param_names_latex (*SLIT_Starlets attribute*), 297
param_names_latex (*Uniform attribute*), 297
param_transform() (*SPEMD method*), 231
parameter_state (*UpdateManager attribute*), 392
partial_array() (*PartialImage method*), 89
PartialImage (*class in lenstronomy.ImSim.Numerics.partial_image*), 88
pdf() (*SkewGaussian method*), 373
pdf_skew() (*SkewGaussian method*), 373
PEMD (*class in lenstronomy.LensModel.Profiles.pemd*), 213
phi_n() (*CartShapelets method*), 220
phi_n() (*Shapelets method*), 291
phi_q2_ellipticity (*in module lenstronomy.Util.param_util*), 372
phys2arcsec_lens() (*LensCosmo method*), 47
physical2lensing_conversion() (*ModelAPI method*), 351
PhysicalLocation (*class in lenstronomy.LensModel.MultiPlane.multi_plane*), 107
pixel_area (*Coordinates attribute*), 51
pixel_coordinates (*PixelGrid attribute*), 55
pixel_grid_joint (*MultiPatchReconstruction attribute*), 38
pixel_kernel() (*in module lenstronomy.Util.kernel_util*), 367
pixel_kernel() (*MultiGaussianConvolution method*), 84
pixel_kernel() (*PixelKernelConvolution method*), 83
pixel_width (*Coordinates attribute*), 51
PixelGrid (*class in lenstronomy.Data.pixel_grid*), 54
PixelKernelConvolution (*class in lenstronomy.ImSim.Numerics.convolution*), 83

PJaffe (*class in lenstronomy.LensModel.Profiles.p_jaffe*), 209
PJaffe (*class in lenstronomy.LightModel.Profiles.p_jaffe*), 285
PJaffe_Ellipse (*class in lenstronomy.LensModel.Profiles.p_jaffe_ellipse*), 212
PJaffeEllipse (*class in lenstronomy.LightModel.Profiles.p_jaffe*), 286
plot_arc() (*in module lenstronomy.Plots.lens_plot*), 301
plot_chain() (*in module lenstronomy.Plots.chain_plot*), 299
plot_chain_list() (*in module lenstronomy.Plots.chain_plot*), 299
plot_extinction_map() (*ModelBandPlot method*), 305
plot_extinction_map() (*ModelPlot method*), 308
plot_line_set() (*in module lenstronomy.Plots.plot_util*), 310
plot_main() (*ModelBandPlot method*), 305
plot_main() (*ModelPlot method*), 308
plot_mcmc_behaviour() (*in module lenstronomy.Plots.chain_plot*), 300
plot_separate() (*ModelBandPlot method*), 305
plot_separate() (*ModelPlot method*), 308
plot_subtract_from_data_all() (*ModelBandPlot method*), 305
plot_subtract_from_data_all() (*ModelPlot method*), 308
point_like_source_cutouts() (*PsfFitting static method*), 390
point_source() (*ImageModel method*), 97
point_source_linear_response_set() (*ImageLinearFit method*), 95
point_source_list() (*PointSource method*), 318
point_source_model_class (*ModelAPI attribute*), 351
point_source_rendering() (*PointSourceRendering method*), 89
point_source_time() (*PointSourceVariability method*), 355
PointMass (*class in lenstronomy.LensModel.Profiles.point_mass*), 215
points_on_circle() (*in module lenstronomy.Util.util*), 380
PointSource (*class in lenstronomy.PointSource.point_source*), 316
PointSourceCached (*class in lenstronomy.PointSource.point_source_cached*), 319
PointSourceParam (*class in lenstronomy.PointSource.point_source_param*), 321
PointSourceRendering (*class in lenstronomy.ImSim.Numerics.point_source_rendering*), 89
PointSourceVariability (*class in lenstronomy.SimulationAPI.point_source_variability*), 354
pol_to_cart (*in module lenstronomy.LensModel.Util.epl_util*), 258
pol_to_ell (*in module lenstronomy.LensModel.Util.epl_util*), 258
polar2cart() (*in module lenstronomy.Util.param_util*), 371
PolarShapelets (*class in lenstronomy.LensModel.Profiles.shapelet_pot_polar*), 221
poly2index() (*ShapeletsPolar static method*), 294
poly2index() (*ShapeletsPolarExp static method*), 294
PositionLikelihood (*class in lenstronomy.Sampling.Likelihoods.position_likelihood*), 323
pot_ext() (*ElliSLICE method*), 155
pot_in() (*ElliSLICE static method*), 155
potential() (*LensModel method*), 263
potential() (*SinglePlane method*), 272
potential() (*SinglePlaneLOS method*), 103
potential_from_kappa_grid() (*in module lenstronomy.LensModel.convergence_integrals*), 258
potential_from_kappa_grid_adaptive() (*in module lenstronomy.LensModel.convergence_integrals*), 258
potential_kernel() (*in module lenstronomy.LensModel.convergence_integrals*), 259
power_spectrum_1d() (*in module lenstronomy.Util.correlation*), 360
power_spectrum_2d() (*in module lenstronomy.Util.correlation*), 360
PowerLaw (*class in lenstronomy.LightModel.Profiles.power_law*), 286
PowerLawFixedShear (*class in lenstronomy.LensModel.QuadOptimizer.param_manager*), 249
PowerLawFixedShearMultipole (*class in lenstronomy.LensModel.QuadOptimizer.param_manager*), 249
PowerLawFreeShear (*class in lenstronomy.LensModel.QuadOptimizer.param_manager*), 250
PowerLawFreeShearMultipole (*class in lenstronomy.LensModel.QuadOptimizer.param_manager*),

250
PowerLawParamManager (*class in lenstronomy.LensModel.QuadOptimizer.param_manager*), 250
pre_calc() (*CartShapelets method*), 220
pre_calc() (*Shapelets method*), 291
print_setting() (*Param method*), 340
prior() (*DynestySampler method*), 330
prior() (*DyPolyChordSampler method*), 332
prior() (*MultiNestSampler method*), 331
prior() (*NestedSampler method*), 329
PriorLikelihood (*class in lenstronomy.Sampling.Likelihoods.prior_likelihood*), 325
profile_center() (*in module lenstronomy.Util.analysis_util*), 357
profile_name (*GNFW attribute*), 179
profile_name (*NFW attribute*), 198
profile_name (*NFW_ELLIPSE attribute*), 200
profile_name (*NFW_ELLIPSE_CSE attribute*), 201
profile_name (*TNFW attribute*), 241
profile_name (*TNFW_ELLIPSE attribute*), 243
profile_slope() (*LensProfileAnalysis method*), 35
ProfileIntegrals (*class in lenstronomy.LensModel.profile_integrals*), 269
ProfileListBase (*class in lenstronomy.LensModel.profile_list_base*), 270
project2d_random() (*in module lenstronomy.GalKin.velocity_util*), 75
projected_mass() (*SersicEllipseKappa method*), 218
ps (*in module lenstronomy.LensModel.Util.epl_util*), 258
PSBase (*class in lenstronomy.PointSource.Types.base_ps*), 312
PSF (*class in lenstronomy.Data.psf*), 55
PSF (*class in lenstronomy.GalKin.psf*), 73
psf_class (*Observation attribute*), 352
psf_error_map (*PSF attribute*), 56
psf_error_map() (*PointSourceRendering method*), 89
psf_estimate_individual() (*PsfFitting method*), 390
psf_iteration() (*FittingSequence method*), 385
psf_iteration_compare() (*in module lenstronomy.Plots.chain_plot*), 300
PsfFitting (*class in lenstronomy.Workflow.psf_fitting*), 387
PSFGaussian (*class in lenstronomy.GalKin.psf*), 74
PSFMoffat (*class in lenstronomy.GalKin.psf*), 74
pso() (*AlignmentFitting method*), 381
pso() (*FittingSequence method*), 385
pso() (*Sampler method*), 342

R

r200_M() (*NFWParam method*), 49
radec_at_xy_0 (*Coordinates attribute*), 51
Radial (*class in lenstronomy.GalKin.anisotropy*), 60
radial_lens_profile() (*LensProfileAnalysis method*), 35
radial_light_profile() (*LightProfileAnalysis method*), 37
radial_profile() (*in module lenstronomy.Util.analysis_util*), 356
radial_profile() (*in module lenstronomy.Util.image_util*), 366
radial_tangential_differentials() (*LensModelExtensions method*), 267
radial_tangential_stretch() (*LensModelExtensions method*), 267
ray_shooting() (*LensModel method*), 263
ray_shooting() (*MultiPlane method*), 106
ray_shooting() (*SinglePlane method*), 272
ray_shooting_fast() (*MultiplaneFast method*), 248
ray_shooting_partial() (*MultiPlane method*), 106
ray_shooting_partial() (*MultiPlaneBase method*), 108
re_size() (*in module lenstronomy.Util.image_util*), 365
re_size_array() (*in module lenstronomy.Util.image_util*), 365
re_size_convolve() (*AdaptiveConvolution method*), 82
re_size_convolve() (*MultiGaussianConvolution method*), 84
re_size_convolve() (*Numerics method*), 88
re_size_convolve() (*PixelKernelConvolution method*), 83
re_size_convolve() (*SubgridKernelConvolution method*), 84
rebin_coord_transform() (*in module lenstronomy.Util.image_util*), 365
rebin_image() (*in module lenstronomy.Util.image_util*), 365
reconstruction_all_bands() (*ModelPlot method*), 309
reduced_chi2() (*ImageLinearFit method*), 95
reduced_residuals() (*ImageLinearFit method*), 95
reduced_residuals() (*MultiDataBase method*), 78
RegularGrid (*class in lenstronomy.ImSim.Numerics.grid*), 86
reset_point_source_cache() (*ImageLikelihood method*), 323
reset_point_source_cache() (*ImageModel method*), 97

reset_point_source_cache() (*MultiDataBase method*), 78
result_string() (*in module lenstronomy.Plots.plot_util*), 311
rho02alpha() (*GNFW method*), 179
rho02alpha() (*NFW static method*), 198
rho02alpha() (*TNFW static method*), 241
rho0_c() (*NFWParam method*), 49
rho2sigma() (*Hernquist method*), 182
rho2sigma() (*PJaffe method*), 212
rho2theta() (*SIS static method*), 228
rho2theta() (*SPP static method*), 238
rho_crit (*Background attribute*), 43
rhoc (*NFWParam attribute*), 50
rhoc_z() (*NFWParam method*), 50
rhotilde() (*CoredDensityExp static method*), 137
rhotilde() (*Uldm static method*), 246
right_multiply() (*LOS static method*), 101
rotate (*in module lenstronomy.Util.util*), 376
rotateImage() (*in module lenstronomy.Util.image_util*), 365
rotmat (*in module lenstronomy.LensModel.Util.epl_util*), 258
run() (*DynestySampler method*), 330
run() (*DyPolyChordSampler method*), 332
run() (*MultiNestSampler method*), 331
run() (*NestedSampler method*), 329

S

sample_ball() (*in module lenstronomy.Util.sampling_util*), 375
sample_ball_truncated() (*in module lenstronomy.Util.sampling_util*), 375
Sampler (*class in lenstronomy.Sampling.sampler*), 341
scale_bar() (*in module lenstronomy.Plots.plot_util*), 309
scale_limits() (*in module lenstronomy.Util.sampling_util*), 374
select_best() (*in module lenstronomy.Util.util*), 379
select_kwargs() (*SingleBandMultiModel method*), 81
selectBest() (*in module lenstronomy.Util.util*), 379
Sersic (*class in lenstronomy.LensModel.Profiles.sersic*), 216
Sersic (*class in lenstronomy.LightModel.Profiles.sersic*), 288
sersic_k_eff2m_star() (*LensCosmo method*), 47
sersic_m_star2k_eff() (*LensCosmo method*), 47
SersicEllipse (*class in lenstronomy.LensModel.Profiles.sersic_ellipse_potential*), 218
SersicEllipseGaussDec (*class in lenstronomy.LensModel.Profiles.gauss_decomposition*), 166
SersicEllipseKappa (*class in lenstronomy.LensModel.Profiles.sersic_ellipse_kappa*), 217
SersicElliptic (*class in lenstronomy.LightModel.Profiles.sersic*), 288
SersicUtil (*class in lenstronomy.LensModel.Profiles.sersic_utils*), 218
set_amplitudes() (*PointSource method*), 318
set_dynamic() (*Chameleon method*), 114
set_dynamic() (*DoubleChameleon method*), 117
set_dynamic() (*EPL method*), 157
set_dynamic() (*LensModel method*), 264
set_dynamic() (*LensProfileBase method*), 111
set_dynamic() (*LOS method*), 101
set_dynamic() (*MultiPlane method*), 107
set_dynamic() (*NFWMC method*), 202
set_dynamic() (*NIE method*), 205
set_dynamic() (*NIE_POTENTIAL method*), 207
set_dynamic() (*ProfileListBase method*), 270
set_dynamic() (*TripleChameleon method*), 119
set_init_state() (*UpdateManager method*), 393
set_param_value() (*FittingSequence method*), 386
set_params() (*ArrayParam method*), 346
set_params() (*LensParam method*), 269
set_params() (*LightParam method*), 299
set_params() (*ModelParamGroup method*), 345
set_params() (*PointSourceParam method*), 321
set_params() (*SingleParam method*), 346
set_params() (*SpecialParam method*), 343
set_pixel_size() (*PSF method*), 56
set_save_cache() (*PointSource method*), 318
set_save_cache() (*PointSourceCached method*), 320
set_static() (*Chameleon method*), 114
set_static() (*DoubleChameleon method*), 117
set_static() (*EPL method*), 157
set_static() (*LensModel method*), 264
set_static() (*LensProfileBase method*), 111
set_static() (*LOS method*), 101
set_static() (*MultiPlane method*), 107
set_static() (*NFWMC method*), 202
set_static() (*NIE method*), 205
set_static() (*NIE_POTENTIAL method*), 207
set_static() (*ProfileListBase method*), 270
set_static() (*TripleChameleon method*), 119
setup() (*AlignmentLikelihood method*), 382
sgn() (*GaussianEllipseKappa static method*), 171
shapelet_basis_2d() (*ShapeletSet method*), 293
Shapelets (*class in lenstronomy.LightModel.Profiles.shapelets*), 290
ShapeletSet (*class in lenstronomy.LightModel.Profiles.shapelets*), 292
ShapeletSetPolar (*class in lenstronomy.LightModel.Profiles.shapelets_polar*),

295
ShapeletsPolar (*class in lenstronomy.LightModel.Profiles.shapelets_polar*), 293
ShapeletsPolarExp (*class in lenstronomy.LightModel.Profiles.shapelets_polar*), 294
Shear (*class in lenstronomy.LensModel.Profiles.shear*), 221
shear_cartesian2polar() (*in module lenstronomy.Util.param_util*), 372
shear_polar2cartesian() (*in module lenstronomy.Util.param_util*), 371
ShearGammaPsi (*class in lenstronomy.LensModel.Profiles.shear*), 222
ShearReduced (*class in lenstronomy.LensModel.Profiles.shear*), 223
Shell (*class in lenstronomy.GalKin.aperture_types*), 64
shell_ifu_select() (*in module lenstronomy.GalKin.aperture_types*), 65
shell_select() (*in module lenstronomy.GalKin.aperture_types*), 65
Shift (*class in lenstronomy.LensModel.Profiles.constant_shift*), 125
shift_coordinate_system() (*Coordinates method*), 51
shift_coordinate_system() (*PixelGrid method*), 55
SIE (*class in lenstronomy.LensModel.Profiles.sie*), 224
sigma2fwhm() (*in module lenstronomy.Util.util*), 380
sigma2rho() (*Hernquist method*), 182
sigma2rho() (*PJaffe method*), 212
sigma_crit (*LensCosmo attribute*), 47
sigma_crit_angle (*LensCosmo attribute*), 47
sigma_function() (*GaussianEllipseKappa method*), 171
sigma_kwargs (*UpdateManager attribute*), 393
sigma_r2() (*AnalyticKinematics method*), 57
sigma_r2() (*NumericKinematics method*), 72
sigma_s2() (*AnalyticKinematics method*), 58
sigma_s2() (*NumericKinematics method*), 72
sigma_s2_project() (*NumericKinematics method*), 72
sigma_s2_r() (*NumericKinematics method*), 73
sign() (*ElliSLICE static method*), 155
SimAPI (*class in lenstronomy.SimulationAPI.sim_api*), 355
simplex() (*FittingSequence method*), 386
simplex() (*Sampler method*), 342
simulate_simple() (*in module lenstronomy.Util.simulation_util*), 375
SingleBand (*class in lenstronomy.SimulationAPI.observation_api*), 352
SingleBandMultiModel (*class in lenstronomy.ImSim.MultiBand.single_band_multi_model*), 80
SingleParam (*class in lenstronomy.Sampling.param_group*), 345
SinglePlane (*class in lenstronomy.LensModel.single_plane*), 270
SinglePlaneLOS (*class in lenstronomy.LensModel.LineOfSight.single_plane_los*), 102
SIS (*class in lenstronomy.LensModel.Profiles.sis*), 227
sis_mst2stretch() (*CurvedArcSISMST static method*), 147
sis_sigma_v2theta_E() (*LensCosmo method*), 47
sis_theta_E2sigma_v() (*LensCosmo method*), 48
SIS_truncate (*class in lenstronomy.LensModel.Profiles.sis_truncate*), 229
SkewGaussian (*class in lenstronomy.Util.prob_density*), 373
sky_brightness (*SingleBand attribute*), 354
Slit (*class in lenstronomy.GalKin.aperture_types*), 63
slit_select() (*in module lenstronomy.GalKin.aperture_types*), 63
SLIT_Starlets (*class in lenstronomy.LightModel.Profiles.starlets*), 295
solve() (*Solver2Point method*), 256
solve() (*Solver4Point method*), 257
solve() (*SolverFlatLCDM method*), 43
solvequadeq (*in module lenstronomy.LensModel.Util.epl_util*), 258
Solver (*class in lenstronomy.LensModel.Solver.solver*), 255
Solver2Point (*class in lenstronomy.LensModel.Solver.solver2point*), 256
Solver4Point (*class in lenstronomy.LensModel.Solver.solver4point*), 256
SolverFlatLCDM (*class in lenstronomy.Cosmo.cosmo_solver*), 43
sort_arrival_times() (*LensEquationSolver method*), 254
sort_image_index() (*in module lenstronomy.Util.util*), 376
source() (*ModelBandPlot method*), 305
source() (*ModelPlot method*), 309
source() (*MultiPatchReconstruction method*), 38
source_amplitude() (*LensedPositions method*), 313
source_amplitude() (*PointSource method*), 318
source_amplitude() (*PointSourceCached method*), 320
source_amplitude() (*PSBase method*), 312
source_amplitude() (*SourcePositions method*), 315
source_amplitude() (*Unlensed method*), 315

source_model_class (*ModelAPI attribute*), 351
source_plane_chi_square() (*MultiplaneFast method*), 248
source_plot() (*ModelBandPlot method*), 305
source_plot() (*ModelPlot method*), 309
source_position() (*LensedPositions method*), 313
source_position() (*PointSource method*), 318
source_position() (*PointSourceCached method*), 320
source_position() (*PSBase method*), 312
source_position() (*SourcePositions method*), 315
source_position() (*Unlensed method*), 315
source_position_likelihood() (*PositionLikelihood method*), 325
source_position_plot() (*in module lenstronomy.Plots.plot_util*), 311
source_surface_brightness() (*ImageModel method*), 97
SourcePositions (*class in lenstronomy.PointSource.Types.source_position*), 314
SpecialParam (*class in lenstronomy.Sampling.special_param*), 343
SPEMD (*class in lenstronomy.LensModel.Profiles.spemd*), 229
SPEP (*class in lenstronomy.LensModel.Profiles.spep*), 231
SPLCORE (*class in lenstronomy.LensModel.Profiles.splcore*), 233
split_kernel() (*in module lenstronomy.Util.kernel_util*), 368
split_lens_los() (*SinglePlaneLOS method*), 104
SPP (*class in lenstronomy.LensModel.Profiles.spp*), 236
spp2stretch() (*CurvedArcSPP static method*), 149
sqrt() (*in module lenstronomy.Plots.plot_util*), 309
stack_images() (*in module lenstronomy.Util.image_util*), 365
stretch2sie_mst() (*CurvedArcTanDiff static method*), 152
stretch2sis_mst() (*CurvedArcSISMST static method*), 147
stretch2spp() (*CurvedArcSPP static method*), 149
subgrid_kernel() (*in module lenstronomy.Util.kernel_util*), 367
SubgridKernelConvolution (*class in lenstronomy.ImSim.Numerics.convolution*), 83
subtract_from_data_plot() (*ModelBandPlot method*), 306
subtract_from_data_plot() (*ModelPlot method*), 309
supersampling_factor (*RegularGrid attribute*), 86
symmetry_average() (*in module lenstronomy.Util.image_util*), 365

T

T_xy() (*Background method*), 42
tangential_average() (*LensModelExtensions method*), 268
TDCosmography (*class in lenstronomy.Analysis.td_cosmography*), 39
text_description() (*in module lenstronomy.Plots.plot_util*), 309
theta2rho() (*SIE static method*), 227
theta2rho() (*SIS static method*), 228
theta2rho() (*SPP static method*), 238
time_delay2fermat_pot() (*LensCosmo method*), 48
time_delay_units() (*LensCosmo method*), 48
time_delays() (*TDCosmography method*), 40
TimeDelayLikelihood (*class in lenstronomy.Sampling.Likelihoods.time_delay_likelihood*), 326
TNFW (*class in lenstronomy.LensModel.Profiles.tnfw*), 238
TNFW_ELLIPSE (*class in lenstronomy.LensModel.Profiles.tnfw_ellipse*), 241
to_vary_index (*PowerLawFixedShearMultipole attribute*), 250
to_vary_index (*PowerLawFreeShearMultipole attribute*), 250
to_vary_index (*PowerLawParamManager attribute*), 251
total_flux() (*Gaussian method*), 277
total_flux() (*GaussianEllipse method*), 278
total_flux() (*Interpol static method*), 283
total_flux() (*MultiGaussian method*), 279
total_flux() (*MultiGaussianEllipse method*), 280
total_flux() (*SersicUtil method*), 219
transform() (*in module lenstronomy.LightModel.Profiles.starlets_util*), 297
transform_angle2pix (*Coordinates attribute*), 51
transform_e1e2_product_average() (*in module lenstronomy.Util.param_util*), 372
transform_e1e2_square_average() (*in module lenstronomy.Util.param_util*), 372
transform_fg() (*Flexionfg static method*), 162
transform_kappa_ext() (*KinematicsAPI static method*), 32
transform_pix2angle (*Coordinates attribute*), 51
transverse_distance_start_stop() (*MultiPlane method*), 107
transverse_distance_start_stop() (*MultiPlaneBase method*), 109
TripleChameleon (*class in lenstronomy.LensModel.Profiles.chameleon*), 117
TripleChameleon (*class in lenstronomy.LightModel.Profiles.chameleon*), 274

U

Uldm (*class in lenstronomy.LensModel.Profiles.uldm*), 243
uldm_angular2phys() (*LensCosmo method*), 48
uldm_mphys2angular() (*LensCosmo method*), 48
Uniform (*class in lenstronomy.LightModel.Profiles.uniform*), 297
uniform2unit() (*in module lenstronomy.Util.sampling_util*), 374
unit2uniform() (*in module lenstronomy.Util.sampling_util*), 374
Unlensed (*class in lenstronomy.PointSource.Types.unlensed*), 315
update_data() (*AlignmentLikelihood method*), 382
update_data() (*ImageData method*), 54
update_data() (*ImageLinearFit method*), 95
update_fixed() (*UpdateManager method*), 393
update_iterative() (*PsfFitting method*), 390
update_kwargs_model() (*Param method*), 340
update_lens_model() (*PointSource method*), 319
update_lens_model() (*PointSourceCached method*), 320
update_lens_model() (*PSBase method*), 312
update_lens_scaling() (*Param method*), 340
update_limits() (*UpdateManager method*), 393
update_linear() (*PointSource method*), 319
update_linear_kwargs() (*ImageLinearFit method*), 95
update_multi_band() (*AlignmentLikelihood method*), 382
update_observation() (*Observation method*), 352
update_options() (*UpdateManager method*), 393
update_param_state() (*UpdateManager method*), 394
update_param_value() (*UpdateManager method*), 394
update_pixel_kwargs() (*ImageLinearFit method*), 96
update_psf() (*ImageModel method*), 98
update_psf() (*PsfFitting method*), 391
update_search_window() (*PointSource method*), 319
update_settings() (*FittingSequence method*), 386
update_sigmas() (*UpdateManager method*), 394
update_solver() (*Solver method*), 255
update_source_redshift() (*MultiPlane method*), 107
update_state() (*FittingSequence method*), 387
UpdateManager (*class in lenstronomy.Workflow.update_manager*), 392
upper_limit_default (*CartShapelets attribute*), 220
upper_limit_default (*Chameleon attribute*), 114, 273
upper_limit_default (*CNFW attribute*), 123
upper_limit_default (*CNFW_ELLIPSE attribute*), 123
upper_limit_default (*ConstMag attribute*), 124
upper_limit_default (*Convergence attribute*), 126
upper_limit_default (*CoreBurkert attribute*), 128
upper_limit_default (*CoredDensity attribute*), 131
upper_limit_default (*CoredDensity2 attribute*), 134
upper_limit_default (*CoredDensityExp attribute*), 137
upper_limit_default (*CoredDensityMST attribute*), 138
upper_limit_default (*CoreSersic attribute*), 290
upper_limit_default (*CSE attribute*), 139
upper_limit_default (*CSEMajorAxis attribute*), 140
upper_limit_default (*CSEProductAvg attribute*), 142
upper_limit_default (*CTNFWGaussDec attribute*), 169
upper_limit_default (*CurvedArcConst attribute*), 145
upper_limit_default (*CurvedArcConstMST attribute*), 144
upper_limit_default (*CurvedArcSISMST attribute*), 147
upper_limit_default (*CurvedArcSPP attribute*), 149
upper_limit_default (*CurvedArcSPT attribute*), 151
upper_limit_default (*CurvedArcTanDiff attribute*), 153
upper_limit_default (*Dipole attribute*), 153
upper_limit_default (*DoubleChameleon attribute*), 117, 274
upper_limit_default (*DoubleChameleonPointMass attribute*), 121
upper_limit_default (*ElliSLICE attribute*), 155
upper_limit_default (*EPL attribute*), 158
upper_limit_default (*EPL_numba attribute*), 160
upper_limit_default (*Flexion attribute*), 161
upper_limit_default (*Flexionfg attribute*), 162
upper_limit_default (*Gaussian attribute*), 176
upper_limit_default (*GaussianEllipse attribute*), 278
upper_limit_default (*GaussianEllipseKappa attribute*), 172
upper_limit_default (*GaussianEllipseKappaSet attribute*), 164
upper_limit_default (*GaussianEllipsePotential attribute*), 174

upper_limit_default (*GaussianKappa attribute*), 175
upper_limit_default (*GNFW attribute*), 179
upper_limit_default (*Hernquist attribute*), 182
upper_limit_default (*Hernquist_Ellipse attribute*), 184
upper_limit_default (*HernquistEllipse attribute*), 282
upper_limit_default (*HernquistEllipseCSE attribute*), 184
upper_limit_default (*Hessian attribute*), 186
upper_limit_default (*Interpol attribute*), 188, 283
upper_limit_default (*InterpolScaled attribute*), 190
upper_limit_default (*LOS attribute*), 101
upper_limit_default (*LOSMinimal attribute*), 101
upper_limit_default (*MultiGaussian attribute*), 279
upper_limit_default (*MultiGaussianEllipse attribute*), 281
upper_limit_default (*MultiGaussianKappa attribute*), 191
upper_limit_default (*MultiGaussianKappaEllipse attribute*), 193
upper_limit_default (*Multipole attribute*), 194
upper_limit_default (*NFW attribute*), 198
upper_limit_default (*NFW_ELLIPSE attribute*), 200
upper_limit_default (*NFW_ELLIPSE_CSE attribute*), 201
upper_limit_default (*NFWEllipseGaussDec attribute*), 168
upper_limit_default (*NFWMC attribute*), 202
upper_limit_default (*NIE attribute*), 205, 285
upper_limit_default (*NIE_POTENTIAL attribute*), 207
upper_limit_default (*PEMD attribute*), 215
upper_limit_default (*PJaffe attribute*), 212, 285
upper_limit_default (*PJaffe_Ellipse attribute*), 213
upper_limit_default (*PJaffeEllipse attribute*), 286
upper_limit_default (*PointMass attribute*), 216
upper_limit_default (*PolarShapelets attribute*), 221
upper_limit_default (*PowerLaw attribute*), 287
upper_limit_default (*Sersic attribute*), 217, 288
upper_limit_default (*SersicEllipse attribute*), 218
upper_limit_default (*SersicEllipseGaussDec attribute*), 167
upper_limit_default (*SersicEllipseKappa attribute*), 218
upper_limit_default (*SersicElliptic attribute*), 289
upper_limit_default (*Shapelets attribute*), 291
upper_limit_default (*ShapeletSet attribute*), 293
upper_limit_default (*ShapeletSetPolar attribute*), 295
upper_limit_default (*ShapeletsPolar attribute*), 294
upper_limit_default (*ShapeletsPolarExp attribute*), 295
upper_limit_default (*Shear attribute*), 222
upper_limit_default (*ShearGammaPsi attribute*), 223
upper_limit_default (*ShearReduced attribute*), 224
upper_limit_default (*Shift attribute*), 125
upper_limit_default (*SIE attribute*), 227
upper_limit_default (*SIS attribute*), 228
upper_limit_default (*SIS_truncate attribute*), 229
upper_limit_default (*SLIT_Starlets attribute*), 297
upper_limit_default (*SPEMD attribute*), 231
upper_limit_default (*SPEP attribute*), 233
upper_limit_default (*SPLCORE attribute*), 235
upper_limit_default (*SPP attribute*), 238
upper_limit_default (*TNFW attribute*), 241
upper_limit_default (*TNFW_ELLIPSE attribute*), 243
upper_limit_default (*TripleChameleon attribute*), 119, 276
upper_limit_default (*Uldm attribute*), 246
upper_limit_default (*Uniform attribute*), 297

V

velocity_dispersion() (*KinematicsAPI method*), 32
velocity_dispersion_analytical() (*KinematicsAPI method*), 32
velocity_dispersion_dimension_less() (*TDCosmography method*), 41
velocity_dispersion_map() (*KinematicsAPI method*), 33
velocity_dispersion_map_dimension_less() (*TDCosmography method*), 41

W

w_f_approx() (*GaussianEllipseKappa static method*), 172
wait_timeout (*MultiPool attribute*), 328
width (*PixelGrid attribute*), 55

X

xy_at_radec_0 (*Coordinates attribute*), 51

Z

zoom_source() (*LensModelExtensions method*), 268